KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

PARADIGMATA PROGRAMOVÁNÍ 1A

JAN KONEČNÝ, VILÉM VYCHODIL

VÝVOJ TOHOTO UČEBNÍHO TEXTU JE SPOLUFINANCOVÁN
EVROPSKÝM SOCIÁLNÍM FONDEM A STÁTNÍM ROZPOČTEM ČESKÉ REPUBLIKY

Olomouc 4. 10. 2010 (aktualizace)

Abstrakt

Série učebních textů z paradigmat programování seznamuje čtenáře s vybranými styly v programování. Tento text je zaměřen na problematiku funkcionálního programování. Text je koncipován jako cvičebnice, teoretické partie jsou prezentovány v redukované podobě a důraz je kladen na prezentaci příkladů a protipříkladů, na kterých jsou dané problémy demonstrovány. V textu se nacházejí klasické partie z funkcionálního programování (například procedury vyšších řádů, rekurze, indukce, práce s hierarchickými daty), ale i partie, které jsou v soudobé literatuře zastoupeny minimálně nebo vůbec (například detailní popis vyhodnocovacího procesu a konstrukce interpretu funkcionálního jazyka). Text je rozdělen do dvanácti na sebe navazujících lekcí. Pro správné pochopení a procvičení problematiky je vhodné číst lekce postupně bez přeskakování a snažit se řešit všechny úkoly. Text u čtenářů nepředpokládá žádné znalosti programování. Pro pochopení většiny příkladů stačí mít znalost středoškolské matematiky. Příklady vyžadující znalosti (úvodních kurzů) vysokoškolské matematiky jsou doplněny o odkazy na vhodnou literaturu.

Jan Konečný, <jan.konecny@upol.cz>
Vilém Vychodil, <vilem.vychodil@upol.cz>

Cílová skupina

Text je určen pro studenty oborů Aplikovaná informatika a Informatika uskutečňovaných v prezenční a kombinované formě na Přírodovědecké fakultě Univerzity Palackého v Olomouci. Může být užitečný i studentům jiných informatických, matematických a inženýrských oborů a všem, kteří se chtějí seznámit s různými styly programování.

Obsah

Lekce 1: Program, jeho syntax a sémantika

Obsah lekce: Tato lekce je úvodem do paradigmat programování. Vysvětlíme jaký je rozdíl mezi programem a výpočetním procesem, co jsou programovací jazyky a čím se od sebe odlišují. Uvedeme několik základních paradigmat (stylů) programování a stručně je charakterizujeme. Dále objasníme rozdíl mezi syntaxí (tvarem zápisu) a sémantikou (významem) programů. Představené pojmy budeme demonstrovat na jazyku Scheme. Programy v jazyku Scheme zavedeme jako posloupnosti symbolických výrazů a popíšeme jejich interpretaci. Zavedeme pojem elementy jazyka a ukážeme několik důležitých typů elementů jazyka, mimo jiné procedury a speciální formy. V závěru kapitoly popíšeme dvě speciální formy sloužící k zavádění nových definic a k podmíněnému vyhodnocování výrazů.

Klíčová slova: abstraktní interpret, aplikace procedur a speciálních forem, cyklus REPL, elementy jazyka, evaluator, externí reprezentace, interní reprezentace, interpretace, jazyk Scheme, paradigma, printer, procedury, program, překlad, reader, speciální formy symbolické výrazy, syntaktická chyba, syntaxe, sémantická chyba, sémantika, výpočetní proces.

1.1 Programovací jazyk, program a výpočetní proces

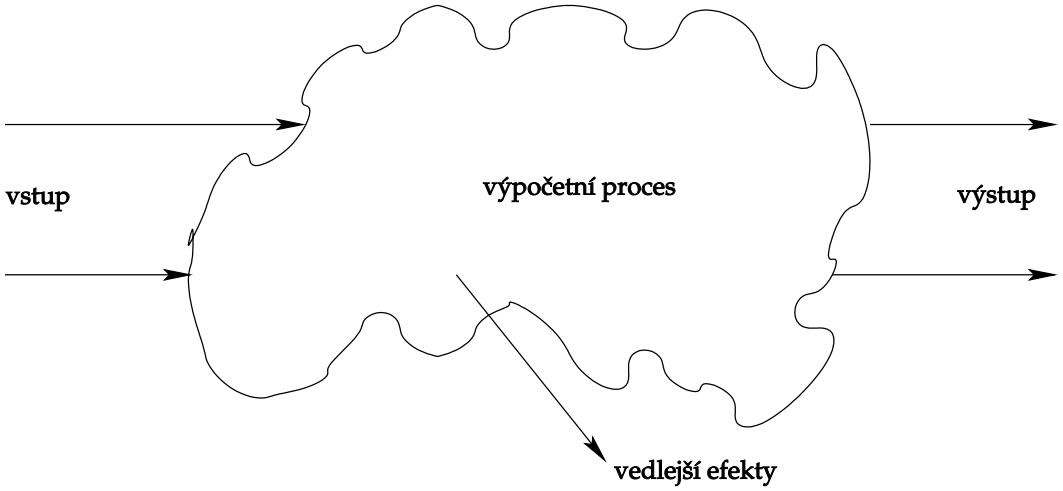
V současnosti je zřejmé, že úlohou počítačů není jen „provádět výpočty“, jak by jejich název mohl napovídat. Z dnešního pohledu se lze na počítače dívat obecně jako na *stroje zpracovávající data.* Samotný proces, při kterém jsou (nějaká) vstupní data zpracovávána nazýváme *výpočetní proces.* Výpočetní proces je dost abstraktní pojem. Pro nás však není příliš důležité rozebírat, jak si tento pojem „představit“. Důležité je chápat výpočetní proces jako proces s počátkem a koncem, probíhající v nějakých elementárních krocích, přetvářející vstupní data (vstupní parametry úlohy) na data výstupní (řešení). Takový pohled na výpočetní proces má své historické kořeny v prvopočátcích rozvoje počítačů, kde skutečně šlo jen o načtení vstupních dat, jejich zpracování a následný zápis výstupních dat. Z dnešního pohledu je tato charakterizace poněkud zjednodušená, protože uživatelé počítačů nepracují s počítači jen tímto primitivním „dávkovým způsobem“. V současné době mají výpočetní procesy celou řadu *vedlejších efektů,* jejichž účelem je například interakce s uživatelem. Viz schéma na obrázku 1.1.

Z pohledu informatika je důležité vědět, jak výpočetní procesy vznikají: výpočetní proces je důsledkem *vykonávání programu.* Každý *program* je předpisem s přesně daným tvarem a významem, podle kterého vzniká výpočetní proces. Smluvená pravidla, v souladu se kterými je program vytvořen, nazýváme *programovací jazyk.* Říkáme, že program *je napsán* v programovacím jazyku.

Na rozdíl od výpočetního procesu je tedy program „pasivní entita“ (sám o sobě nic nevykonává). V soudobých počítačích jsou programy reprezentovány soubory s přesně danou strukturou, které jsou uložené na disku počítače (ačkoliv tomu tak vždy nebylo). To jest samotné programy lze chápat jako data a obráceně (jak bude patrné především v dalších lekcích). Účelem tohoto učebního textu je seznámit studenty se základními *styly* (*paradigmaty*) *programování,* to jest styly vytváření programů, které jsou sdíleny mezi různými programovacími jazyky. Tvůrčí činnost vytváření programů se nazývá *programování,* člověk zabývající se programováním je *programátor.*

Úkolem programátora je vytvořit program tak, aby vykonáváním programu vznikl požadovaný výpočetní proces řešící přesně stanovený úkol. Kvalita programu se bezprostředně promítá do kvality výpočetního procesu a v důsledku do kvality práce člověka (uživatele) s počítačem. Ačkoliv je při počítačovém zpracování dat ústředním zájmem *výpočetní proces,* který samotné zpracování provádí, pozornost programátorů je obvykle směřována na *program,* to jest na předpis, podle kterého výpočetní proces vzniká. Program je potřeba vytvořit dostatečně kvalitně tak, aby neobsahoval chyby, aby byl efektivní, a tak dále. Ačkoliv tato kritéria zní na první poslech triviálně, s programy jejichž činnost je komplikována množstvím chyb, malou efektivitou, nebo nevhodnou komunikací s uživatelem, se bohužel setkáváme dnes a denně.

Obrázek 1.1. Výpočetní proces jako abstraktní entita



Klíčem k vytváření kvalitních programů jenž povedou na žádoucí výpočetní procesy je pochopení pravidel daných programovacím jazykem (jak program zapisujeme, jaký význam mají konkrétní části programu, . . .) a pochopení, jak na základě vytvořeného programu vzniká výpočetní proces, a jak výpočetní proces probíhá. Těmito problematikami se budeme věnovat ve zbytku textu (a v dalších jeho dílech).

Při studiu problematiky programů a výpočetních procesů jimi generovaných budeme sledovat linii *programovacích jazyků.* Důvody jsou ryze pragmatické. Programovacích jazyků existuje celá řada (v současné době řádově v tisících), přitom různé programovací jazyky vyžadují od programátora při vytváření programů jiný styl myšlení, jiný způsob analýzy problému a jiné metody vytváření a testování programu. Není pochopitelně v našich silách zabývat se všemi možnými programovacími jazyky. Proto rozdělíme (téměř) všechny (současně používané) programovací jazyky do malého počtu (částečně se překrývajících) skupin. Každou skupinu budou tvořit jazyky s podobnými vlastnostmi – jazyky vyžadující od programátora stejnou metodu vývoje programu a myšlení nad problémem a programem samotným. Jak dále uvidíme, každá tato skupina jazyků bude korespondovat s jedním hlavním *typickým stylem programování – paradigmatem.*

Poznámka 1.1. Všimněte si, že doposud nepadl pojem *algoritmus.* Ani jsme se nezmínili o tom, jaké *typy problémů* se chystáme řešit (pomocí počítače). Důvodem je fakt, že pojmy „algoritmus“ a „problém“ lze chápat na jednu stranu neformálně (což je poněkud ošidné) a na druhou stranu formálně, což ale vyžaduje hlubší studium problematiky, kterou se v tomto kurzu nezabýváme. Studenti se budou algoritmy, problémy, řešitelností problémů a jejich složitostí zabývat v samostatném kurzu *vyčíslitelnosti a složitosti*. K této poznámce dodejme jen tolik, že výsledky dosažené v teoretické informatice ukázaly několik důležitých faktů, které je potřeba mít neustále na paměti:

(i) zdaleka ne každý problém je řešitelný,

(ii) zdaleka ne každý řešitelný problém lze řešit v přijatelné době,

(iii) všechny programovací jazyky, *které jsou Turingovsky úplné,* mají stejnou vyjadřovací sílu.

Bod (iii) potřebuje detailnější vysvětlení. Programovací jazyk nazýváme *Turingovsky úplný,* pokud je z hlediska řešitelnosti problémů ekvivalentní tak zvanému Turingovu stroji, což je jeden z formálních modelů algoritmu. Všechny běžně používané programovací jazyky jsou Turingovsky úplné (samozřejmě, že lze záměrně vytvořit i jazyk „chudší“, který Turingovsky úplný nebude, ale takovými jazyky se nebudeme zabývat). To tedy znamená, že všechny tyto jazyky jsou vzájemně ekvivalentní z hlediska své výpočetní síly. Neformálně řečeno, je-li nějaký problém možné vyřešit pomocí jednoho z těchto jazyků, je možné jej vyřešit v libovolném z nich. Všechna předchozí pozorování budou zpřesněna v kurzu *vyčíslitelnosti a složitosti.* Pro nás je v tuto chvíli důležité vědět, že z hlediska možnosti řešit problémy jsou si „všechny programovací jazyky rovny“.

Pozorování o rovnocennosti programovacích jazyků má dalekosáhlé důsledky. Říká, mimo jiné, že pokud nemůžeme problém vyřešit v jednom jazyku, pak nám přechod k jinému nikterak nepomůže, což je mimochodem dost důležitý fakt, o kterém řada „ostřílených programátorů z praxe“ *vůbec neví.* Na druhou stranu je ale potřeba zdůraznit, že mezi programovacími jazyky jsou propastné rozdíly v tom, jaký umožňují programátorovi použít aparát při psaní programu a (v důsledku) při řešení problému. Program, řešící jeden typ problému, může být v jednom programovacím jazyku napsán jednoduše, stručně a elegantně, kdežto v jiném jazyku může být napsání programu generujícího stejný výpočetní proces řádově složitější.

Nyní si stručně popíšeme, jakými programovacími jazyky se vlastně budeme zabývat. Z historického hlediska je možné rozdělit počítače (a metody programování) do dvou etap, z nichž první byla podstatně kratší než druhá:

Éra neprogramovatelných počítačů. V prvopočátcích vývoje počítačů byly počítače sestaveny jednoúčelově k vůli řešení konkrétního problému. Program byl fyzicky (hardwarově) součástí počítače. Pokud po vyřešení problémů již počítač nebyl potřeba, byl zpravidla demontován a rozebrán na součástky. Příkladem jednoúčelového neprogramovatelného počítače byly tzv. „Turingovy bomby“, což byly počítače vyráběné během válečných let 1939–1945, které byly během II. světové války používány k dešifrování německého kódovacího zařízení Enigma. Počítače společně navrhli britští vědci Alan Turing a Gordon Welchman, k jejich úplnému odtajnění došlo až v 90. letech minulého století.

Éra programovatelných počítačů. Jednorázová konstrukce počítačů byla velmi neekonomická, takže přirozeným vývojem bylo vytvořit univerzální počítač, který by mohl být programovatelný pro provádění různých úloh. Tento koncept se udržel až do současnosti, i když pochopitelně mezi programovatelností a schopnostmi počítačů v 50. letech minulého století a v současnosti je propastný rozdíl. Prvenství mezi programovatelnými počítači bývá přičítáno několika skupinám vývojářů. Snad úplně prvním počítačem v této kategorii byl počítač Z3, který uvedl do chodu v roce 1941 německý inženýr Konrad Zuse (1910–1995). Počítač byl sestrojen na zakázku koncernu BMW, technicky byl proveden pomocí relé, ale byl plně programovatelný. Původní počítač byl zničen během spojeneckého náletu na Berlín v roce 1944 (v roce 1960 byla vyrobena jeho replika). Zajímavé je, že v roce 1998 se podařilo dokázat, že Z3 (respektive jeho strojový jazyk, viz dále) byl Turingovsky úplný. Mezi další klasické zástupce patří britský počítač Colossus (1944), který byl sice konstruovaný jako jednoúčelový, ale byl omezeně programovatelný (účelem sestrojení počítače bylo dešifrování německých kódovacích systémů). Americký počítač ENIAC (1946), sestrojený k výpočtům dělostřeleckých tabulek v rámci amerického balistického výzkumného centra, byl programovatelný, pracoval však (na dnešní dobu netypicky) v desítkové soustavě. Počítač Z3 byl svou koncepcí mnohem blíže soudobým počítačům, protože pracoval v dvojkové soustavě.

V počátcích vývoje číslicových počítačů stálo několik významných odborníků s českým původem. Největší osobností byl bezpochyby Antonín Svoboda (1907–1980), který již v předválečných letech pracoval na automatickém zaměřovači pro protiletadlové dělostřelectvo a po jeho emigraci do USA pracoval na protiletadlovém zaměřovači pro válečné lodě, který byl vyvíjen na MIT v Bostonu. Po návratu do vlasti se stal ředitelem Ústavu matematických strojů Akademie Věd, kde vedl vývoj prvního československého číslicového počítače ve jménem SAPO (zkratka pro SAmočinný POčítač). Počítač SAPO byl v mnoha ohledech unikátní – jednalo se o první počítač, který využíval redundantních komponent ke zvýšení spolehlivosti. Počítač měl například tři aritmetické jednotky, které o výsledku operace rozhodovaly „hlasováním“ (dva stejné výsledky přehlasovaly třetí rozdílný, v případě tří různých výsledků se celý výpočet opakoval). Tím se výrazně zvýšila spolehlivost výpočtů prováděných aritmetickými jednotkami, které byly ve své době složeny ze součástek s nízkou spolehlivostí. Podobný princip byl použit podruhé až v počítačích řídících dráhu letu vesmírných lodí Apollo. Politické důvody donutily Antonína Svobodu v šedesátých letech podruhé emigrovat z vlasti. Dožil v USA, kde vychoval generaci počítačových expertů, a je právem považován za průkopníka informatiky ve světovém měřítku [KV08].

Programovací jazyky lze rozdělit na *nižší* a *vyšší* podle toho, jak moc jsou vázány na konkrétní hardware počítače a podle toho, jaké *prostředky abstrakce* dávají k dispozici programátorovi.

Nižší programovací jazyky jsou těsně vázané na hardware počítače. Programové konstrukce, které lze v nižších jazycích provádět jsou spjaty s instrukční sadou procesoru. Nižší programovací jazyky neobsahují prakticky žádné prostředky abstrakce, programování v nich je zdlouhavé, často vede k zanášení chyb do programu, které se velmi špatně odhalují (a někdy i špatně opravují). Naopak výhodou programu napsaného v nižším programovacím jazyku je jeho přímá vazba na hardware počítače, což umožňuje „plně počítač využít". V následujícím přehledu jsou uvedeni typičtí zástupci *nižších programovacích jazyků.*

Kódy stroje. Kódy stroje jsou historicky nejstarší. Představují vlastně programovací jazyky dané přímo konkrétním počítačem respektive jeho centrálním procesorem. Program v kódu stroje je de facto *sekvence jedniček a nul,* ve které jsou kódované instrukce pro procesor a data, nad kterými jsou instrukce prováděny. Kód stroje je vykonáván přímo procesorem. To jest, výpočetní proces vzniká z kódu stroje tím, že jej procesor postupně vykonává. Příklad programu v jazyku stroje (nějakého fiktivního počítače) by mohl vypadat takto:

`001 01000`	vlož číslo 8 do paměťové buňky
`011 00101`	přičti k obsahu buňky číslo 5
`100 00000`	vytiskni obsah buňky na obrazovku
`111 00000`	ukonči program

V tomto případě je každá instrukce i s hodnotou, se kterou je provedena (s tak zvaným *operandem*) kódována osmi bity. Přitom první tři bity tvoří kód instrukce a zbylých pět bitů je zakódovaný operand. Například v prvním řádku je „`001`" kódem instrukce pro vkládání hodnoty do paměťové buňky (registru) a „`01000`" je binární rozvoj čísla 8. Celý výše uvedený program zapsaný jako sekvence nul a jedniček tedy vypadá následovně: „`0010100001100101100000001110000`". Podle výše uvedeného slovního popisu se vlastně jedná o program, který sečte čísla 8 a 5 a výsledek vytiskne na obrazovku. Na to, že se jedná o primitivní program, je jeho zápis velmi nepřehledný a pouhá záměnou jedné nuly za jedničku nebo obráceně způsobí vznik chyby.

Assemblery. Assemblery vznikly jako pomůcka pro vytváření programů v kódu stroje. V programech psaných v assembleru se na rozdíl od kódu stroje používají pro zápis instrukcí jejich mnemotechnické (snadno zapamatovatelné) *zkratky.* Rovněž operandy není nutné uvádět pomocí jejich binárních rozvojů: například čísla je možné zapisovat v běžné dekadické notaci a na paměťové buňky (registry) se lze odkazovat jejich zkratkami. Dalším posunem oproti kódu stroje je možnost symbolického adresování. To znamená, že při použití instrukce skoku „skoč na jiné místo programu" se nemusí ručně počítat adresa, na kterou má být skočeno, ale assemblery umožňují pojmenovat cíl skoku pomocí *návěstí.* Předchozí program v kódu stroje by v assembleru pro příslušný fiktivní počítač mohl vypadat následovně:

```
load  8     vlož číslo 8 do paměťové buňky
add   5     přičti k obsahu buňky číslo 5
print       vytiskni obsah buňky na obrazovku
stop        ukonči program
```

Tento program je již výrazně čitelnější než sekvence jedniček a nul. Přesto se programování v assembleru v současnosti omezuje prakticky jen na programování (částí) operačních systémů nebo programů, které jsou extrémně kritické na výkon. Stejně jako v případě kódu stroje jsou však programy v assembleru náchylné ke vzniku chyb a psaní programů v assembleru je náročné a zdlouhavé.

Programy v assembleru jsou „textové soubory" se symbolickými názvy instrukcí. Nejedná se tedy přímo o kód stroje, takže procesor přímo nemůže program v assembleru zpracovávat. Při programování v assembleru je tedy nutné mít k dispozici překladač (kterému se říká rovněž *assembler*), který převede *program v čitelné zdrojové formě* do *kódu stroje,* viz předchozí ukázky. Vygenerovaný kód stroje je již přímo zpracovatelný počítačem. Důležité je uvědomit si, že samotný překladač je rovněž *program,* který „nějak pracuje" a někdo jej musí vytvořit.

Autokódy, bajtkódy, a jiné. Existují další představitelé nižších programovacích jazyků, například *autokódy,* které se během vývoje počítačů příliš nerozšířily, některé z této třídy jazyků poskytují programátorovi

prvky, které již lze z mnoha úhlů pohledu považovat za rysy vyšších jazyků. Stejně tak jako assemblery, programy zapsané v těchto jazycích musí být přeloženy do kódu stroje. *Bajtkódy* (anglicky *bytecode*) lze chápat jako jazyky stroje pro *virtuální procesory* (fyzicky neexistující). Abychom mohl být na základě programu v bajtkódu generován výpočetní proces, je potřeba mít opět k dispozici buď překladač bajtkódu do kódu stroje nebo program, který přímo provádí virtuální instrukce uvedené v bajtkódu.

Vyšší programovací jazyky naopak (od těch nižších) nejsou těsně vázané na hardware počítače a poskytují programátorovi výrazně vyšší stupeň abstrakce než nižší programovací jazyku. To umožňuje, aby programátor snadněji a rychleji vytvářel programy s menším rizikem vzniku chyb.

Čtyři nejstarší vyšší programovací jazyky, které doznaly většího rozšíření, byly (respektive jsou):

ALGOL První variantou byl ALGOL 58, který byl představen v roce 1958. Původně se jazyk měl jmenovat IAL (International Algorithmic Language), ale tato zkratka byla zavržena, protože se (anglicky) špatně vyslovovala. Autory návrhu jazyka byli A. Perlis a K. Samelson [PeSa58, Pe81]. Mnohem známější varianta jazyka, ALGOL 60, se objevila o dva roky později. ALGOLem bylo inspirováno mnoho dalších jazyků včetně PASCALu. Samotný ALGOL byl navržen jako obecný „algoritmický jazyk" pro vědecké výpočty.

COBOL Název COBOL je zkratkou pro „Common Business Oriented Language", což naznačuje, jaký byl účel vyvinutí tohoto jazyka. Jazyk COBOL vznikl z podnětu ministerstva obrany Spojených států jako obecný jazyk pro vytváření obchodních aplikací. Jedno z hlavních setkání, na kterých byl jazyk ustaven se konalo v Pentagonu v roce 1959. COBOL je v současnosti jedním z nejpoužívanějších jazyků pro programování obchodních a ekonomických aplikací. I když je vývoj nových programů na ústupu, stále existuje velké množství programů v COBOLu, které se neustále udržují.

FORTRAN Jazyk FORTRAN (původní název jazyka byl „The IBM Mathematical Formula Translating System") byl navržen k snadnému programování numerických výpočtů. Navrhl jej v roce 1953 J. Backus. FORTRAN doznal během svého více než padesátiletého vývoje velkou řadu změn. Poslední revize standardu jazyka byla provedena v roce 1995 (FORTRAN 95).

LISP První návrh jazyka LISP (z List Processing) vznikl jen o něco později než návrh jazyka FORTRAN. Autorem návrhu byl J. McCarthy [MC60]. LISP byl původně navržen také jako jazyk pro vědecké výpočty, ale na rozdíl od FORTRANu, který byl numericky založený a umožňoval pracovat pouze s čísly, byl LISP již od počátku orientován na práci se symbolickými daty. Na rozdíl od FORTRANu byl LISP vyvíjen v široké komunitě autorů a nebyla snaha jej standardizovat. V současnosti se pod pojmem LISP rozumí celá rodina jazyků, jejichž dva nejvýznamnější zástupci jsou jazyky Common LISP a Scheme.

Otázkou je, zdali můžeme nějak srozumitelně podchytit, co mají vyšší programovací jazyky společného (zatím jsme se vyjadřovali v dost obecných pojmech jako byla například „abstrakce" a „přenositelnost"). Odpověď na tuto otázku je v podstatě záporná, obecných rysů je velmi málo, protože vyšších programovacích jazyků je velké množství. Každý vyšší programovací jazyk by ale měl dát programátorovi k dispozici:

(i) *primitivní výrazy* (například čísla, symboly, . . .),

(ii) *prostředky pro kombinaci* primitivních výrazů do složitějších,

(iii) *prostředky pro abstrakci* (možnost pojmenování složených výrazů a možnost s nimi dále manipulovat).

Každý vyšší programovací jazyk má však svou vlastní technickou realizaci předchozích bodů. To co je v jednom jazyku primitivní výraz již v druhém jazyku být primitivní výraz nemusí a tak podobně. Programovací jazyky se jeden od druhého liší stylem zapisování jednotlivých výrazů i jejich významem, jak ukážeme v další sekci.

Stejně tak jako v případě assemblerů, programy napsané ve vyšších programovacích jazycích jsou psány jako textové soubory a pro vygenerování výpočetního procesu na základě daného programu je potřeba provést buďto jejich *překlad* nebo *interpretaci*:

Interpretace. Interpret programovacího jazyka je program, který čte výrazy programu a postupně je přímo vykonává. Interprety tedy (obvykle) z daného programu neprodukují kód v jazyku stroje.

Překlad přes nižší programovací jazyk. Překladač (*kompilátor*; anglicky *compiler*) *programovacího jazyka* je program, který načte celý program a poté provede jeho překlad do některého nižšího programovacího jazyka, typicky do *assembleru* nebo do nějakého *bajtkódu.* V tomto případě rozlišujeme tři situace:

(a) překlad byl proveden do *kódu stroje,* pak překlad končí, protože jeho výsledkem je již program zpracovatelný procesorem, který může být přímo spuštěn;

(b) překlad byl proveden do *assembleru*, pak je potřeba ještě dodatečný překlad do kódu stroje, který obvykle provádí rovněž překladač;

(c) překlad byl proveden do *bajtkódu*, pak překlad končí. Bajtkód ale není přímo zpracovatelný procesorem, musí tedy být ještě *interpretován* nebo *přeložen* (méně časté) dalším programem.

Z konstrukčního hlediska jsou překladače výrazně složitější než interprety, v praxi jsou však častěji používány, protože vykonávání překládaných programů je výrazně rychlejší než jejich interpretace. S neustálým nárůstem výkonu procesorů je však do budoucna možný obrat ve vývoji. Nevýhodou překladačů je, že s výjimkou překladačů do bajtkódu, generují programy přímo pro konkrétní procesory, takže překladače je o něco složitější „přenést" mezi dvěma počítačovými platformami. U překladu do bajtkódu tento problém odpadá, ale za cenu, že výsledný bajtkód je nakonec interpretován (pomalé).

Překlad přes vyšší programovací jazyk. Situace je analogická jako při překladu do nižšího jazyka, pouze teď překladače překládají programy do některého cílového vyššího programovacího jazyka. Typickou volbou bývá překlad do některého z cílových jazyků, jejichž překladače jsou široce rozšířené. Takovým jazykem je například jazyk C. Pro použití jazyka C jako cílového jazyka překladu také hovoří jeho jednoduchost, jasná specifikace a přenositelnost. Překladače tohoto typu (překladače generující programy ve vyšším programovacím jazyku) jsou z hlediska složitosti své konstrukce srovnatelné s interprety. Oproti interpretaci je výhoda ve větší rychlosti vykonávání výsledného kódu. Nevýhodou je, že je potřeba mít k dispozici ještě překladač cílového jazyka, kterým je z programu v cílovém jazyku nakonec sestaven výsledný kód stroje.

Z pohledu předchozího rozdělení je dobré zdůraznit, že překlad ani interpretace se pevně neváže k programovacímu jazyku. Pro jeden programovací jazyk mohou existovat jak překladače, tak interprety. Rozdíly mezi interprety a překladači se někdy smývají. Některé přeložené kódy v sobě mají interpretační prvky. Naopak, některé interprety programovacích jazyků dokážou přeložit některé části programů s vykonávat je jako kód stroje, přísně vzato jde tedy o hybridní metodu vytváření výpočetního procesu. Tato dodatečná kompilace během interpretace, která se provádí především kvůli efektivitě výsledného výpočetního procesu, se označuje anglickým termínem *just in time compilation*. My se během dalšího výkladu budeme zabývat většinou interpretací, protože je z technického pohledu jednodušší.

V následujícím textu a v navazujících dílech cvičebnice se budeme zabývat pouze vyššími programovacími jazyky. Nižší programovací jazyky budou rozebrány v rámci kurzu *operačních systémů.* Na závěr úvodní sekce podotkněme, že přesun od nižších k vyšším programovacím jazykům nastal velmi brzy po zkonstruování programovatelných počítačů, protože jejich programátoři si dobře uvědomovali, jak bylo programování v nižších jazycích komplikované. Například již Konrad Zuse pro svůj počítač Z3 navrhl vyšší programovací jazyk zvaný Plankalkül [Gi97, Ro00, Zu43, Zu72]. Ačkoliv k jazyku ve své době neexistoval překladač, je nyní Plankalkül považován za *nejstarší vyšší programovací jazyk* (navržený pro fyzicky existující počítač). Konstrukce prvních překladačů si na počátku éry vyšších programovacích jazyků vyžádala obrovské úsilí. Z dnešního pohledu je konstrukce překladačů rutinní záležitost, kterou by měl (alespoň rámcově) znát každý informatik.

Vyšší programovací jazyky s sebou přinesly nové (a původně netušené) možnosti. Brzy se zjistilo, že části některých programů jsou napsány natolik obecně, že je lze používat v jiných programech. To vedlo velmi brzy k vytváření *knihoven* – speciálních částí programů, které obsahovaly předprogramované funkce, které mohly být programátory ihned používány. Další milník padl ve chvíli, kdy se podařilo použít vyšší programovací jazyky pro naprogramování operačních systémů (jazyk C a operační systém UNIX), protože ještě dlouho po nástupu vyšších programovacích jazyků se věřilo, že nejsou pro psaní operačních systémů (a základního softwarového vybavení počítačů) vhodné.

1.2 Syntax a sémantika programů

Při úvahách o programech zapsaných v programovacích jazycích musíme vždy důsledně odlišovat dva úhly pohledu na tyto programy. Prvním je jejich *syntaxe* čili způsob nebo tvar, ve kterém se programy zapisují. Druhým úhlem pohledu je *význam programů*, neboli jejich *sémantika*. Sémantiku někdy nazýváme *interpretace*, ačkoliv my tento pojem nebudeme používat, protože koliduje s pojmem *interpretace* představeným v sekci 1.1, jehož význam je „vykonávání programu“ (což je význam posunutý trochu někam jinam). Syntaxe a sémantika jsou dvě různé složky, které v žádném případě nelze zaměňovat nebo ztotožňovat. Důvod je v celku zřejmý, pouhý tvar, ve kterém zapisujeme programy nic neříká o jejich významu, jak blíže ukážeme v následujících příkladech. Důrazné odlišování syntaxe a sémantiky není jen věcí programovacích jazyků, ale je běžně používáno v *logice, teoretické informatice* i v aplikacích, například při konstrukci *analyzátorů textu, elektronických slovníků* nebo *překladačů*.

Syntax programovacího jazyka je souborem přesně daných pravidel definujících jak zapisujeme programy v daném programovacím jazyku. Syntax většiny programovacích jazyků je popsána v jejich standardech pomocí takzvaných *formálních gramatik*, případně jejich vhodných rozšíření, například *Backus-Naurovy formy*. Obecný popis problematiky je obsahem kurzu *formálních jazyků a automatů*, viz také [Ho01, Ko97].

V sekci 1.1 jsme uvedli, že každý vyšší programovací jazyk obsahuje primitivní výrazy a prostředky pro jejich kombinaci. Syntaxe programovacího jazyka z tohoto pohledu popisuje, jak vypadají primitivní výrazy a vymezuje jakým způsobem lze z jednodušších výrazů konstruovat složitější.

Příklad 1.2. Uvažujme smyšlený programovací jazyk, ve kterém můžeme, kromě jiného, zapisovat datum. Můžeme provést úmluvu, že povolená syntaxe data bude ve tvaru

$$DD/DD/DDDD,$$

kde každé D představuje cifru z množiny $\{0, \ldots, 9\}$. Například tedy výrazy $26/03/1979$ a $17/11/1989$ odpovídají tomuto tvaru. Stejně tak ale třeba i výrazy $00/00/0000$, $01/99/6666$, $54/13/1002$ a podobně, odpovídají smluvenému tvaru. V tuto chvíli bychom se neměli nechat splést tím, že poslední tři výrazy nereprezentují žádné „skutečné datum“; doposud jsme se pouze bavili o syntaxi data (jeho zápisu), nikoliv jeho sémantice (jeho významu). Na tomto příkladu můžeme taky dobře vidět, že pouhý „smluvený tvar“, ve kterém datum zapisujeme nic neříká o jeho konkrétní hodnotě. Vezmeme-li si výraz $03/11/2006$, pak by bylo dost dobře možné přisoudit mu dva zcela různé významy:

(a) 3. listopadu 2006 (první údaj v $DD/DD/DDDD$ značí číslo dne),

(b) 11. března 2006 (první údaj v $DD/DD/DDDD$ značí číslo měsíce).

Tímto příkladem jsme chtěli demonstrovat, že tvar (částí) programů nelze směšovat s jeho významem, ani z něj jeho význam nelze nijak „přímočaře určit“.

Poučení, které plyne z předchozího příkladu, je v podstatě následující: každý programovací jazyk musí mít popsánu svojí syntax a sémantiku. Pokud se v daném jazyku budeme snažit psát programy, měli bychom se vždy se syntaxí a sémantikou daného jazyka seznámit. Předejdeme tak nebezpečným chybám, které by v programu mohly vzniknout díky tomu, že jsme předpokládali, že „něco se *píše/chová* jinak, než jak jsme mysleli (než na co jsme byli zvyklí u jiného jazyka)“. V následujícím příkladu ukážeme, jakých rozdílných významů nabývá tentýž výraz z pohledu sémantiky různých programovacích jazyků.

Příklad 1.3. Uvažujme výraz ve tvaru

$$x = y + 3.$$

Tento výraz lze chápat jako část programů zapsaných v různých programovacích jazycích. V následujícím přehledu uvedeme několik odlišných významů tohoto výrazu tak, jak jej definují vybrané programovací jazyky.

(a) V jazycích jakými jsou C, C++ a další by měl výraz „$x = y + 3$“ význam *příkazu přiřazení.* Význam výrazu bychom mohli konkrétně vyjádřit takto: „do paměťového místa označeného identifikátorem x je zapsána hodnota vzniklá zvětšením hodnoty uložené v paměťovém místě označeném identifikátorem y o tři.“ Symboly x a y tedy hrají úlohy *označení paměťových míst* a výše uvedený výraz je příkaz přiřazení hodnoty vypočtené pomocí obsahu jednoho paměťového místa do jiného paměťového místa.

(b) V jazycích jako jsou Algol 60 a Pascal je symbol „=“ vyhrazen pro *porovnávání číselných hodnot.* V tomto případě má „$x = y + 3$“ význam „porovnej, zdali je hodnota uložená v x rovna hodnotě uložené v y zvětšené o tři.“ Oproti předchozímu případu je tedy výsledkem zpracování výrazu „$x = y + 3$“ pravdivostní hodnota *pravda* (a to v případě, že rovnost skutečně platí) nebo *nepravda* (rovnost neplatí). K žádnému přiřazení hodnot (mezi paměťovými místy) nedochází.

(c) V jazyku METAPOST, což je speciální jazyk vyvinutý pro kreslení vektorových schémat (takzvaných pérovek), má výraz „$x = y + 3$“ význam *příkazu pro řešení lineární rovnice* nebo soustavy (dříve uvedených) lineárních rovnic. Výraz „$x = y + 3$“ uvedený v programu má význam „pokus se řešit rovnici $x = y + 3$“. Pokud by například hodnota x byla známá, pak by po uvedení tohoto výrazu došlo k výpočtu hodnoty y, což by bylo $x - 3$. To je významný rozdíl oproti příkazu přiřazení, viz bod (a), protože v tomto případě došlo k dopočtení hodnoty y a hodnota x (levá strana rovnosti) se nezměnila. Pokud by byly známé obě hodnoty x i y, pak by došlo ke kontrole konzistence rovnice (pokud by neplatila rovnost, výpočet by končil chybou: „neřešitelná rovnice nebo soustava rovnic“). Pokud by ani jedna z proměnných x a y nebyla známá, pak by byla rovnice pouze „zapamatována“ a byla by použita při pokusu o nalezení řešení při výskytu další rovnice.

(d) V logických programovacích jazycích, jakým je například jazyk PROLOG, by měl výraz „$x = y + 3$“ význam *deklarace,* že hodnota navázaná na proměnnou x musí být ve stejném tvaru jako $y + 3$. Taková podmínka je splněna například v případě, kdy na x bude navázána hodnota $f(z) + 3$ (chápaná jako symbolický výraz, v němž f je pouze jméno a neoznačuje žádnou konkrétní funkci) a na y by byla navázána hodnota $f(z)$, protože pak by po dosazení $f(z) + 3$ za x do „$x = y + 3$“ a po následném dosazení $f(z)$ za y do „$f(z) + 3 = y + 3$“ přešel původní výraz „$x = y + 3$“ do tvaru „$f(z) + 3 = f(z) + 3$“. Uvědomme si, že deklarace stejného tvaru dvou výrazů v tomto smyslu je něco jiného než řešení lineární rovnice z bodu (c). V jazyku PROLOG musí být přísně vzato proměnné x a y značeny velkými písmeny, ale když si odmyslíme tento detail, tak máme dohromady čtyři různé interpretace téhož výrazu.

V předchozím příkladu jsme viděli různé interpretace stejného výrazu. Platí samozřejmě i opačná situace a to ta, že konstrukce s jedním významem se v různých programovacích jazycích zapisuje různě. Tento jev je dobře pozorovatelný například na aritmetických výrazech, což ukazuje následující příklad.

Příklad 1.4. Uvažujme aritmetický výraz $2 \cdot (3 + 5)$ čili „vynásob dvěma součet tří a pěti“. V každém rozumném programovacím jazyku je výsledkem zpracování takového výrazu hodnota 16. Jazyky se ale leckdy liší v samotném zápisu výrazu $2 \cdot (3 + 5)$. Jedním ze zdrojů rozdílů je *pozice symbolů označujících operace* „$\cdot$“ a „$+$“ vůči *operandům* čili podvýrazům, se kterými operace provádíme. Následující přehled obsahuje čtyři typické zápisy výrazu $2 \cdot (3 + 5)$ v programovacích jazycích.

`2 * (3 + 5)` Tento tvar je v souladu s běžným psaním výrazů, symboly pro operace stojí mezi operandy se kterými operace chceme provádět. Proto tomuto zápisu říkáme *infixová notace.* Při psaní výrazů v infixové notaci hrají důležitou roli *priority operací* a jejich *asociativita.* Drtivá většina jazyků, například jazyky C a C++, umožňují psát pouze `2 * 3 + 4` místo `(2 * 3) + 4`, protože symbol „`*`“ označující násobení má vyšší prioritu než symbol „`+`“ označující sčítání. Stejně tak lze psát `2 + 3 + 4` místo `2 + (3 + 4)`, což je zaručeno tím, že operace „`+`“ je asociativní. Infixový zápis s sebou z hlediska konstrukce interpretů a překladačů programovacích jazyků přináší komplikace, protože u výrazů je potřeba správně rozpoznávat prioritu a asociativitu operací. Další technickou komplikací je použití operací, které mají víc jak dva argumenty, které již nelze psát „čistě infixově“, protože „jeden symbol“ nelze napsat „mezi tři operandy“. Například jazyky C, C++, PERL a další mají *ternární operátor*, který se zapisuje ve tvaru „$\langle operand_1 \rangle \,?\, \langle operand_2 \rangle : \langle operand_3 \rangle$“, tedy pomocí dvou různých symbolů vložených mezi operandy.

`(* 2 (+ 3 5))` Při použití této notace, takzvané *prefixové notace,* píšeme operaci před všechny operandy, samotné operandy jsou od operace a mezi sebou odděleny mezerami, nebo obecněji „bílými znaky“ (anglicky *whitespace*). Za bílé znaky se v informatickém žargonu považují libovolné sekvence mezer, tabelátorů a nových řádků (případně jiných speciálních znaků nemajících čitelnou reprezentaci). Každý výraz v prefixové notaci skládající se z operace a operandů je navíc celý uzavřen do závorek. Prefixová notace má proti infixové notaci jednu nevýhodu – a tou je počáteční nezvyk programátorů. Na druhou stranu má prefixová notace velké množství výhod, které oceníme postupně v dalších lekcích. Nyní podotkněme, že notace je velmi jednoduchá a nemá v ní například smysl řešit prioritu operací. Jednoduše lze také psát operace s libovolným počtem operandů, například výraz `(+ 2 3 5)` je „součet tří operandů“, `(+ (* 2 3) 5)` je „součet dvou operandů“, dále `(op (+ 1 2) 3 4 5)` je „operace *op* provedená se čtyřmi operandy `(+ 1 2)`, `3`, `4` a `5`“ a podobně.

`(2 (3 5 +) *)` Tato notace se nazývá *postfixová* a je analogická notaci prefixové s tím rozdílem, že operace se píše až za všechny operandy. Dále platí pravidlo o oddělování operandů od operace a mezi sebou a pravidlo o uzávorkování vnějších výrazů jako u prefixové notace. Tato notace je ze všech notací nejméně používaná.

`2 3 5 + *` Poslední uvedená notace se vyznačuje tím, že operace se píší opět za operandy, ale každá operace má pevný počet přípustných operandů, tím pádem je možné bez újmy vynechat ve výrazu všechny závorky. Notace se nazývá *reverzní bezzávorková notace,* nebo též *polská bezzávorková notace,* protože ji proslavil polský logik Jan Lukasiewicz. Výrazy v tomto tvaru lze jednoduše načítat a vyhodnocovat. I přes svou zdánlivou těžkou čitelnost je bezzávorková notace poměrně oblíbená a je na ní založeno několik programovacích jazyků, například jazyk PostScript – specializovaný grafický jazyk, který umí interpretovat většina (lepších) tiskáren a jiných výstupních zařízení počítačů.

Při vytváření programů musí mít každý programátor neustále na paměti syntax a sémantiku jazyka. Syntax proto, aby věděl, jak má zapsat danou číst kódu. Sémantiku proto, aby vždy znal přesný význam toho, co píše a aby uměl rozhodnout, zdali má právě vytvářený program skutečně *zamýšlený význam.*

I přes veškerou snahu se nelze vyhnout vzniku chyb. Chyby při vytváření programů lze obecně rozdělit na chyby syntaktické a sémantické. *Syntaktická chyba* je chybou v zápisu programu. Kód, který není z pohledu daného jazyka syntakticky správný, nelze v podstatě chápat ani jako program v daném jazyku. Syntaktické chyby lze odhalit poměrně snadno. Překladače programovacích jazyků odhalí syntaktické chyby již během překladu. Interprety programovacích jazyků odhalí syntaktické chyby až v momentě, kdy se interpret (neúspěšně) pokusí načíst chybný vstupní výraz. V případě překladačů i interpretů tedy k odhalení syntaktické chyby dochází ještě před započetím vykonávání programu (který v případě syntaktické chyby de facto programem není).

Příklad 1.5. Kdybychom se vrátili k příkladu 1.2, pak výraz $12/3/2003$ je syntakticky chybným protože prostřední složka „3“ má jen jednu cifru. Stejně tak by byly syntakticky chybné výrazy $24/k6/1234$, $12/06/20/03$ a $12–04–2003$. Zdůvodněte proč.

Druhou kategorií chyb jsou *sémantické chyby.* Jak již název napovídá, jedná se o chyby ve významu programu. Mezi typické sémantické chyby patří například provádění operací nad daty neslučitelných typů, například sčítání čísel a řetězců znaků (neformálně řečeno, jde o míchání jablek a hrušek). Sémantické chyby lze odhalit během překladu či interpretace pouze v omezené míře. Například překladače programovacího jazyka C jsou během překladu schopny zjistit všechny konflikty v typech dat se kterými program pracujeme. Technicky ale již třeba není možné při překladu vyloučit chyby způsobené „dělením nulou,“ protože hodnota dělitele může být proměnlivá a může během činnosti programu nabývat různých hodnot.

Překladače některých jazyků, natož jejich interprety, neumějí rozpoznat konflikty v typech předem (to jest, třeba během překladu nebo před započetím interpretace). Tento jev bychom však neměli chápat

jako „chybu“ nebo „nedokonalost“ programovacího jazyka, ale jako jeho *rys* – v některých programovacích jazycích není dopředná analýza typů z principu možná a programátor musí vytvářet program s ohledem na to, že za něj nikdo „typovou kontrolu neprovádí “.

Chyba projevující se až za běhu programu se anglicky označují jako *run-time error*. Asi není třeba příliš zdůrazňovat, že se jedná o vůbec nejnebezpečnější typ chyb, protože chyby způsobené až za běhu programu zpravidla:

(a) mohou být delší dobu *latentní,* tedy programátor si jich nemusí vůbec všimnout,

(b) obvykle způsobí havárii či nekorektní činnost programu,

(c) jejich odstraňování je náročné z hlediska lidských i finančních zdrojů.

Chyby, které se někdy dlouho neprojeví, viz bod (a), jsou často způsobeny podceněním triviálních případů. To například znamená, že při načítání dat ze souboru programátor zapomene ošetřit případ, kdy soubor na disku sice existuje ale je prázdný. Programátor může například ihned po otevření souboru načítat data aniž by zkontroloval, zdali tam nějaká jsou, což může posléze vést k chybě. Jako odstrašující příklad k bodům (b) a (c) můžeme uvést příklad vesmírné sondy Mars Climate Orbiter (MCO), která v září 1999 shořela v atmosféře Marsu vlivem chyby způsobené neprovedením přepočtu vzdálenosti v kilometrech na míle (škoda vzniklá touto „chybou za běhu výpočtu“ byla cca 125 milionů USD).

1.3 Přehled paradigmat programování

Vyšší programovací jazyky lze rozdělit do kategorií podle toho, jakým styl (paradigma) při programování v daném jazyku převládá. Znalost programovacího stylu je důležitá z pohledu vytváření kvalitních programů, protože pouze na základě dobré *vnitřní kvality programu* (to jest toho, jak je program napsán) lze dosáhnout požadované *vnější kvality programu* (to jest toho, jak se program chová z uživatelského pohledu). Za základní dvě kvality programů jsou někdy považovány:

- *korektnost* (program reaguje správně na správné vstupy),
- *robustnost* (program se umí vyrovnat i se špatnými vstupy).

Robustnost je tedy silnější vlastnost a zaručuje, že daný program nehavaruje v případě, kdy dostane na svůj vstup špatná data (třeba data, která nejsou v požadovaném formátu), ale umí danou situaci ošetřit (třeba požádáním o zadání nového vstupu, nebo automatickou opravou špatných dat).

Upozorněme nyní, jak je potřebovat chápat programovací paradigmata ve vztahu k programovacím jazykům. Paradigma je de facto styl, kterým lze programovat v jakémkoliv jazyku, který jej podporuje (umožňuje). Některé programovací jazyky jsou „šité na míru“ jednomu paradigmatu a nepředpokládá se, že by v nich programátoři chtěli pracovat jiným stylem (i když i to je možné). Vezmeme-li si jako příklad *procedurální paradigma* (viz dále), pak například jazyk PASCAL byl vytvořen s ohledem na to, že se v něm bude programovat procedurálním stylem. Proto taky někdy říkáme, že PASCAL je *procedurální jazyk.* Tuto terminologii budeme používat i nadále.

Základních paradigmat programování je několik. Nejprve si uvedeme skupinku tří paradigmat, ke kterým existují teoreticky dobře propracované formální modely.

procedurální Procedurální paradigma je jedním ze dvou nejstarších paradigmat programování, někdy též bývá nazýváno paradigmatem *imperativním.* Programy napsané v procedurálním stylu jsou tvořeny *sekvencemi příkazů,* které jsou postupně vykonávány. Klíčovou roli při programování v procedurálním stylu hraje *příkaz přiřazení* (zápis hodnoty na dané místo v paměti) a související *vedlejší efekt* (některým typem přiřazení může být modifikována datová struktura a podobně). Sekvenční styl vykonávání příkazů lze částečně ovlivnit pomocí *operací skoků* a *podmíněných skoků.* Formálním modelem procedurálních programovacích jazyků je *von Neumannův RAM stroj,* viz [vN46]. Z hlediska stylu programování ještě procedurální paradigma můžeme jemněji rozdělit na:

naivní Naivní paradigma bývá někdy chápané jako „samostatné paradigma“, častěji se mezi paradigmata programování ani neuvádí. Naivní procedurální jazyky se vyznačují jakousi všudypřítomnou chaotičností, mají obvykle nesystematicky navrženou syntaxi a sémantiku, v některých rysech jsou podobné nestrukturovaným procedurálním jazykům, viz dále. Mezi typické zástupce patří programovací jazyky rodiny BASIC (první jazyk BASIC navrhli J. Kemeny a T. Kurtz v roce 1968 za účelem zpřístupnit počítače a programování mimo komunitu počítačových vědců).

nestrukturované Nestrukturované procedurální paradigma je velmi blízké assemblerům, programy jsou skutečně lineární sekvence příkazů a skoky jsou v programech realizovány příkazem typu „GO TO“, to jest „jdi na řádek.“ V raných programovacích jazycích držících se tohoto stylu byly navíc všechny řádky programu číslované a skoky šlo realizovat pouze na základě uvedení konkrétního čísla řádku, což s sebou přinášelo velké komplikace. Později se objevily příkazy skoku využívající návěstí. Typickými zástupci v této kategorii byly rané verze jazyků FORTRAN a COBOL.

strukturované Ve svém článku [Di68] upozornil E. Dijkstra na nebezpečí spjaté s používáním příkazu „GO TO“. Poukázal zejména na to, že při používání příkazu „GO TO“ je prakticky nemožné ladit program, protože struktura programu nedává příliš informací o jeho vykonávání (v nějakém bodě výpočtu). Strukturované procedurální paradigma nahrazuje příkazy skoku pomocí podmíněných cyklů typu „opakuj $\sim$ dokud platí podmínka“ a jiných strukturovaných konstrukcí, které se do sebe *vnořují.* Typickými zástupci programovacích jazyků z této rodiny jsou jazyky C (D. Ritchie, B. Kernighan, 1978), PASCAL (N. Wirth, 1970), Modula-2 (N. Wirth, 1978), a ADA (J. Ichbiah a kolektiv, 1977–1983).

funkcionální Funkcionální paradigma je zhruba stejně staré jako paradigma procedurální. Programy ve funkcionálních jazycích sestávají ze symbolických výrazů. Výpočet je potom tvořen *postupnou aplikací funkcí,* kdy funkce jsou aplikovány s hodnotami, které vznikly v předchozích krocích aplikací, výsledné hodnoty aplikace jsou použity při aplikaci dalších funkcí a tak dále. Při čistém funkcionálním programování nedochází k vedlejším efektům, příkaz přiřazení se nepoužívá. Díky absenci vedlejších efektů je možné chápat programy ve funkcionálních jazycích jako *ekvacionální teorie* [Wec92]. Rovněž se nepoužívají příkazy skoku ani strukturované cykly. Cykly se nahrazují *rekurzivními funkcemi* a *funkcemi vyšších řádů.* V některých funkcionálních jazycích také hraje důležitou roli *líné vyhodnocování* (jazyk Haskell, 1987) a *makra* (zejména dialekty LISPu). Nejznámější teoretický model funkcionálního programování je λ-kalkul (A. Church, 1934, viz [Ch36, Ch41]), i když existují i modely jiné, například modely založené na přepisovacích systémech a uspořádaných algebrách (přehled lze nalézt ve [Wec92]). Typickými zástupci funkcionálních programovacích jazyků jsou Common LISP, Scheme, ML a Anfix (dříve Aleph). Mezi čistě funkcionální jazyky patří Miranda, Haskell, Clean a další.

logické Teoretické základy logického programování byly položeny v článcích [Ro63, Ro65] J. Robinsona popisujících *rezoluční metodu.* Později byla teorie logického programování dále rozpracována a zkonstruovány byly první překladače logických jazyků, viz také [Ll87, NS97, NM00, SS86]. Pro logické paradigma je charakteristický jeho *deklarativní charakter*, při psaní programů programátoři spíš popisují problém, než aby vytvářeli předpis, jak problém řešit (to samozřejmě platí jen do omezené míry). Programy v logických programovacích jazycích lze chápat jako *logické teorie* a výpočet lze chápat jako *dedukci důsledků,* které z této teorie plynou. Formálním modelem logického programování jsou speciální inferenční mechanismy, jejichž studium je předmětem *matematické logiky.* Programy se skládají ze speciálních formulí – tak zvaných *klauzulí.* Výpočet se v logických jazycích zahajuje zadáním cíle, což je existenční dotaz, pro který je během výpočtu nalezena odpověď, nebo nikoliv. Typickým představitelem logického programovacího jazyka je PROLOG (A. Colmerauer, R. Kowalski, 1972). Logické programovací jazyky rovněž nalezly uplatnění v deduktivních databázích, například jazyk DATALOG (H. Gallaire, J. Minker, 1978).

Další dvě významná paradigmata jsou:

objektově orientované Objektové paradigma je založené na myšlence vzájemné *interakce objektů*. Objekty jsou nějaké entity, které mají svůj vnitřní stav, který se během provádění výpočetního procesu může měnit. Objekty spolu komunikují pomocí *zasílání správ.* Objektově orientované paradigma není „jen jedno“, ale lze jej rozdělit na řadu podtypů. Ze všech programovacích paradigmat má objektová orientace

nejvíc modifikací (které jdou leckdy myšlenkově proti sobě). Například při vývoji některých jazyků byla snaha odstranit veškeré rysy připomínající procedurální paradigma: cykly byly nahrazeny *iterátory*, podmíněné příkazy byly nahrazeny *výjimkami*, a podobně. Jiné jazyky se této striktní představy nedrží. Prvním programovacím jazykem s objektovými rysy byla Simula (O. J. Dahl a K. Nygaard, 1967). Mezi čistě objektové jazyky řadíme například Smalltalk (A. Kay a kol., 1971) a Eiffel (B. Meyer, 1985). Dalšími zástupci jazyků s objektovými rysy jsou C++, Java, Ruby, Sather, Python, Delphi a C♯.

paralelní Paralelní paradigma je styl programování, ve kterém je kladen důraz na *souběžné* (*paralelní*) *vykonávání* programu. Paralelní paradigma má opět mnoho podtříd a nejde o něm říct příliš mnoho obecně. Jednou z hlavních myšlenek je, že výpočetní proces se skládá z několika souběžně pracujících částí. Programy generující takové výpočetní procesy musí mít k dispozici prostředky pro jejich vzájemnou synchronizaci, sdílení dat, a podobně. Mezi zástupce čistých paralelních jazyků patří například MPD, Occam a SR. Paralelní prvky se jinak vyskytují ve všech moderních programovacích jazycích nebo jsou k dispozici formou dodatečných služeb (poskytovaných operačními systémy).

Předchozí dvě paradigmata jsou významná z toho pohledu, že velká většina soudobých programovacích jazyků v sobě zahrnuje jak objektové tak paralelní rysy. Zajímavé ale je, že ani objektové ani paralelní paradigma za sebou nemají žádné výrazné formální modely, jako je tomu v případě předchozích tří paradigmat (formálních modelů existuje sice několik, jsou ale vzájemně těžko slučitelné a pokrývají jen část problematiky, navíc obecné povědomí o techto modelech je malé). Další pozoruhodná věc je, že čistě objektové ani paralelní jazyky se (v praxi) de facto vůbec nepoužívají.

Obecně lze říct, že žádné paradigma se prakticky nepoužívá ve své čisté podobě. V dnešní době je typické používat programovací jazyky, které umožňují programátorům programovat více styly. Například jazyky jako jsou C++, Java, C♯, Delphi a jiné jsou procedurálně-objektové, jazyky Anfix a Ocaml jsou funkcionálně-objektové a podobně. Někdy proto hovoříme o *multiparadigmatech.* Čím více stylů programátor během svého života vyzkouší, tím větší bude mít potenciál při vytváření programů.

Naším cílem bude postupně projít všemi paradigmaty a ukázat si jejich hlavní rysy. Máme v zásadě několik možností, jak to udělat. Mohli bychom si pro každé zásadní paradigma zvolit jeden (typický) jazyk, popsat jeho syntax a sémantiku a na tomto jazyku si programovací paradigma představit. Z praktických důvodů se při našem výkladu vydáme jinou cestou. Zvolíme si na začátku jeden multiparadigmový jazyk s vysokým modelovacím potenciálem a na tomto jazyku budeme postupně demonstrovat všechny hlavní styly programování.

Pro naše účely si zvolíme jazyk Scheme. Scheme je dialekt jazyka LISP, který byl částečně inspirován jazykem ALGOL 60. Jazyk Scheme navrhli v 70. letech minulého století Guy L. Steele a Gerald Jay Sussman původně jako experimentální jazyk pro ověření některých vlastností teorie aktorů. Scheme byl inspirovaný LISPem, ale byl na rozdíl od něj výrazně jednodušší. I tak se ukázalo, že v jazyku se dají velmi rychle vytvářet prototypy programů – jazyk se tedy výborně hodil pro experimentální programování. Jazyk byl poprvé úplně popsán v revidovaném reportu [SS78] v roce 1978. Od té doby vznikají další revize tohoto revidovaného reportu, poslední report je „šestý revidovaný report o jazyku Scheme“. Pro revidované reporty se vžilo značení R^{n}RS, kde „n“ je číslo revize. Současná verze se tedy označuje jako R^{6}RS, viz [R6RS]. Další návrhy na úpravy a vylepšení jazyka Scheme se shromažďují prostřednictvím tak zvaných SRFI (zkratka pro *Scheme Requests for Implementation*), které lze nalézt na `http://srfi.schemers.org/`. Revidovaným reportům předcházela série článků, které vešly do historie pod názvem „lambda papers“, dva nejvýznamnější z nich jsou [St76, SS76].

Při našem exkurzu paradigmaty programování však nebudeme používat jazyk Scheme podle jeho současného de facto standardu R^{6}RS, ale budeme uvažovat jeho vlastní *zjednodušenou variantu.* To nám umožní plně se soustředit na vlastní problematiku jazyka a jeho interpretace. Jednodušší verze nám navíc umožní zabývat se problémy, kterými bychom se v plnohodnotném interpretu zabývat nemohli (nebo by to bylo obtížné). Během našeho zkoumání se na jazyk Scheme budeme dívat z *dvou různých úhlů pohledu*:

- *pohled programátora v jazyku Scheme* – z tohoto úhlu pohledu si budeme postupně představovat jazyk Scheme a budeme si ukazovat typické ukázky programů napsaných v jazyku Scheme; budeme přitom postupovat od jednoduchého ke složitějšímu;

- *pohled programátora interpretu jazyka Scheme* – z tohoto úhlu pohledu budeme vysvětlovat, jak funguje interpret jazyka Scheme. Tento úhel pohledu bývá v literatuře často opomíjen, je však důležitý i z předchozího pohledu. Detailním popisem mechanismu, kterým z programu vzniká výpočetní proces, získá programátor úplný přehled o tom, jak jsou programy vykonávány. Tyto znalosti lze pak využít v mnoha oblastech: při ladění programů, při programování vlastních vyhodnocovacích procesů (nejen v jazyku Scheme) a podobně. Tím, že budeme jazyk představovat od nejjednodušších částí, bude popis jeho interpretace stravitelný i pro čtenáře, kteří nikdy zkušenost s programováním neměli.

V temto textu budeme studovat *funkcionální paradigma.* To je zároveň primární paradigma Jazyka Scheme. Po dvanácti lekcích budeme schopni (mimo jiné) naprogramovat vlastní interpret ryze funkcionální podmnožiny jazyka Scheme.

1.4 Symbolické výrazy a syntaxe jazyka Scheme

V této části lekce popíšeme syntaxi jazyka Scheme, to jest způsob, jak zapisujeme programy ve Scheme. Významem (interpretací) programů se budeme zabývat v další části lekce. Jazyk Scheme, stejně tak jako ostatní dialekty LISPu, má velmi jednoduchou syntaxi, zato má velmi bohatou sémantiku. To s sebou přináší několik příjemných efektů:

(a) zápis programů je snadno pochopitelný,

(b) při programování se dopouštíme jen minimálního množství syntaktických chyb,

(c) mechanická kontrola správného zápisu programu je přímočará.

Body (a) a (b) mají praktické dopady při tom, když v jazyku Scheme programujeme. Bod (c) je neméně podstatný, je však zajímavý spíš z pohledu konstrukce překladače/interpretu jazyka Scheme.

Programy v jazyku Scheme se skládají ze *symbolických výrazů.* Nejprve popíšeme jak tyto výrazy mohou vypadat a pak pomocí nich zavedeme pojem program. Neformálně řečeno, za symbolické výrazy budeme považovat čísla a symboly, což jsou základní symbolické výrazy a dále seznamy, což jsou složené symbolické výrazy. Přesná definice následuje:

Definice 1.6 (symbolický výraz)**.**

(i) Každé *číslo* je symbolický výraz
(zapisujeme `12`, `-10`, `2/3`, `12.45`, `4.32e-20`, `-5i`, `2+3i`, a pod.);

(ii) každý *symbol* je symbolický výraz
(zapisujeme `sqrt`, `+`, `quotient`, `even?`, `muj-symbol`, `++?4tt`, a pod.);

(iii) jsou-li $e_1, e_2, \ldots, e_n$ symbolické výrazy (pro $n \geq 1$),
pak výraz ve tvaru `(`$e_1 \sqcup e_2 \sqcup \cdots \sqcup e_n$`)` je symbolický výraz zvaný *seznam*. ■

Poznámka 1.7. (a) Symbolickým výrazům budeme rovněž zkráceně říkat *S-výrazy* (anglicky: *symbolic expression*, případně *S-expression*). Vrátíme-li se opět k primitivním výrazům a prostředkům pro jejich kombinaci, pak čísla a symboly jsou primitivní symbolické výrazy, které lze dále kombinovat do seznamů. Pojem symbolický výraz (S-výraz) se objevuje už v prvním publikovaném návrhu jazyka LISP z roku 1960, popsaném v článku [MC60]. Ačkoliv byl původní LISP vyvinut na počítač IBM 704, popis jazyka a jeho interpretace pomocí S-výrazů umožnil odhlédnout od konkrétního hardware a později snadno implementovat další varianty LISPu na jiných počítačích.

(b) Z důvodů přehlednosti budeme symbolické výrazy zapisovat s barevným odlišením čísel (zelená), symbolů (modrá) a závorek (červená) tak, jak je to vyznačeno v definici 1.6.

(c) Za symboly budeme považovat libovolnou sekvenci znaků neobsahující závorky `(` a `)`, kterou *nelze přirozeně interpretovat jako zápis čísla.* Například tedy `-8.5` chápeme jako číslo „mínus osm celých a pět desetin“, kdežto `-8.5.`, `-8.5.6`, `8-8.5`, `1-` a podobně jsou symboly. Například `2*3` je rovněž symbol, protože se nejedná o sekvenci znaků, kterou bychom chápali jako „zápis čísla“. Pro úplný popis syntaxe

čísel a symbolů bychom v tuto chvíli měli správně udělat detailní specifikaci všech přípustných tvarů čísel. Pro účely dalšího výkladu si však vystačíme s tímto intuitivním odlišením čísel a symbolů. Zájemce o podrobnou specifikaci čísel a symbolů v R^{6}RS Scheme odkazuji na specifikaci [R6RS].

(d) V seznamu $(e_1 \sqcup e_2 \sqcup \cdots \sqcup e_n)$ uvedeném v definici 1.6 jsme označovali pomocí „$\sqcup$“ libovolnou, ale neprázdnou, sekvenci bílých znaků, které slouží jako oddělovače jednotlivých prvků seznamů. Číslo n budeme nazývat *délka seznamu* $(e_1 \sqcup e_2 \sqcup \cdots \sqcup e_n)$. Symbolické výrazy $e_1, \ldots, e_n$ jsou *prvky* seznamu $(e_1 \sqcup e_2 \sqcup \cdots \sqcup e_n)$. Například `(10ahoj20)` je seznam s jediným prvkem, jímž je symbol `10ahoj20`. Naproti tomu třeba seznam `(10 ahoj20)` má dva prvky, první je číslo `10` a druhým je symbol `ahoj20`. Dále třeba `(10 aho j 20)` má čtyři prvky, první je číslo `10`, druhý je symbol `aho`, třetí je symbol `j` a poslední je číslo `20`. Upozorněme také na fakt, že prvky seznamů mohou být opět seznamy a je to plně v souladu s definicí 1.6. Například tedy výraz `(a (1 2) b)` je tříprvkový seznam, jehož první prvek je symbol `a`, druhý prvek je dvouprvkový seznam `(1 2)` a třetí prvek je symbol `b`.

(e) Při psaní symbolických výrazů připustíme jednu výjimku pro vynechání bílých znaků mezi prvky seznamů a to kolem prvků, které jsou samy o sobě seznamy. Například v případě seznamu `(a (3 2) b)` bychom mohli psát rovněž `(a(3 2) b)`, `(a (3 2)b)` nebo `(a(3 2)b)` a pořád by se jednalo o týž seznam. Na druhou stranu, `(a(32)b)` už by stejný seznam nebyl, protože druhý prvek `(32)` je nyní jednoprvkový seznam obsahující číslo `32`.

Příklad 1.8. Následující výrazy *jsou* symbolické výrazy:

```
-2/3      3.141459    list?              (= 1 2)                 (2+ (3 + 3))
-3+2i     10.2.45     inexact->exact     (* 2 (sqrt (/ 2 3)))    ((10) (20))
-20.      1+          ahoj-svete         (1 + 2)                 (1 (ahoj (3)))
```

Podotkněme, že `-2/3` označuje racionální číslo $-\frac{2}{3}$, `-3+2i` označuje komplexní číslo $-3 + 2i$. Čísla tvaru `-20.0` můžeme psát zkráceně `-20.` a podobně.

Následující výrazy *nejsou* symbolické výrazy:

`)`	výraz začíná pravou závorkou,
`(+ 1 2))`	výraz má jednu pravou závorku navíc,
`(10 (20 ahoj)`	výrazu chybí jedna pravá závorka,

Všimněte si, že do předchozích výrazů můžeme doplnit závorky tak, aby se z nich staly symbolické výrazy a to dokonce mnoha způsoby. Například do `(10 (20 ahoj)` můžeme doplnit závorky `(10 (20) ahoj)` nebo `(10 (20 ahoj))` a tak dále.

Nyní můžeme říct, co budeme mít na mysli pod pojmem *program v jazyku Scheme.*

Definice 1.9 (program)**.** Program (v jazyku Scheme) je konečná posloupnost symbolických výrazů. ■

Programy v jazyku Scheme jsou posloupnosti symbolických výrazů. Pokud by byl v posloupnosti obsažen výraz, který by nebyl symbolickým výrazem dle definice, pak daná posloupnost není programem v jazyku Scheme. Z pohledu vzniku chyb se jedná o syntaktickou chybu – špatný zápis části programu.

V další části lekce se budeme zabývat sémantikou symbolických výrazů a s tím související sémantikou programů v jazyku Scheme. Podotkněme, že zatím jsme o významu symbolických výrazů vůbec nic neřekli, dívali jsme se na ně pouze z pohledu jejich zápisu. Na tomto místě je dobré poukázat na odlišnost programů tak, jak jsme je zavedli, od programů v R^{6}RS Scheme, viz [R6RS]. Definice R^{6}RS vymezuje syntaxi symbolických výrazů mnohem přísněji. My si ale plně vystačíme s naší definicí.

1.5 Abstraktní interpret jazyka Scheme

Nejpřesnější cestou popisu sémantiky programu v jazyku Scheme je stanovit, jak interprety jazyka Scheme, případně překladače, zpracovávají symbolické výrazy, protože programy se skládají právě ze symbolických výrazů. Jelikož se budeme během výkladu vždy zabývat především *interpretací,* k popisu sémantiky programu stačí popsat proces, jak jsou symbolické výrazy *postupně vyhodnocovány.* Pokud budeme mechanismus vyhodnocování přesně znát, pak budeme vždy schopni určit (i bez toho aniž bychom měli k dispozici konkrétní interpret jazyka Scheme), jak budou symbolické výrazy zpracovávány, co bude v důsledku daný program dělat, případně jestli při vyhodnocování programu dojde k chybě.

Z praktického hlediska není možné rozebírat zde implementaci nějakého konkrétního interpretu Scheme. Implementace i těch nejmenších interpretů by pro nás zatím byly příliš složité. Místo toho budeme uvažovat „abstraktní interpret Scheme", který přesně popíšeme, ale při popisu (zatím) odhlédneme od jeho technické realizace. Před tím, než představíme abstraktní interpret a proces vyhodnocování symbolický výrazů, si řekneme, jaká bude naše *zamýšlená interpretace* symbolických výrazů. Tuto zamýšlenou interpretaci potom zpřesníme popisem činnosti abstraktního interpretu.

Budeme-li uvažovat nejprve *čísla* a *symboly,* pak můžeme neformálně říct, že úkolem čísel v programu je označovat „svou vlastní hodnotu". Tedy například symbolický výraz `12` označuje hodnotu „číslo dvanáct", symbolický výraz `10.6` označuje hodnotu „číslo deset celých šest desetin" a tak dále. Z pohledu vyhodnocování můžeme říct, že „čísla se vyhodnocují na sebe sama", nebo přesněji *„čísla se vyhodnocují na hodnotu, kterou označují ".*

Symboly lze chápat jako „jména hodnot". Některé symboly, jako třeba `+` a `*` jsou svázány s hodnotami jimiž jsou operace (procedury) pro sčítání a násobení čísel. Symboly `pi` a `e` by mohly být svázány s přibližnými hodnotami čísel π a e (Eulerovo číslo). Jiné symboly, třeba jako `ahoj-svete`, nejsou svázány s žádnou hodnotou[1]. Z pohledu vyhodnocování říkáme, že *„symboly se vyhodnocují na svou vazbu*". Tím máme na mysli, že pokud je například symbol `*` svázán s operací (procedurou) „násobení čísel", pak je vyhodnocením symbolu `*` právě operace (procedura) „násobení čísel".

Zbývá popsat zamýšlený význam seznamů. Seznamy jsou z hlediska vyhodnocovacího procesu „příkazem pro provedení operace", přitom daná operace je vždy určena *prvním prvkem seznamu,* respektive hodnotou vzniklou jeho vyhodnocením. Hodnoty, se kterými bude operace provedena jsou dány hodnotami dalších prvků seznamu – *operandů*. Při psaní programů v jazyku Scheme bychom tedy vždy měli mít na paměti, že seznamy musíme psát ve tvaru

$$(\langle operace\rangle\ \langle operand_1\rangle\ \langle operand_2\rangle \cdots \langle operand_n\rangle)$$

Všimněte si, že tento tvar přirozeně vede na prefixovou notaci výrazů, protože píšeme „operaci" před „operandy". Například seznam `(+ 10 20 30)` je z hlediska vyhodnocování příkaz pro provedení operace „sčítání čísel" jež je hodnotou navázanou na symbol `+`. V tomto případě je operace provedena se třemi hodnotami: „deset", „dvacet" a „třicet", což jsou číselné hodnoty operandů `10`, `20` a `30`. Výsledkem vyhodnocení symbolického výrazu `(+ 10 20 30)` by tedy měla být hodnota „šedesát".

Je dobré uvědomit si rozdíl mezi symbolickým výrazem a jeho hodnotou, což jsou samozřejmě dvě různé věci. Například uvážíme-li výraz `(* 10 (+ 5 6) 20)`, pak se z hlediska vyhodnocení jedná o příkaz provedení operace „násobení čísel" s hodnotami „deset", „výsledek vyhodnocení symbolického výrazu `(+ 5 6)`" a „dvacet". To jest, druhou hodnotou při provádění operace násobení není samotný symbolický výraz `(+ 5 6)`, ale *hodnota vzniklá jeho vyhodnocením.* Výsledkem vyhodnocení celého výrazu je tedy číselná hodnota „dva tisíce dvě sta".

V některých případech se můžeme dostat do situace, že pro některé symbolické výrazy nemůžeme uvažovat „hodnotu" na kterou se vyhodnotí, protože pro ně vyhodnocení není popsáno. V takovém případě říkáme, že se v programu (či v daném symbolické výrazu) nachází *sémantická chyba.*

[1] Alespoň je tomu tak až do okamžiku, kdy takovým symbolům nějakou hodnotu přiřadíme. Tímto problémem se budeme zabývat v další části této lekce. Zatím pro jednoduchost předpokládejme, že symboly buď jsou nebo nejsou svázány s hodnotami.

Příklad 1.10. Uvažujme symbolický výraz `(10 + 20)`. Tento výraz má na prvním místě symbolický výraz `10`, jehož hodnota je „číslo deset“, což není operace. V tomto případě není vyhodnocení symbolického výrazu `(10 + 20)` definováno, protože první prvek seznamu se nevyhodnotil na operaci – symbolický výraz `(10 + 20)` je sice syntakticky správně (jedná se o symbolický výraz), ale z hlediska vyhodnocovacího procesu obsahuje sémantickou chybu.

Poznámka 1.11. Výrazy v prefixové notaci lze vzhledem k jejich zamýšlené interpretaci snadno číst:

`(+ 2 (* 3 4))`
„sečti dvojku se součinem trojky a čtyřky“

`(+ (* 2 3) 4)`
„sečti součin dvojky a trojky s číslem čtyři“

a podobně.

Při psaní symbolických výrazů a hodnot, na které se vyhodnotí, příjmeme následující konvenci. Symbolické výrazy, které budeme vyhodnocovat, budeme psát nalevo od symbolu „$\longmapsto$“ a hodnoty vzniklé vyhodnocením budeme psát napravo od tohoto symbolu. Přitom se hodnoty budeme snažit zapisovat pokud možno tak, jak je vypisuje drtivá většina interpretů jazyka Scheme. Pokud daný výraz nelze vyhodnotit v důsledku chyby, budeme za pravou stranu „$\longmapsto$“ uvádět stručný popis chyby. Samotný symbol „$\longmapsto$“ můžeme číst „se vyhodnotí na“. Viz následující ukázky.

`-10.6`	$\longmapsto$	`-10.6`
`128`	$\longmapsto$	`128`
`*`	$\longmapsto$	„procedura násobení čísel“
`(+ 1 2)`	$\longmapsto$	`3`
`(+ 1 2 34)`	$\longmapsto$	`37`
`(- 3 2)`	$\longmapsto$	`1`
`(- 2 3)`	$\longmapsto$	`-1`
`(* pi 2)`	$\longmapsto$	`6.283185307179586`
`(+ 2 (* 3 4))`	$\longmapsto$	`14`
`(blah 2 3)`	$\longmapsto$	chyba: symbol `blah` nemá vazbu
`((+ 2 3) 4 6)`	$\longmapsto$	chyba: hodnota „číslo pět“ není operace

Poznámka 1.12. Při vyhodnocení posledních dvou výrazů došlo k chybě. U předposledního výrazu zřejmě protože, že na symbol `blah` nebyla navázaná operace. U posledního výrazu si všimněte, že symbolický výraz `(+23)` se vyhodnotí na hodnotu „číslo pět“, což není operace. Vyhodnocení proto opět končí chybou, protože se očekává, že první prvek seznamu `((+ 2 3) 4 6)` se vyhodnotí na operaci, která by byla dále použita s hodnotami „čtyři“ a „šest“.

Při vyhodnocování výrazů si člověk, na rozdíl od počítače, může dopomoci svým vhledem do výrazu. Například u symbolického výrazu `(* 2 (/ 3 (+ 4 5)))` téměř okamžitě vidíme, že k jeho vyhodnocení nám stačí „sečíst čtyřku s pětkou, tímto výsledkem podělit trojku a nakonec tuto hodnotu vynásobit dvojkou.“ Tento postup je zcela v souladu s naší intuicí a s tím, jak jsme vyhodnocování výrazů zatím poněkud neformálně popsali. Interpret jazyka Scheme, což je opět program, ale tuto možnost „vhledu“ nemá. Abychom byli schopni interprety programovacích jazyků konstruovat, musíme sémantiku jazyka (v našem případě proces vyhodnocování výrazů) přesně popsat krok po kroku. Postup vyhodnocování musí být univerzální a použitelný pro libovolný symbolický výraz.

Nyní popíšeme interpretaci symbolických výrazů formálněji. Představíme zjednodušený model interpretu jazyka Scheme – *abstraktní interpret Scheme* a popíšeme vyhodnocování výrazů krok za krokem.

Interprety jazyka Scheme pracují interaktivně. Postupně načítají výrazy (buď ze souboru nebo čekají na vstup uživatele) a vyhodnocují je. Po načtení jednotlivého výrazu je výraz zpracován, vyhodnocen a výsledná hodnota je oznámena uživateli. Poté se opakuje totéž pro další výraz na řadě dokud není vyčerpán celý vstup. Interprety ovšem nepracují se symbolickými výrazy přímo, to jest ve formě sekvencí znaků

tak, jak je zapisujeme (což by bylo neefektivní), ale převádějí je vždy do své vnitřní efektivní reprezentace. Této vnitřní reprezentaci symbolických výrazů budeme říkat *interní reprezentace.* Pro ilustraci, například číslo -123 je dostatečné malé na to, aby jej bylo možné efektivně kódovat čtyřbajtovým datovým typem „integer“, se kterým umějí pracovat všechny běžně používané 32-bitové procesory. Takže interní reprezentací symbolického výrazu `-123` může být třeba 32-bitová sekvence nul a jedniček kódující číslo -123. Seznamy jsou v interpretech reprezentovány zpravidla dynamickými spojovými datovými strukturami (podrobnosti uvidíme v dalších lekcích).

Konkrétní tvar interní reprezentace symbolických výrazů nás ale až na výjimky zajímat nebude. Pro nás je důležité uvědomit si, že každý výraz je na počátku svého zpracování převeden do své interní reprezentace a pak už se výhradně pracuje jen s ní. Část interpretu, která je odpovědná za načtení výrazu, případné odhalení syntaktických chyb a převedení výrazu do interní reprezentace se nazývá *reader*.

Při neformálním popisu interpretace symbolických výrazů jsme uvedli, že při vyhodnocování seznamů dochází k „provádění operací.“ Teď tento pojem poněkud zpřesníme, protože ono „provádění operací“ je ve své podstatě to, co způsobuje vytváření *kvalitativního výpočetního procesu* (vyhodnocování symbolů a čísel by samo o sobě příliš zajímavé nebylo), proto je mu potřeba věnovat patřičnou pozornost. Nejprve se umluvíme, že místo pojmu „operace“ budeme od této chvíle používat obecnější pojem *procedura.* Procedury, které budeme uvažovat, jsou v podstatě nějaké abstraktní elementy, které budou součástí interpretu. Proto jim také někdy budeme říkat *primitivní procedury.* Příklady primitivních procedur si ukážeme později.

Používání pojmu procedura namísto operace jsme zvolili ze dvou důvodů. První důvod je terminologický. V řadě programovacích jazyků je pojem operace spojován prakticky pouze jen s aritmetickými a logickými operacemi (tak je tomu třeba v jazyku C). Naopak pojem *procedura* bývá obvykle chápán obecněji. V terminologii některých jazyků, jako je například Pascal, jsou procedury chápány jako „podprogramy“, které mohou být opakovaně „volány“. V našem pojetí jsou (primitivní) procedury elementy interpretu, které nám umožňují vypočíst novou hodnotu z jiných hodnot. V terminologii většiny funkcionálních programovacích jazyků, jsou obvykle procedury nazývány *funkce* (tak je tomu třeba v jazycích Common LISP, ML a Haskell), odtud taky plyne název funkcionálního paradigma. My pojem „funkce“ používat nebudeme, protože koliduje s matematickým pojmem funkce. O vztahu procedur a matematických funkcí (čili zobrazení) se budeme bavit v další lekci. Druhým důvodem je, že slovo „procedura“ je používáno ve standardu jazyka Scheme a v drtivé většině dostupné literatury, viz například [SICP, Dy96, R6RS, SF94].

Operace, které jsme neformálně používali v úvodu této lekce při nastínění vyhodnocování seznamů, neoperovaly se symbolickými výrazy, nýbrž s hodnotami vzniklými vyhodnocováním symbolických výrazů. Otázkou je, jak bychom se na ony „hodnoty“ měli přesně dívat. Doposud jsme vše demonstrovali pouze na aritmetických operacích, nabízelo by se tedy uvažovat jako hodnoty *interní reprezentace čísel.* Jelikož však budeme chtít, aby byly (primitivní) procedury schopny manipulovat i s jinými daty, než s čísly, zavedeme obecnější pojem *element jazyka.* Na elementy jazyka budeme pohlížet jako na přípustné hodnoty, se kterými mohou (ale nemusejí) manipulovat primitivní procedury. Za jedny z elementů budeme považovat i samotné primitivní procedury, což bude mít zajímavé důsledky jak uvidíme v dalších lekcích. Následující definice pojem element jazyka zavádí.

Definice 1.13 (elementy jazyka)**.**

(i) Interní reprezentace každého symbolického výrazu je element jazyka,

(ii) každá primitivní procedura je element jazyka. ■

Dle definice 1.13 jsou tedy všechny interní reprezentace čísel, symbolů a seznamů elementy jazyka. Rovněž primitivní procedury jsou elementy jazyka. Množinu všech elementů jazyka budeme během výkladu postupně doplňovat o nové elementy. Z pohledu elementů jazyka je *reader* částí interpretu, která převádí symbolické výrazy (posloupnosti znaků) na elementy jazyka (interní reprezentace symbolických výrazů). Naopak jedna z částí interpretu, zvaná *printer*, má opačnou úlohu. Jejím úkolem je pro daný element jazyka

vrátit (třeba vytisknout na obrazovku) jeho externí reprezentaci. Pod pojmem *externí reprezentace* elementu máme na mysli pro člověka čitelnou reprezentaci daného elementu. U symbolických výrazů se smluvíme na tom, že pokud je element E interní reprezentací symbolického výrazu e, pak bude externí reprezentace elementu E právě symbolický výraz e. Převodem symbolického výrazu do jeho interní reprezentace a zpět do externí tedy získáme výchozí výraz. Navíc pokud je element E interní reprezentací symbolického výrazu e, tak potom externí reprezentace elementu E bude opět pro *reader* čitelná. U procedur je situace jiná. Procedury jsou abstraktní elementy u nichž se domluvíme na tom, že jejich externí reprezentace (ať už je jakákoliv) bude pro *reader* nečitelná. Tím zaručíme jakousi „čistotu“, protože *reader* slouží k načítání *právě* symbolických výrazů, tedy *ničeho jiného.* Během výkladu budeme při uvádění externí reprezentace procedur psát jejich stručný slovní popis. Uděláme tedy následující úmluvu.

Úmluva 1.14 (o čitelnosti externích reprezentací)**.** Pokud neuvedeme jinak, externí reprezentace elementu E je čitelná readerem, právě když je E interní reprezentací některého symbolického výrazu. ■

Symbolický výraz je čistě syntaktický pojem. Symbolické výrazy jsou zavedeny jako sekvence znaků, splňující podmínky definice 1.6. Naproti tomu elementy jazyka jsou sémantické pojmy. Elementy jazyka jsou prvky množiny (všech elementů jazyka). Elementy jazyka budeme dále používat jako hodnoty a budou hrát klíčovou roli ve vyhodnocovacím procesu. Role elementů jazyka je úzce svázaná s jejich významem (sémantikou). *Reader* převádí symbolické výrazy na elementy jazyka, které jsou interními reprezentacemi symbolických výrazů. Naopak *printer* slouží k převodu elementů do jejich externí reprezentace.

Nyní popíšeme, co budeme mít na mysli pod pojmem *aplikace primitivní procedury.* V podstatě se jedná o formalizaci již dříve uvedeného „provádění operací“. Roli operací hrají primitivní procedury. Místo provádění „operací s danými hodnotami“ budeme hovořit o *aplikaci procedur s danými argumenty,* přitom konkrétní *argumenty* zastupují konkrétní elementy jazyka (hodnoty), se kterými je daná operace aplikována. Výsledkem aplikace procedury je buď *výsledný element* (pokud aplikace proběhna v pořádku), nebo chybové hlášení „CHYBA: Proceduru nelze aplikovat s danými argumenty.“, pokud během aplikace došlo k chybě. Výsledný element je chápán jako *výsledná hodnota* provedení aplikace. Někdy říkáme, že operace *vrací* výslednou hodnotu. Viz následující ilustrativní příklad.

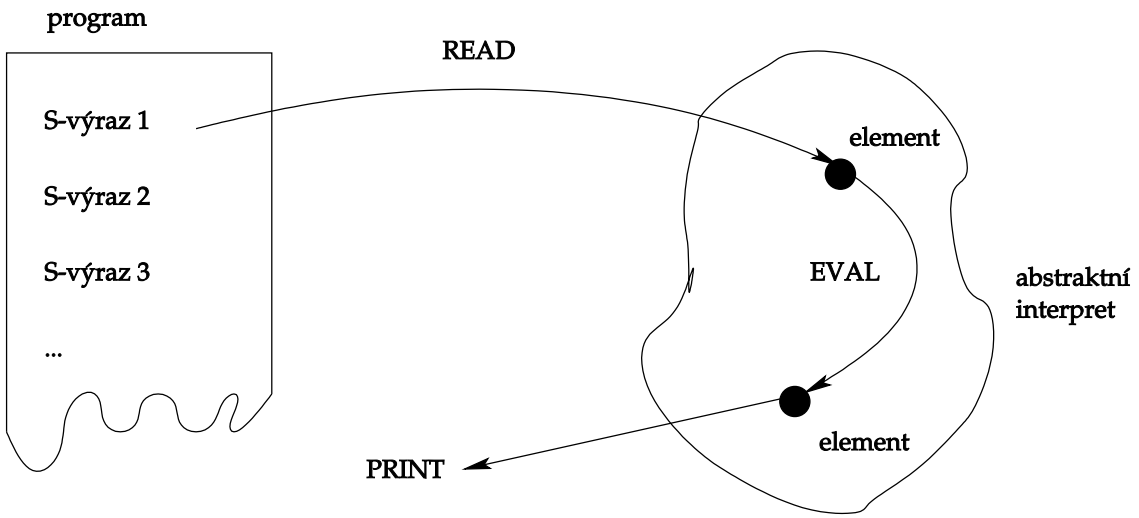
Příklad 1.15. Aplikací procedury „sečti čísla“ na argumenty jimiž jsou elementy zastupující číselné hodnoty 1, 10 a -30 je výsledný element zastupující číselnou hodnotu -19. Aplikací procedury „zjisti zdali je dané číslo sudé“ na číselný argument 12.7 je hlášení „CHYBA: Procedura aplikována se špatným argumentem (vstupní argument musí být celé číslo).“. Aplikací procedury „vypočti druhou odmocninu“ na dva argumenty číselné hodnoty 4 a 10 je hlášení „CHYBA: Procedura aplikována se špatným počtem argumentů (vstupní argument musí být jediné číslo).“.

Je samozřejmé, že při aplikaci primitivní procedury s danými argumenty je obecně nutné specifikovat pořadí jednotlivých argumentů (představte si situaci pro proceduru „odečti dvě čísla“). U některých procedur budeme vždy uvažovat *pevný počet argumentů*, u některých procedur *proměnlivý.* Pro zjednodušení zavedeme následující

Označení 1.16. Nechť E je primitivní procedura a $E_1, \ldots, E_n$ jsou libovolné elementy jazyka. Výsledek *aplikace primitivní procedury E na argumenty $E_1, \ldots, E_n$ v tomto pořadí,* budeme značit $\mathrm{Apply}[E, E_1, \ldots, E_n]$. Pokud je výsledkem této aplikace element F, pak píšeme $\mathrm{Apply}[E, E_1, \ldots, E_n] = F$.

Z matematického pohledu je Apply zavedené v předchozím označení částečné zobrazení z množiny všech konečných posloupností elementů do množiny všech elementů. Pro některé n-tice elementů však toto zobrazení není definované (proto jej označujeme jako „částečné zobrazení“).

Obrázek 1.2. Schéma cyklu REPL



U primitivních procedur se nebudeme zabývat tím, jak při aplikaci (s danými argumenty) vznikají výsledné hodnoty. Na primitivní procedury se z tohoto úhlu pohledu budeme dívat jako na „černé skříňky." To jest, budeme mít přesně popsané, co při aplikaci konkrétní primitivní procedury vznikne, jaké argumenty jsou přípustné a podobně, ale nebudeme se zabývat samotným výpočtem (nebudeme vidět „dovnitř" primitivních procedur – odtud taky plyne jejich název). S konceptem elementu jako „černé skříňky" se v budoucnu budeme setkávat častěji.

Nyní můžeme upřesnit, co máme na mysli pod pojmem abstraktní interpret jazyka Scheme. Abstraktní interpret chápeme jako množinu všech elementů jazyka s pravidly jejich vyhodnocování:

Definice 1.17 (abstraktní interpret)**.** Množinu všech elementů jazyka spolu s pravidly jejich vyhodnocování budeme nazývat *abstraktní interpret jazyka Scheme.* ■

Posledním logickým krokem, který provedeme v této sekci je popis vyhodnocování, tím budeme mít úplně popsaný abstraktní interpret. Vyhodnocování programů, které se skládají z posloupností symbolických výrazů probíhá v cyklu, kterému říkáme REPL. Jméno cyklu je zkratka pro „Read" (načti), „Eval" (vyhodnoť), „Print" (vytiskni) a „Loop" (opakuj), což jsou jednotlivé části cyklu. Budeme-li předpokládat, že máme vstupní program, jednotlivé fáze cyklu REPL můžeme popsat následovně:

Read: Pokud je prázdný vstup, činnost interpretu končí. V opačném případě *reader* načte první vstupní symbolický výraz. Pokud by se to nepodařilo v důsledku syntaktické chyby, pak je činnost interpretu ukončena chybovým hlášením „CHYBA: Syntaktická chyba". Pokud je symbolický výraz úspěšně načten, je převeden do své interní reprezentace. Výsledný element, označíme jej E, bude zpracován v dalším kroku.

Eval: Element E z předchozího kroku je vyhodnocen. Vyhodnocení provádí část interpretu, které říkáme *evaluator.* Pokud během vyhodnocení došlo k chybě, je napsáno chybové hlášení a ukončena činnost interpretu. Pokud vyhodnocení končí úspěchem, pak je výsledkem vyhodnocení element. Označme tento element F. Výsledný element F bude zpracován v dalším kroku.

Print: Provede se převod elementu F z předchozího kroku do jeho externí reprezentace. Tuto část činnosti obstarává *printer.* Výsledná externí reprezentace je zapsána na výstup (vytištěna na obrazovku, nebo zapsána do souboru, a podobně).

Loop: Vstupní symbolický výraz, který byl právě zpracován v krocích „Read", „Eval" a „Print" je odebrán ze vstupu. Dále pokračujeme krokem „Read".

Poznámka 1.18. (a) Průběh cyklu REPL je symbolicky naznačen na obrázku 1.2.

(b) Části cyklu „Read“, „Print“ a „Loop“ jsou více méně triviální a jejich provedením se nebudeme zabývat, opět se na ně můžeme dívat jako na černé skříňky. Nutné je přesně specifikovat „Eval“, což je část odpovědná za vyhodnocování a v důsledku za vznik výpočetního procesu. Zopakujme, že vstupem pro „Eval“ není symbolický výraz, ale jeho interní reprezentace (tedy element jazyka). Výstupem „Eval“ je opět element. Při neformálním popisu vyhodnocování na začátku této sekce jsme se bavili o vyhodnocování symbolických výrazů: čísel, symbolů a seznamů; nyní vidíme, že přísně vzato vyhodnocování probíhá nad elementy jazyka. To je z hlediska oddělení syntaxe a sémantiky (a nejen z tohoto hlediska) žádoucí.

(c) V některých případech, zejména v případě reálných interpretů pracujících interaktivně, není žádoucí, aby byla interpretace při výskytu chyby úplně přerušena. Interprety obvykle dávají uživateli k dispozici řadu možností, jak blíž prozkoumat vznik chyby, což může být v mnoha případech dost složité. Místo ukončení činnosti při vzniku chyby by i náš abstraktní interpret nemusel zareagovat okamžitým ukončením činnosti. Místo toho by mohl odebrat vstupní výraz ze vstupu a pokračovat ve fázi „Read“ dalším výrazem.

(d) Práce s interpretem, která je řízená cyklem REPL má své praktické dopady při práci se skutečným interpretem. Bývá zvykem, že programy jsou postupně laděny po částech. To jest programátoři postupně vkládají symbolické výrazy, a sledují jejich vyhodnocování. Takto je obvykle postupně sestavován program, který je uložen v souboru (souborech) na disku. Poté, co je program vytvořen a odladěn, jej lze předat interpretu tak, aby cyklus REPL běžel neinteraktivně přímo nad souborem (výrazy jsou čteny přímo ze souboru). Při čtení výrazů ze souboru vede obvykle chyba rovnou k zastavení programu.

Nyní obrátíme naši pozornost na část „Eval“ cyklu REPL.

Poznámka 1.19. V dalším výkladu budeme někdy (bez dalšího upozorňování) kvůli přehlednosti ztotožňovat interní reprezentaci symbolických výrazů se samotnými symbolickými výrazy. To nám umožní stručnější vyjadřování, přitom nebude možné, abychom oba pojmy smíchali, protože se budeme zabývat vyhodnocováním elementů, kde již (vstupní) symbolické výrazy nehrají roli. Pořád je ale nutné mít na paměti, že jde o dvě různé věci. Pokud se tedy dále budeme bavit například o seznamu `(+ 2 (* 3 4))` a jeho třetím prvku, máme tím na mysli interní reprezentaci tohoto seznamu a interní reprezentaci jeho třetího prvku a tak podobně.

Během vyhodnocování elementů budeme potřebovat vyhodnocovat symboly. Jak jsme již předeslali, symboly slouží k „pojmenování hodnot“. Při vyhodnocování výrazů je vždy k dispozici tabulka zachycující vazby mezi symboly a elementy. Tuto tabulku nazýváme *prostředí.* Prostředí si lze představit jako tabulku se dvěma sloupci, viz obrázek 1.3.

Obrázek 1.3. Prostředí jako tabulka vazeb mezi symboly a elementy

symbol	*element*
E_1	F_1
E_2	F_2
$\vdots$	$\vdots$
E_k	F_k
$\vdots$	$\vdots$

V levém sloupci prostředí se nacházejí symboly (jejich interní reprezentace), v pravém sloupci se nacházejí libovolné elementy. Pokud je symbol E uveden v tabulce v levém sloupci, pak hodnotě v pravém sloupci na témže řádku říkáme *aktuální vazba symbolu E v prostředí*. Pokud symbol E není uveden v levém sloupci, pak říkáme, že *E nemá vazbu v prostředí*, nebo případně, že *aktuální vazba symbolu E není definovaná.* Na počátku interpretace jsou v prostředí (někdy mu říkáme *počáteční prostředí*) dány *počáteční vazby symbolů*. Jedná se zejména o vazby symbolů na primitivní procedury. Například vazbou symbolu `+` v počátečním prostředí je primitivní procedura „sečti čísla“, aktuální vazbou symbolu `sqrt` je procedura „vypočti druhou

odmocninu“, a podobně. Při práci s interpretem je potřeba znát základní vazby symbolů, abychom znali „jména primitivních procedur“, se kterými budeme chtít pracovat.

Označení 1.20. Pokud chceme říct, že F bude označovat výsledek vyhodnocení elementu E, budeme tento fakt zapisovat $F := \mathrm{Eval}[E]$. Pokud chceme říct, že výsledek vyhodnocení elementu E definujeme jako element F, pak tento fakt zapíšeme $\mathrm{Eval}[E] := F$.

Nyní můžeme popsat vyhodnocování elementů (v počátečním prostředí).

Definice 1.21 (vyhodnocení elementu E)**.**
Výsledek vyhodnocení elementu E, značeno $\mathrm{Eval}[E]$, je definován:

(A) Pokud je E *číslo*, pak $\mathrm{Eval}[E] := E$.

(B) Pokud je E *symbol*, mohou nastat dvě situace:

(B.1) Pokud E *má aktuální vazbu* F, pak $\mathrm{Eval}[E] := F$.

(B.e) Pokud E *nemá vazbu*, pak ukončíme vyhodnocování hlášením „CHYBA: Symbol E nemá vazbu.“.

(C) Pokud je E *seznam* tvaru $(E_1\ E_2 \cdots E_n)$, pak nejprve vyhodnotíme jeho první prvek a výsledek vyhodnocení označíme F_1, formálně: $F_1 := \mathrm{Eval}[E_1]$. Vzhledem k hodnotě F_1 rozlišíme dvě situace:

(C.1) Pokud je F_1 *procedura*, pak se v nespecifikovaném pořadí vyhodnotí zbylé prvky $E_2, \ldots, E_n$ seznamu E, výsledky jejich vyhodnocení označíme $F_2, \ldots, F_n$. Formálně:

$$\begin{aligned} F_2 &:= \mathrm{Eval}[E_2], \\ F_3 &:= \mathrm{Eval}[E_3], \\ &\vdots \quad\ \vdots \\ F_n &:= \mathrm{Eval}[E_n]. \end{aligned}$$

Dále položíme $\mathrm{Eval}[E] := \mathrm{Apply}[F_1, F_2, \ldots, F_n]$, tedy výsledkem vyhodnocení E je element vzniklý *aplikací procedury* F_1 *na argumenty* $F_2, \ldots, F_n$.

(C.e) Pokud F_1 není *procedura*, skončíme vyhodnocování hlášením
„CHYBA: Nelze provést aplikaci: první prvek seznamu E se nevyhodnotil na proceduru.“.

(D) Ve všech ostatních případech klademe $\mathrm{Eval}[E] := E$. ■

Poznámka 1.22. (a) Všimněte si, že bod (A) je vlastně pokryt bodem (D), takže bychom jej přísně vzato mohli bez újmy vynechat. Na druhou stranu bychom mohli ponechat bod (A) a zrušit bod (D), protože elementy, které vyhodnocujeme jsou (zatím) interní formy symbolických výrazů (viz cyklus REPL). Do budoucna by to ale nebylo prozíravé, takže bod (D) ponecháme, i když momentálně nebude hrát roli (jeho úlohu uvidíme už v další sekci).

(b) Vyhodnocování symbolů jsme zavedli v souladu s naším předchozím neformálním popisem. Symboly se vyhodnocují na své aktuální vazby. Nebo končí jejich vyhodnocování chybou (v případě, když vazba neexistuje).

(c) Seznamy se rovněž vyhodnocují v souladu s předešlým popisem. Vyhodnocení seznamu probíhá tak, že je nejprve vyhodnocen jeho první prvek. V případě, že se první prvek vyhodnotí na (*primitivní*) *proceduru*, se v nespecifikovaném pořadí vyhodnotí ostatní prvky seznamu; potom je procedura aplikována na argumenty, kterými jsou právě výsledky vyhodnocení zbylých prvků seznamu – tedy všech prvků kromě prvního jímž je samotná procedura.

V této sekci jsme představili interpret primitivního funkcionálního jazyka, který budeme dále rozšiřovat až budeme mít k dispozici úplný programovací jazyk, který bude z výpočetního hlediska stejně silný jako ostatní programovací jazyky. V tuto chvíli si už ale můžeme všimnout základní ideje funkcionálního programování: *výpočetní proces* generovaný programem se skládá z po sobě jdoucích aplikací procedur. Hodnoty vzniklé aplikací jedné procedury jsou použité jako argumenty při aplikaci jiné procedury. Pořadí v jakém jsou procedury aplikovány a samotnou aplikaci řídí *evaluátor.*

Příklad 1.23. V tomto příkladu si rozebereme, jak interpret vyhodnocuje vybrané seznamy. Budeme přitom postupovat přesně podle definice 1.21.

1. Seznam `(+ 1 2)` se vyhodnotí na `3`. Podrobněji: `(+ 1 2)` je seznam, tedy vyhodnocování postupuje dle bodu (C). Vyhodnotíme první prvek seznamu, to jest stanovíme Eval[+]. Jelikož je `+` symbol, postupujeme dle bodu (B). Na symbol `+` je navázána procedura pro sčítání. V bodu (C) tedy postupujeme po větvi (C.1) s hodnotou F_1 = „procedura sčítání čísel“. Nyní v nespecifikovaném pořadí vyhodnotíme zbylé prvky seznamu `1` a `2`. Oba argumenty jsou čísla, takže Eval[`1`] = `1` a Eval[`2`] = `2`. Nyní aplikujeme proceduru sčítání čísel na argumenty `1` a `2` a dostaneme výslednou hodnotu `3`. Výsledkem vyhodnocení `(+ 1 2)` je `3`.
2. Seznam `(+ (* 3 (+ 1 1)) 20)` se vyhodnotí na `26`. Jelikož se jedná o seznam, postupuje se krokem (C). První prvek seznamu, symbol `+`, je vyhodnocen na proceduru sčítání, takže vyhodnocení pokračuje krokem (C.1). V nespecifikovaném pořadí jsou vyhodnoceny argumenty `(* 3 (+ 1 1))` a `20`. Seznam `(* 3 (+ 1 1))` se bude vyhodnocovat následovně. První je opět vyhodnocen jeho první prvek, což je symbol `*`. Jeho vyhodnocením je „procedura násobní čísel“, pokračujeme tedy vyhodnocením zbylých prvků seznamu. `3` se vyhodnotí na sebe sama a seznam `(+ 1 1)` se opět vyhodnocuje jako seznam. Takže nejprve vyhodnotíme jeho první prvek, což je symbol `+`, jeho vyhodnocením je procedura „sčítání “. Takže vyhodnocujeme zbylé dva prvky seznamu. Jelikož jsou to dvě čísla (obě jedničky), vyhodnotí se na sebe sama. V tuto chvíli dochází k aplikaci sčítání na argumenty `1` a `1`, takže dostáváme, že vyhodnocením seznamu `(+ 1 1)` je číslo `2`. Nyní vyhodnocování pokračuje bodem, který způsobil vyhodnocování seznamu `(+ 1 1)`. To je bod, ve kterém jsme vyhodnocovali seznam `(* 3 (+ 1 1))`. Jelikož již jsme vyhodnotili všechny jeho prvky, aplikujeme proceduru násobení na hodnotu `3` a `2` (výsledek předchozího vyhodnocení). Výsledkem vyhodnocení `(* 3 (+ 1 1))` je tedy hodnota `6`. Nyní se vracíme do bodu, který způsobil vyhodnocení výrazu `(* 3 (+ 1 1))`, což je bod, ve kterém jsme vyhodnocovali postupně prvky seznamu `(+ (* 3 (+ 1 1)) 20)`. Hodnotu druhého prvku jsme právě obdrželi, to je číslo `6`. Třetí prvek seznamu je číslo `20`, které se vyhodnotí na sebe sama. V tuto chvíli aplikujeme proceduru sčítání s hodnotami `6` a `20`. Výsledkem je hodnota `26`, což je výsledek vyhodnocení původního seznamu.
3. Vyhodnocení `(10 + 20)` končí chybou, která nastane v kroku (C.e). Jelikož se jedná o seznam, je nejprve vyhodnocen jeho první prvek. To je číslo `10`, které se vyhodnotí na sebe sama. Jelikož se první prvek seznamu nevyhodnotil na proceduru, vyhodnocení končí bodem (C.e).
4. Vyhodnocení `((+ 1 2) (+ 1 3))` také končí chybou, která nastane v kroku (C.e). Při vyhodnocování seznamu `((+ 1 2) (+ 1 3))` je nejprve vyhodnocen jeho první prvek. Tím je seznam `(+ 1 2)`, který se vyhodnotí na číslo `3` (detaily jsme vynechali). Nyní jsme se dostali do bodu, kdy byl opět první prvek seznamu, konkrétně seznamu `((+ 1 2) (+ 1 3))`, vyhodnocen na něco jiného než je procedura, takže vyhodnocení končí neúspěchem v bodě (C.e).
5. Vyhodnocení `((* 2 3 4))` končí rovněž chybou, která nastane v kroku (C.e). Důvod je stejný jako v předchozím bodě. První prvek seznamu je totiž seznam `(* 2 3 4)`, jeho vyhodnocením je číslo `24`. První prvek seznamu `((* 2 3 4))` se tedy nevyhodnotil na proceduru.
6. Vyhodnocení `(+ 2 (odmocni 10))` končí chybou, která nastane v kroku (B.e), protože použitý symbol `odmocni` nemá v počátečním prostředí vazbu. Důležité je ale uvědomit si, ve kterém momentu k chybě dojde. Je to pochopitelně až při vyhodnocování prvního prvku seznamu `(odmocni 10)`. Kdybychom místo `(+ 2 (odmocni 10))` vyhodnocovali seznam `(+ (2) (odmocni 10))`, pak by rovněž došlo k chybě, ale za předpokladu, že by náš interpret vyhodnocoval prvky seznamů zleva doprava, se vyhodnocování zastavilo v bodě (C.e) a k pokusu o vyhodnocení symbolu `odmocni` by vůbec nedošlo. Vysvětlete sami proč.
7. Vyhodnocení `(* (+ 1 2) (sqrt 10 20))` končí chybou při pokusu o aplikaci procedury „vypočti druhou odmocninu“, která je navázaná na symbol `sqrt`, vypsáno bude hlášení: „CHYBA: Nepřípustný počet argumentů.“

V předchozím příkladu si lze všimnout jednoho významného rysu při programování v jazyku Scheme. Tím je pouze malá libovůle v umístění závorek. Ve většině programovacích jazyků není „příliš velký

hříchy“, když provedeme nadbytečné uzávorkování výrazu. Například v jazyku C mají aritmetické výrazy $2 + x$, $(2 + x)$, $((2 + x))$, $((2 + (x)))$, a tak dále, stejný význam. V jazyku Scheme je z pohledu vyhodnocování každý pár závorek separátním *příkazem pro provedení aplikace procedury*. Tím pádem si nemůžeme dovolit „jen tak přidat do výrazu navíc závorky“. Srovnejte výraz `(+ 1 2)` a naproti tomu výrazy jejichž vyhodnocování končí chybami: `((+ 1 2))`, `(+ (1) 2)`, `((+) 1 2)`, a podobně.

Nyní si představíme některé základní primitivní procedury, které jsou vázány na symboly v počátečním prostředí. Pokud budeme hovořit o procedurách, budeme je pro jednoduchost nazývat jmény symbolů, na které jsou navázány. Například tedy primitivní proceduře „sečti čísla“ budeme říkat procedura `+`. Správněji bychom měli samozřejmě říkat „procedura navázaná na symbol `+`“, protože symbol není procedura. Nyní ukážeme, jak lze používat základní aritmetické procedury `+` (sčítání), `*` (násobení), `-` (odčítání) a `/` (dělení).

Procedura `+` pracuje s libovolným počtem číselných argumentů. Sčítat lze tedy libovolný počet sčítanců. Viz následující příklady:

```
(+ 1 2 3)       ⟹  6
(+ (+ 1 2) 3)   ⟹  6
(+ 1 (+ 2 3))   ⟹  6
(+ 20)          ⟹  20
(+)             ⟹  0
```

U prvních tří příkladů si všimněte toho, že díky možnosti aplikovat `+` na větší počet argumentů než dva se nám výrazně zkracuje zápis výrazu. Kdybychom tuto možnost neměli, museli bychom použít jeden z výrazů na druhém a třetím řádku. Aplikací `+` na jednu číselnou hodnotu je vrácena právě ona hodnota. Poslední řádek ukazuje aplikaci `+` bez argumentů. V tomto případě je vrácena hodnota `0`, protože se jedná o *neutrální prvek* vzhledem k operaci sčítání čísel, to jest 0 splňuje

$$0 + x = x + 0 = x.$$

To jest součet n sčítanců

$$x_1 + x_2 + \cdots + x_n$$

je ekvivalentní součtu těchto n sčítanců, ke kterým ještě přičteme nulu:

$$x_1 + x_2 + \cdots + x_n + 0.$$

Když v posledním výrazu odstraníme všechny $x_1, \ldots, x_n$, zbude nám právě nula, kterou můžeme chápat jako „součet žádných sčítanců“. Pokud se vám zdá zavedení `(+)` umělé nebo neužitečné, pak věřte, že v dalších lekcích uvidíme jeho užitečnost. Důležité je si tento *mezní případ sčítání* zapamatovat.

Analogická pravidla jako pro sčítání platí pro násobení.

```
(* 4 5 6)       ⟹  120
(* (* 4 5) 6)   ⟹  120
(* 4 (* 5 6))   ⟹  120
(* 4)           ⟹  4
(*)             ⟹  1
```

Z pochopitelných důvodů je výsledkem vyhodnocení `(*)` jednička, protože v případě násobení je jednička neutrálním prvkem (platí $1 \cdot x = x \cdot 1 = x$).

Odčítání již na rozdíl od sčítání a odčítání není komutativní ani asociativní operace, to jest obecně *neplatí* $x - y = y - x$ ani $x - (y - z) = (x - y) - z$. Pro odčítání neexistuje neutrální prvek. Primitivní procedura realizující odčítání tedy bude definovaná pro jeden a více argumentů. Na následujících příkladech si všimněte, že odčítání aplikované na jeden argument vrací jeho *opačnou hodnotu* a při odčítání aplikovaném na tři a více argumentů se prvky odčítají postupně zleva-doprava:

```
(- 1 2)         ⟹  -1
(- 1 2 3)       ⟹  -4
(- (- 1 2) 3)   ⟹  -4
```

```
(- 1 (- 2 3))   ⟼  2
(- 1)           ⟼  -1
(-)             ⟼  „CHYBA: Při odčítání je potřeba aspoň jeden argument.“
```

U dělení je situace analogická. Při dělení jednoho argumentu je výsledkem převrácená hodnota:

```
(/ 4 5)         ⟼  4/5
(/ 4 5 6)       ⟼  2/15
(/ (/ 4 5) 6)   ⟼  2/15
(/ 4 (/ 5 6))   ⟼  24/5
(/ 4)           ⟼  1/4
(/)             ⟼  „CHYBA: Při dělení je potřeba aspoň jeden argument.“
```

Všimněte si, jak interprety Scheme vypisují zlomky. Například `4/5` znamená $\frac{4}{5}$, `2/3` znamená $-\frac{2}{3}$ a podobně. Ve stejném tvaru můžeme čísla ve tvaru zlomků i zapisovat. Například:

```
(* -1/2 3/4)   ⟼  -3/8
(/ 1/2 -3/4)   ⟼  -2/3
```

Pozor ale na výraz `2/-3` což je symbol, při jeho vyhodnocení bychom dostali:

```
2/-3   ⟼  „CHYBA: symbol 2/-3 nemá vazbu“
```

Kromě dělení `/` máme ve Scheme k dispozici procedury `quotient` a `modulo` provádějící celočíselné podíl (procedura `quotient`) a vracející zbytek po celočíselném podílu (procedura `modulo`). Srovnejte:

```
(/ 13 5)          ⟼  13/5   podíl
(quotient 13 5)   ⟼  2      celočíselný podíl
(modulo 13 5)     ⟼  3      zbytek po celočíselném podílu
```

Při manipulaci s čísly je vždy nutné mít na paměti *matematická čísla* na jedné straně a jejich *počítačovou reprezentaci* na straně druhé. Je dobře známo, že iracionální čísla jako je π a $\sqrt{2}$ v počítači nelze přesně zobrazit, protože mají nekonečný neperiodický desetinný rozvoj. Taková čísla je možné reprezentovat pouze přibližně (nepřesně). Na druhou stranu třeba číslo $0.\overline{6}$ můžeme v počítači reprezentovat přesně, protože $0.\overline{6} = \frac{2}{3}$. Interprety jazyka Scheme se snaží, pokud je to možné, provádět všechny operace s *přesnou číselnou reprezentací*. Pouze v případě, když přesnou reprezentaci nelze dál zachovat, ji převedou na nepřesnou, pro detaily viz specifikaci přesných a nepřesných čísel uvedenou v [R6RS]. Za přesná čísla považujeme racionální zlomky (a celá čísla), za nepřesná čísla považujeme čísla zapsaná s pohyblivou řádovou tečkou. Viz příklad.

```
(/ 2 3)           ⟼  2/3                 přesná hodnota 2/3
(/ 2 3.0)         ⟼  0.6666666666666666  nereprezentuje přesnou hodnotu 2/3
(* 1.0 (/ 2 3))  ⟼  0.6666666666666666  přesná hodnota 2/3 převedena na nepřesnou
(sqrt 4)          ⟼  2                   přesná hodnota 2
(* -2e-20 4e40)  ⟼  -8e+20              nepřesná hodnota -8 · 10^20
```

Pro vzájemný převod přesných hodnot na nepřesné slouží procedury `exact->inexact` a `inexact->exact`. Samozřejmě, že takový převod je v obou směrech obecně vždy jen přibližný. Viz příklad.

```
(exact->inexact 2/3)                  ⟼  0.6666666666666666
(inexact->exact 0.6666666666666666)   ⟼  6004799503160661/9007199254740992
```

V druhém případě v předchozím výpisu bychom možná očekávali, že nám interpret vrátí přesnou hodnotu $\frac{2}{3}$. To ale není přesná hodnota vstupního čísla `0.6666666666666666`. Místo toho nám byl vrácen komplikovaný racionální zlomek. Pro převedení tohoto zlomku na zlomek jednodušší, který se od něj svou hodnotou příliš neliší, bychom mohli použít primitivní proceduru `rationalize`. Tato procedura se používá se dvěma argumenty, první z nich je *racionální číslo*, druhým parametrem je číslo vyjadřující *maximální odchylku*. Procedura vrací nové racionální číslo, které má co možná nejkratší zápis, a které se od výchozího liší maximálně o danou odchylku. Následující příklad ukazuje, jak by se pomocí `rationalize` dalo konvertovat předchozí nepřehledný zlomek na přesnou hodnotu $\frac{2}{3}$:

```
(rationalize 6004799503160661/9007199254740992 1/1000)   ⟼  2/3
(rationalize (inexact->exact 2/3) 1/1000)                 ⟼  2/3
```

K přesné reprezentaci čísel dodejme, že v jazyku Scheme je možné používat libovolně velká celá čísla a taktéž libovolně přesná racionální čísla (za předpokladu, že se jejich reprezentace vejde do paměti počítače). V praxi to například znamená, že přičtením jedničky k danému celému číslu vždy získáme číslo větší. Ačkoliv toto tvrzení může znít triviálně, ve většině programovacích jazyků, které používají pouze čísla s pevným rozsahem hodnot, tomu tak není.

Ve Scheme je možné počítat rovněž s imaginárními a komplexními čísly a to opět v jejich přesné a nepřesné reprezentaci. Za přesnou reprezentaci komplexního čísla považujeme reprezentaci, kde je reálná a imaginární složka komplexního čísla vyjádřena přesnou reprezentací racionálních čísel. Například tedy `3+2i`, `-3+2i`, `-3-2i`, `+2i`, `-2i`, `3+1/2i`, `-2/3+4/5i` jsou přesně reprezentovaná komplexní čísla. Při zápisu imaginárních čísel je vždy potřeba začít znaménkem, například `2i` je tedy symbol, nikoliv číslo. Na druhou stranu třeba `0.4e10+2i`, `2/3-0.6i` a podobně jsou komplexní čísla v nepřesné reprezentaci. S čísly se dále operuje prostřednictvím již představených primitivních procedur:

```
(sqrt -2)                ⟹  0+1.4142135623730951i
(* +2i (+ 0.4-3i -7i))   ⟹  20.0+0.8i
```

a podobně. V interpretu jsou k dispozici další primitivní procedury, které zde již představovat nebudeme, jedná se o procedury pro výpočet goniometrických, cyklometrických a jiných funkcí, procedury pro zaokrouhlování a jiné. Zájemce tímto odkazuji na [R6RS].

Nakonec této sekce upozorněme na zápis dlouhých symbolických výrazů. Pokud uvážíme například aritmetický výraz

$$1 + \frac{2 \cdot (x + 6)}{\sqrt{x} + 10},$$

pak jej přímočaře přepisujeme ve tvaru:

```
(+ 1 (/ (* 2 (+ x 6)) (+ (sqrt x) 10)))
```

Ačkoliv je výraz pořád ještě relativně krátký, není napsán příliš přehledně, protože jsme jej celý napsali do jednoho řádku. Při psaní složitějších symbolických výrazů je zvykem výrazy vhodně strukturovat a psát části výrazů pod sebe tak, abychom zvětšili jejich čitelnost, například takto:

```
(+ 1
   (/ (* 2 (+ x 6))
      (+ (sqrt x) 10)))
```

Při dělení symbolických výrazů do víc řádků platí úzus, že pokud to není nutné z nějakých speciálních důvodů, pak by žádný řádek neměl začínat závorkou „`)`“.

Z hlediska „programátorského komfortu“ jsou podstatnou součástí téměř každého programu *komentáře*. Pomocí komentářů programátor v programu slovně popisuje konkrétní části program tak, aby se v programu i po nějaké době vyznal. Komentáře jsou ze syntaktického hlediska speciální sekvence znaků vyskytující se v programu. Ze sémantického hlediska jsou komentáře části programů, které jsou zcela ignorovány. To jest překladače a interprety během své činnosti komentáře nijak nezpracovávají a program s komentáři se zpracovává jako stejný program, ve kterém by byly komentáře vynechány.

V jazyku Scheme jsou komentáře sekvence znaků začínající středníkem (znak „`;`“) a končící koncem řádku. Znak středník tedy hraje speciální roli v programu – označuje počátek komentáře. Následující příklad ukazuje část programu s komentáři:

```
;; toto je priklad nekolika komentaru
(+ (* 3 4)  ; prvni scitanec
   (/ 5 6)) ; druhy scitanec
;; k vyhodnoceni (+ 10 20) nedojde, protoze to je soucast tohoto komentare
```

1.6 Rozšíření Scheme o speciální formy a vytváření abstrakcí pojmenováním hodnot

V předchozí sekci jsme ukázali vyhodnocovací proces. Interpret jazyka Scheme můžeme nyní používat jako „kalkulačku“ pro vyhodnocování jednoduchých výrazů. Samozřejmě, že komplexní programovací jazyk

musí umožňovat daleko víc než jen prosté vyhodnocování výrazů představené v předchozí kapitole. Důležitým prvkem každého programovacího jazyka jsou *prostředky pro abstrakci.* Abstrakcí při programování obvykle myslíme možnost vytváření obecnějších programových konstrukcí, které je možné používat ke zvýšení efektivity při programování. Efektivitou ale nemáme nutně na mysli třeba jen rychlost výpočtu, ale třeba provedení programu, které je snadno pochopitelné, rozšiřitelné a lze jej podle potřeby upravovat.

Prvním prostředkem abstrakce, který budeme používat bude *definice vazeb symbolů.* Při práci s interpretem bychom totiž leckdy potřebovali upravit vazby symbolů v prostředí. Uvažme například následující výraz:

```
(* (+ 2 3) (+ 2 3) (+ 2 3))
```

Ve výrazu se třikrát opakuje tentýž podseznam. Při vyhodnocování celého výrazu se tedy bude seznam `(+ 2 3)`. Vyhodnocovat hned třikrát. Kdyby se jednalo o výraz, jehož výpočet by byl časově náročný, pak bychom při výpočtu čekali na jeho výpočet třikrát. V takové situaci by se hodilo mít k dispozici aparát, kterým bychom mohli na nový symbol bez vazby, řekněme `value`, navázat hodnotu vzniklou vyhodnocením `(+ 2 3)`. Jakmile by byla vazba provedena, stačilo by pouze vyhodnotit:

```
(* value value value)
```

Při vyhodnocení posledního uvedeného výrazu už by se seznam `(+ 2 3)` nevyhodnocoval, takže zdržení související s vyhodnocením tohoto seznamu by nastalo pouze jednou.

Mnohem důležitějším důvodem pro možnost definice nových vazeb je zpřehlednění kódu. Ve větších programech se obvykle vyskytují nějaké hodnoty, které se během výpočtu nemění, ale během životního cyklu programu mohou nabývat různých hodnot. Takovou hodnotou může být například sazba DPH. Bylo by krajně nepraktické, kdyby programátor napsal program (například účetní program), ve kterém by byla sazba DPH vyjádřená číslem na každém místě, kde by ji bylo potřeba. Takových míst by ve větším programu byla zcela jistě celá řada a při změně sazby DPH by to vedlo k velkým problémům. Uvědomte si, že problém by zřejmě nešel vyřešit hromadnou náhradou čísla `10` za číslo `20` v programu. Co kdyby se v něm číslo `10` vyskytovalo ještě v nějakém jiném významu než je hodnota DPH? Čisté řešení v takovém případě je použít abstrakci založenou na pojmenování hodnoty symbolem. To jest, v programu je lepší místo číselné hodnoty používat symbol pojmenovaný třeba `vat-value` a pouze na jednom zřetelně označeném místě programu mít provedenu definici vazby čísla reprezentující konkrétní sazbu na symbol `vat-value`.

Z toho co jsme nyní řekli je zřejmé, že potřebujeme mít k dispozici prostředky pro definici nových vazeb nebo předefinování již existujících vazeb novými hodnotami. Samotné „provedení definice“ by měl mít na starosti nějaký element jazyka, který bychom chtěli během programování používat. Řekněme ale již v tuto chvíli, že takovým elementem *nemůže být procedura.* Důvod je v celku prostý. Před aplikací procedury dojde vždy k vyhodnocení všech prvků seznamu. Jedním z nich by mělo být jméno symbolu, na který chceme novou hodnotu nadefinovat. Pokud by ale symbol zatím neměl vazbu (což obecně mít nemusí), pak by vyhodnocení končilo neúspěchem. Proberme tuto situaci podrobněji. Předpokládejme, že bychom chtěli mít k dispozici proceduru navázanou na symbol `define` a chtěli bychom zavádět nové definice takto:

```
(define vat-value 20)
```

Z pohledu vyhodnocovacího procesu je předchozí seznam vyhodnocen tak, že se nejprve vyhodnotí jeho první prvek. Tím je dle našeho předpokladu symbol, který se vyhodnotí na proceduru provádějící definice. Dále by byly vyhodnoceny v nespecifikovaném prostředí zbylé dva argumenty. `20` se vyhodnotí na sebe sama, zde žádný zádrhel neleží. Při vyhodnocování symbolu `vat-value` by ovšem došlo k chybě v kroku (B.e), viz definici 1.21, protože symbol nemá vazbu.

Jednoduchou analýzou problému jsme dospěli do situace, kdy potřebujeme mít v jazyku Scheme nové elementy, při jejichž aplikaci se nebude vyhodnocovat zbytek seznamu tak, jako u procedur. Tyto nové elementy nyní představíme a budeme jim říkat *speciální formy.*

Nyní již známe tři typy elementů jazyka: interní reprezentace symbolických výrazů, primitivní procedury a speciální formy. Speciální formy mají podobný účel jako procedury – jejich aplikacemi lze generovat, případně modifikovat, výpočetní proces. Zcela zásadně se však od procedur liší v tom, jak probíhá jejich aplikace během vyhodnocování.

Speciální formy jsou aplikovány s argumenty jimiž jsou *elementy v jejich nevyhodnocené podobě* a každá *speciální forma si sama určuje* jaké argumenty a v jakém pořadí (zdali vůbec) bude vyhodnocovat. Proto je nutné každou speciální formu vždy detailně popsat. Při popisu se na rozdíl od procedur musíme zaměřit i na to, jak speciální forma manipuluje se svými argumenty. Aplikaci speciální formy E na argumenty $E_1, \ldots, E_n$ budeme značit v souladu s označením aplikace primitivních procedur výrazem $\mathrm{Apply}[E, E_1, \ldots, E_n]$.

Poznámka 1.24. Všimněte si toho, že procedury jsou aplikovány s argumenty, které jsou předem získány vyhodnocením. Tedy procedura přísně vzato nemůže nijak zjistit, vyhodnocením jakých elementů dané argumenty vznikly. To je patrné pokud se podíváme na definici 1.21, bod (C.1).

Jelikož jsme zavedli nový typ elementu – speciální formy, musíme rozšířit vyhodnocovací proces tak, abychom mohli zavést aplikaci speciální formy, který nastane v případě, že první prvek seznamu se vyhodnotí na speciální formu. Pro použití speciálních forem rozšíříme bod (C) části „Eval“ cyklu REPL následovně:

Definice 1.25 (doplnění vyhodnocovacího procesu o speciální formy)**.**
Výsledek vyhodnocení elementu E, značeno $\mathrm{Eval}[E]$, je definován:

(A) Stejné jako v případě definice 1.21.

(B) Stejné jako v případě definice 1.21.

(C) Pokud je E *seznam* tvaru $(E_1\ E_2 \cdots E_n)$, pak nejprve vyhodnotíme jeho první prvek a výsledek vyhodnocení označíme F_1, formálně: $F_1 := \mathrm{Eval}[E_1]$. Vzhledem k hodnotě F_1 rozlišíme tři situace:

- (C.1) Pokud je F_1 *procedura*, pak se postup shoduje s (C.1) v definici 1.21.
- (C.2) Pokud je F_1 *speciální forma*, pak $\mathrm{Eval}[E] := \mathrm{Apply}[F_1, E_2, \ldots, E_n]$.
- (C.e) Pokud F_1 není *procedura ani speciální forma*, skončíme vyhodnocování hlášením „CHYBA: Nelze provést aplikaci: první prvek seznamu E se nevyhodnotil na proceduru ani na speciální formu.“.

(D) Stejné jako v případě definice 1.21. ■

V novém bodu (C.2) jasně vidíme, že výsledkem vyhodnocení E je aplikace speciální formy F_1 na elementy $E_2, \ldots, E_n$, což jsou prvky seznamu E (kromě prvního) v jejich nevyhodnocené podobě.

V jazyku Scheme budeme mít vždy snahu mít co možné nejméně (nezbytné minimum) speciálních forem, protože zavádění speciálních forem do jazyka a jejich využívání programátorem je potenciálním zdrojem chyb. Důvod je zřejmý: jelikož si každá speciální forma určuje vyhodnocování svých argumentů, na uživatele formy (to jest programátora) to na rozdíl od procedur klade další požadavek (pamatovat si, jak se argumenty zpracovávají). Druhý úhel pohledu je více méně epistemický – zařazováním nadbytečných forem do jazyka se zvyšuje jeho celková složitost.

Pokud budeme dále hovořit o speciálních formách, budeme je pro jednoduchost nazývat stejně jako symboly, na které jsou speciální formy navázány v počátečním prostředí. Příslušné symboly budeme značit tmavě modrou barvou, abychom zřetelně viděli, že na uvažovaný symbol je v počátečním prostředí navázána speciální forma. Jako první zavedeme speciální formu `define`, která slouží k definici vazby na symbol.

Definice 1.26 (speciální forma `define`)**.** Speciální forma `define` se používá se dvěma argumenty ve tvaru:

`(define` ⟨*jméno*⟩ ⟨*výraz*⟩`)`

Při aplikaci speciální forma nejprve ověří zdali je argument ⟨*jméno*⟩ symbol. Pokud tomu tak není, aplikace speciální formy je ukončena hlášením „CHYBA: První výraz musí být symbol.“ V opačném případě je vyhodnocen argument ⟨*výraz*⟩. Označme F výsledek vyhodnocení tohoto elementu, to jest $F := \mathrm{Eval}[\langle \textit{výraz} \rangle]$. Dále je v prostředí vytvořena nová vazba symbolu ⟨*jméno*⟩ na element F. Pokud již symbol ⟨*jméno*⟩ měl vazbu, tato původní vazba je nahrazena elementem F. Ve standardu jazyka Scheme [R6RS] se uvádí, že výsledná hodnota aplikace speciální formy `define` není definovaná. ■

Poznámka 1.27. (a) Speciální forma `define` má při své aplikaci *vedlejší efekt*: modifikuje prostředí.

(b) Specifikace jazyka Scheme R^{6}RS neupřesňuje, co je pod pojmem „nedefinovaná hodnota“ myšleno. Respektive specifikace uvádí, že výsledná hodnota může být „jakákoliv“. Jednotlivé interprety proto vracejí při použití `define` různé hodnoty. V našem abstraktním interpretu Scheme vyřešíme situaci s nedefinovanou hodnotou následovně. Budeme uvažovat speciální element jazyka *„nedefinovaná hodnota“*, tento element se bude v souladu s bodem (D) definice 1.25 vyhodnocovat na sebe sama. Externí reprezentací tohoto elementu bude „prázdný řetězec znaků“. Dodejme, že tímto způsobem je nedefinovaná hodnota řešena i v mnoha skutečných interpretech jazyka Scheme.

Příklad 1.28. Tento příklad ukazuje vyhodnocování seznamů obsahujících `define`.

1. Po vyhodnocení `(define ahoj (* 2 3))` se na symbol `ahoj` naváže hodnota `6`. Podrobněji, celý seznam se vyhodnocuje dle bodu (C), takže jako první je vyhodnocen jeho první prvek. Tím je symbol `define`, který se vyhodnotí na speciální formu. Speciální forma je aktivována s argumentem `ahoj` a druhým argumentem je seznam `(* 2 3)`. Jelikož je `ahoj` symbol, je vyhodnocen seznam `(* 2 3)`, jehož vyhodnocením získáme hodnotu `6`. Do prostředí je zavedena vazba symbolu `ahoj` na hodnotu `6`. Výsledkem aplikace speciální formy je nedefinovaná hodnota.
2. V případě seznamu `(define a (* 2 a))` je definován výsledek vyhodnocení seznamu `(* 2 a)` na symbol `a`. V případě, že by během vyhodnocování výrazu symbol `a` doposud neměl žádnou vazbu, dojde k ukončení vyhodnocování výrazu `(* 2 a)` v bodě (B.e), protože `a` nemá vazbu. Pokud by již `a` vazbu mělo a pokud by na `a` bylo navázáno číslo, pak by byla po aplikaci speciální formy hodnota navázaná na `a` nadefinovaná na dvojnásobek původní hodnoty.
3. Po vyhodnocení `(define blah +)` se na symbol `blah` naváže hodnota, která byla navázaná na symbolu `+`. Pokud měl symbol `+` doposud svou počáteční vazbu, pak by byla na symbol `blah` navázána primitivní procedura sčítání.
4. Po postupném vyhodnocení seznamů `(define tmp +)`, `(define + *)`, a `(define * tmp)` se zamění vazby na `+` a `*`. Na pomocný symbol `tmp` bude navázána původní vazba symbolu `+`.
5. Vyhodnocení `(define 10 (* 2 3))` končí chybou při aplikaci speciální formy (`10` není symbol).
6. Seznam `((define a 10) 20)` se vyhodnocuje dle bodu (C), takže jako první se vyhodnotí jeho první prvek, čímž je seznam `(define a 10)`. Vyhodnocením tohoto seznamu vznikne nová vazba v prostředí a vrácena je „nedefinovaná hodnota“. Jelikož se první prvek seznamu `((define a 10) 20)` vyhodnotil na nedefinovanou hodnotu, což není ani procedura, ani speciální forma, je vyhodnocení ukončeno chybou dle bodu (C.e).
7. Po vyhodnocení `(define define 20)` dojde k redefinici symbolu `define`, na který bude navázaná hodnota `20`. Pokud bychom neměli speciální formu `define` navázanou ještě na jiném symbolu, pak bychom ji po provedené této definice nenávratně ztratili.

Nyní ukážeme několik použití `define` bez detailního rozboru vyhodnocování. Budeme předpokládat, že výrazy vyhodnocujeme postupně během jednoho spuštění interpretu.

```
(define a 10)
a                ⟼  10
(* 2 a)          ⟼  20
(sqrt (+ a 5))   ⟼  3.8729833462074
```

Redefinicí hodnoty symbolu `a` bude předešlá vazba „zapomenuta“:

```
(define a 20)
(* 2 a)          ⟼  40
(sqrt (+ a 5))   ⟼  5
```

V následujícím příkladu je nová vazba symbolu `a` vypočtena pomocí jeho aktuální vazby:

```
(define a (+ a 1))
a                ⟼  21
```

Nyní můžeme nadefinovat vazbu symbolu `b` na hodnotu součtu čtyřky s aktuální vazbou `a`. Samotná aktuální vazba symbolu `a` se po provedení následující definice nijak nemění:

```
(define b (+ 4 a))
a                    ⟼  21
b                    ⟼  25
(sqrt b)             ⟼  5
```

Pokud nyní změníme vazbu symbolu `a`, vazba symbolu `b` nebude změněna. Nic na tom nemění ani fakt, že hodnota navázaná na `b` byla v jednom z předchozích kroků stanovena pomocí hodnoty `a`.

```
(define a 666)
a                ⟼  666
b                ⟼  25
```

Následující definicí bude na symbol `plus` navázána primitivní procedura, která je navázána na symbol `+`:

```
(define plus +)
(plus 1 2)       ⟼  3
(plus 1 b)       ⟼  26
(plus 1 2 3)     ⟼  6
```

Nyní bychom mohli předefinovat vazbu symbolu `+` třeba takto:

```
(define + *)
(+ 1 1)          ⟼  1
(+ 2 3)          ⟼  6
```

Primitivní proceduru sčítání, však máme neustále navázanou na symbolu `plus`, takže jsme ji „neztratili“.

Poznámka 1.29. Kdybychom v našem abstraktním interpretu dali vyhodnotit symbol `define`, interpret by nám měl odpovědět jeho vazbou. Tedy měl by napsat něco ve smyslu „speciální forma pro vytváření definic“. Skutečné interprety jazyka Scheme na vyhodnocení symbolu `define` reagují obvykle chybovým hlášením, protože nepředpokládají vyhodnocení symbolu `define` na jiné pozici než na prvním místě seznamu. Například interprety Guile a Elk však reagují stejně jako náš abstraktní interpret.

1.7 Pravdivostní hodnoty a podmíněné výrazy

V předchozích sekcích jsme pracovali pouze s číselnými hodnotami a ukázali jsme princip vazby hodnoty na symbol. V praxi je pochopitelně potřeba pracovat i s jinými daty než s čísly, například s pravdivostními hodnotami. Velmi často je totiž v programech potřeba provádět podmíněné vyhodnocování v závislosti na vyhodnocení jiných elementů. Podmíněným vyhodnocováním se budeme zabývat v této části lekce. Abychom mohli uvažovat podmínky a jejich pravdivost, musíme do našeho interpretu zavést nové elementy jazyka reprezentující *pravdivostní hodnoty – pravda* a *nepravda*.

V počátečním prostředí bude element „pravda“ navázaný na symbol `#t` („*t*“ z anglického *true*) a element „nepravda“ bude navázaný na symbol `#f` („*f*“ z anglického *false*). Samotné elementy „pravda“ a „nepravda“ se budou vyhodnocovat na sebe sama dle bodu (D) definice 1.25, vyhodnocovací proces tedy nemusíme nijak upravovat. Pravdivostní hodnoty budeme zapisovat ve tvaru `#t` a `#f` a podle potřeby je budeme ztotožňovat se samotnými symboly `#t` a `#f`, které budou jejich externí reprezentací. V tomto případě jsme tedy udělali výjimku proti obecné úmluvě 1.14 na straně 24.

Pravdivostní hodnoty „pravda“ a „nepravda“ jsou speciální elementy, které se vyhodnocují na sebe sama. Pro jednoduchost uvažujeme, že pravdivostní hodnoty jsou v počátečním prostředí navázány na symboly `#t` (pravda) a `#f` (nepravda). Specifikace R^{6}RS jazyka Scheme zachází s pravdivostními hodnotami odlišně: `#t` a `#f` nejsou symboly, ale speciální sekvence znaků rozpoznatelné readerem, který je při načítání převádí na pravdivostní hodnoty. My tuto koncepci pro jednoduchost nebudeme přebírat, protože bychom museli rozšiřovat pojem *symbolický výraz*, což nechceme.

Abychom mohli formulovat netriviální podmínky, potřebujeme mít k dispozici procedury, které budou při aplikaci vracet pravdivostní hodnoty. Procedury, výsledkem jejichž aplikace jsou pravdivostní hodnoty, nazýváme *predikáty*. Tento pojem se používá ve standardu jazyka Scheme [R6RS], budeme jej používat i my, i když poněkud nevhodně koliduje s pojmem predikát, který je užívaný v *matematické logice* (v jiném smyslu než zde). Z praktického hlediska jsou ale predikáty obyčejné procedury (tak, jak jsme je představili v předchozích sekcích), pouze vracející pravdivostní hodnoty.

V počátečním prostředí je definováno několik predikátů, kterými můžeme porovnávat číselné hodnoty. Jsou to predikáty `<` (ostře menší), `<=` (menší rovno), `=` (rovno), `>=` (větší rovno), `>` (ostře větší), jejichž zamýšlený význam je zřejmě jasný. Všechny tyto predikáty pracují se dvěma argumenty, viz příklady.

```
#t          ⟼  #t
#f          ⟼  #f
(<= 2 3)    ⟼  #t
(< 2 3)     ⟼  #t
(= 2 3)     ⟼  #f
(= 2 2.0)   ⟼  #t
(= 0.5 1/2) ⟼  #t
(>= 3 3)    ⟼  #t
(> 3 3)     ⟼  #f
```

Příklad 1.30. Kdybychom si vzali například seznam `(+ 2 (< 3 3))`, pak při jeho vyhodnocení dojde k chybě. Všimněte si totiž, že seznam `(< 3 3)` se vyhodnotí na `#f`. Po tomto kroku by měla být aplikována procedura sčítání s číselnou hodnotou `2` a druhým argumentem, kterým by byla pravdivostní hodnota „nepravda“. Aplikace procedury sčítání by tedy končila chybovým hlášením „CHYBA: Druhý argument při aplikaci sčítání není číslo.“.

Nyní zavedeme speciální formu `if`, pomocí niž budeme moci provádět podmíněné vyhodnocování.

Definice 1.31 (speciální forma `if`)**.** Speciální forma `if` se používá se dvěma argumenty ve tvaru:

`(if` ⟨*test*⟩ ⟨*důsledek*⟩ ⟨*náhradník*⟩`)`,

přitom ⟨*náhradník*⟩ je *nepovinný argument* a nemusí být uveden. Při aplikaci speciální formy `if` je nejprve vyhodnocen argument ⟨*test*⟩. Pokud je hodnota vzniklá jeho vyhodnocením různá od elementu „nepravda“ (to jest elementu navázaného na `#f`), pak je výsledkem aplikace speciální formy `if` výsledek vyhodnocení argumentu ⟨*důsledek*⟩. V opačném případě, to jest v případě, kdy se ⟨*test*⟩ vyhodnotil na nepravda, rozlišujeme dvě situace. Pokud je přítomen argument ⟨*náhradník*⟩, pak je výsledkem aplikace speciální formy `if` výsledek vyhodnocení argumentu ⟨*náhradník*⟩. Pokud ⟨*náhradník*⟩ není přítomen, pak je výsledkem aplikace speciální formy `if` nedefinovaná hodnota, viz poznámku 1.27 (b). ■

Poznámka 1.32. Všimněte si, že speciální forma `if` zachází při vyhodnocování *testu* s pravdivostními hodnotami ještě o něco obecněji, než jak jsme předeslali. *Náhradník*, což je argument vyhodnocující se v případě, kdy vyhodnocení *testu* skončilo nepravdou, je skutečně vyhodnocen pouze v případě, kdy se *test* vyhodnotí na „nepravda“ (to jest `#f`). V ostatních případech je vždy vyhodnocen *důsledek*. To zahrnuje i případy, kdy výsledkem *testu* není pravdivostní hodnota, ale jakýkoliv jiný element kromě elementu „nepravda“.

Speciální forma `if` slouží k podmíněnému vyhodnocování. Seznamy, ve kterých se používá forma `if` lze přirozeně číst. Například `(if (<= a b) a b)` můžeme číst „pokud je `a` menší nebo rovno `b`, pak vrať `a`, v opačném případě vrať `b`“. Tudíž při vyhodnocení tohoto seznamu obdržíme jako výsledek vždy menší z hodnot navázaných na `a` a `b`. Speciální forma `if` zachází s pravdivostními hodnotami v zobecněné podobě. Jediný element, který je považován za „nepravdu“ je element navázaný na `#f`.

Otázkou je, proč trváme na tom, aby byla speciální forma `if` právě speciální forma. Nabízí se otázka, zdali by nebylo dostačující chápat ji jako proceduru. Odpověď je dvojí. Na jednu stranu bychom ji *zatím*

mohli chápat jako proceduru. S tímto konceptem „`if` jako procedury“ bychom si však nevystačili příliš dlouho. Druhým úhlem pohledu je samotná myšlenka podmíněného vyhodnocení. U něj totiž nejde jen o to, hodnota kterého z výrazů ⟨*důsledek*⟩ a ⟨*náhradník*⟩ je vrácena, ale o to, aby byl vždy *vyhodnocen pouze jeden z nich*. Kdybychom se chtěli v interpretu přesvědčit, že `if` je skutečně speciální forma, mohli bychom zkusit vyhodnotit následující seznam:

```
(if (= 0 0) #t nepouzity-symbol)
```

Pokud by `if` byla procedura, pak by byly vyhodnoceny všechny její argumenty a vyhodnocování by skončilo chybou v bode (B.e), protože symbol `nepouzity-symbol` by neměl vazbu (pokud bychom ji předtím nevytvořili, což nepředpokládáme). V případě `if` jako speciální formy je nejprve vyhodnocena podmínka `(= 0 0)`, která je (vždy) pravdivá. Tím pádem je výsledkem aplikace `if` výsledek vyhodnocení druhého argumentu, což je argument `#t`, který se vyhodnotí na „pravda“. Symbol `nepouzity-symbol` v tomto případě nebude nikdy vyhodnocen.

Nyní uvedeme příklady použití speciální formy `if`.

```
(define a 10)
(define b 13)
(if (> a b) a b)                ⟹  13
(+ 1 (if (> a b) a b))          ⟹  14
(if (<= a b) (+ a b) (- a b))   ⟹  23
(if (<= a b) (- b a) (- a b))   ⟹  3
```

Pokud jsou na symboly `a` a `b` navázána nezáporná čísla, pak se poslední z uvedených podmíněných výrazů vyhodnotí na absolutní hodnotu rozdílu číselných hodnot navázaných na `a` a `b` (rozmyslete si proč). Podmíněné výrazy se někdy k vůli přehlednosti píší tak, že všechny tři argumenty jsou pod sebou:

```
(if (<= a b)
    (- b a)
    (- a b))  ⟹  3
```

Přehlednost zápisu vynikne teprve v případě, kdy máme několik podmínek vnořených. Například:

```
(if (<= a b)
    (if (= a b)
        a
        (- a))
    #f)  ⟹  -10
```

Zkuste si rozebrat, jak by se předchozí výraz vyhodnocoval v případě různých hodnot navázaných na `a` a `b`. Které všechny případy musíme v takové situaci uvažovat?

I když to není příliš praktické, v podmíněných výrazech můžeme uvádět podmínky, které jsou vždy pravdivé (v obecném smyslu) nebo vždy nepravdivé. Viz následující ukázku.

```
(if #t 10 20)   ⟹  10
(if 1 2 3)      ⟹  2
(if #f 10 20)   ⟹  20
(if #f 10)          nedefinovaná hodnota
```

Doteď jsme se při vyhodnocování seznamů setkávali prakticky výhradně se seznamy, které měly na prvním místě symbol, jenž se vyhodnotil na primitivní proceduru nebo na speciální formu. V následujících ukázkách demonstrujeme, že tomu tak obecně být nemusí a přitom vyhodnocovací proces nekončí chybou (jak tomu bylo ve všech předcházejících případech, kdy první prvek seznamu nebyl symbol).

```
(if #t + -)           ⟹  procedura „sčítání čísel“
((if #t + -) 10 20)   ⟹  30
((if #f + -) 10 20)   ⟹  -10
```

V obou posledních případech je nejprve vyhodnocen první prvek celého seznamu, tím je opět seznam, který způsobí aplikaci speciální formy `if`. Po testu podmínky, která v prvním případě dopadne pozitivně

a v druhém negativně, je vyhodnocen symbol `+` nebo symbol `-`. Jejich vyhodnocením vznikají primitivní procedury. V tomto okamžiku se tedy první prvek celého seznamu vyhodnotí na proceduru. Dále jsou v nespecifikovaném pořadí vyhodnoceny ostatní argumenty (čísla `10` a `20`) a procedura získaná vyhodnocením prvního výrazu je s nimi aplikována. To vede buď na jejich součet nebo na jejich rozdíl.

Jako další příklad tohoto fenoménu si můžeme ukázat seznam, jehož vyhodnocením získáme absolutní hodnotu čísla navázaného na symbol `a`. Přímočaře problém vyřešíme třeba takto:

```
(if (< a 0) (- a) a)
```

Na druhou stranu bychom ale mohli použít následující kód:

```
((if (< a 0) - +) a)
```

V tomto případě se první prvek seznamu opět vyhodnotí na proceduru sčítání nebo odčítání v závislosti na tom, zdali byla na `a` navázaná hodnota menší nebo rovna nule. Procedury sčítání nebo odčítání jsou pak aplikovány s jediným argumentem jímž je hodnota vázaná na `a`. Výsledná hodnota je tedy buď tatáž jako hodnota navázaná na `a` nebo její obrácená hodnota.

Shrnutí

V této úvodní lekci jsme ukázali, co je program a jaký je jeho vztah k výpočetnímu procesu. Program je pasivní entita, lze si jej představit jako soubor uložený na disku. Výpočetní procesy jsou generované programy během jejich zpracování počítačem. Výpočetní procesy manipulují s daty, mění vstupy na výstupy a mají vedlejší efekty. Programovací jazyky dělíme na nižší (kódy stroje, assemblery a autokódy) a vyšší. Vyšší programovací jazyky umožňují větší komfort při programování díky možnostem abstrakce, které dávají uživatelům (programátorům). Programy napsané ve vyšších programovacích jazycích jsou buď překládané (kompilované), nebo interpretované. U programovacích jazyků a programů rozeznáváme jejich syntaxi (tvar) a sémantiku (význam), které nelze zaměňovat. Programování v daném jazyku vždy podléhá programovacímu stylu nebo více stylům, kterým říkáme paradigmata programování. Mezi základní paradigmata patří: funkcionální, procedurální, objektové, paralelní a logické paradigma. Syntax a interpretaci v případě konkrétního jazyka jsme demonstrovali na jazyku Scheme. Programy v jazyku Scheme jsme definovali jako sekvence symbolických výrazů, které dělíme na čísla, symboly a seznamy. Dále jsme ukázali model interpretu jazyka Scheme a popsali vyhodnocování. Zavedli jsme pojem element jazyka. Základními elementy pro nás byly primitivní procedury zastupující operace a interní formy symbolických výrazů. Na elementy jazyka se lze dívat jako na hodnoty, se kterými je možné provádět výpočty. Popsali jsme aplikaci primitivních procedur, prostředí a vazby symbolů v počátečním prostředí. Dále jsme poukázali na nutnost vybavit náš interpret novým typem elementů – speciálními formami kvůli možnosti definovat nové vazby symbolů. Nakonec jsme se zabývali podmíněným vyhodnocováním a představili jsme další typy elementů jazyka – pravdivostní hodnoty a nedefinovanou hodnotu.

Pojmy k zapamatování

- program, výpočetní proces,
- nižší programovací jazyk, kód stroje, autokód, assembler, bajtkód,
- vyšší programovací jazyk, překladač, interpret,
- primitivní výrazy, prostředky kombinace, prostředky abstrakce,
- syntax programu, sémantika programu,
- operace, operandy,
- infixová/prefixová/postfixová notace, polská (reverzní) bezzávorková notace
- syntaktická chyba, sémantická chyba,
- paradigmata programování: funkcionální, procedurální, objektové, paralelní, logické
- jazyk Scheme, symbolické výrazy, S-výrazy,
- čísla, symboly, seznamy, program,

- vyhodnocovací proces, abstraktní interpret Scheme,
- primitivní procedury, elementy jazyka
- interní reprezentace symbolických výrazů,
- externí reprezentace elementů,
- reader, printer, evaluator, cyklus REPL,
- aplikace primitivní procedury, argumenty,
- prostředí, počáteční prostředí, vazba symbolu, aktuální vazba,
- speciální formy, definice vazeb, podmíněné vyhodnocování,
- pravdivostní hodnoty, predikáty, nedefinovaná hodnota.

Nově představené prvky jazyka Scheme

- speciální formy: `define` a `if`,
- aritmetické procedury: `+`, `-`, `*`, `/`, `quotient`, `modulo`, `sqrt`,
- procedury pro konverzi číselné reprezentace: `exact->inexact`, `inexact->exact`, `rationalize`,
- pravdivostní hodnoty: `#t`, `#f`,
- predikáty: `<`, `<=`, `=`, `>=`, `>`.

Kontrolní otázky

1. *Jaký je rozdíl mezi syntaxí a sémantikou programu?*
2. *Jak v jazyku Scheme zapisujeme seznamy?*
3. *Z čeho se skládají programy v jazyku Scheme?*
4. *Jak probíhá vyhodnocení symbolických výrazů?*
5. *Co máme na mysli pod pojmem aplikace procedur?*
6. *Jaký je rozdíl mezi procedurami a speciálními formami?*
7. *Proč nemůže být* `define` *procedura?*
8. *Co je aktuální vazba symbolu?*
9. *Jaké znáte typy chyb a jaký je mezi nimi rozdíl?*
10. *Co jsou to predikáty?*
11. *Z jakých částí se skládá cyklus REPL?*
12. *Jaké má výhody prefixová notace výrazů?*

Cvičení

1. Napište, jak interpret Scheme vyhodnotí následující symbolické výrazy:
```
(/ 2 3)          (2 + 3)    (+ -)           (1 2 3 4)      (modulo 10 3)
(+ (* 2 3) 4)    (+1 2)     (+ (*))         ((+ 1 2))      (rationalize 2 1/10)
(- (- 2))        (+)        (* (+) 2 (+))   (quotient 5)    (sqrt (* 1/2 0.5))
```
2. Rozeberte vyhodnocení následujících výrazů krok po kroku. Pokud při vyhodnocení dojde k chybě, zdůvodněte proč se tak stalo.
   (a) `(+ (* 2 3) (/ 2 8))`
   (b) `(if (= 0 (+ (- (+ 10 20) 30))) 7 8)`
   (c) `(* 2 sqrt (4))`
3. Napište, jak se vyhodnotí následující výrazy a zdůvodněte proč.
```
(define a 10)
(define b (+ a 1))
(define a 20)
b  ⟹  ???
```
4. Napište, jak dopadne vyhodnocení následujících výrazů a zdůvodněte proč.

```
(define define 1)
(define a 2)
a
```

5. Napište výsledky vyhodnocení následujících výrazů.
```
(if + 1 2)
(sqrt (if + 4 16))
(+ ((if - - +) 20))
(if (if (> 1 0) #f #t) 10 20)
```
6. Následující výraz
```
(define nedef ···)
```
doplňte tak, aby po jeho vyhodnocení byla na symbol `nedef` navázána nedefinovaná hodnota.

Úkoly k textu

1. Uvažujme následující výraz:
```
(if (if a
        (> a b)
        -666)
    (if b
        (- b)
        #t)
    10)
```
Proveďte analýzu předchozího výrazu a napište jak dopadne jeho vyhodnocení v závislosti na různých vazbách symbolů `a` a `b`. Soustřeďte se přitom na případy, kdy vyhodnocení končí úspěchem (tedy nikoliv chybou). Kolik musíme při našich úvahách rozlišit případů?

2. Předpokládejte, že máme na symbolech `a`, `b` a `c` navázány tři číselné hodnoty. Napište symbolický výraz, jehož vyhodnocením je vrácena *největší hodnota z* `a`, `b` *a* `c`. U výrazu důkladně otestujte jeho správnost pro vzájemně různé hodnoty navázané na symboly `a`, `b` a `c`. Je pochopitelné, že na dané symboly můžeme navázat nekonečně mnoho vzájemně odlišných číselných hodnot, ale z hlediska testování správnost výrazu stačí probrat jen konečně mnoho testovacích možností. Kolik jich bude? Zdůvodněte správnost vaší úvahy.

3. Prohlédněte si následující matematické výrazy a rozmyslete si, jak byste je vyjádřili pomocí symbolických výrazů v jazyku Scheme.

$$\frac{x+\sqrt{9}\cdot\dfrac{\log(z+5)}{\log(y)}}{x+y+z+4}, \qquad y\cdot\sqrt{\sin\left(\frac{4\cdot\operatorname{atan}(x)}{y}\right)}, \qquad \left(e^{y\cdot\log(z)}\right)^2\cdot\frac{x}{z\cdot\left(y+\frac{4+5^2-x}{4}\right)}.$$

Proveďte přepis výrazů a pomocí interpretu jazyka Scheme zjistěte, jaké budou jejich hodnoty v případě, že za x, y a z dosadíme číselné hodnoty 1, 2 a 3. Při přepisu použijte procedury navázané na symboly `log`, `exp`, `sqrt` a `atan`.

Řešení ke cvičením

1. Řešení po řadách: `2/3`, chyba, chyba, chyba, `1`; `10`, chyba, `1`, chyba, `2`; `2`, `1`, `2`, chyba, `0.5`.
2. Výsledky: (a) `25/4`; (b) `7`; (c) chyba (při vyhodnocování `(4)`). Podrobný rozbor udělejte dle definice 1.25.
3. Symbol `b` se na konci vyhodnotí na `11`. Protože vazba `b` byla definována jako hodnota první vazby symbolu `a` (což bylo `10`) zvětšená o jedna.

4. Vyhodnocení končí chybou při pokusu vyhodnotit druhý výraz. Při vyhodnocení prvního výrazu se totiž na symbol `define` naváže jednička, takže při vyhodnocení druhého výrazu se první prvek seznamu nevyhodnotí ani na proceduru ani na speciální formu.
5. Výsledky vyhodnocení: `1`, `2`, `-20`, `20`.
6. Například: `(define nedef (if #f #f))`

Lekce 2: Vytváření abstrakcí pomocí procedur

Obsah lekce: V této lekci se seznámíme s uživatelsky definovatelnými procedurami a s vytvářením abstrakcí pomocí nich. Nejprve řekneme motivaci pro zavedení takových procedur a ukážeme několik příkladů. Zavedeme λ-výrazy jako výrazy jejichž vyhodnocováním procedury vznikají. Dále rozšíříme současný model prostředí. Uvedeme, jak lze chápat vyhodnocování elementů relativně vzhledem k prostředí. Dále ukážeme, jak jsou reprezentovány uživatelsky definovatelné procedury, jak vznikají a jak probíhá jejich aplikace. Ukážeme několik typických příkladů procedur a naši pozornost zaměříme na procedury vyšších řádů. Posléze poukážeme na vztah procedur a zobrazení (matematických funkcí). Nakonec ukážeme dva typy rozsahů platnoti symbolů a představíme některé nové speciální formy.

Klíčová slova: aktuální prostředí, currying, disjunkce, dynamicky nadřazené prostředí, dynamický rozsah platnosti, elementy prvního řádu, formální argumenty, globální prostředí, identita, konjunkce, konstantní procedura, λ-výraz, lexikálně nadřazené prostředí, lexikální (statický) rozsah platnosti, lokální prostředí, negace, parametry, procedura vyššího řádu, projekce, předek prostředí, tělo procedury, uživatelsky definovatelná procedura, volné symboly, vázané symboly.

2.1 Uživatelsky definovatelné procedury a λ-výrazy

V sekci 1.6 jsem popsali vytváření abstrakcí pomocí *pojmenování hodnot*. Stručně řečeno šlo o to, že místo používání konkrétních hodnot v programu jsme hodnotu pojmenovali pomocí *symbolu* zastupujícího „jméno hodnoty“ a v programu jsme dále používali tento symbol. Nyní pokročíme o krok dále a ukážeme daleko významnější způsob vytváření abstrakcí, který je založený na *vytváření nových procedur.*

V minulé lekci jsme představili celou řadu *primitivních procedur,* to jest procedur, které byly na počátku vyhodnocování (po spuštění interpretu) navázány na některé symboly (jako třeba `+`, `modulo` a podobně) v počátečním prostředí. Při vytváření programů se často dostáváme do situace, že potřebujeme provádět podobné „operace“ (sekvence aplikací primitivních procedur), které se v programu projevují „podobnými kusy kódu“. V programu tedy dochází k jakési *redundanci* (*nadbytečnosti*) *kódu.* Tato redundance by mohla být potenciálním zdrojem chyb, kdybychom byli nuceni tutéž sekvenci kódu měnit na všech místech programu (třeba vlivem přidání nového parametru úlohy). Snadno bychom na některou část kódu mohli zapomenout a chyba by byla na světě. Nabízí se tedy vytvořit *novou proceduru,* potom ji *pojmenovat* (což již umíme), a dále s ní pracovat, to jest aplikovat ji, jako by se jednalo o primitivní proceduru. Jedná se nám tedy o problém *uživatelského vytváření procedur* (pod pojmem „uživatel“ budeme mít na mysli uživatele jazyku Scheme, tedy programátora). Nový typ procedur, o kterém budeme hovořit v této lekci, budeme tedy nazývat *uživatelsky definovatelné procedury.* Stejně jako u primitivních procedur budeme i uživatelsky definovatelné procedury považovat za *elementy jazyka* Scheme.

> Primitivní procedury i uživatelsky definovatelné procedury budeme dále označovat jedním souhrnným názvem *procedury.* Procedura je tedy obecné označení pro element jazyka zahrnující v sobě jednak procedury, které jsou k dispozici na začátku práce s interpretem (to jest primitivní procedury) a na druhé straně procedury, které lze *dynamicky vytvářet během vyhodnocování programů* a dále je v nich používat (to jest uživatelsky definovatelné procedury).

Uživatelsky definovatelné procedury nyní popíšeme od jejich syntaxe po jejich sémantiku. V této sekci uvedeme pouze základy, motivační příklady a to, jak se na procedury dívat z pohledu programátora. V dalších sekcích této lekce se budeme zabývat tím, jak procedury vznikají a jak probíhá jejich aplikace z pohledu abstraktního interpretu jazyka Scheme. Nejprve uvedeme ilustrativní příklad.

Představme si situaci, že v programu často počítáme hodnoty druhých mocnin nějakých čísel. Přirozeně bychom mohli výpočet druhé mocniny zajistit výrazy tvaru `(*` ⟨*výraz*⟩ ⟨*výraz*⟩`)`, které bychom uváděli na každém místě v programu, kde bychom druhou mocninu chtěli počítat. Toto řešení má mnoho nevýhod. Jednak je to již dříve uvedená redundance kódu, jednak ⟨*výraz*⟩ bude při vyhodnocování celého výrazu

`(* ⟨výraz⟩ ⟨výraz⟩)` vyhodnocován dvakrát. Nabízelo by se tedy „zhmotnit“ novou proceduru jednoho argumentu „vypočti druhou mocninu“ a potom ji pomocí `define` navázat na dosud nepoužitý symbol, třeba `na2` (jako „na druhou“). Pokud by se to podařilo, mohli bychom tuto novu proceduru aplikovat jako by byla primitivní, to jest ve tvaru `(na2 ⟨výraz⟩)`.

Nové procedury budeme v interpretu vytvářet *vyhodnocováním λ-výrazů.* Například proceduru „umocni danou hodnotu na druhou“ bychom vytvořili vyhodnocením následujícího λ-výrazu:

`(lambda (x) (* x x))` ⟹ „procedura, která pro dané `x` vrací násobek `x` s `x`“

V předchozím výrazu je `lambda` symbol, na který je v počátečním prostředí navázána speciální forma vytvářející procedury, jak si podrobně rozebereme dále (zatím se na tuto speciální formu můžeme dívat jako na černou skříňku). Za tímto symbolem následuje jednoprvkový seznam `(x)`, jehož jediným prvkem je symbol `x`, kterým *formálně označujeme hodnotu předávanou proceduře.* Posledním prvkem předchozího λ-výrazu je seznam `(* x x)`, což je *tělo procedury* udávající, co se při aplikaci procedury s hodnotou navázanou na symbol `x` bude dít. V našem případě bude hodnota vynásobena sama se sebou a výsledek této aplikace bude i výsledkem aplikace naší nové procedury. Proceduru bychom mohli použít například takto:

```
((lambda (x) (* x x)) 8)  ⟹  64
```

Předchozí použití je z hlediska vyhodnocovacího proceduru zcela v pořádku. Uvědomte si, že symbolický výraz `(lambda (x) (* x x))`, což je první prvek celého seznamu, se (jak jsme uvedli) vyhodnotí na proceduru. V dalším kroku se číslo `8` vyhodnotí na sebe sama a poté dojde k aplikaci nově vzniklé procedury s hodnotou `8`. Všimněte si, že v tomto případě se jednalo vlastně o jakousi „jednorázovou aplikaci“ procedury, protože nová procedura vznikla, byla jednou aplikována (s argumentem `8`) a další aplikaci téže procedury již nemůžeme provést, protože jsme ji „ztratili“. Pro vysvětlenou, kdybychom dále napsali

```
((lambda (x) (* x x)) 10)  ⟹  100,
```

pak by byla opět (vyhodnocením λ-výrazu) vytvořena nová procedura „umocni na druhou,“ pouze jednou aplikována a poté opět „ztracena“. Abychom tutéž proceduru (to jest proceduru vzniklou vyhodnocením *jediného λ-výrazu* na konkrétním místě v programu) mohli vícekrát aplikovat, musíme jí bezprostředně po jejím vytvoření pojmenovat (v dalších sekcích uvidíme, že to lze udělat i bez pojmenování), například:

```
(define na2
  (lambda (x) (* x x)))
```

Při vyhodnocování předchozího výrazu je aktivována speciální forma `define`, která na symbol `na2` naváže hodnotu vzniklou vyhodnocením druhého argumentu. Tím je v našem případě λ-výraz, který se vyhodnotí na proceduru, takže po vyhodnocení předchozího výrazu bude skutečně na `na2` navázána nově vytvořená procedura „umocni hodnotu na druhou“. Nyní můžeme proceduru používat, jako by byla primitivní:

```
(na2 2)          ⟹  4
(+ 1 (na2 4))    ⟹  17
(na2 (na2 8))    ⟹  4096
```

Sice jsme ještě přesně nepopsali vznik ani detaily aplikace uživatelsky definovatelných procedur, ale základní princip jejich vytváření a používání by již měl být každému zřejmý.

V čem tedy spočívá vytváření abstrakcí pomocí procedur? Spočívá v tom, že *konkrétní části kódu* nahrazujeme *aplikací abstraktnějších procedur.* Například `(* 3 3 3)` a `(* 5 5 5)` bychom mohli nahradit výrazy `(na3 3)` a `(na3 5)` způsobujícími aplikaci procedury „umocni na třetí“ navázané na symbol `na3`. Při úpravě kódu se pak můžeme soustředit pouze na samotnou proceduru a nemusíme se zabývat, kde všude v programu je používána.

Nyní přesně popíšeme, jak vypadají λ-výrazy.

Definice 2.1 (λ-výraz)**.** Každý seznam ve tvaru

`(lambda (`⟨$param_1$⟩ ⟨$param_2$⟩ ⋯ ⟨$param_n$⟩`)` ⟨tělo⟩`)`,

kde n je nezáporné číslo, $\langle param_1 \rangle, \langle param_2 \rangle, \ldots, \langle param_n \rangle$ jsou vzájemně různé symboly a $\langle tělo \rangle$ je libovolný symbolický výraz, se nazývá λ-*výraz* (*lambda výraz*). Symboly $\langle param_1 \rangle, \ldots, \langle param_n \rangle$ se nazývají *formální argumenty* (někdy též *parametry*). Číslo n nazýváme *počet formálních argumentů* (*parametrů*). ■

Poznámka 2.2. (a) Zřejmě každý λ-výraz je symbolický výraz, protože je to seznam (ve speciálně vyžadovaném tvaru). Příklady λ-výrazů jsou třeba: `(lambda (x) (* 2 x))`, `(lambda (x y) (+ (* 2 x) y))`, `(lambda (x y dalsi) (+ x dalsi))` a tak podobně. Všimněte si, že definice 2.1 připouští i nulový počet formálních argumentů. Takže například `(lambda () 10)` je rovněž λ-výraz. Respektive jedná se o λ-výraz (tedy o symbolický výraz) za předpokladu, že v definici 1.6 na straně 19 připustíme i „prázdné seznamy“, to jest seznamy `()`, což budeme od tohoto okamžiku tiše předpokládat.

(b) Účelem formálních argumentů je „pojmenovat hodnoty“ se kterými bude procedura aplikována. Proceduru můžeme aplikovat s různými argumenty, proto je potřeba argumenty při definici procedury zastoupit symboly, aby byly pokryty „všechny možnosti aplikace procedury“. Tělo procedury představuje vlastní *předpis procedury* – neformálně řečeno, tělo vyjadřuje „co bude procedura s danými argumenty provádět“.

(c) Pojem „λ-výraz“ je přebrán z formálního kalkulu zvaného λ-kalkul, který vyvinul v 30. letech 20. století americký matematik Alonzo Church [Ch36, Ch41].

Zopakujme, že uvedením λ-výrazu v programu je z pohledu vyhodnocovacího procesu (tak jak jsme jej uvedli v definici 1.25 na straně 1.25) provedena aplikace speciální formy navázané na symbol `lambda`. Samotná procedura je vytvořená speciální formou `lambda` (záhy popíšeme jak). Je ovšem zřejmé, že element navázaný na `lambda` musí být skutečně speciální forma, *nikoliv* procedura. Kdyby totiž na `lambda` byla navázána procedura, pak by vyhodnocení následujícího výrazu

```
(lambda (x) (* x x))
```

končilo chybou v kroku (B.e), protože symbol `x` by nemá vazbu (rozeberte si podrobně sami).

> Je vhodné dobře si uvědomovat, jaký je rozdíl mezi λ-výrazy a procedurami vzniklými jejich vyhodnocováním. Předně, λ-výrazy jsou seznamy, tedy symbolické výrazy, což nejsou procedury. λ-výrazy uvedené v programech bychom měli chápat jako *předpisy pro vznik procedur*, nelze je ale ztotožňovat s procedurami samotnými.

Nyní neformálně vysvětlíme princip aplikace uživatelských procedur, který dále zpřesníme v dalších sekcích. Nejprve řekněme, že z pohledu symbolů vyskytujících se v těle konkrétního λ-výrazu je můžeme rozdělit na dvě skupiny. První skupinou jsou symboly, které jsou formálními argumenty daného λ-výrazu. Takovým symbolům budeme říkat *vázané symboly*. Druhou skupinou jsou symboly, které se nacházejí v těle λ-výrazu, ale nejedná se o formální argumenty. Takovým symbolům budeme říkat *volné symboly*. Demonstrujme si blíže oba pojmy na následujícím výrazu:

```
(lambda (x y nepouzity) (* (+ 1 x) y))
```

V těle tohoto výrazu se nacházejí celkem čtyři symboly: `+`, `*`, `x` a `y`. Symboly `x` a `y` jsou *vázané*, protože jsou to formální argumenty, to jest jsou to prvky seznamu `(x y z)`. Naproti tomu symboly `+` a `*` jsou *volné*, protože se v seznamu formálních argumentů nevyskytují. Formální argument `nepouzity` se nevyskytuje v těle, takže jej neuvažujeme.

Pomocí volných a vázaných symbolů v λ-výrazech můžeme nyní zjednodušeně popsat aplikaci procedur.

Definice 2.3 (zjednodušený model aplikace uživatelsky definovatelných procedur)**.** Při aplikaci procedury vzniklé vyhodnocením λ-výrazu `(lambda (`$\langle param_1 \rangle$ $\langle param_2 \rangle \cdots \langle param_n \rangle$`)` $\langle tělo \rangle$`)` dojde k vytvoření *lokálního prostředí*, ve kterém jsou na symboly formálních argumentů $\langle param_1 \rangle, \ldots, \langle param_n \rangle$ navázané hodnoty, se kterými byla procedura aplikována. Při aplikaci musí být proceduře předán stejný počet hodnot jako je počet formálních argumentů procedury. V takto vytvořeném lokálním prostředí je *vyhodnoceno* $\langle tělo \rangle$ *procedury*. Při vyhodnocování těla procedury se *vazby vázaných symbolů* hledají v *lokálním prostředí* a vazby *volných symbolů* se hledají *v počátečním prostředí*. Výsledkem aplikace procedury je hodnota vzniklá jako výsledek vyhodnocení jejího těla. ■

Zdůrazněme ještě jednou, že proces aplikace popsaný v definici 2.3 je pouze zjednodušeným modelem. V dalších sekcích uvidíme, že s takto probíhající aplikací bychom si nevystačili. Pro zdůvodnění výsledků aplikace uživatelsky definovatelných procedur nám to v této sekci zatím postačí.

Poznámka 2.4. (a) Definice 2.3 říká, že hodnotou aplikace procedury s danými argumenty je hodnota vyhodnocení jejího těla, za předpokladu, že formální argumenty budou navázány na skutečné hodnoty použité při aplikaci. Otázkou je, proč potřebujeme rozlišovat dvě prostředí – lokální a globální. Je to z toho důvodu, že nechceme směšovat formální argumenty procedur se symboly v počátečním prostředí, protože mají odlišnou roli. Vázané symboly v těle procedur zastupují argumenty předané proceduře. Volné symboly v těle procedur zastupují elementy (čísla, procedury, speciální formy, . . .), které jsou definované *mimo lokální prostředí* (v našem zjednodušeném modelu je to prostředí počáteční).

(b) Jelikož jsme zjistili, že abstraktní interpret jazyka Scheme pouze s jedním (počátečním) prostředím je dále neudržitelný, budeme se v další sekci zabývat tím, jak prostředí vypadají a následně upravíme vyhodnocovací proces.

Ve zbytku sekce ukážeme další příklady uživatelsky definovatelných procedur. Za předpokladu, že na symbolu `na2` máme navázanou proceduru pro výpočet druhé mocniny (viz předchozí text), pak můžeme dále definovat odvozené procedury pro výpočet dalších mocnin:

```
(define na3 (lambda (x) (* x (na2 x))))
(define na4 (lambda (x) (na2 (na2 x))))
(define na5 (lambda (x) (* (na2 x) (na3 x))))
(define na6 (lambda (x) (na2 (na3 x))))
(define na7 (lambda (x) (* (na3 x) (na4 x))))
(define na8 (lambda (x) (na2 (na4 x))))
  ⋮
```

Pomocí procedur pro výpočet druhé mocniny a druhé odmocniny (primitivní procedura navázaná na `sqrt`) můžeme napsat proceduru pro výpočet velikosti přepony v pravoúhlém trojúhelníku. Bude se jedna o proceduru dvou argumentů, jimiž jsou velikosti přepon a která bude provádět výpočet podle známého vzorce $c = \sqrt{a^2 + b^2}$:

```
(define prepona
  (lambda (odvesna-a odvesna-b)
    (sqrt (+ (na2 odvesna-a) (na2 odvesna-b)))))
```

Příklad použití procedury:

```
(prepona 3 4)          ⟹  5
(+ 1 (prepona 3 4))    ⟹  6
(prepona 30 40)        ⟹  50
```

Dále bychom mohli vytvořit další proceduru používající právě vytvořenou proceduru na výpočet velikosti přepony. Třeba následující procedura počítá velikost přeponu pravoúhlých trojúhelníků, jejichž delší odvěsna je stejně dlouhá jako dvojnásobek kratší odvěsny. Je zřejmé, že této proceduře bude stačit předávat pouze jediný argument – délku kratší odvěsny, protože delší odvěsnu si můžeme vypočítat:

```
(define dalsi-procedura
  (lambda (x)
    (prepona x (* 2 x))))
```

Doposud jsme ukazovali procedury, které ve svém těle provedly pouze jednoduchý aritmetický výpočet. Procedury mívají obvykle daleko složitější těla, ve kterých se často vyskytují podmíněné výrazy. Nyní si ukážeme, jak nadefinovat například proceduru pro výpočet absolutní hodnoty reálného čísla. Z matematiky víme, že absolutní hodnota $|x|$ čísla x je číslo dané následujícím vztahem:

$$|x| = \begin{cases} x & \text{pokud } x \geq 0, \\ -x & \text{pokud } x < 0. \end{cases}$$

Tedy absolutní hodnota čísla je jeho „vzdálenost od nuly na souřadné ose“. Podíváme-li se na předchozí

matematický zápis $|x|$, můžeme vidět, že se vlastně jedná o vyjádření hodnoty v závislosti na vztahu x k nule. Tento matematický výraz můžeme zcela přímočaře přepsat pomocí speciální formy `if` následovně:

```
(define abs
  (lambda (x)
    (if (>= x 0)
        x
        (- x))))
```

Jak se můžete sami přesvědčit, procedura při aplikaci skutečně vrací absolutní hodnotu daného argumentu.

Nyní si uvedeme několik procedur, které mají svá *ustálená slovní označení*, která se používají i v matematice. První z nich je procedura zvaná *identita*. Identita je procedura jednoho argumentu, který pro daný argument vrací právě jeho hodnotu. Identitu bychom tedy vytvořili vyhodnocením následujícího λ-výrazu:

`(lambda (x) x)` $\longmapsto$ identita: „procedura která pro dané `x` vrací `x`“

Při použití se tedy identita chová následovně:

```
((lambda (x) x) 20)  ⟼  20
```

```
(define id
  (lambda (x) x))
```

```
(id (* 2 20))        ⟼  40
(id (+ 1 (id 20)))   ⟼  21
(id #f)              ⟼  #f
((id -) 10 20)       ⟼  -10
```

Všimněte si, jak se vyhodnotil poslední výraz. První prvek seznamu `((id -) 10 20)`, to jest `(id -)` se vyhodnotil na proceduru „odčítání“, protože vyhodnocení tohoto seznamu způsobilo aplikaci identity na argument jimž byla právě procedura „odčítání“ navázaná na symbol `-`.

Další procedury s ustáleným názvem jsou *projekce*. Pro každých n formálních argumentů můžeme uvažovat právě n procedur $\pi_1, \ldots, \pi_n$, kde každá π_i je procedura n argumentů vracející hodnotu svého i-tého argumentu. Proceduře π_i říkáme *i-tá projekce*. Uvažme pro ilustraci situaci, kdy $n = 3$ (tři argumenty). Pak budou projekce vypadat následovně:

```
(lambda (x y z) x)   ⟼  první projekce: „procedura vracející hodnotu svého prvního argumentu“
(lambda (x y z) y)   ⟼  druhá projekce: „procedura vracející hodnotu svého druhého argumentu“
(lambda (x y z) z)   ⟼  třetí projekce: „procedura vracející hodnotu svého třetího argumentu“
```

Přísně vzato, procedura identity je vlastně první projekcí (jediného argumentu).

Použití projekcí můžeme vidět na dalších příkladech:

```
(define 1-of-3 (lambda (x y z) x))
(define 2-of-3 (lambda (x y z) y))
(define 3-of-3 (lambda (x y z) z))
```

```
(1-of-3 10 20 30)                          ⟼  10
(2-of-3 10 (+ 1 (3-of-3 2 4 6)) 20)        ⟼  7
((3-of-3 #f - +) 13)                       ⟼  13
((2-of-3 1-of-3 2-of-3 3-of-3) 10 20 30)   ⟼  20
```

K tomu abychom si přesně uvědomili jak se vyhodnotily poslední dva výrazy se potřebujeme zamyslet nad vyhodnocením jejich prvního prvku (proveďte podrobně sami).

Dalšími procedurami jsou *konstantní procedury*, která ignorují své argumenty a vracejí nějakou konstantní hodnotu (vždy stejný element). Například:

```
(lambda (x) 10)  ⟼  konstantní procedura: „vrať číslo 10“
(lambda (x) #f)  ⟼  konstantní procedura: „vrať pravdivostní hodnotu #f“
```

```
(lambda (x) +)    ⟹    konstantní procedura: „vrať hodnotu navázanou na symbol +“
  ⋮
```

Viz následující příklady definice a aplikací konstantních procedur:

```
(define c
  (lambda (x) 10))

(+ 1 (c 20))                    ⟹   11
(((lambda (x) -) 10) 20 30)     ⟹   -10
```

Důležité je neplést si konstantní proceduru jednoho argumentu s *procedurou bez argumentu*. Například následující dvě definice zavádějí konstantní proceduru jednoho argumentu a analogickou proceduru bez argumentu:

```
(define const-proc (lambda (x) 10))
(define noarg-proc (lambda () 10))
```

Zásadní rozdíl je však v jejich aplikaci:

```
(const-proc 20)   ⟹   10
(noarg-proc)      ⟹   10
```

Kdybychom aplikovali proceduru navázanou na `const-proc` bez argumentu, vedlo by to k chybě. Stejně tak by vedl k chybě pokus o aplikaci procedury `noarg-proc` s jedním (nebo více) argumenty.

2.2 Vyhodnocování elementů v daném prostředí

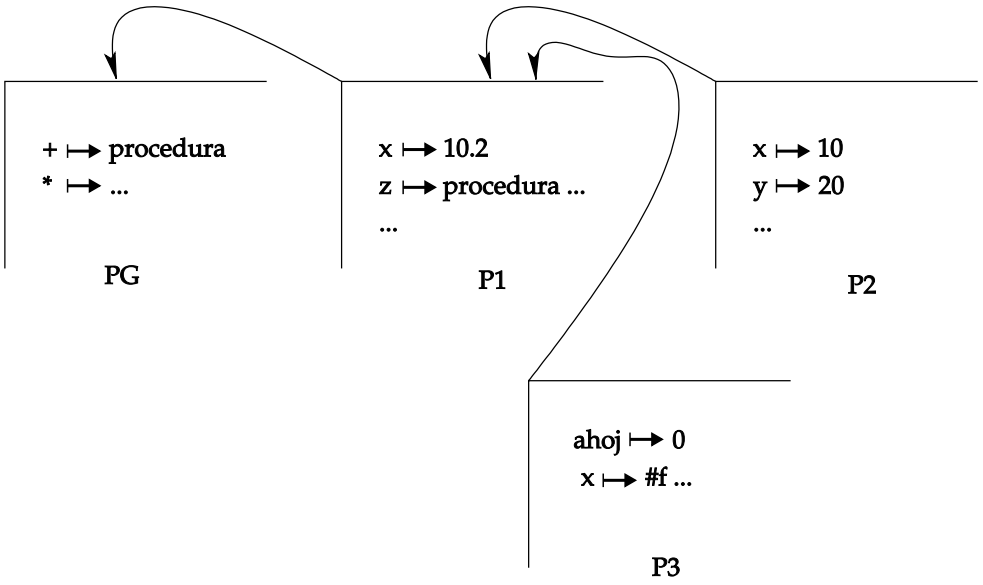
V předchozí sekci jsme nastínili problém potřeby pracovat s více prostředími. Jedno prostředí je počáteční prostředí, ve kterém jsou definovány počáteční vazby symbolů. Dalšími prostředími jsou lokální prostředí procedur, ve kterých jsou vázány na formální argumenty hodnoty se kterými byly procedury aplikovány. Situace je ve skutečnosti ještě o něco komplikovanější. Budeme potřebovat pracovat obecně s několika prostředími současně. Tato prostředí mezi sebou navíc budou *hierarchicky provázaná.* V této sekci se proto zaměříme na samotná prostředí a následné zobecnění vyhodnocovacího procesu. Pracujeme-li totiž s více jak jedním prostředím, je potřeba vztáhnout vyhodnocování elementů (část „Eval“ cyklu REPL, viz definici 1.25) *relativně vzhledem k prostředí.* Budeme se tedy zabývat zavedením *vyhodnocení elementu E v prostředí $\mathcal{P}$.*

Nejprve uvedeme, co budeme mít od tohoto okamžiku na mysli pod pojmem „prostředí“.

Definice 2.5 (prostředí)**.** *Prostředí $\mathcal{P}$* je tabulka vazeb mezi symboly a elementy, přitom každé uvažované prostředí, kromě jediného, má navíc ukazatel na svého *předka,* což je opět prostředí. Prostředí, které nemá svého předka, budeme označovat $\mathcal{P}_G$ a budeme jej nazývat *globální* (*počáteční*) *prostředí.* Fakt, že prostředí $\mathcal{P}_1$ je předkem prostředí $\mathcal{P}_2$, budeme značit $\mathcal{P}_1 \prec \mathcal{P}_2$. Pokud $\mathcal{P}_1 \prec \mathcal{P}_2$, pak také říkáme, že prostředí $\mathcal{P}_1$ *je nadřazeno prostředí* $\mathcal{P}_2$. Pokud je na symbol s v prostředí $\mathcal{P}$ navázán element E, pak řekneme, že *E je aktuální vazba symbolu s v prostředí $\mathcal{P}$* a budeme tento fakt značit $s \mapsto_{\mathcal{P}} E$. Pokud je prostředí $\mathcal{P}$ zřejmé z kontextu, pak budeme místo $s \mapsto_{\mathcal{P}} E$ psát jen $s \mapsto E$. ■

Poznámka 2.6. Prostředí je tedy tabulka zachycující vazby mezi symboly a hodnotami (elementy) jako doposud, k této tabulce navíc ale přibyl ukazatel na předka (na nadřazené prostředí). Pouze *globální prostředí* žádného *předka nemá.* Prostředí a vazby v nich obsažené včetně ukazatelů na předky budeme někdy zobrazovat pomocí diagramů. Příklad takového diagramu je na obrázku 2.1. V tomto obrázku jsou zachyceny čtyři prostředí: $\mathcal{P}_G$ (globální), $\mathcal{P}_1$, $\mathcal{P}_2$ a $\mathcal{P}_3$. Šipky mezi prostředími ukazují na jejich předky. Z prostředí $\mathcal{P}_G$ proto žádná šipka nikam nevede, předkem prostředí $\mathcal{P}_1$ je globální prostředí (tedy $\mathcal{P}_G \prec \mathcal{P}_1$); předkem prostředí $\mathcal{P}_2$ a $\mathcal{P}_3$ je shodně prostředí $\mathcal{P}_1$ (zapisujeme $\mathcal{P}_1 \prec \mathcal{P}_2$ a $\mathcal{P}_1 \prec \mathcal{P}_3$). V samotných prostředích budeme pomocí již uvedeného značení „$s \mapsto E$“ zapisovat obsažené vazby symbolů, budeme přitom znázorňovat jen ty vazby, které jsou pro nás nějakým způsobem zajímavé (například v globálním prostředí nebudeme vypisovat všechny počáteční vazby symbolů, to by bylo nepřehledné).

Obrázek 2.1. Prostředí a jejich hierarchie



Všimněte si, že díky tomu, že každé prostředí (kromě globálního) má svého předka, na množinu všech prostředí (během výpočtu) se lze dívat jako na hierarchickou strukturu (strom). Globální (počáteční) prostředí, je prostředí ve kterém jsou nastaveny počáteční vazby symbolů a ve kterém jsme doposud uvažovali vyhodnocování. Jelikož globální prostředí *nemá předka*, je v hierarchii prostředí „úplně na vrcholu“. Pod ním se v hierarchii nacházejí prostředí jejichž předkem je právě globální prostředí. Na další úrovni hierarchie jsou prostředí jejichž předky jsou prostředí na předchozí úrovni a tak dále. Vznik nových prostředí úzce souvisí s aplikací procedur, jak uvidíme v další sekci.

Nyní uvedeme modifikaci vyhodnocování elementů. Místo vyhodnocení daného elementu E budeme uvažovat vyhodnocení elementu E v *prostředí* $\mathcal{P}$:

Definice 2.7 (vyhodnocení elementu E v prostředí $\mathcal{P}$)**.**
Výsledek vyhodnocení elementu E v prostředí $\mathcal{P}$, značeno $\mathrm{Eval}[E, \mathcal{P}]$, je definován:

(A) Pokud je E *číslo*, pak $\mathrm{Eval}[E, \mathcal{P}] := E$.

(B) Pokud je E *symbol*, mohou nastat tři situace:

(B.1) Pokud $E \mapsto_{\mathcal{P}} F$, pak $\mathrm{Eval}[E, \mathcal{P}] := F$.

(B.2) Pokud E *nemá vazbu v* $\mathcal{P}$ a pokud $\mathcal{P}' \prec \mathcal{P}$, pak $\mathrm{Eval}[E, \mathcal{P}] := \mathrm{Eval}[E, \mathcal{P}']$.

(B.e) Pokud E *nemá vazbu v* $\mathcal{P}$ a pokud $\mathcal{P}$ *je globální prostředí*, pak ukončíme vyhodnocování hlášením „CHYBA: Symbol E nemá vazbu.“.

(C) Pokud je E *neprázdný seznam* tvaru $(E_1\ \ E_2 \cdots E_n)$, pak $F_1 := \mathrm{Eval}[E_1, \mathcal{P}]$. Dále rozlišujeme tři situace:

(C.1) Pokud F_1 je *procedura*, pak se v nespecifikovaném pořadí vyhodnotí $E_2, \ldots, E_n$:

$$\begin{aligned} F_2 &:= \mathrm{Eval}[E_2, \mathcal{P}], \\ F_3 &:= \mathrm{Eval}[E_3, \mathcal{P}], \\ &\vdots \quad\ \vdots \\ F_n &:= \mathrm{Eval}[E_n, \mathcal{P}]. \end{aligned}$$

Potom položíme $\mathrm{Eval}[E, \mathcal{P}] := \mathrm{Apply}[F_1, F_2, \ldots, F_n]$.

(C.2) Pokud F_1 je *speciální forma*, pak $\mathrm{Eval}[E, \mathcal{P}] := \mathrm{Apply}[F_1, E_2, \ldots, E_n]$.

(C.e) Pokud F_1 není *procedura ani speciální forma*, skončíme vyhodnocování hlášením „CHYBA: Nelze provést aplikaci: E se nevyhodnotil na proceduru ani na speciální formu.“.

(D) Ve všech ostatních případech klademe $\mathrm{Eval}[E, \mathcal{P}] := E$. ■

Poznámka 2.8. (a) Všimněte si, že předchozí popis „Eval“ jsme rozšířili v podstatě jen velmi nepatrně. Oproti „Eval“ pracujícímu se speciálními formami, viz definici 1.25 na straně 33, jsme pouze přidali bod (B.2) zajišťující možnost hledat vazby v nadřazených prostředích. U všech ostatních výskytů „Eval“ v definici 2.7 provádíme vyhodnocení v témže prostředí $\mathcal{P}$. Další úpravou, kterou jsme provedli, je vyžadování neprázdného seznamu na počátku bodu (C). To jsme provedli kvůli tomu, že jsme se dohodli na možnost uvažovat prázdné seznamy (kvůli možnosti vyjádřit procedury bez argumentů).

(b) Prozkoumáme-li podrobněji bod (B), zjistíme, že říká následující. Pokud má daný symbol aktuální vazbu v prostředí, ve kterém jej vyhodnocujeme, pak je výsledkem vyhodnocení jeho vazba, viz bod (B.1). Pokud symbol v daném prostředí vazbu nemá, pak bude vazba hledána v jeho *nadřazeném prostředí*, viz bod (B.2). Pokud ani tam nebude nalezena, bude se hledat v dalším nadřazeném prostředí, dokud se v hierarchii prostředí nedojde až k prostředí globálnímu. Pokud by vazba neexistovala ani v globálním prostředí, vyhodnocování by končilo chybou, viz bod (B.e).

Příklad 2.9. Vraťme se k prostředím uvedeným na obrázku 2.1. Z obrázku můžeme snadno vyčíst, že platí například $\mathrm{Eval}[\texttt{ahoj}, \mathcal{P}_3] = \texttt{0}$, $\mathrm{Eval}[\texttt{x}, \mathcal{P}_2] = \texttt{10}$, $\mathrm{Eval}[\texttt{z}, \mathcal{P}_1]$ je „nějaká procedura“, $\mathrm{Eval}[\texttt{+}, \mathcal{P}_G]$ je procedura sčítání a tak podobně. Dále například $\mathrm{Eval}[\texttt{x}, \mathcal{P}_3] = \texttt{\#f}$, protože symbol `x` má vazbu přímo v prostředí $\mathcal{P}_3$, hodnota navázaná na `x` v nadřazeném prostředí $\mathcal{P}_1$ tedy nehraje roli. Naproti tomu $\mathrm{Eval}[\texttt{z}, \mathcal{P}_3]$ je „nějaká procedura“, přitom zde již symbol `z` v prostředí $\mathcal{P}_3$ vazbu nemá, vazba byla tedy hledána v nadřazeném prostředí, což bylo $\mathcal{P}_1$, ve kterém byla vazba nalezena. Kdybychom uvažovali $\mathrm{Eval}[\texttt{+}, \mathcal{P}_3]$, pak nebude vazba symbolu `+` nalezena ani v prostředí $\mathcal{P}_1$, takže se bude hledat jeho nadřazeném prostředí, což je prostředí globální, ve kterém již vazba nalezena bude.

V tuto chvíli je potřeba objasnit ještě dvě věci. V cyklu REPL, který řídí vyhodnocovací proces abstraktního interpretu, se ve fázi „Eval“ (následující po načtení symbolického výrazu a jeho převedení do interní formy) provádí vyhodnocování elementů. Teď, když jsme rozšířili vyhodnocování o dodatečný parametr, kterým je prostředí, musíme přesně říct, ve kterém prostředí evaluator spouštěný v cyklu REPL elementy vyhodnocuje. Není asi překvapující, že to bude *globální* (*počáteční*) *prostředí* $\mathcal{P}_G$.

Další místo, kde může docházet k vyhodnocením, jsou speciální formy. Připomeňme, že každá speciální forma si sama vyhodnocuje (dle potřeby) své argumenty, takže speciální forma může rovněž používat evaluator. Obecně k tomu nelze nic říct (ve kterém prostředí bude docházet k vyhodnocování argumentů si každá speciální forma bude určovat sama). Jelikož jsme doposud představili jen tři speciální formy (`define`, `if` a `lambda`), můžeme specifikovat, jak vyhodnocování probíhá u nich. Zaměříme se pouze na speciální formy `define` a `if`, protože přesná činnost speciální formy `lambda` bude popsaná až v další sekci. O formách `define` a `if` můžeme říct, že veškerá vyhodnocování provádějí v prostředí, *ve kterém byly aplikovány*. Toto prostředí, tedy prostředí v němž byla vyvolána aplikace speciální formy, budeme dále nazývat *aktuální prostředí* (*aplikace speciální formy*). Fakt, že speciální formy `define` a `if` provádějí vyhodnocování výrazů v aktuálním prostředí [2] je plně v souladu s naší intuicí, protože uvážíme-li například znovu kód

```
(define abs
  (lambda (x)
    (if (>= x 0)
        x
        (- x))))
```

definující proceduru absolutní hodnoty, pak je žádoucí, aby při vyhodnocování těla pracovala forma `if` právě v lokálním (aktuálním) prostředí, to jest v prostředí kde je k dispozici vazba symbolu `x`. V globálním

[2] Přísně vzato by tedy aplikace speciální formy $\mathrm{Apply}[F_1, E_2, \ldots, E_n]$ uvedená v bodě (C.2) definice 2.7 měla obsahovat, jako jeden z argumentů, i prostředí $\mathcal{P}$, tedy měla by být například ve tvaru $\mathrm{Apply}[\mathcal{P}, F_1, E_2, \ldots, E_n]$. Pro jednoduchost značení však tento technický detail pomineme. Podotkněme, že tato poznámka se týká pouze speciálních forem, nikoliv procedur.

prostředí je tento symbol nedefinovaný. Analogicky, vyhodnocením `(define name` ⟨*výraz*⟩`)` v prostředí $\mathcal{P}$ (to jest $\mathrm{Eval}[$`(define name` ⟨*výraz*⟩`)`$, \mathcal{P}]$) se v prostředí $\mathcal{P}$ provede vazba symbolu `name` na výsledek vyhodnocení argumentu ⟨*výraz*⟩. Více se o vzniku a aplikaci procedur dozvíme v další sekci.

V této sekci jsme ukázali obecný koncept hierarchicky uspořádaných prostředí, kterého se již budeme držet dál během výkladu. Ve skutečnosti již nebudeme potřebovat vůbec upravovat vyhodnocovací proces. Od teď již budeme pouze uvažovat další speciální formy a procedury, kterými budeme náš abstraktní interpret postupně obohacovat.

2.3 Vznik a aplikace uživatelsky definovatelných procedur

V této sekci přesně popíšeme vznik procedur. Zatím jsme uvedli, že uživatelsky definovatelné procedury vznikají vyhodnocováním λ-výrazů, ale neuvedli jsme, jak samotné procedury vypadají ani jak konkrétně vznikají. Stejně tak jsme z technických důvodů zatím nespecifikovali *aplikaci uživatelsky definovatelných procedur*. To jest, neřekli jsme, co je výsledkem provedení $\mathrm{Apply}[F_1, F_2, \ldots, F_n]$ pokud je F_1 procedura vzniklá vyhodnocením λ-výrazu, viz bod (C.1) definice 2.7. Na tyto problémy se zaměříme nyní.

Abychom mohli při aplikaci procedur správně rozlišovat vazby vázaných a volných symbolů, musíme každou proceduru vybavit dodatečnou informací o prostředí jejího vzniku. Uživatelsky definovatelné procedury tedy budeme chápat jako elementy obsahující v sobě seznam formálních argumentů, tělo a odkaz na prostředí svého vzniku, viz následující definici:

Definice 2.10 (uživatelsky definovaná procedura)**.** Každá trojice ve tvaru

$$\langle \langle \textit{parametry} \rangle, \langle \textit{tělo} \rangle, \mathcal{P} \rangle,$$

kde ⟨*parametry*⟩ je *seznam formálních argumentů* (to jest seznam po dvou různých symbolů), ⟨*tělo*⟩ je *libovolný element* a $\mathcal{P}$ je *prostředí*, se nazývá *uživatelsky definovaná procedura.* ■

Dále specifikujeme, jak uživatelsky definované procedury přesně vznikají. Přísně vzato teď vlastně popíšeme činnost speciální formy `lambda`, která je překvapivě jednoduchá:

Definice 2.11 (speciální forma `lambda`)**.** Speciální forma `lambda` se používá se dvěma argumenty tak, jak to bylo popsáno v definici 2.1. Při aplikaci speciální formy `lambda` vyvolané vyhodnocením λ-výrazu

`(lambda (`⟨*param*$_1$⟩ ⟨*param*$_2$⟩ $\cdots$ ⟨*param*$_n$⟩`)` ⟨*tělo*⟩`)`

v prostředí $\mathcal{P}$ vznikne procedura $\langle$`(`⟨*param*$_1$⟩ ⟨*param*$_2$⟩ $\cdots$ ⟨*param*$_n$⟩`)`$, \langle$*tělo*$\rangle, \mathcal{P}\rangle$. ■

Všimněte si, že předchozí definice říká, jak vznikají uživatelsky definované procedury. Při vyhodnocení λ-výrazu se vezme seznam formálních parametrů a tělo (to jest druhý a třetí prvek λ-výrazu) a spolu s prostředím, ve kterém byl λ-výraz vyhodnocen, se zapouzdří do nově vzniklé procedury. Uživatelsky definované procedury jsou tedy elementy jazyka skládající se právě ze *seznamu formálních argumentů*, *těla*, a *prostředí vzniku procedury*. Další zajímavý rys vzniku procedur je, že speciální forma `lambda` během své aplikace *neprovádí žádné vyhodnocování elementů.*

Z předchozího je zřejmé, že každá procedura vzniklá vyhodnocením λ-výrazu si v sobě nese informaci o prostředí, ve kterém vznikla. Tato informace bude dále použita při aplikaci procedury.

Definice 2.12 (aplikace procedury)**.** Mějme dánu proceduru E a nechť $E_1, \ldots, E_n$ jsou libovolné elementy jazyka. *Aplikace procedury* E *na argumenty* $E_1, \ldots, E_n$ (v tomto pořadí), bude značena $\mathrm{Apply}[E, E_1, \ldots, E_n]$ v souladu s definicí 1.16 na straně 24. V případě, že E je primitivní procedura, pak je hodnota její aplikace $\mathrm{Apply}[E, E_1, \ldots, E_n]$ vypočtena touto primitivní procedurou (nezabýváme se přitom tím *jak* je vypočtena). Pokud je E uživatelsky definovaná procedura ve tvaru $\langle$`(`⟨*param*$_1$⟩ $\cdots$ ⟨*param*$_m$⟩`)`$, \langle$*tělo*$\rangle, \mathcal{P}\rangle$, pak hodnotu $\mathrm{Apply}[E, E_1, \ldots, E_n]$, definujeme takto:

Program 2.1. Výpočet délky přepony v pravoúhlém trojúhelníku.

```
(define na2
  (lambda (x)
    (* x x)))

(define soucet-ctvercu
  (lambda (a b)
    (+ (na2 a) (na2 b))))

(define prepona
  (lambda (odvesna-a odvesna-b)
    (sqrt (soucet-ctvercu odvesna-a odvesna-b))))
```

(1) Pokud se m (*počet formálních argumentů* procedury E) neshoduje s n (*počet argumentů* se kterými chceme proceduru aplikovat), pak aplikace končí chybovým hlášením „CHYBA: Chybný počet argumentů, proceduře bylo předáno n, očekáváno je m“. V opačném případě se pokračuje dalším krokem.

(2) Vytvoří se *nové prázdné prostředí* $\mathcal{P}_l$, které nazýváme *lokální prostředí procedury.* Tabulka vazeb prostředí $\mathcal{P}_l$ je v tomto okamžiku prázdná (neobsahuje žádné vazby), předek prostředí $\mathcal{P}_l$ není nastaven.

(3) *Nastavíme předka prostředí* $\mathcal{P}_l$ *na hodnotu* $\mathcal{P}$ (předkem prostředí $\mathcal{P}_l$ je *prostředí vzniku procedury* E).

(4) V prostředí $\mathcal{P}_l$ se zavedou vazby $\langle param_i \rangle \mapsto E_i$ pro $i = 1, \ldots, n$.

(5) Položíme $\mathrm{Apply}[E, E_1, \ldots, E_n] := \mathrm{Eval}[\langle tělo \rangle, \mathcal{P}_l]$. ■

Poznámka 2.13. Výsledkem aplikace uživatelsky definované procedury je tedy hodnota vzniklá vyhodnocením jejího těla v prostředí, ve kterém jsou na formální argumenty procedury navázané hodnoty předané proceduře, přitom předek tohoto nového prostředí je nastaven na prostředí vzniku procedury. Při vyhodnocování samotného těla v novém (lokálním) prostředí jsou zřejmě všechny vazby vázaných symbolů nalezeny přímo v lokálním prostředí. Vazby volných symbolů se hledají v prostředích, které jsou nadřazené (počínaje prostředím vzniku procedury).

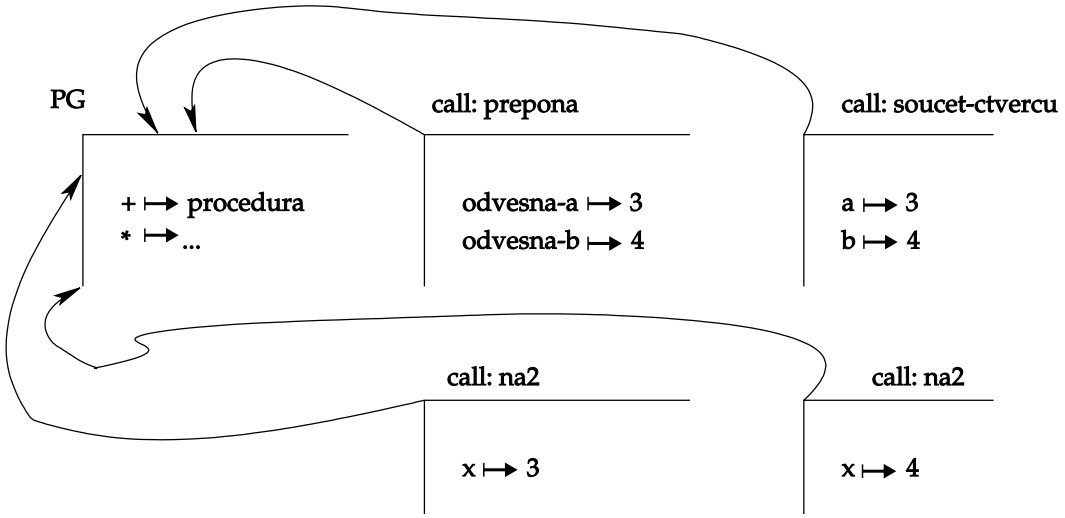
Všimněte si, že při aplikaci procedur může dojít k „překrytí globálně definovaných symbolů.“ Například pokud bychom provedli následující aplikaci:

```
((lambda (+) (- +)) 10)  ⟹  -10,
```

pak při aplikaci procedury vzniklé vyhodnocením `(lambda (+) (- +))` s hodnotou `10` vznikne prostředí, ve kterém bude na symbol `+` navázána hodnota `10`. V tomto prostředí bude vyhodnocen výraz `(- +)`. Vazba volného symbolu `-` bude nalezena v globálním prostředí (což je prostředí vzniku naší procedury), naproti tomu vazba symbolu `+` bude nalezena už v lokálním prostředí (hodnota `10`). Výsledkem aplikace je tedy skutečně `-10`. V lokálním prostředí tedy vazba symbolu `+` překryla vazbu, která existuje v globálním prostředí, kde je na symbol navázána primitivní procedura sčítání. Překrytí definice symbolů v tomto smyslu je někdy nechtěné, ale v mnoha případech je účelné, jak uvidíme v příští lekci.

Nová prostředí vznikají aplikací (uživatelsky definovaných) procedur, tedy nikoliv při jejich vytváření, ale až v momentě, kdy jsou procedury aplikovány. Tomu je potřeba rozumět tak, že s *každou novou aplikací* (jedné) procedury *vznikne nové prostředí.* Dále si všimněte, že z hlediska vazeb symbolů v těle procedury je úplně jedno, odkud proceduru aplikujeme, protože vazby symbolů se při vyhodnocování těla hledají počínaje lokálním prostředím – pokud v něm nejsou nalezeny, tak se přejde do *nadřazeného prostředí,* což je *prostředí vzniku procedury.* Prostředí odkud byla aplikace vyvolána tedy nemá uplatnění.

Obrázek 2.2. Vznik prostředí během aplikace procedur z programu 2.1



Příklad 2.14. Uvažujme nyní program 2.1 skládající se z několika procedur. Jde opět o program pro výpočet velikosti přepony pravoúhlého trojúhelníka, přitom procedura navázaná na symbol `prepona` v sobě používá pomocnou proceduru počítající součet čtverců. Procedura pro výpočet součtu čtverců v sobě jako pomocnou proceduru používá proceduru pro výpočet druhé mocniny. Pokud bychom v interpretu zadali výraz `(prepona 3 4)`, tak bude jeho vyhodnocování probíhat následovně. Nejprve bude aplikována procedura navázaná na `prepona` s hodnotami `3` a `4`, takže vznikne nové prostředí v němž budou tyto hodnoty navázané na formální argumenty `odvesna-a` a `odvesna-b`. Předchůdcem tohoto prostředí bude globální prostředí, protože to je prostředí, ve kterém tato procedura vznikla (vznikly v něm všechny uvedené procedury, takže to již dál nebudeme zdůrazňovat). Viz obrázek 2.2. V lokálním prostředí je tedy vyhodnoceno tělo, to jest výraz `(sqrt (soucet-ctvercu odvesna-a odvesna-b))`. Vyhodnocení tohoto výrazu povede na aplikaci procedury pro součet čtverců. Až k její aplikaci dojde, vznikne nové prostředí, ve kterém budou na symboly `a` a `b` (což jsou formální argumenty dané procedury) navázány hodnoty vzniklé vyhodnocením `odvesna-a` a `odvesna-b` v lokálním prostředí procedury pro výpočet přepony (což jsou hodnoty `3` a `4`). V lokálním prostředí procedury pro součet čtverců je pak vyhodnoceno její tělo. Při jeho vyhodnocování dojde ke dvojí aplikaci procedury pro výpočet druhé mocniny, jednou s argumentem `3` a jednou s argumentem `4`. Takže vzniknou dvě další prostředí. V každém z těchto dvou prostředí se vyhodnotí výraz `(* x x)`, což povede na výsledky `9` (v případě prostředí, kde `x` ↦ `3`) a `16` (v případě prostředí, kde `x` ↦ `4`). Výsledné hodnoty vzniklé voláním druhých mocnin jsou použity při aplikaci sčítání, které je provedeno v těle procedury pro součet čtverců, výsledkem její aplikace tedy bude hodnota `25`. Tato hodnota je potom použita v těle procedury pro výpočet délky přepony při aplikaci procedury pro výpočet odmocniny. Výraz `(prepona 3 4)` se tedy vyhodnotí na `5`. Během jeho vyhodnocení vznikly čtyři nová prostředí, každé mělo za svého předka globální prostředí, viz obrázek 2.2.

2.4 Procedury vyšších řádů

V této sekci se budeme zabývat dalšími aspekty uživatelsky definovatelných procedur. Konkrétně se budeme zabývat procedurami, kterým jsou při aplikaci předávány další procedury jako argumenty nebo které vracejí procedury jako výsledky své aplikace. Souhrnně budeme procedury tohoto typu nazývat *procedury vyšších řádů*. Z hlediska jazyka Scheme však procedury vyšších řádů *nejsou* „novým typem procedur“, jedná se o standardní procedury tak, jak jsme je chápali doposud. Tyto procedury „pouze“ pracují s dalšími procedurami jako s hodnotami (buď je dostávají formou argumentů nebo je vrací jako výsledky aplikace). Než přistoupíme k samotné problematice přijmeme nejprve zjednodušující konvenci týkající se označování (pojmenovávání) procedur:

Úmluva 2.15 (o pojmenovávání procedur)**.** V dalším textu budeme procedury, které jsou po svém vytvoření navázány na symboly, pojmenovávat stejně jako samotné symboly. ■

Podle předchozí úmluvy budeme tedy hovořit o proceduře `+` místo přesnějšího: „procedura navázaná na symbol `+`“. Jako první příklad procedury vyššího řádu si uvedeme proceduru, které bude předána další procedura jako argument. Viz program 2.2. Procedura `infix` v programu 2.2 je aplikována se třemi argumenty.

Program 2.2. Procedura provádějící infixovou aplikaci procedury dvou argumentů.

```
(define infix
  (lambda (x operace y)
    (operace x y)))
```

Formální argumenty jsme nazvali `x`, `operace` a `y`. Přeneseme-li svou pozornost na tělo této procedury, vidíme, že jeho vyhodnocení (k němuž dochází při aplikaci procedury) povede na pokus o aplikaci procedury navázané na symbol `operace` s argumenty navázanými na symboly `x` a `y`. Skutečně je tomu tak, protože symbol `operace` se v těle procedury nachází na první pozici v seznamu. Bude-li tedy procedura `infix` aplikována s druhým argumentem jímž nebude procedura (nebo speciální forma, přísně vzato), pak vyhodnocení těla skončí v bodě (C.e) chybou. Smysluplná aplikace procedury je tedy taková, při které jako druhý argument předáváme proceduru dvou argumentů a první a třetí argument budou hodnoty, se kterými má být předaná procedura aplikována. Jaký má tedy účel procedura `infix`? Jedná s o proceduru, pomocí které můžeme vyhodnocovat jednoduché výrazy v „infixové notaci“. Připomeňme, že problematika výrazů zapsaných v různých notacích, byla probrána v sekci 1.2 začínající na straně 13. Viz příklady použití procedury `infix`:

```
(infix 10 + 20)                    ⟹  30
(infix 10 - (infix 2 + 5))         ⟹  3
(infix 10 (lambda (x y) x) 20)     ⟹  10
(infix 10 (lambda (x y) 66) 20)    ⟹  66
(infix 10 (lambda (x) 66) 20)      ⟹  „CHYBA: Chybný počet argumentů při aplikaci.“
(infix 10 20 30)                   ⟹  „CHYBA: Nelze aplikovat: 20 není procedura.“
```

Na předchozím příkladu si všimněte toho, že v prvních dvou případech jsme předávali proceduře `infix` primitivní procedury sčítání a odčítání, v dalších případech jsme předávali procedury vzniklé vyhodnocením λ-výrazů (konkrétně to byly projekce a konstantní funkce). V hlediska vyhodnocovacího procesu a aplikace procedur je skutečně jedno, zdali procedura, kterou předáváme je primitivní procedura nebo uživatelsky definovaná procedura.

Nyní si ukážeme proceduru, která bude vracet další proceduru jako *výsledek své aplikace.* Podívejte se na proceduru `curry+` definovanou v programu 2.3. Procedura `curry+` má pouze jediný argument. Tělo pro-

Program 2.3. Rozložení procedury sčítání na dvě procedury jednoho argumentu.

```
(define curry+
  (lambda (c)
    (lambda (x)
      (+ x c))))
```

cedury `curry+` obsahuje λ-výraz `(lambda (x) (+ x c))`. To znamená, že při aplikaci procedury `curry+` bude vyhodnoceno její tělo, jímž je tento λ-výraz. Jeho vyhodnocením vznikne procedura (jednoho argumentu), která bude vrácena jako výsledná hodnota aplikace `curry+`. Při každé aplikaci `curry+` tedy *vznikne nová procedura.* Proceduru vzniklou aplikací `(curry+` ⟨*základ*⟩`)` bychom mohli slovně označit jako „proceduru, která hodnotu svého argumentu přičte k číslu ⟨*základ*⟩“. Abychom lépe pochopili, co vlastně

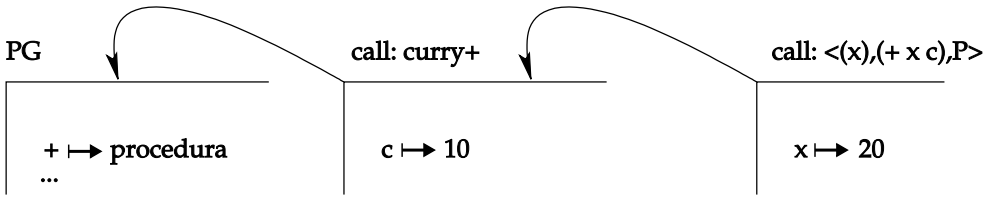
dělá procedura curry+, jak probíhá její aplikace a jak probíhá aplikace procedur, které curry+ vrací, si podrobně rozebereme následující příklad použití curry+:

```
(define f (curry+ 10))
f                          ⟼   „procedura, která hodnotu svého argumentu přičte k hodnotě 10“
(f 20)                     ⟼   30
```

Nejprve rozebereme vyhodnocení prvního výrazu. Na symbol f bude navázána hodnota vznklá vyhodnocením (curry+ 10). Při vyhodnocování tohoto výrazu dojde k aplikaci uživatelsky definované procedury curry+ s argumentem 10. Při její aplikaci vznikne prostředí $\mathcal{P}$, jehož předkem bude globální prostředí (protože v něm vznikla curry+), a ve kterém bude na symbol c navázána hodnota 10. V prostředí $\mathcal{P}$ bude vyhodnoceno tělo procedury curry+, to jest výraz (lambda (x) (+ x c)). Jeho vyhodnocením vznikne procedura $\langle$(x), (+ x c), $\mathcal{P}\rangle$, která je vrácena jako výsledek aplikace curry+. Tím pádem bude po vyhodnocení prvního výrazu na symbol f navázána procedura $\langle$(x), (+ x c), $\mathcal{P}\rangle$.

Zaměříme-li se teď na třetí řádek v předchozí ukázce, to jest na vyhodnocení (f 20), pak je zřejmé, že jde o aplikaci procedury navázané na symbol f (naše nově vytvořená procedura) s hodnotou 20. Při aplikaci $\langle$(x), (+ x c), $\mathcal{P}\rangle$ dojde k vytvoření nového prostředí, nazvěme jej třeba $\mathcal{P}'$. Jeho předkem bude prostředí $\mathcal{P}$ a prostředí $\mathcal{P}'$ bude obsahovat vazbu symbolu x na hodnotu 20. Nyní tedy již uvažujeme tři prostředí: $\mathcal{P}_G$ (globální), $\mathcal{P}$ (prostředí vzniku procedury navázané na symbol f) a $\mathcal{P}'$ (prostředí poslední aplikace procedury), pro která platí $\mathcal{P}_G \prec \mathcal{P} \prec \mathcal{P}'$, viz obrázek 2.3. V prostředí $\mathcal{P}'$ bude vyhodnoceno tělo procedury,

Obrázek 2.3. Vznik prostředí během aplikace procedur z programu 2.3



to jest výraz (+ x c). Nyní je dobře vidět, že symbol x má vazbu přímo v prostředí $\mathcal{P}'$ (lokální prostředí aplikované procedury), symbol c v $\mathcal{P}'$ nemá vazbu, ale má vazbu v jeho předchůdci $\mathcal{P}$ a konečně symbol + nemá vazbu ani v $\mathcal{P}'$ ani v $\mathcal{P}$, ale až v globálním prostředí $\mathcal{P}_G$. Snadno tedy nahlédneme, že výsledkem aplikace je hodnota součtu 20 a 10, což je číslo 30.

Z předchozího rozboru procedury curry+ plyne, že ji můžeme chápat jako proceduru, která *rozloží proceduru sčítání dvou argumentů* na *dvě procedury jednoho argumentu* tak, že první sčítanec je dán hodnotou, se kteru aplikujeme curry+ a druhý sčítanec je dán hodnotou, se kterou aplikujeme vrácenou proceduru. Proceduru vrácenou při volání curry+ lze tedy skutečně chápat jako proceduru, která k danému základu přičte předávaný argument. Pomocí curry+ bychom mohli snadno nadefinovat řadu užitečných procedur pro přičítání konstant, viz ukázku:

```
(define 1+ (curry+ 1))
(define 2+ (curry+ 2))
(define pi+
  (curry+
   (* 4 (atan 1))))
(1+ 10)                    ⟼   11
(2+ 10)                    ⟼   12
(pi+ 10)                   ⟼   13.14159265359
```

Proceduru vrácenou při aplikaci curry+ můžeme samozřejmě (jednorázově) volat i bez nutnosti vytvářet pomocnou vazbu procedury na symbol:

```
((curry+ 1) 2)             ⟼   3
((curry+ 1) ((curry+ 2) 3))  ⟼   6
```

Dalším rysem práce s procedurami, na který bychom měli upozornit, je šíření chyb v programu. Uvažujme následující kód:

```
(define f (curry+ #f))
(f 10)                    ⟼   „CHYBA: Nelze sčítat čísla a pravdivostní hodnoty.“
```

Zcela zřejmě došlo k chybě vzhledem k pokusu o sčítání čísla `10` s pravdivosntí hodnotu `#f`. Nutné je ale dobře si rozmyslet, kde k chybě došlo. Všimněte si, že aplikace `curry+` proběhla zcela v pořádku. To by pro nás v tuto chvíli nemělo být překvapující, protože `curry+` pouze vytváří novou proceduru. Při tomto procesu nedochází k vyhodnocování těla vytvářené procedury. Žádná chyba se v tomto bodě výpočtu neprojevila. Až při volání vytvořené procedury se interpret pokouší sečíst hodnotu `10` navázanou na `x` v lokálním prostředí s hodnotou `#f` navázanou na `c` v prostředí vzniku vrácené procedury. A až zde dojde k chybě. To je výrazný rozdíl proti tomu, kdybychom uvažovali výsledek vyhodnocení výrazu `(+ #f 10)`, zde by k chybě došlo okamžitě. Kdybychom nyní provedli globální redefinici `+`:

```
(define + (lambda (x y) x)),
```

pak bychom při aplikaci `(f 10)` dostali:

```
(f 10)  ⟼  10,
```

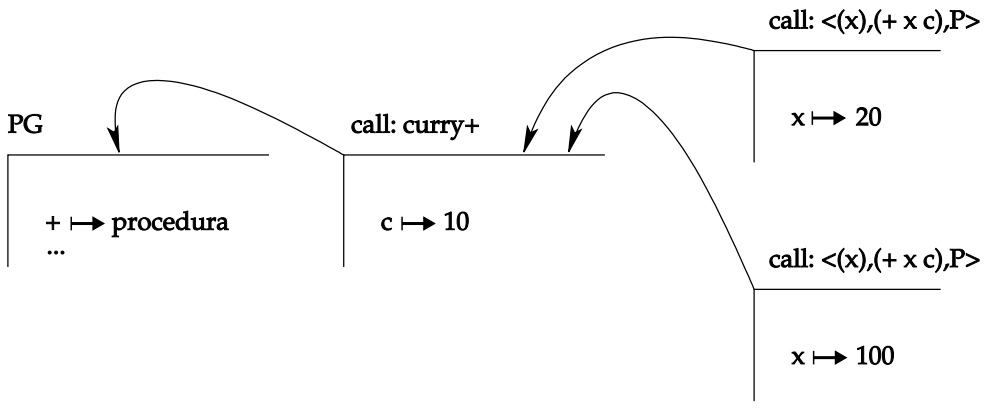
protože jsme symbol `+` v globálním prostředí redefinovali na projekci prvního argumentu ze dvou a tím je hodnota navázaná na symbol `x` (hodnota `#f` navázaná v předcházejícím prostředí na `c` tedy nebude hrát roli), viz program 2.3 a obrázek 2.3.

Poslední věc, na kterou poukážeme, je rys související s hierarchií prostředí, která vznikají při aplikaci těchto procedur. Vezmeme-li si kus kódu:

```
(define f (curry+ 10))
(f 20)                    ⟼   30
(f 100)                   ⟼   110,
```

pak bychom si měli být vědomi toho, že procedura `curry+` byla aplikována *pouze jednou*, kdežto procedura vzniklá její (jedinou) aplikací byla aplikována dvakrát, vznikne nám tedy hierarchie prostředí jako na obrázku 2.4. Nyní můžeme objasnit, proč jsme vlastně uživatelsky definovatelné procedury chápali jako

Obrázek 2.4. Vznik prostředí během aplikace procedur z programu 2.3



trojice, jejichž jednou složkou bylo prostředí jejich vzniku. Bylo to z toho důvodu, že jsme chtěli, aby vazby všech volných symbolů v těle procedur byly hledány v *prostředích vzniku procedur.* Bez použití procedur vyšších řádů byla situace poněkud triviální, protože prostředí vzniku procedury bylo vždy globální prostředí[3]. Nyní vidíme, že uvážíme-li procedury vracející jiné procedury jako výsledky svých aplikací, pak musíme mít u každé procedury podchyceno prostředí jejího vzniku. Bez této informace bychom nebyli schopni hledat vazby volných symbolů.

[3]V příští lekci uvidíme, že to tak úplně není pravda. V jazyku Scheme zavedeme pojem *interní definice* a zjistíme, že i bez použití procedur vyšších řádů mohou být prostředím vzniku procedur lokální prostředí vzniklá aplikací jiných procedur.

Jak již bylo řečeno, vazby v daném prostředí překrývají definice vazeb v prostředích nadřazených. V příkladě z obrázku 2.4 to má následující žádoucí efekt – pokud provedeme globální definici symbolu `c`, pak tím nijak neovlivníme činnost procedury `curry+`, ani procedur vzniklých její aplikací. Viz příklad:

```
(define f (curry+ 10))
(define c 666)
(f 20)                  ⟹  30
(f 100)                 ⟹  110
```

V druhém kroku je zavedena vazba $\texttt{c} \mapsto_{\mathcal{P}_G} \texttt{666}$, která ale nemá vliv na hodnotu vazby v prostředí volání `curry+`. Při vyhodnocení těla procedury navázané na `f` bude opět vazba symbolu `c` nalezena v prostředí $\mathcal{P}$. Fakt, že `c` má nyní vazbu i v prostředí nadřazeném je nepodstatný.

Poznámka 2.16. (a) Analogickým způsobem, jako jsme v programu 2.3 rozložili proceduru sčítání na proceduru jednoho argumentu vracející proceduru druhého argumentu, bychom mohli rozložit libovolnou proceduru dvou argumentů. Rozklad procedury dvou argumentů tímto způsobem je výhodný v případě, kdy první argument je při řadě výpočtů *pevný* a druhý argument může nabývat různých hodnot. Tento princip rozkladu poprvé použil americký logik Haskell Brooks Curry (1900–1982). Princip se nyní na jeho počest nazývá *currying* (termín nemá český ekvivalent). Po tomto logikovi je rovněž pojmenován významný představitel funkcionálních jazyků – Haskell.

(b) Z předchozích příkladů je zřejmé, že po ukončení aplikace procedury nemůže být lokální prostředí nějak „zrušeno“. To jest skutečné interprety jazyka Scheme musí lokální prostředí udržovat například v případě, kdy jako výsledek aplikace procedury vznikla další procedura pro niž se dané prostředí tím pádem stává prostředím jejího vzniku.

(c) Ve většině programovacích jazyků, ve kterých existuje nějaká analogie procedur (snad všechny funkcionální jazyky a drtivá většina procedurálních jazyků), lze s procedurami manipulovat pouze omezeně. Je například možné předávat procedury jako argumenty jiným procedurám. Například v procedurálním jazyku C je možné předat proceduře (v terminologii jazyka C se procedurám říká *funkce*) jinou proceduru formou *ukazatele* (*pointeru*). V drtivé většině jazyků však procedury nemohou během výpočtu *dynamicky vznikat* tak, jako tomu je v našem jazyku (světlou výjimkou jsou dialekty LISPu a jiné funkcionální jazyky).

(d) Procedury jsou ve většině vyšších programovacích jazyků (z těch co disponují procedurami) vytvářeny jako *pojmenované* – to jest jsou definovány vždy s jejich jménem. Naproti tomu v našem jazyku *všechny procedury vznikají* jako *anonymní*, to jest, nejsou nijak pojmenované. Svázání jména (symbolu) s procedurou můžeme ale následně provést pomocí speciální formy `define` tak, jak jsme to v této lekci již na mnoha místech udělali. Pouze málo vyšších jazyků, které nespadají do rodiny funkcionálních jazyků, disponuje možností vytvářet procedury anonymně (například do lze udělat v jazyku Python).

(e) V předchozích ukázkách procedur vracejících procedury jsme vždy vraceli uživatelsky definované procedury. Nic samozřejmě nebrání tomu, abychom vraceli i primitivní procedury, i když to není zdaleka tak užitečné. Například vyhodnocením výrazu `(lambda (x) sqrt)` vzniká (konstantní) procedura jednoho argumentu, která vždy vrací primitivní proceduru pro výpočet druhé odmocniny.

Pro každý vyšší programovací jazyk můžeme uvažovat jeho *elementy prvního řádu.* Viz následující definici:

Definice 2.17 (element prvního řádu)**.** *Element prvního řádu* je každý element jazyka, pro který platí:

(i) element může být *pojmenován*,

(ii) element může být *předán proceduře jako argument*,

(iii) element může *vzniknout aplikací* (*voláním*) *procedury*,

(iv) element může být *obsažen v hierarchických datových strukturách.*

S výjimkou bodu (iv), jehož smysl objasníme v jedné z dalších lekcí, je zřejmé, že procedury jsou z pohledu jazyka Scheme elementy prvního řádu, protože je můžeme pojmenovat, předávat jako argumenty, vytvářet aplikací jiných procedur (a poslední podmínka platí také). Jak jsme již naznačili v předchozí poznámce, většina programovacích jazyků tento „luxus“ neumožňuje. Procedury jako elementy prvního řádu jsou až na výjimky doménou funkcionálních programovacích jazyků.

Za *procedury vyšších řádů* považujeme procedury, které jsou aplikovány s jinými procedurami jako se svými argumenty nebo vracejí procedury jako výsledky své aplikace. Termín „procedury vyšších řádů“ je inspirovaný podobným pojmem používaným v matematické logice (logiky vyšších řádů). Samotné vytváření procedur vyšších řádů se nijak neliší od vytváření procedur, které jsme používali doposud. Nijak jsme nemuseli upravovat vyhodnocovací proces ani aplikaci. Z praktického hlediska chce vytváření procedur vyšších řádů od programátora o něco vyšší stupeň abstrakce – programátor se musí plně sžít s faktem, že *s procedurami se pracuje jako s hodnotami*.

2.5 Procedury versus zobrazení

V této sekci se zaměříme na vztah procedur a (matematických) funkcí. Připomeňme, pod pojmem *funkce* (*zobrazení*) množiny $X \neq \emptyset$ do množiny $Y \neq \emptyset$ máme na mysli relaci $R \subseteq X \times Y$ takovou, že pro každé $x \in X$ existuje právě jedno $y \in Y$ tak, že $\langle x, y\rangle \in R$. Obvykle přijímáme následující notaci. Zobrazení značíme obvykle malými písmeny $f, g, \ldots$ Pokud je relace $f \subseteq X \times Y$ zobrazením, pak píšeme $f(x) = y$ místo $\langle x, y\rangle \in f$. Fakt, že relace $f \subseteq X \times Y$ je zobrazení zapisujeme $f : X \to Y$. Pokud $X = Z_1 \times \cdots \times Z_n$, pak každé zobrazení $f : Z_1 \times \cdots \times Z_n \to Y$ nazýváme *n-ární zobrazení*. Místo $f(\langle z_1, \ldots, z_n\rangle) = y$ píšeme poněkud nepřesně, i když bez újmy, $f(z_1, \ldots, z_n) = y$.

Slovně řečeno, n-ární zobrazení $f : X_1 \times \cdots \times X_n \to Y$ přiřazuje každé n-tici prvků $x_1 \in X_1, \ldots, x_n \in X_n$ (v tomto pořadí) prvek y z Y označovaný $f(x_1, \ldots, x_n)$. Pokud $X = X_1 = \cdots = X_n$, pak píšeme stručně $f : X^n \to Y$. Studentům budou asi nejznámější speciální případy zobrazení používané v úvodních kurzech matematické analýzy – *funkce jedné reálné proměnné*. Z pohledu zobrazení jsou funkce jedné reálné proměnné zobrazení ve tvaru $f : S \to \mathbb{R}$, kde $\emptyset \neq S \subseteq \mathbb{R}$.

Uvážíme-li nyní (nějakou) proceduru (navázanou na symbol) `f`, která má n argumentů, nabízí se přirozená otázka, zdali lze tuto proceduru chápat jako nějaké zobrazení. Tato otázka je zcela na místě, protože při aplikaci se n-ární proceduře předává právě n-tice elementů, pro kterou je vypočtena hodnota, což je opět element. Odpověď na tuto otázku je (za dodatečných podmínek) kladná, jak uvidíme dále.

Při hledání odpovědi si předně musíme uvědomit, že aplikace n-ární procedury nemusí být pro každou n-tici elementů proveditelná. Aplikace třeba může *končit chybou* nebo *vyhodnocování* těla procedury *nemusí nikdy skončit* (v tom případě se výsledku aplikace „nedočkáme“). Abychom demonstrovali, že druhá situace může skutečně nastat, uvažme, že procedura `f` má své tělo ve tvaru následujícího výrazu:

```
((lambda (y) (y y)) (lambda (x) (x x)))
```

Pokud se zamyslíme nad průběhem vyhodnocování předchozího výrazu, pak zjistíme, že po první aplikaci procedury vzniklé vyhodnocením λ-výrazu `(lambda (y) (y y))` bude neustále dokola aplikována procedura vzniklá vyhodnocením λ-výrazu `(lambda (x) (x x))`, která si bude při aplikaci předávat sebe sama prostřednictvím svého argumentu (rozmyslete si podrobně proč). Pokus o vyhodnocení předchozího výrazu tedy vede k nekončící sérii aplikací téže procedury.

Označíme-li nyní M množinu všech elementů jazyka, pak z toho co jsme teď uvedli, je zřejmé, že n-ární procedury obecně nelze chápat jako zobrazení $f : M^n \to M$, protože nemusejí být definované pro každou n-tici hodnot z M. I kdybychom se soustředili pouze na n-ární procedury, které *jsou* definované pro každou n-tici elementů (nebo se pro každou proceduru omezili jen na podmnožinu $N \subseteq M^n$ na které je definovaná), i tak není zaručeno, že proceduru lze chápat jako zobrazení. V jazyku Scheme budeme například uvažovat primitivní proceduru navázanou na symbol `random`, která pro daný argument r jímž je přirozené číslo, vrací jedno z pseudo-náhodně vybraných nezáporných celých čísel, které je ostře menší než r. Viz příklad:

```
(random 5)  ⟹  3
(random 5)  ⟹  2
(random 5)  ⟹  1
(random 5)  ⟹  3
   ⋮
```

Pokud nyní budeme uvažovat proceduru, která ignoruje svůj jediný argument a výsledek její aplikace závisí na použití `random`, třeba proceduru vzniklou vyhodnocením

```
(lambda (x) (+ 1 (random 10))),
```

pak tuto proceduru nelze chápat jako zobrazení $f : M \to M$ i když výsledná hodnota je vrácena pro jakýkoliv argument. Předchozí procedura totiž může pro *dva stejné argumenty* vrátit *různé výsledné hodnoty.*

Dobrou zprávou je, že na hodně procedur se *lze dívat* jako na zobrazení. K čemu je toto pozorování vůbec dobré? Pokud víme, že procedura se chová jako zobrazení, pak se při *testování* její *funkčnosti* můžeme soustředit pouze na hodnoty jejich argumentů a dané výsledky. Pokud pro danou n-tici argumentů procedura vrací očekávaný výsledek, pak jej bude *vždy vracet* (pokud neprovedeme globální redefinici některého symbolu, který je *volný* v těle procedury), je přitom jedno, na jaké pozici se v programu procedura nachází nebo v jaké fázi výpočtu bude aplikována. Pohled na procedury jako na zobrazení (je-li to možné) je tedy důležitý z pohledu samotného programování.

Pohled na proceduru jako na zobrazení (pokud je to možné) je významnou přidanou hodnotou. Procedury chovající se jako zobrazení se snáze ladí a upravují. Při vytváření programu ve funkcionálním jazyku bychom se měli snažit tento pohled maximálně uplatňovat a omezit na minimum tvorbu procedur, které se jako zobrazení nechovají. Výhodou tohoto postupu je možnost ladit jednotlivé procedury po částech bez nutnosti provádět ladění v závislosti na pořadí aplikací, které by potenciálně během spuštění programu mohly nastat (což je někdy velmi těžké nebo dokonce nemožné zjistit). Pohled na procedury jako na zobrazení tedy v žádném případě není jen nějakou „nadbytečnou matematizací problému“.

Na druhou stranu můžeme *využít procedury k reprezentaci* některých *zobrazení.* Například proceduru pro výpočet součtu čtverců z programu 2.1 na straně 51 lze chápat jako proceduru přibližně reprezentující zobrazení $f : \mathbb{R}^2 \to \mathbb{R}$, definované $f(x, y) = x^2 + y^2$. Slovo „přibližně“ je zde samozřejmě na místě, protože v počítači nelze zobrazit iracionální čísla, takže bychom měli f chápat spíš jako zobrazení $f : \mathbb{S}^2 \to \mathbb{S}$ kde $\mathbb{S}$ je množina čísel reprezentovatelných v jazyku Scheme. Stejně tak třeba zobrazení $f : \mathbb{R} \to \mathbb{R}$ definovaná předpisy $f(x) = x^2$, $f(x) = \frac{x+1}{2}$, $f(x) = |x|$, ... budou přibližně reprezentovatelná procedurami vzniklými vyhodnocením:

```
(lambda (x) (* x x))
(lambda (x) (/ (+ x 1) 2))
(lambda (x) (if (>= x 0) x (- x)))
  ⋮
```

Ze středoškolské matematiky známe řadu praktických metod, jak na základě dané funkce $f : \mathbb{R} \to \mathbb{R}$ vyjádřit další funkci $g : \mathbb{R} \to \mathbb{R}$. Jedním ze způsobů je například vyjádření funkce g „posunem“ funkce f po ose x nebo y. Podrobněji, je-li $f : \mathbb{R} \to \mathbb{R}$ funkce, pak pro každé $k \in \mathbb{R}$ definujeme funkce $f_{X,k} : \mathbb{R} \to \mathbb{R}$ a $f_{Y,k} : \mathbb{R} \to \mathbb{R}$ tak, že pro každé $x \in \mathbb{R}$ položíme

$$f_{X,k}(x) = f(x - k),$$
$$f_{Y,k}(x) = f(x) + k.$$

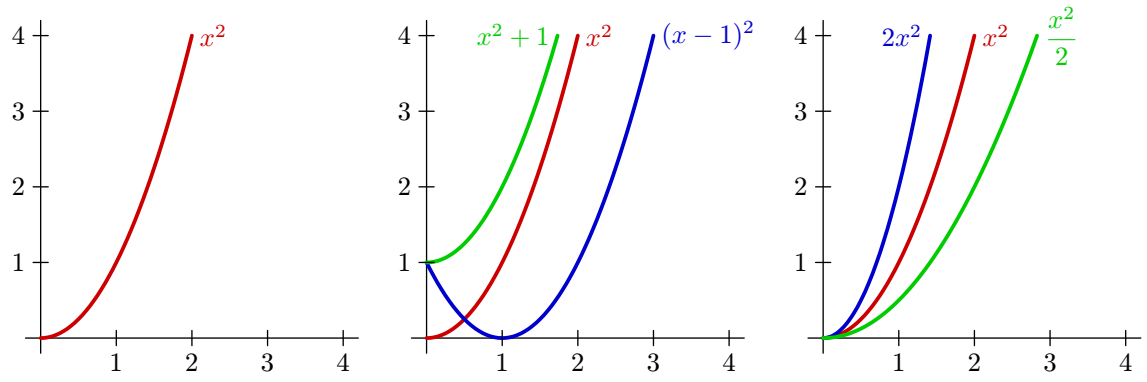
Funkce $f_{X,k}$ reprezentuje funkci f posunutou o k podél osy x a funkce $f_{Y,k}$ reprezentuje funkci f posunutou o k podél osy y. Na obrázku 2.5 (vlevo) je zobrazena část grafu funkce f dané předpisem $f(x) = x^2$ ($x \in \mathbb{R}$). Na témže obrázku uprostřed máme ukázky částí grafů funkcí $f_{X,1}$ a $f_{Y,1}$, které jsou tím pádem dány předpisy $f_{X,1}(x) = (x - 1)^2$ a $f_{Y,1}(x) = x^2 + 1$. Řadu dalších funkcí bychom mohli například vyrobit „násobením funkčních hodnot“. Pro každé $m \in \mathbb{R}$ a funkci f tedy můžeme uvažovat funkci $f_{*,m} : \mathbb{R} \to \mathbb{R}$ danou, pro každé $x \in \mathbb{R}$, následujícím předpisem:

$$f_{*,m}(x) = m \cdot f(x).$$

Na obrázku 2.5 (vpravo) máme zobrazeny funkce $f_{*,2}$ a $f_{*,\frac{1}{2}}$ příslušné funkci f (opět $f(x) = x^2$).

Z hlediska procedur vyšších řádů je toho vyjadřování „funkcí pomocí jiných funkcí“ zajímavé, protože můžeme snadno naprogramovat procedury, které budou z procedur reprezentující tato zobrazení vytvářet

Obrázek 2.5. Vyjádření funkcí pomocí posunu a násobení funkčních hodnot.



Program 2.4. Procedury vytvářející nové procedury pomocí posunu a násobení.

```
(define x-shift
  (lambda (f k)
    (lambda (x)
      (f (- x k)))))

(define y-shift
  (lambda (f k)
    (lambda (x)
      (+ k (f x)))))

(define scale
  (lambda (f m)
    (lambda (x)
      (* m (f x)))))
```

procedury reprezentující nová zobrazení. Například trojice procedur uvedená v programu 2.4 reprezentuje právě procedury vytvářející nové procedury pomocí posunutí a násobku. Konkrétně procedura `x-shift` akceptuje jako argumenty proceduru (jednoho argumentu) a číslo a vrací novou proceduru vzniklou z předané procedury jejím „posunutím po ose x“ o délku danou číslem. Všimněte si, jak koresponduje tělo procedury `x-shift` s předpisem zobrazení $f_{X,k}$. Analogicky procedura `y-shift` vrací proceduru „posunutou po ose y“ a konečně procedura `scale` vrací procedury s „násobenými funkčními hodnotami“. Reprezentaci funkcí zobrazených na obrázku 2.5 bychom mohli pomocí `x-shift`, `y-shift` a `scale` vyrobit následovně:

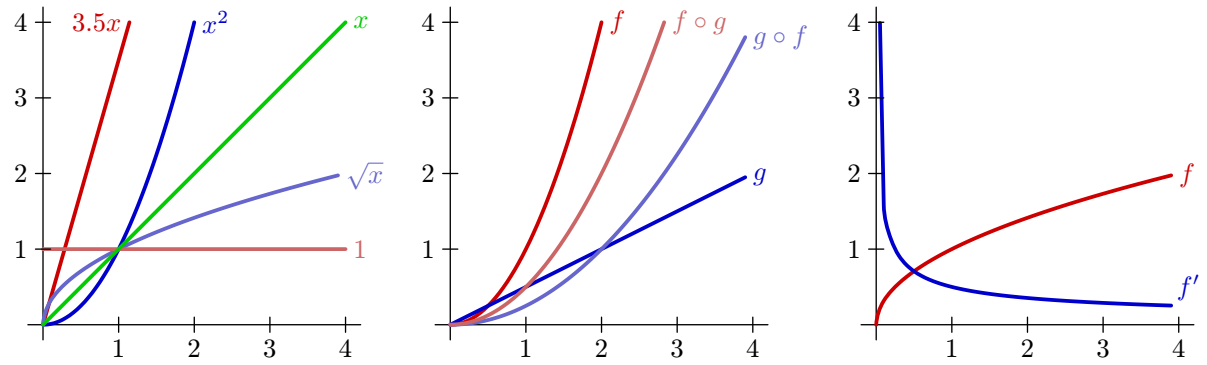
`(x-shift na2 1)` $\longmapsto$ „druhá mocnina posunutá o `1` na ose x“
`(y-shift na2 1)` $\longmapsto$ „druhá mocnina posunutá o `1` na ose y“
`(scale na2 2)` $\longmapsto$ „druhá mocnina násobená hodnotou `2`“
`(scale na2 1/2)` $\longmapsto$ „druhá mocnina násobená hodnotou `1/2`“

Takto vytvořené procedury bychom dále mohli třeba pojmenovat, nebo s nimi přímo pracovat:

```
((x-shift na2 1) 1)                ⟼  0
((x-shift na2 1) 2)                ⟼  1

(define na2*1/2 (scale na2 1/2))
(na2*1/2 1)                        ⟼  1/2
(na2*1/2 2)                        ⟼  2
```

Obrázek 2.6. Různé polynomické funkce, skládání funkcí a derivace funkce.



⋮

Řadu funkcí, jako například funkce dané předpisy $f(x) = x$ (identita), $f(x) = x^n$, $f(x) = \sqrt[n]{x}$, $f(x) = ax$, $f(x) = c$ (konstantní funkce) a jiné, lze chápat jako speciální případy funkce $f : \mathbb{R} \to \mathbb{R}$ definované

$$f(x) = ax^n$$

pro různé hodnoty parametrů $a, n \in \mathbb{R}$. Grafy některých z těchto funkcí jsou zobrazeny na obrázku 2.6 (vlevo). Procedury reprezentující všechny tyto funkce (a řadu dalších) bychom mohli v programu vytvářet pomocnou procedurou, které bychom předávali pouze různé hodnoty parametrů a a n. Z hlediska technické realizace (naprogramování ve skutečném interpretu jazyka Scheme) bychom museli vyřešit pouze problém, jak vypočítat hodnotu x^n. Ve standardu R^{6}RS jazyka Scheme je k dispozici procedura `expt`, se dvěma argumenty *základ* a *exponent*, která počítá mocninu (danou exponentem) z daného základu. Proceduru pro vytváření procedur reprezentující výše uvedená zobrazení bychom tedy mohli vytvořit třeba tak, jak je to uvedeno v programu 2.5. Procedury reprezentující funkce z obrázku 2.6 (vlevo) bychom mohli vytvořit

Program 2.5. Vytváření procedur reprezentujících polynomické funkce.

```
(define make-polynomial-function
  (lambda (a n)
    (lambda (x) (* a (expt x n)))))
```

pomocí procedury `make-polynomial-function` vyhodnocením následujících výrazů:

`(make-polynomial-function 3.5 1)`	$\Longmapsto$	„procedura reprezentující $f(x) = 3.5x$“
`(make-polynomial-function 1 2)`	$\Longmapsto$	„procedura reprezentující $f(x) = x^2$“
`(make-polynomial-function 1 1)`	$\Longmapsto$	„procedura reprezentující $f(x) = x$“
`(make-polynomial-function 1 1/2)`	$\Longmapsto$	„procedura reprezentující $f(x) = \sqrt{x}$“
`(make-polynomial-function 1 0)`	$\Longmapsto$	„procedura reprezentující $f(x) = 1$“

Dalším typickým způsobem vyjadřování funkcí je *skládání* (*kompozice*) dvou (nebo více) funkcí do jedné. Z matematického hlediska je jedná o skládání dvou zobrazení. Jsou-li $f : X \to Y$ a $g : Y' \to Z$ zobrazení, kde $Y \subseteq Y'$, pak definujeme zobrazení $(f \circ g) : X \to Z$ tak, že pro každé $x \in X$ klademe

$$(f \circ g)(x) = g(f(x)).$$

Zobrazení $f \circ g$ se potom nazývá *kompozice zobrazení f a g (v tomto pořadí)*. Předtím, než ukážeme program, který pro dvě procedury reprezentující zobrazení vrací proceduru reprezentující jejich kompozici, upozorněme na některé důležité vlastnosti složených funkcí.

Uvažujme neprázdnou množinu X a označme $\mathcal{F}(X)$ množinu všech zobrazení z X do X (to jest zobrazení ve tvaru $f : X \to X$). Pak pro složení libovolných dvou funkcí $f, g \in \mathcal{F}(X)$ platí $f \circ g \in \mathcal{F}(X)$. Skládání funkcí „$\circ$“ lze tedy chápat jako binární operaci na množině $\mathcal{F}(X)$. Tato operace má navíc další zajímavé

vlastnosti. Označme ι identitu na X (to jest $\iota(x) = x$ pro každé $x \in X$). Pak platí:

$$f \circ (g \circ h) = (f \circ g) \circ h,$$
$$\iota \circ f = f \circ \iota = f,$$

pro každé $f, g, h \in \mathcal{F}(X)$, což si můžete snadno dokázat sami. Skládání zobrazení je tedy *asociativní* a je *neutrální vzhledem k identickému zobrazení*. Z algebraického hlediska je struktura $\langle \mathcal{F}(X), \circ, \iota \rangle$ *pologrupa s neutrálním prvkem* neboli *monoid*.

Asociativita ($f \circ (g \circ h) = (f \circ g) \circ h$) říká, že při skládání více funkcí nezáleží na jejich uzávorkování, místo $f \circ (g \circ h)$ můžeme tedy bez újmy psát $f \circ g \circ h$, protože ať výraz uzávorkujeme jakkoliv, výsledné složení bude vždy jednoznačně dané. Upozorněme ale důrazně na fakt, že skládání funkcí *není komutativní*, nelze tedy zaměňovat pořadí, v jakém funkce skládáme (to jest obecně $f \circ g$ a $g \circ f$ *nejsou* stejná zobrazení). Příklad nekomutativity skládání funkcí je vidět na obrázku 2.6 (uprostřed). Neutralita vůči identitě ($\iota \circ f = f \circ \iota = f$) říká, že složením funkce s identitou (a obráceně) obdržíme výchozí funkci. Operacím, které spolu s množinou na níž jsou definované a se svým neutrálním prvkem tvoří monoid, budeme dále říkat *monoidální operace* a během dalších lekcí jich objevíme ještě několik.

Nyní již slíbená procedura `compose2`, která provádí „složení dvou procedur“, viz program 2.6. Opět je vidět, že proceduru jsme naprogramovali pouze pomocí vhodné formalizace definičního vztahu $f \circ g$ v jazyku Scheme. Při programování můžeme navíc proceduru `compose2` použít i s argumenty jimiž jsou

Program 2.6. Kompozice dvou procedur.

```
(define compose2
  (lambda (f g)
    (lambda (x)
      (g (f x)))))
```

procedury nereprezentující zobrazení (i když to asi nebude příliš účelné). V následující ukázce použití procedury `compose2` jsou definovány procedury korespondující s grafy funkcí v obrázku 2.6 (uprostřed) a pomocí `compose2` jsou vytvořeny procedury vzniklé jejich složením:

```
(define f na2)
(define g (make-polynomial-function 1/2 1))
(define f*g (compose2 f g))
(define g*f (compose2 g f))
```

Poslední ukázkou procedury vyššího řádu v této sekci bude procedura, které pro danou proceduru reprezentující funkci $f : \mathbb{R} \to \mathbb{R}$ bude vracet proceduru reprezentující přibližnou derivaci funkce f. Připomeňme, že derivace funkce f v daném bodě x je definována vztahem

$$f'(x) = \lim_{\delta \to 0} \left(\frac{f(x+\delta) - f(x)}{\delta} \right).$$

Pokud tedy v předchozím vztahu odstraníme limitní přechod a za δ zvolíme dost malé kladné číslo (blízko nuly), pak získáme odhad hodnoty derivace v bodě x:

$$g(x) = \frac{f(x+\delta) - f(x)}{\delta}.$$

Z geometrického pohledu je hodnota $g(x)$ směrnicí sečny (tangens úhlu, který svírá sečna s osou x), která prochází v grafu funkce body ($[x, f(x)]$ a $[x+\delta, f(x+\delta)]$). Jako funkci přibližné derivace funkce f tedy můžeme považovat funkci g (roli samozřejmě hraje velikost δ). Program 2.7 ukazuje proceduru `smernice`, která pro danou proceduru (reprezentující f) a dvě číselné hodnoty (reprezentující hodnoty x_1, x_2) vrací směrnici sečny procházející body $[x_1, f(x_1)]$ a $[x_2, f(x_2)]$. Procedura `derivace` akceptuje jako argument proceduru (reprezentující opět f) a hodnotu δ určující *přesnost*. Jako výsledek své aplikace procedura vrací proceduru jednoho argumentu reprezentující přibližnou derivaci, viz program 2.7 a předchozí diskusi. Následující kód ukazuje vytvoření procedury reprezentující přibližnou funkci derivace funkce $f(x) = \sqrt{x}$ s přesností $\delta = 0.001$ a její použití. Funkce a její derivace jsou zobrazeny na obrázku 2.6 (vpravo).

Program 2.7. Přibližná směrnice tečny a přibližná derivace.

```
(define smernice
  (lambda (f a b)
    (/ (- (f b) (f a))
       (- b a))))

(define derivace
  (lambda (f delta)
    (lambda (x)
      (smernice f x (+ x delta)))))
```

```
(define f-deriv
  (derivace sqrt 0.001))

(f-deriv 0.01)  ⟹  4.8808848170152
(f-deriv 0.1)   ⟹  1.5772056245761
(f-deriv 0.5)   ⟹  0.70675358090799
(f-deriv 2)     ⟹  0.3535092074645
(f-deriv 4)     ⟹  0.24998437695292
```

V této sekci jsme ukázali, že některé procedury lze chápat jako zobrazení a jiné procedury tak chápat nemůžeme. Poukázali jsme na výhody toho, když se procedury chovají jako zobrazení a ukázali jsme, jak lze zobrazení reprezentovat (nebo přibližně reprezentovat) pomocí procedur.

2.6 Lexikální a dynamický rozsah platnosti

V této sekci se ještě na skok vrátíme k prostředím a hledání vazeb symbolů. V předchozích sekcích jsme představili aplikaci uživatelsky definovaných procedur, která spočívala ve vyhodnocení jejich těla v novém lokálním prostředí. Každé lokální prostředí vytvořené při aplikaci procedury mělo jako svého předka nastaveno *prostředí vzniku procedury.* Všimli jsme si, že to bezprostředně vede k tomu, že pokud není vazba symbolu (uvedeného v těle procedury) nalezena v lokálním prostředí procedury, pak je hledána v prostředí vzniku této procedury. Prostředí vzniku procedury je buď globální prostředí nebo se jedná o lokální prostředí jiné procedury (tato procedura musí být zřejmě procedura vyššího řádu, protože v ní naše výchozí procedura vznikla). Pokud není symbol nalezen ani v tomto prostředí, je opět hledán v prostředí předka. Zde opět může nastat situace, že se jedná o globální prostředí nebo o lokální prostředí nějaké procedury vyššího řádu, . . .

Důležitým faktem je, že vazby symbolů, které nejsou nalezeny v lokálním prostředí, se postupně hledají *v prostředích vzniku procedur*. Pokud programovací jazyk používá tento princip hledání vazeb symbolů, pak říkáme, že programovací jazyk používá *lexikální* (*statický*) *rozsah platnosti symbolů* (ve mnoha jazycích se ovšem místo pojmu „symbol“ používá pojem „proměnná“). Prostředí vzniku procedury se v terminologii spojené s lexikálním rozsahem platnosti nazývá obvykle *lexikálně nadřazené prostředí.* Takže bychom mohli říct, že lexikální rozsah platnosti spočívá v hledání vazeb symbolů (které nejsou nalezeny v lokálním prostředí) v lexikálně nadřazených prostředích.

Lexikální rozsah platnosti není jediným možným rozsahem platnosti. Programovací jazyky mohou stanovit i jiný typ hledání vazeb symbolů. Druhým typem rozsahu platnosti je *dynamický rozsah platnosti,* který je v současnosti prakticky nepoužívaný. Stručně bychom mohli říci, že „jedinou odlišností“ od předchozího modelu je, že pokud není vazba symbolu nalezena v lokálním prostředí procedury, pak je hledána v *prostředí odkud byla procedura aplikována*. Prostředí, ve kterém byla procedura aplikována, můžeme nazvat *dynamicky*

nadřazené prostředí. Před další diskusí výhod a nevýhod obou typů rozsahů platnosti ukážeme příklad, na kterém bude jasně vidět odlišnost obou typů rozsahů platnosti.

Příklad 2.18. Uvažujme proceduru uvedenou v programu 2.3 na straně 53 a představme si, že v globálním prostředí provedeme navíc tuto definici:

```
(define c 100)
```

Co se stane, pokud dáme vyhodnotit následující dva výrazy?

```
(define f (curry+ 10))
(f 20)                    ⟼   ???
```

Odpověď v případě lexikálního rozsahu již máme: `30`, viz diskusi v sekci 2.4 k programu 2.3. Kdyby náš interpret ale používal dynamický rozsah platnosti, situace by byla jiná. Proceduru navázanou na symbol `f` jsme totiž *aplikovali* v globálním prostředí. Tím pádem bychom se při vyhodnocení jejího těla `(+ x c)` dostali do situace, kdy by byla vazba symbolu `c` (která není k dispozici v lokálním prostředí) hledána v dynamicky nadřazeném prostředí, to jest v prostředí aplikace procedury = v globálním prostředí. V tomto prostředí má ale symbol `c` vazbu `100`, takže výsledkem aplikace procedury navázané na `f` s argumentem `20` by bylo číslo `120`. Dostali jsme tedy *jiný výsledek* než v případě lexikálního rozsahu platnosti.

Lexikální a dynamický rozsah platnosti se od sebe liší ve způsobu hledání vazeb symbolů, pokud tyto vazby nejsou nalezeny v lokálním prostředí procedury. Při použití lexikálního rozsahu platnosti jsou vazby nenavázaných symbolů hledány v prostředí *vzniku* procedur (lexikálně nadřazeném prostředí). Při použití dynamického rozsahu platnosti jsou vazby nenavázaných symbolů hledány v prostředí *aplikace* procedur (dynamicky nadřazeném prostředí). Většina vyšších programovacích jazyků, včetně jazyka Scheme, používá lexikální rozsah platnosti.

Poznámka 2.19. Lexikální rozsah platnosti má z pohledu programátora a tvorby programů mnohem lepší vlastnosti. Předně, každá procedura má jednoznačně určené své lexikálně nadřazené prostředí. Toto prostředí můžeme kdykoliv jednoduše určit pouhým pohledem na strukturu programu. Podíváme se, ve které části programu je umístěn λ-výraz, jehož vyhodnocením procedura vznikne a pak stačí zjistit uvnitř které procedury se tento λ-výraz nachází. Pro jednoduchost můžeme na chvíli „ztotožnit procedury s λ-výrazy“, jejichž vyhodnocením vznikají, a „ztotožnit prostředí se seznamy formálních argumentů“ uvedených v λ-výrazech. Pro nalezení lexikálně nadřazeného prostředí nám stačí k danému λ-výrazu přejít k *nejvnitřnějšímu λ-výrazu, který výchozí λ-výraz obsahuje.* Ze seznamu formálních argumentů pak lze vyčíst, zdali bude vazba symbolu nalezena v prostředí aplikace procedury vzniklé vyhodnocením tohoto λ-výrazu, nebo je potřeba hledat v (dalším) nadřazeném prostředí. Například v případě následujícího programu

```
(lambda (x y)
   (lambda (z a)
      (lambda (d)
         (+ d a y b))))
```

bychom mohli vyčíst, že při aplikaci poslední vnořené procedury by v prostředí jejího lexikálního předka existovaly vazby symbolů `z` a `a` (v lokálním prostředí samotné procedury je k dispozici symbol `d`). V prostředí lexikálního předka jejího lexikálního předka by existovaly vazby symbolů `x` a `y`. Konkrétní hodnoty vazeb samozřejmě z programu vyčíst nemůžeme, ty budou známé až při aplikaci (během výpočetního procesu), ale *program* (sám o sobě) nám *určuje strukturu prostředí.* To je velkou výhodou lexikálního rozsahu platnosti. Naproti tomu při *dynamickém rozsahu platnosti* je *struktura prostředí dána* až samotným *výpočetním procesem.* Jelikož navíc tatáž procedura může být aplikována na více místech v programu, dynamicky nadřazené prostředí *není určeno jednoznačně.* Vezmeme-li kód z příkladu 2.18 a přidáme-li k němu výraz

```
((lambda (c)
   (f 20))
 1000),
```

pak při jeho vyhodnocení dojde k aplikaci procedury navázané na `f` s hodnotou `20`, ale tentokrát bude prostředím aplikace této procedury lokální prostředí procedury vytvořené při aplikaci procedury vznikající vyhodnocením vnějšího λ-výrazu. V tomto prostředí bude na symbol `c` navázána hodnota `1000`, takže procedura `f` nám dá se stejným argumentem jiný výsledek (konkrétně `1020`), než kdybychom ji zavolali v globálním prostředí, jak to bylo původně v příkladu 2.18. Z hlediska programátora je to silně nepřirozené, tatáž procedura aplikovaná na různých místech programu vrací různé hodnoty. To klade na programátora velké nároky při programování a hlavně při ladění programu (a případném hledání chyb). Pro danou proceduru totiž musí uvažovat, kdy a kde může být v programu aplikována. Ve velkých programech může být takových míst velké množství. Mnohem větším problémem ale je, že všechna místa aplikace obecně ani není možné zjistit (důvody nám budou jasnější v dalších lekcích).

Pokud bychom chtěli (z nějakého důvodu) v našem abstraktním interpretu *zavést dynamický rozsah platnosti* místo lexikálního, stačilo by uvažovat uživatelsky definovatelné procedury pouze jako elementy tvořené dvojicí ve tvaru $\langle\langle \textit{parametry}\rangle, \langle \textit{tělo}\rangle\rangle$. Prostředí vzniku procedury by si již nebylo potřeba pamatovat. Aplikaci procedury bychom museli uvažovat *relativně k prostředí* $\mathcal{P}_a$, ve kterém ji chceme provést (při lexikálním rozsahu jsme naopak informaci o prostředí, ve kterém byla procedura aplikována nepotřebovali). To jest v podobném duchu jako jsme rozšířili „Eval“ představený v první lekci o dodatečný parametr reprezentující prostředí, museli bychom nyní rozšířit Apply o další argument reprezentující prostředí – v tomto případě prostředí aplikace procedury. Dále bychom potřebovali upravit bod (3) definice 2.12 na straně 50 následovně:

(3) Nastavíme předka $\mathcal{P}_l$ na $\mathcal{P}_a$ (předkem nového prostředí je *prostředí aplikace procedury* E).

Dále bychom upravili „Eval“, viz definici 2.7 na straně 48, v bodech (C.1) a (C.2) tak, aby se při každé aplikaci předávala informace o prostředí $\mathcal{P}_a$, ve kterém aplikaci provádíme.

Poznámka 2.20. (a) Implementace dynamického rozsahu platnosti je jednodušší než implementace lexikálního rozsahu platnosti. Proto byl dynamický rozsah platnosti populární v rané fázi vývoje interpretů a překladačů programovacích jazyků. Programování s dynamickým rozsahem platnosti však vede k častému vzniku chyb (programátor se musí neustále zamýšlet nad tím, odkud bude daná procedura volána a jaké je potřeba mít v daném prostředí vazby symbolů) a proto jej v současnosti nevyužívá skoro žádný programovací jazyk (jedním z mála vyšších programovacích jazyků s dynamickým rozsahem platnosti, který je v praxi používán, je programovací jazyk FoxPro).

(b) V jazyku Common LISP existuje možnost deklarovat proměnnou jako „dynamickou.“ Z úhlu pohledu naší terminologie to znamená, že jazyk umožňuje hledat vazby nejen podle lexikálního rozsahu platnosti (který je standardní), ale u speciálně deklarovaných proměnných i podle dynamického rozsahu platnosti. Jedná se o jeden z mála jazyků (pokud ne jediný) umožňující v jistém smyslu využívat oba typy rozsahů.

2.7 Další podmíněné výrazy

V této sekci ukážeme další prvky jazyka Scheme, které nám budou sloužit k vytváření složitějších podmínek a složitějších podmíněných výrazů. Připomeňme, že podmíněné vyhodnocování jsme dělali pomocí speciální formy `if`, viz definici 1.31 na straně 36. Při podmíněném vyhodnocování výrazů hraje důležitou roli samotná podmínka. Doposud jsme používali pouze jednoduché podmínky. V mnoha případech se však hodí konstruovat složitější podmínky pomocí vazeb jako „platí . . . a platí . . . “ (*konjunkce* podmínek), „platí . . . nebo platí . . . “ (*disjunkce* podmínek), „neplatí, že . . . “ (*negace* podmínky). Nyní ukážeme, jak budeme vytváření složitějších podmínek provádět v jazyku Scheme.

Jazyk Scheme má k dispozici proceduru navázanou na symbol `not`. Tato procedura jednoho argumentu vrací výsledek negace pravdivostní hodnoty reprezentované svým argumentem. Přesněji řečeno, pokud je předaným argumentem `#f`, pak je výsledkem aplikace `not` pravdivostní hodnota `#t`. Pokud je argumentem jakýkoliv element různý od `#f`, pak je výsledkem aplikace `not` pravdivostní hodnota `#f`. Procedura `not` tedy pro libovolný element vrací buď `#f` nebo `#t`. Viz příklady:

```
(not #t)   ⟹  #f
(not #f)   ⟹  #t
```

```
(not 0)                          ⟼  #f
(not -12.5)                      ⟼  #f
(not (lambda (x) (+ x 1)))       ⟼  #f
(not (<= 1 2))                   ⟼  #f
(not (> 1 3))                    ⟼  #t
```

Procedura `not` tedy provádí negaci pravdivostní hodnoty, přitom pravdivostní hodnota je brána v zobecněném smyslu – vše kromě `#f` je považováno za „pravdu“. Viz komentář v sekci 1.7 na straně 35. Pozorným čtenářům zřejmě neuniklo, že procedura `not` je plně definovatelná. Viz program 2.8. Proceduru `not` bychom

Program 2.8. Procedura negace.

```
(define not
  (lambda (x)
    (if x #f #t)))
```

tedy nemuseli mít v jazyku Scheme dánu jako primitivní proceduru, ale mohli bychom ji dodatečně vytvořit. Na následujícím příkladu je vidět, že při práci s pravdivostními hodnotami v zobecněném smyslu platí „zákon dvojí negace“ (dvojicí negací výchozí pravdivostní hodnoty získáme tutéž pravdivostní hodnotu ve zobecněném smyslu), ale „pravda“ může být reprezentována vzájemně různými elementy (dvojí aplikací `not` tedy obecně nezískáme tentýž element):

```
(not (not #f))      ⟼  #f
(not (not #t))      ⟼  #t
(not (not -12.5))   ⟼  #t
```

V programu 2.9 jsou uvedeny příklady definic dvou predikátů. Připomeňme, že za predikáty považujeme procedury, které pro dané argumenty vracejí buď `#f` nebo `#t`. Predikát `even?` představuje predikát „je dané

Program 2.9. Predikáty sudých a lichých čísel.

```
(define even?
  (lambda (z)
    (= (modulo z 2) 0)))

(define odd?
  (lambda (z)
    (not (even? z))))
```

číslo sude?“, predikát `odd?` představuje predikát „je dané číslo liché?“. V těle predikátu `even?` je výraz vyjadřující podmínku, že číslo je sudé, právě když je jeho zbytek po dělení číslem 2 roven nule. Predikát `odd?` jsme naprogramovali pomocí negace a predikátu `even?`. Viz použití predikátů:

```
(even? 10)      ⟼  #t
(odd? 10)       ⟼  #f
(odd? 10.2)     ⟼  „CHYBA: Argument musí být celé číslo.“
```

Při vyhodnocení posledního výrazu způsobila aplikace procedury `modulo` chybové hlášení – prvním očekávaným argumentem mělo být celé číslo. Při psaní uživatelsky definovatelných predikátů (respektive, při jejich pojmenování) přijímáme následující konvenci, kterou jsme už použili i v programu 2.9.

Úmluva 2.21 (o pojmenovávání predikátů)**.** Pokud budou nově vytvořené predikáty navázané na symboly v globálním prostředí, pak budeme v jejich jménu na konci psát znak otazník „?“. ■

K vytváření složených podmínek ve tvaru *konjunkce* slouží speciální forma `and`:

Definice 2.22 (speciální forma `and`)**.** Speciální forma `and` se používá s libovolným počtem argumentů:

`(and` $\langle test_1 \rangle \cdots \langle test_n \rangle$`)`,

kde $n \geq 0$. Speciální forma `and` při své aplikaci postupně *vyhodnocuje výrazy* $\langle test_1 \rangle, \ldots, \langle test_n \rangle$ v aktuálním prostředí a to v pořadí *zleva doprava.* Pokud se průběžně vyhodnocovaný výraz $\langle test_i \rangle$ vyhodnotí na nepravdu (to jest `#f`), pak je výsledkem aplikace speciální formy `and` hodnota `#f` a další argumenty $\langle test_{i+1} \rangle, \ldots, \langle text_n \rangle$ uvedené za průběžně vyhodnocovaným argumentem se již nevyhodnocují. Pokud se postupně všechny výrazy $\langle test_1 \rangle, \ldots, \langle test_n \rangle$ vyhodnotí na pravdu (hodnotu různou od `#f`), pak je jako výsledek aplikace speciální formy `and` vrácena hodnota vyhodnocení posledního výrazu. Pokud $n = 0$ (`and` byla aplikována bez argumentů), pak je vrácena hodnota `#t`. ■

Následující příklady ukazují použití speciální formy `and`. Neformálně řečeno, výsledkem aplikace speciální formy `and` je „pravda“, pokud se postupně všechny její argumenty vyhodnotí na pravdu. Pokud tomu tak není, je výsledkem aplikace „nepravda“. Navíc platí, že všechny argumenty vyskytující se za argumentem jenž se vyhodnotil na nepravdu se již nevyhodnocují.

```
(and (= 0 0) (odd? 1) (even? 2))        ⟹  #t
(and (= 0 0) (odd? 1) (even? 2) 666)    ⟹  666
(and 1 #t 3 #t 4)                       ⟹  4
(and 10)                                ⟹  10
(and +)                                 ⟹  „procedura sčítání“
(and)                                   ⟹  #t
(and (= 0 0) (odd? 2) (even? 2))        ⟹  #f
(and 1 2 #f 3 4 5)                      ⟹  #f
```

V následujícím případu je vidět, že `and` skutečně řídí vyhodnocování svých argumentů tak, jak bylo řečeno v definici 2.22. Kdyby byl `and` procedura (a ne speciální forma), pak by při vyhodnocení následujícího výrazu došlo k chybě, protože symbol `nenavazany-symbol` by neměl vazbu.

```
(and (= 0 0) 2 #f nenavazany-symbol)    ⟹  #f
```

V našem případě ale k chybě nedochází, protože k vyhodnocení symbolu `nenavazany-symbol` nedojde. Zpracování svých argumentů speciální forma `and` ukončí jakmile narazí na třetí argument jenž se vyhodnotí na nepravdu.

Všimněte si, že speciální formu `and` je možné aplikovat i bez argumentu, v tom případě je konstantně vracena pravda (element `#t`). V sekci 1.5 jsme vysvětlili, co a proč vracejí procedury `+` a `*` pokud jsou aplikovány bez argumentů. Úvahu u speciální formy `and` bychom mohli udělat analogicky. Speciální forma `and` vrací `#t`, pokud je aplikována bez argumentů, protože `#t` se chová neutrálně vzhledem k pravdivostní funkci spojky „konjunkce“ (v logice obvykle značené $\wedge$):

`#t` $\wedge\, p = p \wedge$ `#t` $= p$.

To jest výraz

$p_1 \wedge p_2 \wedge \cdots \wedge p_n$

je ekvivalentní výrazu:

$p_1 \wedge p_2 \wedge \cdots \wedge p_n \wedge$ `#t`.

Když v posledním výrazu odstraníme všechny $p_1, \ldots, p_n$, zbude nám `#t`, což je pravdivostní hodnota kterou chápeme jako „konjunkci žádných argumentů“.

Následující predikát `within?` ukazuje praktické použití speciální formy `and`. Jedná se o predikát tří argumentů testující, zdali první argument leží v uzavřeném intervalu vymezeném dvěma dalšími argumenty:

```
(define within?
  (lambda (x a b)
    (and (>= x a) (<= x b))))
```

Nezkušení programátoři jsou někdy v pokušení psát místo předchozího programu následující:

```
(define within?
  (lambda (x a b)
    (if (and (>= x a) (<= x b))
        #t
        #f)))
```

Z hlediska funkčnosti jde o totéž, v druhém případě se ale v programu vyskytuje jeden zbytečný `if`, což snižuje jeho čitelnost. Psát speciální formu `if` ve tvaru `(if` ⟨*test*⟩ `#t #f)` je skutečně zbytečné, protože pokud je vyhodnocen ⟨*test*⟩ na pravdu, je vrácena „pravda“ a v opačném případě, tedy pokud se ⟨*test*⟩ vyhodnotil na nepravdu, je vrácena „nepravda“. Úplně tedy stačí do programu uvést samotný ⟨*test*⟩.

Ke speciální formě `and` ještě dodejme, že tato speciální forma je v našem jazyku také v podstatě nadbytečná, protože výraz ve tvaru

`(and` ⟨$test_1$⟩ $\cdots$ ⟨$test_n$⟩`)`

bychom mohli nahradit několika vnořenými `if`-výrazy:

```
(if ⟨test_1⟩
    (if ⟨test_2⟩
        (if ···
            (if ⟨test_{n-1}⟩
                ⟨test_n⟩
                #f)
              . . .
            #f)
        #f)
    #f)
```

Z hlediska programátora je však použití `and` samozřejmě mnohem příjemnější a čitelnější.

K vytváření složených podmínek ve tvaru *disjunkce* slouží speciální forma `or`:

Definice 2.23 (speciální forma `or`)**.** Speciální forma `or` se používá s libovolným počtem argumentů:

`(or` ⟨$test_1$⟩ $\cdots$ ⟨$test_n$⟩`)`,

kde $n \geq 0$. Speciální forma `or` při své aplikaci postupně *vyhodnocuje výrazy* ⟨$test_1$⟩, …, ⟨$test_n$⟩ v aktuálním prostředí a to v pořadí *zleva doprava*. Pokud se průběžně vyhodnocovaný výraz ⟨$test_i$⟩ vyhodnotí na pravdu (to jest cokoliv kromě `#f`), pak je výsledkem aplikace speciální formy `or` hodnota vzniklá vyhodnocením ⟨$test_i$⟩ a další argumenty ⟨$test_{i+1}$⟩, …, ⟨$text_n$⟩ uvedené za průběžně vyhodnocovaným argumentem se již nevyhodnocují. Pokud se postupně všechny výrazy ⟨$test_1$⟩, …, ⟨$test_n$⟩ vyhodnotí na nepravdu (to jest `#f`), pak je jako výsledek aplikace speciální formy `or` vrácena hodnota `#f`. Pokud $n = 0$ (`or` byla aplikována bez argumentů), pak je rovněž vrácena hodnota `#f`. ■

Viz příklady použití speciální formy `or`:

```
(or (even? 1) (= 1 2) (odd? 1))   ⟹  #t
(or (= 1 2) (= 3 4) 666)          ⟹  666
(or 1 #f 2 #f 3 4)                ⟹  1
(or (+ 10 20))                    ⟹  30
(or)                              ⟹  #f
(or #f)                           ⟹  #f
(or #f (= 1 2) #f)                ⟹  #f
```

Speciální forma `or` bez argumentu vrací `#f`, protože `#f` se chová neutrálně vzhledem k pravdivostní funkci spojky „disjunkce“ (v logice obvykle značené $\vee$):

`#f` $\vee\, p = p \vee$ `#f` $= p$,

můžeme tedy aplikovat analogickou úvahu jako v případě `and`, `+` nebo `*`. Speciální forma `or` řídí vyhodnocování svých argumentů, nemůže se tedy jednat o proceduru. Jako příklad si můžeme uvést následující kód:

```
(or (even? 2) nenavazany-symbol)  ⟹  #t,
```

při jehož vyhodnocení by v případě, pokud by byla `or` procedura, nastala chyba. V našem případě je však vrácena hodnota `#t`, což je výsledek vyhodnocení `(even? 2)` a argument `nenavazany-symbol` již vyhodnocen nebude. Výsledkem aplikace speciální formy `or` je „nepravda“, pokud se postupně všechny její argumenty vyhodnotí na nepravdu. Pokud tomu tak není, je výsledkem aplikace výsledek vyhodnocení prvního argumentu, který se nevyhodnotil na „pravdu.“ Všechny argumenty vyskytující se za argumentem jenž se vyhodnotil na pravdu se dál nevyhodnocují.

Následující procedura je praktickou ukázkou užití speciální formy `or`. Jedná se o predikát `overlap?`, který pro dva uzavřené intervalu $[a, b]$ a $[c, d]$, jejichž krajní prvky jsou dány čtyřmi argumenty, testuje, zdali se tyto intervalu vzájemně překrývají, tedy jestli platí $[a, b] \cap [c, d] \neq \emptyset$.

```
(define overlap?
  (lambda (a b c d)
    (or (within? a c d)
        (within? b c d)
        (within? c a b)
        (within? d a b))))
```

V tomto případě nám hned „matematický popis“ vzájemného překrytí nebyl při psaní procedury moc platný. Snadno ale nahlédneme, že $[a, b] \cap [c, d] \neq \emptyset$, právě když platí aspoň jedna z podmínek: $a \in [c, d]$, $b \in [c, d]$, $c \in [a, b]$ a $d \in [a, b]$, což je právě podmínka formalizovaná v těle předchozí procedury.

Stejně jako v případě `and` bychom mohli výraz

`(or` $\langle test_1 \rangle \cdots \langle test_n \rangle$`)`

nahradit vnořenými výrazy ve tvaru:

```
(if ⟨test_1⟩
    ⟨test_1⟩
    (if ⟨test_2⟩
        ⟨test_2⟩
        (if ···
            (if ⟨test_n⟩
                ⟨test_n⟩
                #f)···)))
```

Z čistě funkcionálního pohledu, o který se v tomto díle učebního textu budeme snažit, jsou předchozí dva kódy skutečně ekvivalentní. V dalším díle tohoto textu však uvidíme, že v případě zavádění imperativních prvků do našeho jazyka (prvků vyskytujících se v procedurálních jazycích), již bychom tento přepis nemohli provést. Zatím se ale s ním můžeme plně spokojit.

K speciálním formám `and` a `or` ještě podotkněme, jaký je jejich vztah k pravdivostním funkcím logických spojek „konjunkce“ a „disjunkce“. Vzhledem k tomu, že nepracujeme pouze s pravdivostními hodnotami `#f` a `#t`, naše formy se chovají poněkud jinak než výše uvedené pravdivostní funkce používané v logice. Pravdivostní funkce konjunkce a disjunkce například splňují De Morganovy zákony z nichž nám plyne, že disjunkci lze vyjádřit pomocí konjunkce a negace a analogicky konjunkci lze vyjádřit pomocí disjunkce a negace:

$$p_1 \vee \cdots \vee p_n = \neg(\neg p_1 \wedge \cdots \wedge \neg p_n),$$
$$p_1 \wedge \cdots \wedge p_n = \neg(\neg p_1 \vee \cdots \vee \neg p_n).$$

Přepíšeme-li levé a pravé strany předchozích výrazů v notaci jazyka Scheme a zamyslíme-li se nad hodnotami vzniklými jejich vyhodnocením, pak uvidíme, že vyhodnocením mohou vzniknout různé elementy, kupříkladu:

```
(or 1 2 3)                          ⟹  1
(and 1 2 3)                         ⟹  3
(not (and (not 1) (not 2) (not 3)))  ⟹  #t
(not (or (not 1) (not 2) (not 3)))   ⟹  #t
```

Dvojice elementů vzniklých vyhodnocením levé a pravé strany předchozích vztahů ale vždy reprezentují pravdu (v zobecněném smyslu), nebo jsou oboje rovny `#f` (zdůvodněte si proč).

Nyní si ukážeme aplikaci procedur vyšších řádů. Předpokládejme, že máme naprogramovat procedury dvou argumentů `min` a `max`, které budou pro dané dva číselné argumenty vracet menší, případně vetší, hodnotu z nich. Takové procedury můžeme snadno naprogramovat třeba tak, jak je to uvedeno v programu 2.10. Již při letmém pohledu na obě proceduru nás jistě napadne, že vypadají „téměř shodně“

Program 2.10. Procedury vracející minumum a maximum ze dvou prvků.

```
(define min
  (lambda (x y)
    (if (<= x y) x y)))

(define max
  (lambda (x y)
    (if (>= x y) x y)))
```

a liší se pouze v použití (procedur navázaných na) „`<=`“ a „`>=`“, to jest procedur pro porovnávání číselných hodnot. V takovém okamžiku se nabízí zobecnit program v tom smyslu, že vytvoříme abstraktní proceduru vyššího řádu, která bude vytvářet procedury pro vracení „extrémního prvek ze dvou“. To, co budeme mít na mysli pod pojmem „extrémní prvek“, budeme vytvářející proceduře specifikovat pomocí jejího argumentu. Podívejte se nyní na program 2.11. V něm je definována procedura `extrem` jednoho

Program 2.11. Procedura vyššího řádu pro hledání extrémních hodnot.

```
(define extrem
  (lambda (f)
    (lambda (x y)
      (if (f x y) x y))))

(define min (extrem <=))
(define max (extrem >=))
```

argumentu. Tímto argumentem je predikát, který bude použit na otestování, zdali jedna číselná hodnota je „extrémnější“ vůči druhé. Tělem procedury `extrem` je λ-výraz `(lambda (x y) (if (f x y) x y))` jehož vyhodnocením vzniká procedura dvou argumentů. Tato procedura při své aplikaci vyhodnocuje tělo, v němž se nachází podmíněný výraz. V něm je nejprve vyhodnocena podmínka `(f x y)`, která je pravdivá, právě když procedura navázaná na `f` vrátí „pravdu“ pro argumenty se kterými byla zavolána procedura vrácená procedurou `extrem`. Pokud bude na `f` navázána procedura „menší nebo rovno“, pak se bude procedura vrácená procedurou `extrem` chovat stejně jako procedura `min` z programu 2.10. Pokud bude na `f` navázána procedura „větší nebo rovno“, pak se bude procedura vrácená procedurou `extrem` chovat stejně jako procedura `max` z programu 2.10. Procedury `min` a `max` tedy můžeme získat voláním `extrem` a navázat je na příslušné symboly tak, jako v programu 2.11.

Procedura `extrem` nám však umožňuje vytvářet mnohé další procedury pro vracení význačných prvků, což nám oproti prostému minimu a maximu ze dvou dává nové, předtím netušené, možnosti. Pomocí procedury pro výpočet absolutní hodnoty

```
(define abs
  (lambda (x)
    (if (>= x 0)
        x
```

```
        (- x))))
```

kterou jsme již dříve uvedli, bychom mohli definovat dvě nové procedury `absmin` a `absmax`, které budou vracet prvek, který má menší nebo větší absolutní hodnotu:

```
(define absmin
  (extrem (lambda (x y)
            (<= (abs x) (abs y)))))

(define absmax
  (extrem (lambda (x y)
            (>= (abs x) (abs y)))))
```

Rozdíl mezi `min` a `absmin` (případně mezi `max` a `abmax`) je zřejmý:

```
(min -20 10)     ⟼  -20
(absmin -20 10)  ⟼  10
```

Analogicky bychom mohli vytvořit další procedury.

Speciální forma `if` byla představena již v předchozí lekci, viz definici 1.31 na straně 36. Nyní představíme speciální formu `cond`, pomocí které lze jednodušeji zapisovat složitější podmíněné výrazy.

Definice 2.24 (speciální forma `cond`)**.** Speciální forma `cond` se používá ve tvaru:

```
(cond (⟨test_1⟩ ⟨důsledek_1⟩)
      (⟨test_2⟩ ⟨důsledek_2⟩)
        ⋮
      (⟨test_n⟩ ⟨důsledek_n⟩)
      (else ⟨náhradník⟩)),
```

kde $n \geq 0$ a poslední výraz v těle, to jest výraz `(else` ⟨*náhradník*⟩`)` je *nepovinný* a *nemusí být uveden*. Při své aplikaci postupně speciální forma `cond` vyhodnocuje (v aktuálním prostředí) výrazy $\langle test_1 \rangle, \ldots, \langle test_n \rangle$ do toho okamžiku, až narazí na první $\langle test_i \rangle$, který se vyhodnotil na pravdu (na cokoliv kromě `#f`). V tom případě se vyhodnocování dalších $\langle test_{i+1} \rangle, \ldots, \langle test_n \rangle$ neprovádí a výsledkem aplikace speciální formy je hodnota vzniklá vyhodnocením výrazu $\langle důsledek_i \rangle$ v aktuálním prostředí. Pokud se ani jeden z výrazů $\langle test_1 \rangle, \ldots, \langle test_n \rangle$ nevyhodnotil na pravdu pak mohou nastat dvě situace. Pokud je posledním argumentem výraz ve tvaru `(else` ⟨*náhradník*⟩`)`, pak je výsledkem aplikace speciální formy hodnota vzniklá vyhodnocením výrazu ⟨*náhradník*⟩ v aktuálním prostředí. Pokud ⟨*náhradník*⟩ není přítomen, pak je výsledkem aplikace speciální formy `cond` nedefinovaná hodnota, viz poznámku 1.27 (b) na straně 34. ■

Z tvaru, ve kterém se používá speciální forma `cond` lze jasně vyčíst, že by nemohla být zastoupena procedurou, například totiž

```
(cond (1 2))  ⟼  2,
```

kdyby byla `cond` procedura, vyhodnocení předchozího výrazu by končilo chybou. Před uvedením praktických příkladů si nejprve ukažme některé mezní případy použití `cond`. V následujícím případě se neuplatní `else`-větev:

```
(cond ((= 1 1) 20)
      (else 10))  ⟼  20
```

Stejně tak, jako by se neuplatnila v tomto případě:

```
(cond ((+ 1 2) 20)
      ((= 1 1) 666)
      (else 10))  ⟼  20
```

Zde už se větev uplatní:

```
(cond ((= 1 2) 20)
      (else 10))  ⟼  10
```

Všimněte si, že `else`-větev `(else` ⟨*náhradník*⟩`)` bychom mohli ekvivalentně nahradit výrazem ve tvaru

`(`⟨*test*⟩ ⟨*náhradník*⟩`)`,

ve které by se ⟨*test*⟩ vyhodnotil vždy na „pravda“ (cokoliv kromě `#f`). Třeba takto:

```
(cond ((= 1 2) 20)
      (#t 10))  ⟼  10
```

Kdyby ani jeden z testů nebyl pravdivý a `else`-větev chybí, pak je výsledná hodnota aplikace nedefinovaná:

```
(cond ((= 1 2) 20)
      ((even? 1) (+ 1 2)))        nedefinovaná hodnota
```

Speciálním případem předchozího je `cond` bez argumentů:

```
(cond)        nedefinovaná hodnota
```

Nyní si ukážeme praktické použití speciální formy `cond`. Uvažujme funkci sgn (funkce *signum*), která se často používá v matematice a kterou definujeme předpisem:

$$\operatorname{sgn} x = \begin{cases} -1 & \text{pokud } x < 0, \\ 0 & \text{pokud } x = 0, \\ 1 & \text{pokud } x > 0. \end{cases}$$

Výsledkem $\operatorname{sgn} x$ je tedy jedna z hodnot -1, 0 nebo 1 v závislosti na tom, zdali je dané číslo x záporné, nula, nebo kladné. Definiční vztah bychom mohli přímočaře přepsat pomocí speciální formy `cond` a vytvořit tak formalizaci funkce sgn:

```
(define sgn
  (lambda (x)
    (cond ((= x 0) 0)
          ((> x 0) 1)
          (else -1))))
```

Jelikož se všechny podmínky v definici sgn vzájemně vylučují (vždy je právě jedna z nich pravdivá), nezáleží přitom na *pořadí*, v jakém byly jednotlivé podmínky a příslušné důsledky uvedeny. Výše uvedená procedura skutečně reprezentuje funkci signum (přesvědčte se sami).

Speciální forma `cond` je v podstatě taky „nadbytečná“, protože bychom místo ní mohli opět použít sérii do sebe zanořených výrazů používajících speciální formu `if`. Konkrétně bychom mohli výraz ve tvaru

```
(cond (⟨test₁⟩ ⟨důsledek₁⟩)
      (⟨test₂⟩ ⟨důsledek₂⟩)
        ⋮
      (⟨testₙ⟩ ⟨důsledekₙ⟩)
      (else ⟨náhradník⟩)),
```

nahradit výrazem

```
(if ⟨test₁⟩
    ⟨důsledek₁⟩
    (if ⟨test₂⟩
        ⟨důsledek₂⟩
        (if ···
            (if ⟨testₙ⟩
                ⟨důsledekₙ⟩
                ⟨náhradník⟩)···)))
```

Samozřejmě, že použití formy `cond` je mnohem přehlednější. Srovnejte naši výše uvedenou proceduru realizující funkci signum a následující proceduru, ve které je `cond` rozepsán pomocí `if`:

```
(define sgn
  (lambda (x)
```

```
    (if (= x 0)
        0
        (if (> x 0)
            1
            -1))))
```

Nejen, že `cond` je vyjádřitelná pomocí speciální formy `if`, ale lze tak učinit i obráceně, což je samozřejmě jednodušší – použijeme `cond` pouze s jednou podmínkou a případně s `else`-výrazem. V následujícím příklady máme proceduru pro výpočet absolutní hodnoty napsanou s použitím `cond`:

```
(define abs
  (lambda (x)
    (cond ((>= x 0) x)
          (else (- x)))))
```

Dohromady tedy dostáváme, že `if` a `cond` jsou *vzájemně plně zastupitelné.*

V následujícím příkladu si ukážeme použití `cond` při stanovení největšího prvku ze tří. Půjde tedy o zobecnění procedury `max` z programu 2.10 tak, aby pracovala se třemi číselnými hodnotami. Samozřejmě, že úplně nejčistším řešením problému by bylo využití již zavedené procedury `max` takto:

```
(define max3
  (lambda (x y z)
    (max x (max y z))))
```

Výše uvedená procedura skutečně vrací maximum ze tří prvků (rozmyslete si důkladně, proč). My ale v rámci procvičení práce s `cond` vyřešíme tento problém bez použití `max`. Mohli bychom to udělat takto:

```
(define max3
  (lambda (x y z)
    (cond ((and (>= x y) (>= x z)) x)
          ((and (>= y x) (>= y z)) y)
          (else z))))
```

V těle předchozí procedury je pomocí podmíněných výrazů zachycen rozbor problému nalezení maxima ze tří prvků. V první podmínce je testováno, zdali je hodnota navázaná na `x` větší než dvě další. Pokud je tomu tak, je tato hodnota největší. Na dalším řádku provedeme analogický test pro `y`. Na třetím řádku již můžeme s jistotou vrátit hodnotu navázanou na `z`, protože třetí řádek bude zpracován, pokud testy v předchozích dvou selžou, tedy právě když ani `x` ani `y` nejsou největší hodnoty, musí jí potom být `z`. Předchozí proceduru bychom ještě mohli následovně zjednodušit:

```
(define max3
  (lambda (x y z)
    (cond ((and (>= x y) (>= x z)) x)
          ((>= y z) y)
          (else z))))
```

Zde jsme zeštíhlili druhou podmínku. Když první test nebude pravdivý, pak máme jistotu, že `x` nebude největší hodnotou, pak již stačí vzájemně porovnat jen hodnoty `y` a `z`. I když je nový kód kratší, pro někoho může být méně čitelný. Snad nejčitelnější (i když nejméně efektivní z hlediska počtu vyhodnocovaných výrazů) by byla procedura ve tvaru:

```
(define max3
  (lambda (x y z)
    (cond ((and (>= x y) (>= x z)) x)
          ((and (>= y x) (>= y z)) y)
          ((and (>= z x) (>= z y)) z)
          (else sem-bychom-se-nemeli-dostat))))
```

Tato procedura má na svém konci uvedenu jakousi „pojistku“, která by se mohla uplatnit, kdyby někdo (třeba omylem) předefinoval globální vazbu symbolu `>=` třeba procedurou vzniklou vyhodnocením

λ-výrazu `(lambda (x y) #f)`. V každém případě bychom měli po naprogramování procedury se složitějšími podmíněnými výrazy provést její důkladné otestování, abychom mohli okamžitě zjistit všechny případné chyby (které jsme „zcela jistě nechtěli udělat“, ale všechno přece „bylo tak jasné“[4]). Například v případě procedury `max3` bychom měli provést test se třemi vzájemně různými číselnými hodnotami a otestovat všechny jejich permutace, třeba:

```
(max3 1 2 3)
(max3 1 3 2)
(max3 2 1 3)
(max3 2 3 1)
(max3 3 1 2)
(max3 3 2 1)
```

Shrnutí

V této lekci jsme položili základ problematice uživatelsky definovatelných procedur. Pomocí procedur lze účinně vytvářet abstrakce a v důsledku tak psát čistější, menší, univerzálnější a lépe spravovatelné programy. Uživatelsky definované procedury vznikají vyhodnocováním λ-výrazů, což jsou seznamy ve speciálním tvaru skládající se ze seznamu formálních argumentů a těla procedury. Kvůli aplikaci procedur jsme museli rozšířit dosud uvažovaný pojem prostředí. Prostředí už neuvažujeme pouze jedno, ale prostředí obecně vznikají během výpočtu aplikací procedur. Každé prostředí je navíc vybaveno ukazatelem na svého předka, což je prostředí vzniku procedury. Tento přístup nám umožňuje hledat vazby symbolů v lexikálním smyslu. Dále jsme museli rozšířit model vyhodnocování. Vyhodnocení elementů chápeme relativně vzhledem k prostředí. Samotné uživatelské procedury jsou reprezentovány jako trojice: seznam formálních argumentů, tělo procedury, prostředí vzniku procedury. Při každé aplikaci procedury vzniká nové lokální prostředí, jehož předek je nastaven na prostředí vzniku procedury. Ukázali jsme některé procedury s ustálenými názvy: identitu, projekce, konstantní procedury, procedury bez argumentů. Dále jsme se zabývali procedurami vyšších řádů, což jsou procedury, kterým jsou při aplikaci předávány jiné procedury jako argumenty nebo které vrací procedury jako výsledky své aplikace. Ukázali jsme princip rozkladu procedur dvou argumentů na proceduru jednoho argumentu vracející proceduru druhého argumentu. Dále jsme se zabývali vztahem procedur a zobrazení (matematických funkcí). Ukázali jsme, že za nějakých podmínek je možné chápat procedury jako (přibližnou) reprezentaci matematických funkcí a ukázali jsme řadu operací s funkcemi, které jsem schopni provádět na úrovni procedur vyšších řádů (kompozici procedur a podobně). Dále jsme se zabývali dvěma základními typy rozsahu platnosti – lexikálním (statickým) a dynamickým, ukázali jsme výhody lexikálního rozsahu a řadu nevýhod dynamického rozsahu platnosti (který se prakticky nepoužívá). V závěru lekce jsme ukázali nové speciální formy pomocí kterých je možné vytvářet složitější podmínky a složitější podmíněně vyhodnocené výrazy. Ukázali jsme rovněž, že všechny nové speciální formy jsou „nadbytečné“, protože je lze vyjádřit pomocí již dříve představeného typu podmínek.

Pojmy k zapamatování

- redundance kódu, uživatelsky definovatelné procedury,
- λ-výraz, formální argumenty, parametry, tělo, prázdný seznam,
- vázané symboly, volné symboly, vazby symbolů,
- identita, projekce, konstantní procedura, procedura bez argumentu,
- lokální prostředí, globální prostředí, předek prostředí, aktuální prostředí,
- vyhodnocení elementu v prostředí,
- aplikace uživatelsky definované procedury,
- procedura vyššího řádu, currying, elementy prvního řádu,
- pojmenované procedury, anonymní procedury,

[4] Podobná tvrzení jsou typickou ukázkou tak zvané „programátorské arogance“, což je obecně velmi nebezpečný projev samolibosti dost často se vyskytující u programátorů, kteří jakoby zapomínají, že nesou odpovědnost za funkčnost svých výtvorů.

- kompozice procedur, monoidální operace,
- nadřazené prostředí, lexikálně nadřazené prostředí, dynamicky nadřazené prostředí,
- lexikální (statický) rozsah platnosti, dynamický rozsah platnosti,
- konjunkce, disjunkce, negace.

Nově představené prvky jazyka Scheme

- speciální formy `lambda`, `and`, `or` a `cond`,
- procedury `abs`, `random`, `expt`, `not`, `min`, `max`,
- predikáty `even?`, `odd?`

Kontrolní otázky

1. *Co jsou to λ-výrazy a jak vypadají?*
2. *Co vzniká vyhodnocením λ-výrazů?*
3. *Jaké mají prostředí mezi sebou vazby?*
4. *Jak vznikají prostředí?*
5. *Jak se změní vyhodnocování, pokud jej uvažujeme vzhledem k prostředí?*
6. *Co jsou a jak jsou reprezentovány uživatelsky definovatelné procedury?*
7. *Jak probíhá aplikace uživatelsky definované procedury?*
8. *Co jsou to procedury vyšších řádů?*
9. *Jaký mají vztah procedury a matematické funkce?*
10. *Co máme na mysli pod pojmem monoidální operace?*
11. *Co v jazyku Scheme považujeme za elementy prvního řádu?*
12. *Jaký je rozdíl mezi lexikálním a dynamickým rozsahem platnosti?*
13. *Jaké má výhody lexikální rozsah platnosti?*
14. *Jaké má nevýhody dynamický rozsah platnosti?*
15. *Jak by se dal upravit náš interpret tak, aby pracoval s dynamickým rozsahem platnosti?*

Cvičení

1. Napište, jak interpret Scheme vyhodnotí následující symbolické výrazy:

```
(lambda () x)                          ⟼
((lambda (x) (+ x)) 20)                ⟼
(+ 1 ((lambda (x y) y) 20 30))         ⟼
((lambda () 30))                       ⟼
(* 2 (lambda () 20))                   ⟼
((lambda ()
   (+ 1 2)))                           ⟼
(and (lambda () x) 20)                 ⟼
(and (or) (and))                       ⟼
(if (and) 1 2)                         ⟼
(cond ((+ 1 2) 4)
      (else 5))                        ⟼
(and #f #f #f)                         ⟼
(and a #t b)                           ⟼
(not (and 10))                         ⟼
(((lambda (x)
    (lambda (y)
      x))
  10) 20)                              ⟼
((lambda (f)
```

```
   (f 20)) -)                    ⟹
((lambda (f g) (f (g)))
 + -)                            ⟹
((lambda (x y z) y) 1 2 3)       ⟹
(or #f ahoj-svete #t)            ⟹
(lambda () ((lambda ())))        ⟹
(* 2 ((lambda () 40) -100))      ⟹
```

2. Naprogramujte predikáty `positive?` a `negative?`, které jsou pro dané číslo pravdivé, právě když je číslo kladné respektive záporné.
3. Napište proceduru, která má jako argument libovolný element vyjadřující zobecněnou pravdivostní hodnotu a vrací pravdivostní hodnotu (nezobecněnou) tohoto argumentu.
4. Napište predikát `implies` dvou argumentů, který vrací pravdivostní hodnoty tvrzení „pravdivost prvního argumentu implikuje pravdivost druhého argumentu“. Úkolem je tedy napsat proceduru reprezentující pravdivostní funkci logické spojky *implikace*.
   (a) proceduru naprogramujte bez použití `if` a `cond`.
   (b) proceduru naprogramujte bez použití `and`, `or` a `not`.
5. Napište predikát `iff` dvou argumentů, který vrací pravdivostní hodnotu tvrzení „první argument je pravdivý, právě když je druhý argument pravdivý“. Úkolem je tedy napsat proceduru reprezentující pravdivostní funkci logické spojky *ekvivalence*.
   (a) proceduru naprogramujte bez použití `if` a `cond`.
   (b) proceduru naprogramujte bez použití `and` a `or`.
6. Naprogramujte proceduru `sum3g`, která bere jako argumenty tři čísla a vrací hodnotu součtu dvou větších čísel z těchto tří.
   (a) proceduru naprogramujte rozborem případů s použitím `cond`,
   (b) proceduru naprogramujte bez použití `if` a `cond`.
7. Upravte proceduru z programu 2.5 na straně 60 tak, aby nepoužívala `expt`. Hodnotu počítanou pomocí `expt` stanovte za pomocí procedur `log` (logaritmus při základu e) a `exp` (exponenciální funkce při základu e).
8. Obsah trojúhelníka lze snadno spočítat pokud známe velikosti všech jeho stran pomocí tak zvaného Heronova vzorce:

   $$S_\Delta = \sqrt{s(s-a)(s-b)(s-c)}, \quad \text{kde} \quad s = \frac{a+b+c}{2}.$$

   Napište proceduru `heron` se třemi argumenty, jimiž budou délky stran trojúhelníka, která vrací jeho obsah. Napište proceduru tak, aby byla hodnota s počítána *pouze jednou*.
9. Napište proceduru `diskr` na výpočet diskriminantu kvadratické rovnice. Dále napište procedury `koren-1` a `koren-2`, které vracejí první a druhý kořen kvadratické rovnice.
10. Procedury `koren-1` a `koren-2` vhodně zobecněte pomocí procedury vyššího řádu a ukažte jako lze pomocí ní původní procedury získat.

Úkoly k textu

1. Uvažujme proceduru `kdyz` definovanou následujícím způsobem.

```
(define kdyz
  (lambda (podminka vyraz alt)
    (if podminka vyraz alt)))
```

   Prostudujte proceduru a zjistěte, v čem se její chování liší od chování speciální formy `if`. Bez použití interpretu určete, zdali bude možné naprogramovat procedury `abs`, `sgn` a `max3` z této lekce pouze pomocí `kdyz` bez pomocí speciálních forem `cond`, `if`, `and` a `or`. Potom procedury naprogramujte a přesvědčte se, jestli byla vaše úvaha správná.

2. Dokažte pravdivost následujícího tvrzení.

   Pokud v daném λ-výrazu přejmenujeme všechny výskyty téhož vázaného symbolu jiným (ale pokaždé stejným) symbolem nevyskytujícím se v tomto λ-výrazu, pak se procedura vzniklá vyhodnocením tohoto λ-výrazu nezmění.

   Zdůvodněte, (i) proč předchozí tvrzení není možné rozšířit i na volné symboly, (ii) proč vyžadujeme, aby se jméno nového symbolu v λ-výrazu dosud nevyskytovalo, (iii) proč je potřeba nahradit *všechny výskyty* vázaného symbolu (a ne pouze některé).

3. Naprogramujte proceduru `curry-max3`, pomocí niž bude proveden currying procedury `max3` pro hledání maxima ze tří prvků. Účelem procedury `curry-max3` tedy bude rozložit proceduru tří argumentů `max3` na proceduru prvního argumentu vracející proceduru druhého argumentu vracející proceduru třetího argumentu. Svou implementaci důkladně otestujte.

Řešení ke cvičením

1. procedura, `20`, `31`, `30`, chyba, `30`, `20`, `#f`, `1`, `4`, `#f`, chyba, `#f`, `10`, `-20`, chyba, `2`, chyba, chyba, chyba
2. ```
   (define positive? (lambda (a) (> a 0)))
   (define negative? (lambda (a) (< a 0)))
   ```
3. ```
   (lambda (x) (not (not x)))
   ```
4. (a):
   ```
   (define implies (lambda (a b) (or (not a) b)))
   ```
   (b):
   ```
   (define implies (lambda (a b) (if a b #t)))
   ```
5. (a):
   ```
   (define iff
     (lambda (x y)
       (or (and x y)
           (and (not x) (not y)))))
   ```
   (b):
   ```
   (define iff
     (lambda (x y)
       (if x y (not y))))
   ```
6. (a):
   ```
   (define sum3g
     (lambda (x y z)
       (cond ((and (<= x y) (<= x z)) (+ y z))
             ((and (<= y x) (<= y z)) (+ x z))
             (else (+ x y)))))
   ```
   (b):
   ```
   (define sum3g
     (lambda (x y z)
       (- (+ x y z)
          (min x (min y z)))))
   ```
7. ```
   (define make-polynomial-function
     (lambda (a n)
       (lambda (x) (* a (exp (* n (log x)))))))
   ```
8. ```
   (define heron
     (lambda (a b c)
       ((lambda (s)
          (sqrt (* s (- s a) (- s b) (- s c))))
        (/ (+ a b c) 2))))
   ```

9.
```
(define diskr
  (lambda (a b c)
    (- (* b b) (* 4 a c))))

(define koren-1
  (lambda (a b c)
    (/ (+ (- b) (sqrt (diskr a b c)))
       (* 2 a))))

(define koren-2
  (lambda (a b c)
    (/ (- (- b) (sqrt (diskr a b c)))
       (* 2 a))))
```
10.
```
(define vrat-koren
  (lambda (op)
    (lambda (a b c)
      (/ (op (- b) (sqrt (diskr a b c)))
         (* 2 a)))))

(define koren-1 (vrat-koren +))
(define koren-2 (vrat-koren -))
```

Lekce 3: Lokální vazby a definice

Obsah lekce: V této kapitole objasníme důvody, proč potřebujeme lokální vazby, obeznámíme se se dvěma novými speciálními formami pro vytváření lokálních prostředí a vazeb v nich. Dále vysvětlíme pojem abstrakční bariéra a seznámíme se se dvěma důležitými styly vytváření programů: top-down a bottom-up. Dále rozšíříme λ-výrazy tak, abychom v jejich těle mohli používat speciální formu `define`, modifikovat tak lokální prostředí procedur a vytvářet interní definice.

Klíčová slova: abstrakční bariéry, blackbox, interní definice, `let*`-blok, `let`-blok, lokální vazba, rozsah platnosti, styl bottom-up, styl top-down, top-level definice

3.1 Vytváření lokálních vazeb

V prvních dvou lekcích jsme se seznámili s několika způsoby, jakými může být v jazyku Scheme hodnota navázána na symbol. Víme už, že v počátečním prostředí interpretu jazyka Scheme jsou na některé symboly navázány primitivní procedury, zabudované speciální formy a některé další elementy. Také již umíme zavádět nové vazby do globálního prostředí (respektive měnit stávající vazby) použitím speciální formy `define`. Dále víme, že při aplikaci uživatelské procedury vznikají vazby mezi formálními argumenty a hodnotami, na které je procedura aplikována. Tyto vazby přitom vznikají v lokálním prostředí aplikace této procedury, nikoli v globálním prostředí, jako v předchozích dvou případech. Vazby tohoto typu jsou tedy platné jen „v těle procedury“. Takovéto vazby nazýváme *lokální*. Právě lokálním vazbám se budeme věnovat v této lekci.

K čemu vlastně jsou lokální vazby? Důvody jsme si už vlastně řekli v lekci 1, kde jsme hovořili o abstrakcích vytvářených pojmenováním hodnot:

- Pojmenovaní hodnot může vést ke odstranění redundance kódu a *zvýšit tak jeho efektivitu*.
- Názvy symbolů nám mohou říkat, jakou roli jejich hodnota hraje. Tím se kód stává *čitelnějším*.

Demonstrujme si předchozí dva body na příkladech: Předpokládejme, že chceme napsat proceduru, která počítá povrch válce S podle jeho objemu V a výšky h. Ze vzorce pro výpočet objemu válce

$$V = \pi r^2 h$$

vyjádříme poloměr rotačního válce

$$r = \sqrt{\frac{V}{\pi h}}.$$

Poté takto stanovený poloměr dosadíme do vzorce na výpočet povrchu $S = 2\pi r(r + h)$. Všimněte si, že ve vzorci na výpočet povrchu se vyskytuje hodnota poloměru dvakrát. Jedním z řešení by bylo naprogramovat proceduru pro výpočet povrchu tak, že bychom vyšli ze vzorce pro výpočet S a nahradili v něm oba výskyty r vzorcem pro výpočet hodnoty r z objemu válce a výšky:

$$S = 2\pi\sqrt{\frac{V}{\pi h}}\left(\sqrt{\frac{V}{\pi h}} + h\right).$$

To ale není z programátorského hlediska příliš čisté řešení. Proceduru bychom mohli napsat také takto:

```
(define povrch-valce
  (lambda (V h)
    ((lambda (r)
       (* 2 r pi (+ r h)))
     (sqrt (/ V pi h)))))
```

V těle procedury `povrch-valce` tedy vytváříme novou proceduru vyhodnocením výrazu

`(lambda (r) (* 2 r pi (+ r h)))`.

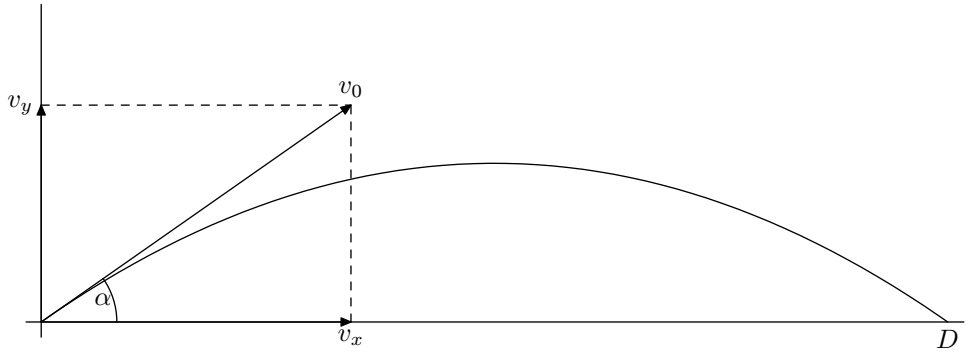
Tu hned aplikujeme na vyhodnocení výrazu na výpočet poloměru r. Nejde nám ani tak o proceduru samotnou, ale o lokální vazbu symbolu `r` na poloměr r, která vznikne její jednorázovou aplikací. V těle procedury

vzniklé vyhodnocením vnitřního λ-výrazu však můžeme zacházet s poloměrem jako s pojmenovanou hodnotou a neuvádět tak vícekrát stejnou část kódu (na výpočet poloměru).

Jiný příklad, kde se lokální vazba používá k tomuto účelu, je úkol č. 8 z předchozí lekce.

A teď druhý příklad. Tentokrát využijeme lokálních vazeb k pojmenování jednotlivých hodnot podle jejich rolí tak, abychom zlepšili čitelnost programu. Chtěli bychom proceduru, ve které počítáme délku šikmého vrhu (ve vakuu). Vstupními hodnotami pro nás bude úhel α, pod kterým je těleso vrženo a počáteční rychlost v_0. V těle této procedury přitom chceme používat symboly, které odpovídají proměnným v následujícím odvození: Okamžitá rychlost v je dána vektorovým součtem svislé $v_y = v\sin(\alpha)$ a vodorovné

Obrázek 3.1. Šikmý vrh ve vakuu



rychlosti $v_x = v\cos(\alpha)$. Vodorovná rychlost v_x je přitom stále stejná. Dostřel je tedy dán vodorovnou rychlostí v_x a časem T_d, po který těleso letí: $D = v_x T_d$. Let tělesa má dvě fáze. V první fázi letí těleso nahoru než dosáhne nulové svisle rychlosti. Pak přechází do druhé fáze a začíná padat. První fáze trvá po dobu $t_1 = \frac{v_y}{g}$, a těleso při ní dosáhne výšky $\frac{{v_y}^2}{2g}$. Doba, po kterou bude těleso z této výšky padat, je $t_2 = \frac{v_y}{g}$. Tedy $T_d = t_1 + t_2 = \frac{2v_y}{g}$. Přímočarým přepisem předchozího odvození a s využitím pomocné procedury pro vytvoření lokálních vazeb, naprogramujeme proceduru pro výpočet dostřelu takto:

```
(define dostrel
  (lambda (v0 alfa g)
    ((lambda (vx vy)
       (* 2 vx vy (/ g)))
     (* v0 (cos alfa))
     (* v0 (sin alfa)))))
```

Vidíme, že ačkoli bylo naším záměrem zpřehlednění kódu, dosáhli jsme efektu spíše opačného. A to jsme ještě nevytvářeli lokální vazbu na symbol `Td`, který by odpovídal proměnné T_d. Přehlednější, a přitom ekvivalentní, kód se dá napsat s použitím *speciální formy* `let`. Takto by vypadala procedura `dostrel` napsaná pomocí této speciální formy:

```
(define dostrel
  (lambda (v0 alfa g)
    (let ((vx (* v0 (cos alfa)))
          (vy (* v0 (sin alfa))))
      (* 2 vx vy (/ g)))))
```

Všimněte si, v čem vlastně spočívalo zpřehlednění programu. Jde především o to, že v kódu máme výrazy a symboly, na které budou vyhodnocení těchto výrazů lokálně navázány, uvedeny hned vedle sebe. Kdežto při použití λ-výrazu (jako v předchozím případě) jsou symboly v seznamu formálních argumentů, a jejich hodnoty jsou pak až za λ-výrazem.

Nyní se podrobně podíváme na použití a aplikaci speciální formy `let`.

Definice 3.1. Speciální forma `let` se používá ve tvaru:

```
(let ((⟨symbol_1⟩ ⟨hodnota_1⟩)
      (⟨symbol_2⟩ ⟨hodnota_2⟩)
       ⋮
      (⟨symbol_n⟩ ⟨hodnota_n⟩)))
 ⟨tělo⟩),
```

kde n je nezáporné celé číslo, $\langle symbol_1\rangle$, $\langle symbol_2\rangle$, ..., $\langle symbol_n\rangle$ jsou vzájemně různé symboly, $\langle hodnota_1\rangle$, $\langle hodnota_2\rangle$, ..., $\langle hodnota_n\rangle$ a $\langle tělo\rangle$ jsou libovolné S-výrazy. Tento výraz se nazývá `let`-blok (někdy též `let`-výraz). Vyhodnocení `let`-bloku v tomto tvaru je ekvivalentní vyhodnocení výrazu

```
((lambda (⟨symbol_1⟩ ⟨symbol_2⟩ ··· ⟨symbol_n⟩) ⟨tělo⟩) ⟨hodnota_1⟩ ⟨hodnota_2⟩ ··· ⟨hodnota_n⟩)
```

■

V definici 3.1 jsme zavedli sémantiku `let`-bloku tím, že jsme uvedli symbolický výraz, který se vyhodnotí stejně. Aplikaci speciální formy `let` v prostředí $\mathcal{P}$ bychom mohli však mohli popsat nezávisle takto:

(1) S-výrazy $\langle hodnota_1\rangle$, $\langle hodnota_2\rangle$ ... $\langle hodnota_n\rangle$ jsou vyhodnoceny *v aktuálním prostředí* $\mathcal{P}$. Pořadí jejich vyhodnocování přitom *není specifikováno*. Výsledky jejich vyhodnocení označme E_i, kde $i = 1, \ldots, n$.

(2) Vytvoří se *nové prázdné prostředí* $\mathcal{P}_l$, Tabulka vazeb prostředí $\mathcal{P}_l$ je v tomto okamžiku prázdná (neobsahuje žádné vazby), předek prostředí $\mathcal{P}_l$ není nastaven.

(3) *Nastavíme předka prostředí* $\mathcal{P}_l$ *na hodnotu* $\mathcal{P}$ (předkem prostředí $\mathcal{P}_l$ je *aktuální prostředí*).

(4) V prostředí $\mathcal{P}_l$ se zavedou vazby $\langle symbol_i\rangle \mapsto E_i$ pro $i = 1, \ldots, n$.

(5) Výsledek vyhodnocení je pak roven $\mathrm{Eval}[\langle tělo\rangle, \mathcal{P}_l]$.

Poznámka 3.2. (a) Zřejmě každý `let`-blok je symbolický výraz, protože je to, obdobně jako v případě λ-výrazů, seznam ve speciálně vyžadovaném tvaru. Příklady `let`-bloků jsou třeba

```
(let ((x 10) (y 20)) (+ x y 1))
(let ((x 10)) (* x x))
(let ((a 1) (b 2) (c 3)) 0)
  ⋮
```

Kvůli čitelnosti píšeme obvykle `let`-bloky do více řádků, každou vazbu na nový řádek a tělo rovněž zvlášť. Viz následující příklad přepisu prvního z výše uvedených `let`-bloků.

```
(let ((x 10)
      (y 20))
   (+ x y 1))
```

(b) Všimněte si, že definice 3.1 připouští i nulový počet dvojic `(`$\langle symbol_i\rangle$ $\langle hodnota_i\rangle$`)`, takže například symbolický výraz ve tvaru `(let () 10)` je také `let`-blok.

(c) Vyhodnocením `let`-bloku je provedena aplikace speciální formy navázané na symbol `let`. Je zřejmé, že element navázaný na symbol `let` nemůže být procedura. V takovém případě by vyhodnocení `let`-bloku ve tvaru `(let ((x 10)) (* x x))` skončilo chybou v kroku (B.e), protože symbol `x` nemá vazbu.

(d) Vidíme, že speciální forma `let` je nadbytečná, protože každý `let`-blok můžeme nahradit ekvivalentním kódem pomocí λ-výrazu. Umožňuje však přehlednější zápis kódu – a to především díky tomu, že symboly vázané `let`-blokem jsou uvedeny hned vedle hodnot, na které jsou navázány.

I když je forma `let` nadbytečná ve smyslu, že vše co lze pomocí ní v programu vyjádřit, bychom mohli vyjádřit i bez ní, v žádném případě to ale neznamená, že je snad „nepraktická“, právě naopak. Všechny soudobé programovací jazyky obsahují spoustu konstrukcí, které jsou tímto způsobem nadbytečné, ale výrazně ulehčují programátorům práci. Z trochou nadsázky můžeme říct, že kvalita a propracovanost programovacího jazyka se pozná právě pomocí množství a potenciálu (nadbytečných) konstrukcí, které jsou k dispozici programátorovi. Strohé programovací jazyky obsahující pouze nezbytně nutné minimum konstrukcí se v praxi nepoužívají a většinou slouží pouze jako teoretické výpočetní formalismy.

Příklad 3.3. (a) Výsledkem vyhodnocení `let`-bloku `(let ((a (+ 5 5))) (+ a 2))` je číslo 12. Průběh jeho vyhodnocování je následující. V aktuálním prostředí – v tomto případě v globálním prostředí – se vyhodnotí výraz `(+ 5 5)` na číslo `10`. Je vytvořeno nové prázdné prostředí $\mathcal{P}$, do něj je přidána vazba `a` $\mapsto_{\mathcal{P}}$ `10`, jako předek je nastaveno aktuální (globální) prostředí. V prostředí $\mathcal{P}$ pak bude vyhodnoceno tělo `(+ a 2)`. Vazba symbolu `+` na proceduru sčítání je nalezena v lexikálním předkovi prostředí $\mathcal{P}$, to jest v globálním prostředí. Vazbu `a` $\mapsto_{\mathcal{P}}$ `10` jsme přidali do lokálního prostředí. Výsledkem vyhodnocení je tedy 12.

(b) `let`-blok `(let ((+ (+ 10 20))) +)` se vyhodnotí tímto způsobem: v globálním prostředí se vyhodnotí výraz `(+ 10 20)` na číslo `30`. Je vytvořeno nové prázdné prostředí s vazbou `+` $\mapsto$ `30`, jako předek bude nastaveno aktuální (globální) prostředí. V novém prostředí pak bude vyhodnoceno tělo `+` a jelikož je v tomto prostředí `+` navázáno na hodnotu `30`, je číslo `30` výsledkem vyhodnocení celého výrazu. Vazba symbolu `+` na proceduru sčítání v globálním prostředí v tomto případě při vyhodnocení těla nehraje žádnou roli. Upozorněme na fakt, že vazba `+` v těle `key`-bloku je skutečně lokální, tedy globální definice `+` na primitivní proceduru sčítání čísel se vně `let`-bloku nijak nezmění.

(c) Uvažujme vyhodnocení následujících dvou výrazů z nichž druhý je `let`-blok.

```
(define x 10)
(let ((x (+ x 1))
      (y (+ x 2)))
  y)  ⟹  12
```

Nejprve jsou oba výrazy `(+ x 1)` a `(+ x 2)` vyhodnoceny v aktuálním (tedy globálním) prostředí. Jelikož je v tomto prostředí `x` navázáno na číslo `10`, budou výsledkem vyhodnocení čísla 11 a 12. Bude vytvořeno nové prázdné prostředí $\mathcal{P}$ a do něj budou přidány vazby `x` $\mapsto_{\mathcal{P}}$ 11 a `y` $\mapsto_{\mathcal{P}}$ 12. Tělo `let`-bloku `y` se pak vyhodnotí na svou aktuální vazbu, tedy číslo 12.

(d) Vyhodnocování výrazu `(let ((x 1) (x 2)) (* x 10))` skončí chybou „CHYBA: Vázáné symboly musí být vzájemně různé“.

Zopakujme, že výrazy $\langle hodnota_1 \rangle$, $\langle hodnota_2 \rangle$, ..., $\langle hodnota_n \rangle$ se vyhodnotí na elementy E_1, E_2, ..., E_n v *aktuálním prostředí*, až poté vznikají v novém prostředí vazby $\langle symbol_i \rangle \mapsto E_i$. To mimo jiné znamená, že tyto vazby nemohou ovlivnit vyhodnocovaní výrazů $\langle hodnota_1 \rangle$, ..., $\langle hodnota_n \rangle$. Za prvé, všechna vyhodnocení proběhnou dříve než vzniknou vazby. Za druhé, tato vyhodnocení proběhnou v prostředí, ze kterého na tyto vazby „nelze vidět“. Například v následujícím kódu

```
(define x 100)
(let ((x 10)
      (y (* 2 x)))
  (+ x y))  ⟹  210
```

máme `let`-výraz, ve kterém vážeme vyhodnocení výrazu `10` na symbol `x` a vyhodnocení výrazu `(* 2 x)` na symbol `y`. Přitom se výraz `(* 2 x)` vyhodnotí v globálním prostředí $\mathcal{P}_G$, kde je na `x` navázáno číslo `100` (viz obrázek 3.2), a výsledkem tohoto vyhodnocení bude číslo `100`. Proto je výsledkem vyhodnocení celého `let`-výrazu číslo `210`, nikoli číslo `30`.

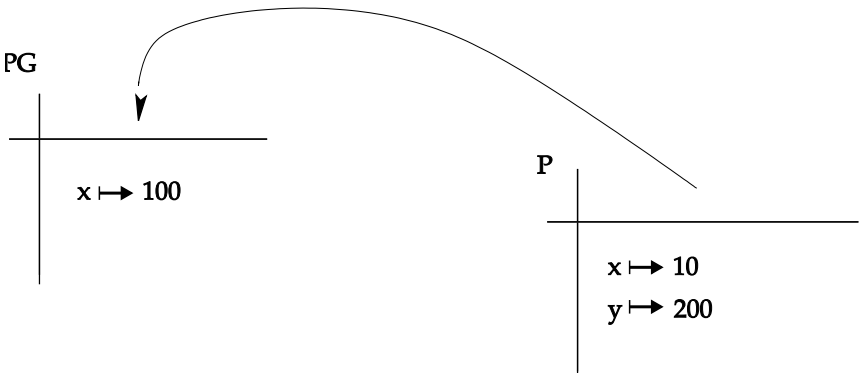
Ke stejnému závěru dospějeme po rozepsání `let`-bloku pomocí λ-výrazu:

```
(define x 100)
((lambda (x y) (+ x y)) 10 (* 2 x))
```

Příklad 3.4. Další ukázka představuje použití speciální formy `let` k vytvoření prostředí, kde jsou zaměněny vazby symbolů `+` a `*`. Jelikož jsou oba tyto symboly vyhodnoceny v globálním prostředí a až poté vznikají vazby, nemohou se tato vyhodnocení vzájemně ovlivnit.

```
(let ((+ *)
      (* +))
  (+ (* 10 10) 20))  ⟹  400
```

Obrázek 3.2. Vznik prostředí během vyhodnocení programu



Všimněte si, že kdybychom k podobné záměně chtěli dospět pomocí speciální formy `define` museli bychom použít nějaký pomocný symbol pro uchování jedné z vazeb. Například takto:

```
(define plus +)
(define + *)
(define * plus)
```

Předchozí sekvence používající `define` ale *neprovádí souběžnou záměnu vazeb* symbolů `+` a `*`.

Vraťme se nyní k sekci 2.6 – tam jsme si vysvětlili rozdíl mezi *lexikálním rozsahem platnosti symbolů* a *dynamickým rozsahem platnosti symbolů*. Také jsme uvedli jednoduchý program, který při vyhodnocení při lexikálním typu platnosti symbolů dával jiný výsledek než při dynamickém a na kterém tak byla vidět odlišnost obou typů rozsahů platnosti, viz příklad 2.18 na straně 63. Teď demonstrujme tuto odlišnost na jiném programu. Uvažujme, následující definici v globálním prostředí:

```
(define na2 (lambda (x) (* x x)))
```

a podívejme se na vyhodnocení následujícího výrazu:

```
(let ((+ *)
      (* +))
  (na2 10))
```

`let`-blok nám vytváří lokální prostředí, ve kterém jsou vzájemně zaměněny vazby symbolů `+` a `*` stejně jako v předchozím příkladu. V těle `let`-bloku `(na2 10)` se však ani jeden z těchto symbolů nevyskytuje. Při lexikálním rozsahu platnosti dostáváme výsledek `100`. To je velice přirozené, protože uvedené vazby se skutečně (z důvodu absence vázaných symbolů v těle) jeví jako zbytečné.

Nyní se podívejme na tento program při dynamickém rozsahu platnosti. V tomto případě dojde k nepříjemnému efektu, který byl popsán už v příkladě 2.18 na straně 63. A to k závislosti chování procedury na prostředí, ve kterém je aplikovaná. Při aplikaci procedury `na2` vyhodnocujeme její tělo v lokálním prostředí, jehož dynamickým předchůdcem je právě lokální prostředí vytvořené `let`-blokem. Vazba na symbol `*` není v prostředí procedury `na2` nalezena a je tedy hledána v tomto předchůdci. Tam je `*` navázán na proceduru sčítání čísel. Proto v případě dynamického rozsahu platnosti dostáváme výsledek `20`. Dostali jsme tedy opětně jiný výsledek než při lexikálním rozsahu platnosti.

Příklad 3.5. Další příklad, který si ukážeme, představuje použití speciální formy `let` k zapamatování si lexikální vazbu symbolu, která bude posléze globálně předefinována. V tomto příkladu chceme navázat na symbol `<=` proceduru porovnávající absolutní hodnoty dvou čísel. Absolutní hodnoty ale chceme porovnávat pomocí procedury, která je v globálním prostředí navázána právě na symbol `<=`. Řešení využívající `let`-blok vypadá takto:

```
(define <=
  (let ((<= <=))
```

```
    (lambda (x y)
      (<= (abs x) (abs y)))))
```

Pomocí speciální formy `let` vytvoříme nové lokální prostředí $\mathcal{P}$. V tomto novém prostředí bude na symbol `<=` navázáno vyhodnocení symbolu `<=` v aktuálním (tedy globálním) prostředí $\mathcal{P}_G$. Tam má symbol `<=` vazbu na proceduru porovnávání čísel. Jinými slovy, v lokálním prostředí vznikne stejná vazba na symbol `<=` jako je v globálním prostředí. Tělem tohoto `let`-bloku je λ-výraz `(lambda (x y) (<= (abs x) (abs y)))`. Ten bude vyhodnocen v tomto novém prostředí $\mathcal{P}$ a tedy $\mathcal{P}$ se stane prostředím vzniku procedury

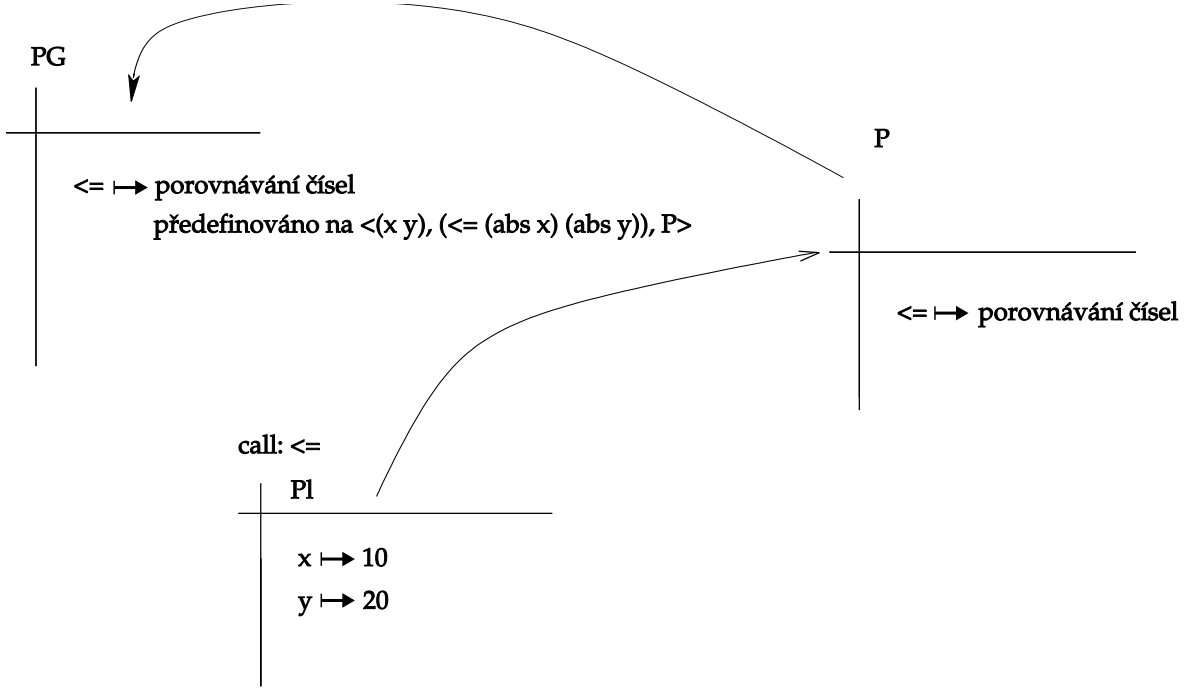
$\langle$`(x y)`, `(<= (abs x) (abs y))`, $\mathcal{P}\rangle$.

Ta pak bude navázána v globálním prostředí aplikací speciální formy `define` na symbol `<=`. Situace je schématicky zobrazena v obrázku 3.3. Rozdíl oproti původní proceduře porovnávání čísel je zřejmý:

```
(<= 10 20)     ⟹  #t        (<= 20 10)     ⟹  #f
(<= -10 20)    ⟹  #t        (<= -20 10)    ⟹  #f
(<= 10 -20)    ⟹  #t        (<= 20 -10)    ⟹  #f
(<= -10 -20)   ⟹  #t        (<= -20 -10)   ⟹  #f
```

Při aplikaci naší nové procedury je vytvořeno nové lokální prostředí $\mathcal{P}_l$, jsou v něm vytvořeny vazby na symboly `x` a `y`. A v něm bude vyhodnoceno tělo `(<= (abs x) (abs y))`. Vazba symbolu `<=` nebude nalezena v lokálním prostředí, takže bude hledána v prostředí předka, jímž je prostředí $\mathcal{P}$, jenž bylo vytvořeno během vyhodnocování `let`-bloku. V tomto prostředí má symbol `<=` vazbu na primitivní proceduru porovnávání čísel. To jest při vyhodnocení těla `(<= (abs x) (abs y))` skutečně dojde k porovnání dvou absolutních hodnot. Téhož efektu bychom dosáhli následujícím kódem, který je možná přehlednější, pro-

Obrázek 3.3. Vznik prostředí během vyhodnocení programu z příkladu 3.5



tože jsem v něm použili nový symbol jiného jména (odlišného od `<=`). Ale, jak jsme si už ukázali, zavádět nový symbol není nutné.

```
(define <=
  (let ((mensi-nez <=))
    (lambda (x y)
      (mensi-nez (abs x) (abs y)))))
```

Na závěr tohoto příkladu si ještě ukážeme, že následující program by k požadovanému cíli nevedl.

```
(define <=
  (lambda (x y)
    (<= (abs x) (abs y))))
```

Při aplikaci takto nadefinované procedury vniká nové lokální prostředí, jehož lexikální předchůdce je prostředí jejího vzniku – tedy globální prostředí $\mathcal{P}_G$. Při vyhodnocování těla procedury dojde k vyhodnocení symbolu `<=`. Vazba na něj je nalezena v $\mathcal{P}_G$ a při vyhodocování těla je tedy opět aplikována stejná procedura. Dochází tedy k nekonečící sérii aplikací procedury `<=` a výpočet tak nikdy neskončí.

Jak už jsme několikrát řekli, vyhodnocování jednotlivých výrazů $\langle hodnota_i \rangle$ určujících hodnoty vazeb v `let`-bloku se navzájem neovlivňují. Někdy ale nastávají situace, kdy jistý typ ovlivnění požadujeme. Můžeme toho dosáhnout třeba postupným vnořováním `let`-bloků:

```
(let ((x 10))
  (let ((y (* x x)))
    (let ((z (- y x)))
      (/ z y))))  ⟹  9/10
```

Nebo můžeme použít další speciální formu `let*`:

```
(let* ((x 10)
       (y (* x x))
       (z (- y x)))
  (/ z y))  ⟹  9/10
```

Aplikaci této speciální formy si můžeme představit takto. Forma `let*` se chová analogicky jako `let`, ale při jejím použití se výrazy $\langle hodnota_1 \rangle, \langle hodnota_2 \rangle, \ldots, \langle hodnota_n \rangle$ nevyhodnocují v nespecifikovaném pořadí a vazby se neprovádějí „současně“. Naopak vyhodnocování výrazů (i vznik vazeb) se provádí „postupně“ v pořadí v jakém jsou uvedeny. Přitom vyhodnocení každého výrazu $\langle hodnota_i \rangle$ probíhá v takovém okamžiku a v takovém prostředí, že vazby symbolů $\langle symbol_j \rangle$ na vyhodnocení $\langle hodnota_j \rangle$ pro $j < i$ jsou již „viditelné“.

Definice 3.6. Syntaxe speciální formy `let*` je stejná, jako u speciální formy `let` (až na jména symbolů):

```
(let* ((⟨symbol_1⟩ ⟨hodnota_1⟩)
       (⟨symbol_2⟩ ⟨hodnota_2⟩)
          ⋮
       (⟨symbol_n⟩ ⟨hodnota_n⟩)))
  ⟨tělo⟩),
```

kde n je nezáporné celé číslo, $\langle symbol_1 \rangle, \ldots, \langle symbol_n \rangle$ jsou symboly a $\langle hodnota_1 \rangle, \ldots, \langle hodnota_n \rangle$, $\langle tělo \rangle$ jsou libovolné výrazy. Tento výraz budeme nazývat `let*`-*blokem*.

Aplikaci speciální formy `let*` zavedeme pomocí speciální formy `let`.

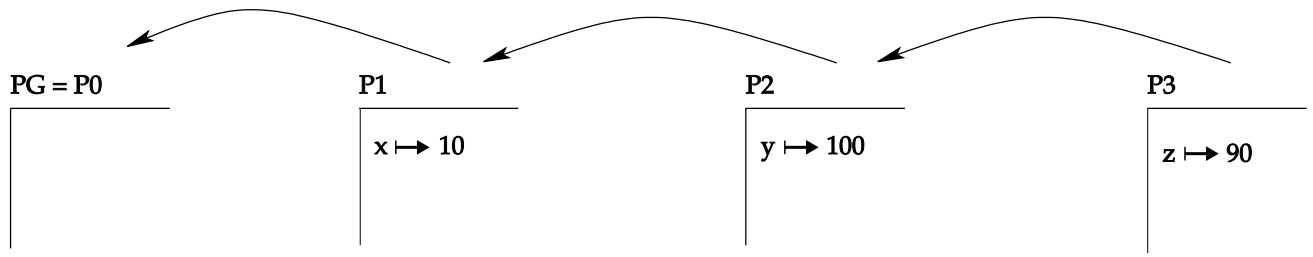
- Jestliže $n = 0$ nebo $n = 1$, je vyhodnocení stejné, jako u speciální formy `let`.
- Jinak je vyhodnocení stejné jako vyhodnocení následujícího výrazu:

```
(let ((⟨symbol_1⟩ ⟨hodnota_1⟩))
  (let* ((⟨symbol_2⟩ ⟨hodnota_2⟩))
            ⋮
        ((⟨symbol_n⟩ ⟨hodnota_n⟩)))
    ⟨tělo⟩))
```

■

Během vyhodnocování `let*`-bloku tedy nevzniká jen jedno prostředí, jako v případě `let`-bloku, nýbrž celá hierarchie prostředí. Je postupně vytvářeno nové prostředí $\mathcal{P}_i$ pro každou dvojici `(`$\langle symbol_i \rangle$ $\langle hodnota_i \rangle$`)`, kde $i = 1, \ldots, n$. Označme aktuální prostředí jako $\mathcal{P}_0$. Předchůdcem prostředí $\mathcal{P}_i$ je prostředí $\mathcal{P}_{i-1}$, pro $i = 1, \ldots, n$. Při vzniku každého z prostředí $\mathcal{P}_i$ je do něj vložena nová vazba symbol $\langle symbol_i \rangle$ na vyhodnocení výrazu $\langle hodnota_i \rangle$ v prostředí $\mathcal{P}_{i-1}$. Poté je nastaven jeho lexikální předek na $\mathcal{P}_{i-1}$. Například prostředí vytvořená vyhodnocením následujícího `let*`-bloku budou vypadat tak, jak je to znázorněno na obrázku 3.4.

Obrázek 3.4. Hierarchie prostředí




```
(let* ((x 10)
       (y (* x x))
       (z (- y x)))
  (/ z y))  ⟹  9/10
```

Hierarchii prostředí si také dobře uvědomíme, pokud předchozí `let*`-blok rozepíšeme pomocí `let`-bloků a ty dále rozepíšeme pomocí λ-výrazů:

```
((lambda (x)
   ((lambda (y)
      ((lambda (z)
         (/ z y))
       (- y x)))
    (* x x)))
 10)  ⟹  9/10
```

Poznámka 3.7. (a) Všimněte si, že narozdíl od `let`-bloku, nemusejí být symboly $\langle symbol_i \rangle$ vzájemně různé. Třeba výraz ve tvaru `(let* ((x 10) (x 20)) x)` je `let*`-blok.

(b) Sémantika `let*`-bloku by samozřejmě šla popsat, podobně jako u `let`-bloku, v bodech popisujících provádění jednotlivých vazeb:

(0) Označme aktuální prostředí $\mathcal{P}_0$, nastavme i na 1.

(1) Pokud $n = 0$, tedy seznam dvojic $(\langle symbol_i \rangle\ \langle vazba_i \rangle)$ je prázdný, je vytvořeno nové prázdné prostředí, jako jeho předchůdce je nastaveno aktuální prostředí $\mathcal{P}_0$ a v tomto prostředí je vyhodnoceno tělo `let*`-bloku. Tím aplikace speciální formy `let*` končí. V opačném případě $n > 0$ pokračujeme bodem (2).

(2) Pokud platí $i > n$ (to jest, už jsme prošli všechny dvojice $(\langle symbol_i \rangle\ \langle hodnota_i \rangle)$), je v prostředí $\mathcal{P}_i$ vyhodnoceno tělo `let*`-bloku. Tím aplikace `let*` končí. V opačném případě ($i \leq n$) pokračujeme bodem (3).

(3) Vyhodnotí se výraz $\langle hodnota_i \rangle$ v prostředí $\mathcal{P}_{i-1}$, výsledek vyhodnocení označme E_i. Dále pokračujeme bodem (4).

(4) Vytvoří se nové prázdné prostředí $\mathcal{P}_i$, je do něj vložena vazba $\langle symbol_i \rangle \mapsto E_i$, předek prostředí $\mathcal{P}_i$ je nastaven na $\mathcal{P}_{i-1}$. Pokračujeme bodem (5).

(5) Zvýšíme i o 1 a pokračujeme bodem (2).

(c) Při aplikaci speciální formy `let*` vzniká tolik prostředí, kolik je dvojic $(\langle symbol_i \rangle\ \langle vazba_i \rangle)$. Zvláštním případem je, pokud je tento počet nulový. V takovém případě vzniká jedno prostředí.

Jak vyplývá z definic 3.1 a 3.6 jsou speciální formy představené `let` a `let*` v této sekci „nadbytečné“. Jejich sémantiku jsme totiž schopni vyjádřit pomocí speciální formy `lambda`. Použití forem `let` a `let*`

ale umožňuje přehlednější zápis kódu. Větší přehlednosti je dosaženo především tím, že symboly vázané `let`-blokem (popřípadě `let*`-blokem) jsou uvedeny hned vedle hodnot, na které jsou navázány.

Příklad 3.8. (a) Uvažujme následující `let*`-bloky a jejich vyhodnocení:

```
(let* ((a (+ 5 5))) (+ a 2))  ⟹  12
(let* ((+ (+ 10 20))) +)      ⟹  30
```

To jest výsledkem vyhodnocení `let*`-bloků jsou čísla `12` a `30` (v tomto pořadí). Jejich vyhodnocení je tedy stejné jako v případě speciální formy `let` – to bylo podrobně rozepsáno v příkladě 3.3 v bodech (a) a (b).

(b) Vyhodnocení následujícího `let*` výrazu už bude odlišné.

```
(let* ((x 10)
       (y (* 2 x)))
  (+ x y))  ⟹  30
```

Zatímco aplikace speciální formy `let` by skončila chybou „CHYBA: Symbol `x` nemá vazbu“, aplikace formy `let*` proběhne následovně: v globálním prostředí je vyhodnocen výraz `10` na hodnotu `10`. Pak je vytvořeno nové prázdné prostředí $\mathcal{P}_1$, do něhož je přidána vazba na tento element, tedy vazba `x` $\mapsto_{\mathcal{P}_1}$ `10`. Jako lexikální předek tohoto prostředí je nastaveno globální prostředí. V tomto prostředí je vyhodnocen výraz `(* 2 x)` na hodnotu `20`. Je vytvořeno další prázdné prostředí $\mathcal{P}_2$ a do něj je vložena vazba `y` $\mapsto_{\mathcal{P}_2}$ `20`. Předkem tohoto prostředí bude $\mathcal{P}_1$. V prostředí $\mathcal{P}_2$ je konečně vyhodnoceno tělo `let*`-výrazu `(+ x y)`, na výslednou hodnotu `30`. K témuž bychom došli po rozepsání `let*`-bloku podle definice 3.6:

```
(let ((x 10))
  (let ((y (* 2 x)))
    (+ x y)))  ⟹  30
```

a k témuž dojdeme i při nahrazení `let`-bloků λ-výrazy:

```
((lambda (x)
   ((lambda (y)
      (+ x y))
    (* 2 x)))
 10)  ⟹  30
```

(c) Při použití speciální formy `let` by následující výraz nebyl `let`-blokem, protože vázané symboly použité v bloku nejsou vzájemně různé.

```
(let* ((x 10)
       (x (+ x 20)))
  (+ x 1))     ⟹  31
```

Definice `let*`-bloku ale připouští i stejné symboly, a proto je tento výraz `let*`-blokem.

3.2 Rozšíření λ-výrazů a lokální definice

Doteď jsme se zabývali jen tím jak vytvořit nové lokální prostředí, ne však tím, jak vytvářet nové vazby v již existujících lokálních prostředích ani tím, jak měnit vazby v existujících lokálních prostředích.

Ze sekce 1.6 již známe speciální formu `define`, jejíž aplikace má *vedlejší efekt*. Tímto vedlejším efektem je modifikace prostředí. Přesněji vyhodnocením `(define name` $\langle$*výraz*$\rangle$`)` v prostředí $\mathcal{P}$ je do prostředí $\mathcal{P}$ vložena vazba symbolu `name` na výsledek vyhodnocení argumentu $\langle$*výraz*$\rangle$: `name` $\mapsto$ Eval($\langle$*výraz*$\rangle$, $\mathcal{P}$) Dosud jsme ale tuto speciální formu používali jen k modifikaci globálního prostředí. Nyní si ukážeme, jak modifikovat i lokální prostředí. Abychom toho dosáhli, potřebujeme aplikovat speciální formu `define` v tom prostředí, které chceme modifikovat. Jinými slovy, potřebujeme vyhodnotit definici v těle procedury. Dále musíme zajistit, aby tato define byla vyhodnocena dříve, než část kódu, ve které chceme vzniklou vazbu využívat. Tento druhý bod je tím, co dělá lokální definice netriviální. Uvědomme si totiž, že nám principiálně nic nebrání vytvořit například proceduru vzniklou vyhodnocením následujícího λ-výrazu:

```
(lambda (x) (define y 20))
```

Při aplikaci této procedury vznikne lokální prostředí $\mathcal{P}$ v němž bude formální argument `x` navázaný na předanou hodnotu. Při vyhodnocení těla procedury v tomto lokálním prostředí dojte k aplikaci `define` a zavedení nové vazby `y` $\mapsto_{\mathcal{P}}$ `20` v lokálním prostředí. Tato vazba ale není nijak užitečná, protože výsledkem vyhodnocení těla (to jeste speciální formy `define`) je *nedefinovaná hodnota*, která je vrácena jako výsledek aplikace celé procedury. Musíme tedy zajistit vytvoření lokální definice a možnost jejího dalšího netriviálního použití.

K tomuto účelu můžeme například použít speciální formu `lambda`. Viz následující program:

```
(lambda ()
  ((lambda (nepouzity-symbol)
     (+ 10 x))
   (define x 20)))
```

Vyhodnocením tohoto λ-výrazu, vznikne procedura. Při aplikaci této procedury je v jejím lokálním prostředí $\mathcal{P}$ vyhodnoceno tělo, tedy výraz `((lambda (ingorovany-symbol) (+ x 10)) (define x 20))`. Vyhodnocením prvního prvku tohoto seznamu dostáváme proceduru, která ignoruje svůj jediný argument a vrací součet čísla `10` a hodnoty navázané na symbol `x`. Prvním prvkem seznamu (těla aplikované procedury) je tedy procedura, vyhodnotí se tedy zbývající prvek seznamu: seznam `(define x 20)`. Jako vedlejší efekt vyhodnocení tohoto výrazu je vytvoření vazby symbolu `x` $\mapsto_{\mathcal{P}}$ `20`. V okamžiku vyhodnocení těla `(+ 10 x)` vnitřního λ-výrazu je tedy vazba na `x` už zavedena.

Jiná možnost je využití aplikace speciální formy `and`:

```
(lambda ()
  (and (define x 10)
       (define y (- x 2))
       (define z (/ x y))
       (+ x y z)))
```

Speciální forma `and`, jak bylo popsáno v definici 2.22, vyhodnocuje své argumenty v pořadí, v jakém jsou uvedeny – zleva doprava. Jelikož výsledkem aplikace speciální formy `define` je *nedefinovaná hodnota*, tedy element různý od `#f`, jsou postupně vyhodnoceny všechny tři definice i výraz `(+ x y z)`. A jelikož se jedná o poslední argument formy `and`, je výsledek vyhodnocení výrazu `(+ x y z)` také výsledkem aplikace formy `and`.

Například proceduru `dostrel` bychom mohli napsat tak, jak je uvedeno v programu 3.1. Ve skutečnosti

Program 3.1. Procedura `dostrel` s lokálními vazbami vytvářenými s využitím `and`.

```
(define dostrel
  (lambda (v0 alfa g)
    (and (define vx (* v0 (cos alfa)))
         (define vy (* v0 (sin alfa)))
         (define Td (* 2 vy (/ g)))
         (* vx Td))))
```

nemusíme použití definic v těle λ-výrazu řešit takovou oklikou, jako je využití jiné speciální formy. Stačí když uděláme následující změny: Rozšíříme λ-výraz tak, aby se jeho tělo mohlo skládat z více než jednoho výrazu a stejným způsobem upravíme definici procedury. Dále upravíme popis aplikace speciální formy `lambda` a aplikaci uživatelsky definovatelné procedury. Pak můžeme proceduru `dostrel` definovat tak, jak je ukázáno v programu 3.2.

Nyní rozebereme uvedené změny podrobněji. V definici 2.1 jsme zavedli tělo λ-výrazu jako jeden libovolný S-výraz. Od tohoto okamžiku budeme uvažovat, že se tělo λ-výrazu bude skládat z libovolného nenulového počtu symbolických výrazů:

Program 3.2. Procedura `dostrel` s lokálními vazbami vytvářenými pomocí speciální formy `define`.

```
(define dostrel
  (lambda (v0 alfa g)
    (define vx (* v0 (cos alfa)))
    (define vy (* v0 (sin alfa)))
    (define Td (* 2 vy (/ g)))
    (* vx Td)))
```

Definice 3.9 (**λ**-výraz)**.** Každý seznam ve tvaru

(lambda (⟨*param*$_1$⟩ ⟨*param*$_2$⟩ ⋯ ⟨*param*$_n$⟩) ⟨*výraz*$_1$⟩ ⟨*výraz*$_2$⟩ ⋯ ⟨*výraz*$_m$⟩),

kde n je nezáporné číslo, m je kladné číslo, ⟨*param*$_1$⟩, ⟨*param*$_2$⟩, …, ⟨*param*$_n$⟩ jsou vzájemně různé symboly a ⟨*výraz*$_1$⟩, ⟨*výraz*$_2$⟩, …, ⟨*výraz*$_m$⟩ jsou symbolické výrazy, tvořící *tělo*, se nazývá *λ-výraz* (*lambda výraz*). Symboly ⟨*param*$_1$⟩, …, ⟨*param*$_n$⟩ se nazývají *formální argumenty* (někdy též *parametry*). Číslo n nazýváme *počet formálních argumentů* (*parametrů*). ■

Vyhodnocením v prostředí $\mathcal{P}$ tohoto výrazu vznikne uživatelsky definovaná procedura reprezentovaná trojicí ⟨(⟨*param*$_1$⟩ ⟨*param*$_2$⟩ ⋯ ⟨*param*$_n$⟩); ⟨*výraz*$_1$⟩, ⟨*výraz*$_2$⟩, …, ⟨*výraz*$_m$⟩; $\mathcal{P}$⟩ stejně tak, jak je popsáno v předchozí lekci. Až na ten detail, že tělo zde není interní reprezentace jednoho S-výrazu, ale jednoho nebo více S-výrazů ⟨*výraz*$_i$⟩.

Aplikace takové procedury je totožná s aplikací popsanou v definici 2.12, a to až na bod 5. Zde se vyhodnotí všechny S-výrazy, z nichž se skládá tělo procedury, v prostředí $\mathcal{P}_l$. Vyhodnocují se postupně v tom pořadí, v jakém jsou v těle procedury uvedeny. Výsledkem je pak vyhodnocení *posledního z nich*.

Všimněte si, že výsledky vyhodnocení S-výrazů v těle – až na poslední z nich – jsou „zapomenuty“. Předpokládá se, že při vyhodnocení těchto S-výrazů dojde k nějakému vedlejšímu efektu, jako je například modifikace aktuálního prostředí při aplikaci speciální formy `define`. Takto užité definice nazýváme *interní* (též *vnitřní*) *definice*. Definice v globálním prostředí nazýváme pro odlišení *globálními* (též *top-level*) *definicemi*.

Příklad 3.10. Nyní si to, co jsme v této sekci řekli, ukážeme na vyhodnocení následujícího kódu:

```
(define proc
  (lambda (x)
    (define y 20)
    (+ x y)
    (* x y)))

(proc 10)  ⟹  200
```

Jedná se o aplikaci procedury, jejíž tělo je tvořeno třemi výrazy: `(define y 20)`, `(+ x y)` a `(* x y)`. Vytvoří se tedy nové lokální prostředí $\mathcal{P}$, v němž bude vytvořena vazba `x` $\mapsto$ `10` a jehož předchůdce bude nastaven na globální prostředí $\mathcal{P}_G$. V tomto lokálním prostředí se budou vyhodnocovat všechny výrazy z těla. Vyhodnocují zleva doprava, tak jak jsou uvedeny – nejdříve se tedy vyhodnotí interní definice a v lokálním prostředí $\mathcal{P}$ procedury bude provedena vazba `y` $\mapsto$ `20`. Dále se vyhodnocuje výraz `(+ x y)`, jeho výsledkem bude číslo `30`, ale protože se nejedná o poslední výraz z těla je tento výsledek ignorován a pokračujeme dalším – posledním výrazem `(* x y)`. Výsledkem vyhodnocení tohoto výrazu je číslo `200`, a protože se jedná o poslední výraz těla procedury, je toto číslo i výsledkem Eval[`(proc 10)`, $\mathcal{P}_G$]. Všimněte si, že výraz `(+ x y)` nemá žádný význam. Jeho výsledek je zapomenut a přitom nemá ani žádný vedlejší efekt. Vazby vytvořené interními definicemi samozřejmě nejsou viditelné zvenčí. Kdybychom se pokusili vyhodnotit v globálním prostředí třeba výraz `(+ y 1)`, skončili bychom chybou „CHYBA: Symbol `y` nemá vazbu“.

Poznámka 3.11. (a) Týmž způsobem, jakým jsme právě rozšířili definici λ-výrazu, rozšiřujeme i definici `let`-bloku a `let*`-bloku. Tedy ⟨*tělo*⟩ `let`-bloku i `let*`-bloku se může skládat z více než jednoho S-výrazu. Rozšíření `let`-bloku a `let*`-bloku je vlastně automatické, protože jejich sémantiku jsme popisovali přes λ--výrazy a aplikaci procedur vzniklých jejich vyhodnocováním.

(b) Specifikace R^{5}RS jazyka Scheme – viz [R5RS] – říká, že interní definice se mohou objevit na začátku těla λ-výrazu (popř. `let`-bloku, let*-výrazu.) To znamená, že například vyhodnocování λ-výrazu

```
(lambda (x) (+ x 1) (define y 2) (+ x y))
```

v některých interpretech může skončit chybou „CHYBA: Speciální forma `define` je použita ve špatném kontextu“. V našem abstratním interpretru se tímto nebudeme omezovat a budeme připouštět i interní definice na jiném místě v těle λ-výrazu. Stejně tak je připouštějí například interprety Elk a Bigloo. Nepovoluje je třeba MIT Scheme.

(c) Některé interpretry dokonce mimo omezení popsané v (b) neumožňují v jednom těle definovat vazbu na symbol a pak ji pomocí další interní definice změnit. Příklady takových interpretů jsou MIT Scheme a Guile. Třeba vyhodnocování výrazu

```
(lambda () (define x 10) (define y 10) (define x 20) y)
```

by skončilo chybou.

Příklad 3.12. (a) Podívejme se na program 3.2 – tedy implementaci procedury `dostrel` s použitím interních definic. Vyhodnocení tohoto kódu bude mít za efekt navázání nové procedury na symbol `dostrel`. Při vyvolání této procedury vznikne nové prostředí a v něm vazby na formální argumenty `v0`, `alfa` a `g`. V tomto prostředí se postupně vyhodnocují všechny výrazy v těle. Vedlejším efektem jsou postupně do lokálního prostředí přidány vazby na symboly `vx`, `vy` a `Td`. Důležité je, že třetí vnitřní definice se vyhodnocuje až po vyhodnocení druhé. V té době už tedy existuje vazba na symbol `vy`. Výsledek vyhodnocení posledního výrazu v těle `(* vx Td)` je pak výsledkem aplikace procedury.

(b) Následující `let`-blok se vyhodnotí na číslo `30`:

```
(let ()
  (define x (lambda () y))
  (define y 10)
  (x))  ⟼  30
```

Do prázdného prostředí vytvořeného při aplikaci `let`-bloku, je vyhodnocením vnitřní definice

```
(define x (lambda () y))
```

přidána vazba `x` na proceduru. Tato procedura nebere žádný argument a vrací vždy vyhodnocení symbolu `y` a prostředím jejího vzniku je právě toto prostředí. Poté je do tohoto prostředí přidána vazba na symbol `y`. Při vyvolání procedury `x` v posledním výrazu v těle už je tedy symbol `y` navázán. Výsledkem je číslo `30`.

3.3 Příklady na použití lokálních vazeb a interních definic

Nyní ukážeme několik příkladů, v nichž je vhodné použít lokální vazby a definice. A to na jednoduché finanční aritmetice: Střádání – na počátku každého úrokovacího období se pravidelně ukládá částka a a na konci období se k úsporám připisuje úrok ve výši $p\%$ úspor. Po n obdobích vzroste vklad na částku a_n danou

$$a_n = ar\frac{r^n - 1}{r - 1}, \text{ kde } r = \left(1 + \frac{p}{100}\right).$$

Následující program vypočítá hodnotu a_n s využitím lokální vazby vytovřené v `let`-bloku:

```
(define sporeni
  (lambda (a n p)
    (let ((r (+ 1 (/ p 100.0))))
      (/ (* a r (- (expt r n) 1)) (- r 1)))))
```

Při aplikaci procedury se vytvoří lokální prostředí s vazbami na formální argumenty `a`, `n` a `p`. V uvedeném vzorci se nám ale vyskytuje několikrát r, které musíme dopočítat. Místo toho, abychom zbytečně psali na každém místě místo r výraz `(+ 1 (/ p 100.0))`, vytvořili jsme vazbu symbolu `r` na vyhodnocení tohoto výrazu. Tím jsme odstranili redundanci (stejný výraz by se nám zde vyskytoval třikrát). Vazba na symbol `r` vzniká v novém lokálním prostředí vyhodnocením `let`-bloku. A v tomto prostředí je také vyhodnoceno tělo procedury.

Lokální vazbu bychom mohli vytvořit i pomocí interní definice:

```
(define sporeni
  (lambda (a n p)
    (define r (+ 1 (/ p 100.)))
    (/ (* a r (- (expt r n) 1)) (- r 1))))
```

Toto řešení je také správné. V tomto případě ale nevzniká nové prostředí. Aplikací formy `define` Je modifikováno aktuální prostředí – tedy prostředí procedury `sporeni`. K vazbám vytvořených navázáním argumentů tak přibude nová vazba na symbol `r`.

Proceduru `sporeni` je vhodné upravit na proceduru vyššího řádu. Takto upravenou procedurou je pak možné vytvářet procedury na výpočet úspor při různém úročení.

```
(define sporeni
  (lambda (p)
    (let ((r (+ 1 (/ p 100))))
      (lambda (a n)
        (/ (* a r (- (expt r n) 1)) (- r 1))))))
```

Pomocí lokálních definic můžeme také upravit program 2.7 na straně 62, kde jsme implementovali proceduru `derivace`, která jako výsledek vracela přibližnou derivaci. Tato procedura používala pomocnou proceduru `smernice`. Tento program můžeme upravit následujícím způsobem. Z definice procedury `smernice` uděláme interní definici.

```
(define derivace
  (lambda (f delta)

    (define smernice
      (lambda (f a b)
        (/ (- (f b) (f a))
           (- b a))))

    (lambda (x)
      (smernice f x (+ x delta)))))
```

Kód můžeme ještě začistit, a to následujícím způsobem: parametr `f`, nemusíme předávat proceduře `smernice`. Ta totiž vzniká v prostředí, ve kterém je symbol `f` už navázán, to jest v prostředí procedury `derivace`, viz program 3.3.

Pro úplnost uvádíme variantu pomocí `let`. Viz program 3.4

Další ukázkou bude procedura na výpočet nákladů vydaných za určitý počet kusů nějakého výrobku (`pocet-kusu`), který má nějakou cenu (`cena-kus`). V nejprimitivnější formě vypadá implementace takto:

```
(define naklady
  (lambda (cena-kus pocet-kusu)
    (* cena-kus pocet-kusu)))
```

Nyní zahrneme nový fakt. A to, že při zakoupení daného množství kusů (`sleva-kus`) dostaneme množstevní slevu (`sleva-%`):

```
(define naklady
```

Program 3.3. Procedura `derivace` s použitím interní definice.

```
(define derivace
  (lambda (f delta)

    (define smernice
      (lambda (a b)
        (/ (- (f b) (f a))
           (- b a))))

    (lambda (x)
      (smernice x (+ x delta)))))
```

Program 3.4. Procedura `derivace` s použitím speciální formy `let`.

```
(define derivace
  (lambda (f delta)
    (let ((smernice
           (lambda (a b)
             (/ (- (f b) (f a))
                (- b a)))))
      (lambda (x)
        (smernice x (+ x delta))))))
```

V mnoha případech se vyplatí vytvářet pomocné procedury v těle hlavních procedur. Často tak lze snížit celkový počet předávaných argumentů. To díky tomu, že elementy, s kterými pracuje pomocná procedura, mohou být navázány symbol v prostředí vzniku této procedury. Pomocná procedura tedy může některé symboly vázat ve svém prostředí a některé brát z prostředí nadřazeného, to jest prostředí hlavní procedury. Snížením počtu předávaných argumentů dochází k zpřehlednění kódu.

```
(lambda (cena-kus pocet-kusu sleva-kus sleva-%)
  (- (* cena-kus pocet-kusu)
     (if (>= pocet-kusu sleva-kus)
         (* cena-kus pocet-kusu (/ sleva-% 100))
         0))))
```

Tedy od původní ceny, která je `(* cena-kus pocet-kusu)`, odečteme buďto nulu, nebo slevu o velikosti `(* cena-kus pocet-kusu (/ sleva-% 100))`. To v závislosti na tom, jestli počet kupovaných kusů překročil či nepřekročil množství potřebné na slevu: `(>= pocet-kusu sleva-kus)`.

Ačkoli se vlastně jedná o jednoduchou proceduru, její kód je docela nepřehledný. Navíc zde dvakrát počítáme součin `(* cena-kus pocet-kusu)`. Proto jej nahradíme následujícím kódem:

```
(define naklady
  (lambda (cena-kus pocet-kusu sleva-kus sleva-%)
    (let* ((bez-slevy (* cena-kus pocet-kusu))
           (sleva-mult (/ sleva-% 100))
           (sleva (* sleva-mult bez-slevy)))
      (if (>= pocet-kusu sleva-kus)
          (- bez-slevy sleva)
```

```
        bez-slevy))))
```

Použitím lokálních vazeb jsme kód zpřehlednili tím, že jsme hodnoty pojmenovali podle rolí, které mají. Také jsme odstranili redundanci v kódu.

3.4 Abstrakční bariéry založené na procedurách

V této sekci se nebudeme zabývat jazykem Scheme, ale zamyslíme se nad problémy souvisejícími s vytvářením velkých programů. Při vytváření větších programů je potřeba programové celky vhodně strukturovat a organizovat, aby se v nich programátoři vyznali. Při psaní jednoduchých programů, se kterými jsme se zatím setkali, tato potřeba není příliš markantní. Při programování velkých programů však rychle vzrůstá jejich složitost a hrozí postupné snižování čitelnosti programu a následné zanášení nechtěných chyb do programu vlivem neznalosti nebo nepochopení některých jeho částí.

Předchozího jevu si programátoři všimli záhy po vytvoření vyšších programovacích jazyků a po jejich nasazení do programátorské praxe. Velmi brzy se proto programátoři začali zamýšlet nad metodami strukturování programů do menších celků, které by byly snadno pochopitelné a bylo by je možné (do jisté míry) programovat a ladit samostatně.

Snad historicky nejstarší metoda granularizace větších programových celků se nazývá *top-down*. Tento styl vytváření programů vychází z toho, že si velký program nejprve představíme jako „celek“ a navrhneme jej rozdělit do několika samostatných částí (podprogramů). Každou z těchto částí opět prohlédneme a opět navrhneme její rozdělení. Takto pokračujeme dokud nedosáhneme požadovaného stupně „rozbití programu“ na mají celky. Tyto malé celky se potom obvykle programují samostatně (na jejich vytvoření může pracovat několik kolektivů programátorů). Pro dokončení všech celků se provede jejich spojení do výsledného programu.

Ve funkcionálních jazycích a zejména v dialektech LISPu, což jsou jedny z nejflexibilnějších programovacích jazyků, se však obvykle používá jiný přístup, který se nazývá *bottom-up*. V tomto případě je myšlenkový postup jakoby „obrácený“. Programátoři vytvářejí program po vrstvách. Nejnižší vrstvou programu je vždy samotný programovací jazyk (v našem případ jazyk Scheme). Nad ním je druhá vrstva, která obsahuje nově definované procedury řešící jistou třídu problémů. Tuto vrstvu si lze v podstatě představit jako rozšíření jazyka Scheme. Další vrstva bude tvořena jinými procedurami, které budou řešit další problémy a které již budou používat nejen primitivní procedury jazyka Scheme, ale i procedury vytvořené v předchozí vrstvě. Při programování ve stylu *bottom-up* je tedy zvykem postupně *obohacovat samotný programovací jazyk* o nové schopnosti a postupně tak dospět k dostatečně bohatému jazyku, ve kterém bude již snadné naprogramovat zamýšlený program. Pokud jsou procedury v jednotlivých vrstvách navrženy dostatečně abstraktně, změna jejich kódu nebo reimplementace celé jedné vrstvy programu (třeba kvůli efektivitě nebo kvůli změně zadání od uživatele) nečiní příliš velký problém.

S vytvářením programů metodou *bottom-up* souvisí dva důležité pojmy, z nichž první jsme již slyšeli.

černá skříňka Jakmile máme vytvořeny procedury v jedné vrstvě programu a začneme vytvářet další (vyšší) vrstvu, na procedury v nižší vrstvě bychom se správně měli dávat jako na „černé skříňky“. To jest, neměli bychom se zabývat tím, jak jsou naprogramované, ale měli bychom je pouze používat na základě toho, jaké argumenty je jim možné předávat a na základě znalosti výsledků jejich aplikací. Tím, že odhlédneme od implementace procedur na nižší úrovni, budeme vytvářet kvalitnější procedury na vyšší úrovni, které nebudou zasahovat do nižší vrstvy. Když potom provedeme reimplementaci procedur na nižší vrstvě při zachování jejich rozhraní (argumentů a výstupů), pak se funkčnost procedur na vyšší vrstvě nezmění. Připomeňme, že pohled na procedury jako na černé skříňky není nic „umělého“. Při programování ve Scheme jsme se doposud dívali na všechny primitivní procedury (to jest na procedury na nejnižší vrstvě), jako na černé skříňky.

abstrakční bariéra je pomyslný mezník mezi dvěma vrstvami programu. Pokud vytváříme procedury na vyšší vrstvě programu, procedury nižších vrstev jsou pro nás „za abstrakční bariérou.“ Otázkou je, na jakých místech v programu je vhodné tyto bariéry „vytvářet“. Jinými slovy, otázkou je, kde od sebe

oddělovat jednotlivé vrstvy programu. Odpověď na tuto otázku není jednoduchá a závisí na konkrétním problému, zkušenostech programátora a jeho představivosti. Dobrou zprávou ale je, že vrstvy v programu se mohou během života programu vyvíjet. Obzvlášť pro programování ve funkcionálních jazycích je typické, že se začne vytvářet program, který je pouze částečně navržený a během jeho vývoje se postupně segregují vrstvy tak, jak je to zrovna potřeba nebo podle toho, když se zjistí, že je to „praktické“. Taková situace může například nastat v momentě, když si programátoři všimnou, že několik procedur „vypadá podobně“. V takovém případě je vhodné zamyslet se nad jejich zobecněním, při kterém je možné například použít procedury vyšších řádů.

V tomto kursu se budeme zabývat základy programovacích stylů a až na výjimky budeme vše demonstrovat na malých programech, to jest programech majících maximálně desítky řádků. Výhody metody *bottom-up* bychom ocenili až při vytváření větších programů. Ve dvanácté lekci ukážeme implementaci interpretu funkcionální podmnožiny jazyka Scheme, která bude mít několik set řádků. To je z hlediska velikosti programů opět pouze malý program (za „velké programy“ se obvykle považují programy mající minimálně stovky tisíc řádků; za největší program současnosti je považován program pro řízení letu raketoplánu, který má údajně přes 150 milionů řádků). Na druhou stranu, program z dvanácté lekce už bude „natolik velký“, že v něm budeme moci rozlišit několik přirozených vrstev a abstrakčních bariér.

Shrnutí

V této kapitole jsme se zabývali lokálními vazbami. Uvedli jsme důvody, proč je potřebujeme. Zopakovali jsme, jak vznikají při aplikaci uživatelsky definovaných procedur. Ukázali jsme si dvě speciální formy – `let` a `let*`. Uvedli jsme příklady použití těchto forem a také jsme ukázali, že obě tyto nové speciální formy jsou nahraditelné použitím speciální formy `lambda`. Dále jsme rozšířili λ-výrazy, tím že jsme umožnili jejich tělo sestávat z libovolného nenulového počtu výrazů. Ukázali jsme použití interních definic, tedy speciální formy `define` pro modifikaci lokálních prostředí.

Pojmy k zapamatování

- lokální vazba
- interní definice, top-level definice
- `let`-blok, `let*`-blok

Nově představené prvky jazyka Scheme

- Speciální formy `let` a `let*`

Kontrolní otázky

1. *Co jsou lokální vazby? K čemu jsou dobré?*
2. *Proč nemohou být* `let` *a* `let*` *procedury?*
3. *Jak se* `let`*-bloky přepisují na λ-výrazy?*
4. *Jak se na ně přepisují* `let*`*-bloky?*
5. *Jak probíhá aplikace formy* `let`*?*
6. *Jak probíhá aplikace formy* `let*`*?*
7. *Jak lze přepsat* `let*`*-blok pomocí formy* `let`*?*
8. *Jak jsme rozšířili λ-výrazy?*
9. *Co jsou interní definice, jak se liší od top-level definic?*

Cvičení

1. Bez použití interpretu určete výsledky vyhodnocení následujících výrazů:

```
(let () 10)                                                    ⟹
(let () x)                                                     ⟹
(let () (define x 5))                                          ⟹
(let () (define x 5) (* x x))                                  ⟹
(let ((x (+ 5 5))) (* x x))                                    ⟹
(let* ((x (+ 5 5))) (* x x))                                   ⟹
(let ((x (+ 5 5))) (* x x) (+ x x))                            ⟹
(let ((x 10)) (define x 5) (* x x))                            ⟹
(let* ((x 10)) (define x (+ x 1)) (* x x))                     ⟹
(let ((x 10)) (define x x) (* x x))                            ⟹

(let ((x 10) (y 20)) (+ x y))                                  ⟹
(let* ((x 10) (y 20)) (+ x y))                                 ⟹
(let ((x 10) (y x)) (* x y))                                   ⟹
(let* ((x 10) (y x)) (* x y))                                  ⟹
(let ((x 10) (y +)) (* x y))                                   ⟹
(let ((x 10) (y +)) (define y x) (* x y))                      ⟹
(let* ((x 10) (y +)) (define y x) (* x y))                     ⟹
(let* ((x 10) (y x)) (define y 3) (* x y))                     ⟹
(let ((x 10)) (define y 3) (* x y))                            ⟹
(let ((x 10) (y (lambda () x))) (+ x (y)))                     ⟹

(let* ((x 10) (y (lambda () x))) (+ x (y)))                    ⟹
(let* ((x 10) (y (lambda (x) x))) (+ x (y x)))                 ⟹
(let* ((x 10) (y (lambda () x)) (x 5)) (+ x (y)))              ⟹
(let* ((x 10) (y (lambda (x) x)) (x 5)) (+ x (y x)))           ⟹
(let ((x 10)) (define y (lambda () x)) (define x 5) (+ x (y)))  ⟹
(let ((x 10)) (define y (lambda (x) x)) (define x 5) (+ x (y x)))  ⟹
(let ((x 10)) (+ (let ((y (+ x 1))) (* y 2)) x))               ⟹
(let ((x 10) (x 20)) (+ x x))                                  ⟹
(let* ((x 10) (x 20)) (+ x x))                                 ⟹
(if #f (let ((x 10) (x 20)) (+ x x)) 10)                       ⟹
```

2. Přepište následující výrazy na ekvivalentní výrazy bez použití speciální forem `let` a `let*`.

   - ```
     (let* ((x (+ 10 y))
            (y (let ((x 20)
                     (y 30))
                 (* 2 (+ x y)))))
       (/ x y z))
     ```

   - ```
     (let ((a 3)
           (b 4)
           (c (let* ((a 10)
                     (b 20))
                (+ a b 10))))
       (+ a b c))
     ```

   - ```
     (let* ((x (let ((x 10))
                 (define x 2)
                 (+ x x)))
            (y (+ x x)))
       (+ x y))
     ```

3. Napište proceduru, která vypočte, v jaké výšce dosáhne těleso vržené vzhůru počáteční rychlostí v_0 rychlosti v (ve vakuu). Vstupními parametry budou v_0 a v. Použijte vzorce:
     ```

$$t = \frac{v_0 - v}{g}, \quad s = v_0 t - \frac{1}{2} g t^2.$$

Lokální vazbu pro t vytvořte jednou pomocí interní definice, podruhé pak pomocí speciální formy `let`. Tíhové zrychlení g definujte globálně.

4. Napište proceduru řešící tento typ slovních úloh: Objekt se pohybuje rovnoměrně zrychleným pohybem, s počáteční rychlostí v_0 se zrychlením a. Vypočtěte dráhu, po které dosáhne rychlosti v. Použijte vzorce:

$$t = \frac{v - vo}{a}, \quad s = v_0 t + \frac{1}{2} a t^2.$$

Lokální vazbu pro t vytvořte jednou pomocí interní definice, podruhé pomocí speciální formy `let`.

5. Předefinujte proceduru navázanou na symbol `+` na proceduru sčítání druhých mocnin dvou čísel.
6. Napište λ-výraz, který má ve svém těle více než jeden výraz a jehož přímočarý přepis na λ-výraz obsahující ve svém těle jeden výraz s požitím speciální formy `and` (viz například programy 3.1 a 3.2) se vyhodnotí na jinou proceduru. Jinou procedurou v tomto případě myslíme to, že aplikací na nějaký argument bude vracet jiný výsledek.

Úkoly k textu

1. Uvažujme speciální formu `let+`, která se používá ve stejném tvaru jako speciální forma `let*`:

```
(let+ ((⟨symbol_1⟩ ⟨hodnota_1⟩)
       (⟨symbol_2⟩ ⟨hodnota_2⟩)
       ⋮
       (⟨symbol_n⟩ ⟨hodnota_n⟩)))
  ⟨tělo⟩)
```

ale přepisuje se na:

```
(let ()
  (define ⟨symbol_1⟩ ⟨hodnota_1⟩)
  (define ⟨symbol_2⟩ ⟨hodnota_2⟩)
  ⋮
  (define ⟨symbol_n⟩ ⟨hodnota_n⟩)
 ⟨tělo⟩)
```

Zamyslete se nad rozdílem oproti speciální formě `let*`. Napište výraz, který bude mít různé výsledky vyhodnocení při použití `let+` a `let*`.

2. Popište jak se dá nahradit `let*`-blok pomocí λ-výrazů.
3. Na začátku sekce 3.2 jsme uvedli, jak použít definice v těle λ-výrazů, kdybychom nezměnili definici λ-výrazu. Použili jsme přitom speciálních forem `lambda` a `and`. Popište, jak by se k tomu dalo využít jiných speciálních forem, např. `if` nebo `cond`.

Řešení ke cvičením

1. 10, chyba, nespecifikovaná hodnota, 25, 100, 100, 20, 25, 36, 100,
30, 30, chyba, 100, chyba, 100, 100, 30, 30, chyba,
20, 20, 15, 10, 10, 10, 32, chyba, 40, 10

2.
```
((lambda (x)
   ((lambda (y)
      (/ x y z))
```

```
((lambda (x y)
   (* 2 (+ x y)))
 20 30)))

(+ 10 y))
```

- ```
  ((lambda (a b c)
     (+ a b c))
   3 4
   ((lambda (a)
      ((lambda (b)
         (+ a b 10))
       20))
    10))
  ```

- ```
  ((lambda (x)
     ((lambda (y)
        (+ x y))
      (+ x x)))

     ((lambda (x)
        (define x 2)
        (+ x x))
      10))
  ```

3. ```
   (define g 9.80665)

   (define vrh-vzhuru
     (lambda (v0 v)
       (let ((t (/ (- v0 v) g)))
         (- (* v0 t) (* 1/2 g t t)))))
   ```

   ```
   (define vrh-vzhuru
     (lambda (v0 v)
       (define t (/ (- v0 v) g))
       (- (* v0 t) (* 1/2 g t t))))
   ```

4. ```
   (define pohyb-rovnomerne-zrychleny
     (lambda (v0 v a)
       (let ((t (/ (- v0 v) a)))
         (+ (* v0 t) (* 1/2 a t t)))))
   ```

   ```
   (define pohyb-rovnomerne-zrychleny
     (lambda (v0 v a)
       (define t (/ (- v0 v) a))
       (+ (* v0 t) (* 1/2 a t t))))
   ```

5. ```
   (define +
     (let ((+ +))
       (lambda (x y)
         (+ (* x x) (* y y)))))
   ```

6. Například: `(lambda () #f 1)`
   ```

Lekce 4: Tečkové páry, symbolická data a kvotování

Obsah lekce: V této lekci se budeme zabývat vytvářením abstrakcí pomocí dat. Představíme tečkové páry pomocí nichž budeme vytvářet hierarchické datové struktury. Ukážeme rovněž, že tečkové páry bychom mohli plně definovat pouze pomocí procedur vyšších řádů. Dále se budeme zabývat kvotováním a jeho zkrácenou notací jenž nám umožní chápat symboly jako data. Jako příklad datové abstrakce uvedeme implementaci racionální aritmetiky.

Klíčová slova: datová abstrakce, konstruktory, kvotování, selektory, symbolická data, tečkové páry.

4.1 Vytváření abstrakcí pomocí dat

Doteď jsme vytvářeli abstrakce pomocí procedur: používali jsme procedury k vytváření složitějších procedur, přitom jsme se nemuseli zabývat tím, jak jsou jednodušší procedury vytvořeny (pokud fungují správně). Teď provedeme totéž s daty: budeme pracovat se *složenými* (*hierarchickými*) *daty* a budeme konstruovat procedury, které je budou zpracovávat – budou je brát jako argumenty nebo vracet jako výsledek své aplikace. Přitom to opět budeme provádět tak, aby bylo procedurám de facto jedno, jak jsou složená data konstruována z jednodušších. Nejprve si ukážeme několik motivačních příkladů, z nichž pak vyplyne potřeba vytvářet a zpracovávat *složená data*.

Příklad 4.1. Uvažujme, že bych chtěli napsat proceduru, která má nalézt kořeny kvadratické rovnice

$$ax^2 + bx + c = 0$$

na základě jejich koeficientů podle důvěrně známého vzorce

$$\frac{-b \pm \sqrt{D}}{2a}, \quad \text{kde } D = b^2 - 4ac.$$

Takové kořeny jsou v oboru komplexních čísel dva: x_1 a x_2 (popřípadě jeden dvojitý). Proceduru pro výpočet kořenů, která má tři argumenty reprezentující koeficienty kvadratické rovnice, bychom mohli napsat například následujícím způsobem. Kód je ponechán neúplný, protože nám zatím schází prostředek, jak vrátit oba kořeny současně:

```
(define koreny
  (lambda (a b c)
    (let* ((diskr (- (* b b) (* 4 a c)))
           (koren1 (/ (+ (- b) (sqrt diskr)) (* 2 a)))
           (koren2 (/ (- (- b) (sqrt diskr)) (* 2 a))))
      teď bychom chtěli vrátit dvě hodnoty současně
     )))
```

V tuto chvíli se samozřejmě nabízí celá řada „hloupých řešení“, jak obejít problém vracení dvou hodnot. Mohli bychom například proceduru rozšířit o další parametr, který by určoval, který kořen máme vrátit. Kód by pak vypadal následovně:

```
(define koreny
  (lambda (a b c p)
    (let ((diskr ((- (* b b) (* 4 a c)))))
       (/ ((if p + -) (- b) (sqrt diskr)) 2 a))))
```

Předchozí řešení problému není šťastné, kdybychom totiž dále chtěli pracovat s oběma kořeny, museli bychom proceduru aplikovat dvakrát a pokaždé by docházelo k vyhodnocování téhož kódu se stejnými argumenty (jedná se třeba o výpočet diskriminantu). Jiným takovým řešením je namísto jedné procedury `koreny` mít dvě procedury `koren1` a `koren2`, přitom by každá vracela jeden z kořenů. Zavádění dvou procedur namísto jedné ale znesnadňuje údržbu kódu a stává se zdrojem chyb. Navíc by nám to nijak nepomohlo, pořád by to vedlo k problému násobného vyhodnocování téhož kódu. V případě dvojice kořenů kvadratické rovnice není tento problém zas až tak markantní. Jeho význam bude zřejmější v následujícím příkladě.

Příklad 4.2. Uvažme, že bychom ve Scheme neměli racionální aritmetiku, a chtěli bychom navrhnout systém na provádění aritmetických operací nad racionálními čísly (sčítání, odčítání, násobení, dělení a tak dále). Chtěli bychom například implementovat proceduru `r+`, která bere jako parametry dvě racionální čísla a vrací jejích součet. Z pohledu primitivních dat uvažujeme racionální číslo jako dvě celá čísla – čitatele a jmenovatele. S touto dvojicí potřebujeme zacházet jako s jednou hodnotou. Pro bližší ilustraci uvedeme neúplnou definici procedury `r+` s vloženým textem poukazujícím na potřebu používat hierarchická data:

```
(define r+
  (lambda (cit1 jmen1 cit2 jmen2)
          proceduře bychom chtěli předávat jen dva argumenty = dvě racionální čísla
    (let* ((cit (+ (* cit1 jmen2) (* cit2 jmen1)))
           (jmen (* jmen1 jmen2))
           (delit (gcd cit jmen))))
      teď bychom chtěli vrátit dvě hodnoty současně: cit a jmen
    )))
```

Obdobně jako v předchozím příkladě bychom mohli mít zvlášť proceduru na výpočet čitatele a zvlášť proceduru na výpočet jmenovatele, nebo pomocí dalšího argumentu určovat, kterou složku racionálního čísla (čitatele nebo jmenovatele) má procedura vrátit. Ve většině výpočtů ale budeme potřebovat i čitatele i jmenovatele. Navíc kdybychom neustále pracovali se dvěma hodnotami namísto jedné a pro každou operaci bychom museli mít dvě procedury (vracející čitatele a jmenovatele z výsledné hodnoty). Program by se stal velmi brzy nepřehledným. Tato podvojnost procedur a čísel by také velice znesnadňovala udržování kódu a stala by se potenciálním zdrojem chyb.

Příklad 4.3. V dalším příkladě se snažíme navrhnout program pracující s geometrickými útvary, ve kterém bychom chtěli pracovat s pojmy jako bod, úsečka, kružnice a tak dále. Bod v rovině je z pohledu primitivních dat dvojice reálných čísel; úsečku můžeme reprezentovat (třeba) jako dvojici bodů; kružnici (třeba) jako bod (střed) a reálné číslo (poloměr). Rozumným požadavkem na takový geometrický program by bylo mít v něm například k dispozici procedury pro výpočet různých průsečíků: třeba procedura na výpočet průsečíku dvou úseček bude vracet bod. Při vytváření programu bychom záhy narazili na potřebu „slepit“ dvě souřadnice dohromady, abychom je mohli považovat za bod; „slepit“ dva body dohromady a vytvořit tak reprezentaci úsečky a podobně. Z pohledu tohoto příkladu je tady patrné, že jako programátoři bychom měli mít k dispozici prostředky, které nám umožní reprezentovat abstraktnější data než jen „numerické hodnoty“. Konec konců, prakticky všechny soudobé programy pracují s mnohem složitějšími daty než jsou jen čísla.

Z těchto tří motivačních příkladů vyplývá zřejmý požadavek: Obdobně jako jsme v lekci 2 vytvářeli nové procedury, potřebujeme teď vytvářet *větší datové celky z primitivních dat* a pracovat s nimi jako by se jednalo o *jeden element*. Potřebujeme tedy vytvářet *abstrakce pomocí dat*. Jakmile to budeme moci udělat, můžeme například přestat uvažovat racionální číslo jako dvě čísla – čitatele a jmenovatele – a můžeme začít s ním pracovat na vyšší úrovni abstrakce jako s celkem, to jest s elementem reprezentujícím „racionální číslo“. Stejně tak v ostatních motivačních příkladech můžeme pracovat s elementy reprezentující „dvojice kořenů kvadratické rovnice“, „bod“, „úsečka“, „kružnice“, a tak dále.

Abychom docílili datové abstrakce, musíme odpovědět na dvě otázky:

1. Jak implementovat abstraktní data pomocí konkrétní (fyzické) datové reprezentace?
   Neboli potřebujeme prostředek, který nám umožní z *primitivních dat* vytvářet *složená data*.
2. Jak odstínit program od této konkrétní datové reprezentace?
   Když budeme pracovat se složenými daty, budeme s nimi chtít zacházet jako s pojmem, který reprezentují. Nebude nás zajímat jak a kde jsou data konkrétně uložena. Například v případě kořenů kvadratických rovnic chceme mít k dispozici procedury jako „vytvoř řešení“, „vrať první kořen z řešení“, „vrať druhý kořen z řešení“, a další procedury, které pracují s kořeny. Z pohledu uživatelů těchto procedur nás už ale nebude zajímat jakým způsobem a kde jsou hodnoty uloženy.

   Kvůli odstínění programu od konkrétní datové reprezentace proto pro každá složená data vytváříme:

konstruktory – procedury sloužící k vytváření složených dat z jednodušších, s vytvořenými daty dále pracujeme jako s jednotlivými elementy;

selektory – procedury umožňují přistupovat k jednotlivým složkám složených dat.

4.2 Tečkové páry

Ve Scheme máme k dispozici elementy jazyka nazývané *tečkové páry* (též jen *páry*). Tečkový pár je vytvořen spojením dvou libovolných elementů do uspořádané dvojce. Prvky v páru pak nazýváme *první prvek páru* a *druhý prvek páru*. Pár je zkonstruován primitivní procedurou `cons`, procedury pro přístup k jeho jednotlivým prvkům jsou navázané na symboly `car` a `cdr`. Z pohledu terminologie v předchozí sekci je `cons` *konstruktor páru* (vytváří pár z jednoduchých dat) a `car` a `cdr` jsou *selektory* (umožňují přistupovat k prvkům párů). Viz následující upřesňující definici.

Definice 4.4. Procedura `cons` se používá se dvěma argumenty

`(cons` ⟨*první prvek*⟩ ⟨*druhý prvek*⟩`)`

a vrací pár obsahující tyto dva argumenty ⟨*první prvek*⟩ a ⟨*druhý prvek*⟩ jako svoje prvky.
Máme-li pár ⟨*pár*⟩, můžeme extrahovat tyto části pomocí primitivních procedur `car` a `cdr`. Procedury `car` a `cdr` se používají s jedním argumentem

`(car` ⟨*pár*⟩`)`,
`(cdr` ⟨*pár*⟩`)`.

Procedura `car` jako výsledek své aplikace vrací první prvek předaného páru. Procedura `cdr` jako výsledek své aplikace vrací druhý prvek předaného páru. V případě, že by `car` nebo `cdr` byly aplikovány s elementem, který není pár, pak dojde k chybě „CHYBA: Argument předaný proceduře musí být pár“. ■

Poznámka 4.5. Původ jmen *cons*, *car* a *cdr*: Jméno procedury `cons` znamená „construct“ (vytvoř). Jména `car` a `cdr` pocházejí z původní implementace LISPu na počítači IBM 704. Tento stroj měl adresovací schéma, které povolovalo odkazovat se na místa v paměti nazvané „address“ a „decrement“. Zkratka „car“ znamená „Contents of Address part of Register“ a zkratka „cdr“ (čteno „kudr“) znamená „Contents of Decrement part of Register,“ pro detaily viz například [SICP].

Příklad 4.6. Uvažujme, že na symboly `par1` a `par2` navážeme dva páry následujícím způsobem:

```
(define par1 (cons 1 2))
(define par2 (cons par1 3))
```

(a) Výraz `(car par1)` se vyhodnotí na číslo `1`, protože na symbol `par1` je navázán pár, který má první prvek číslo `1` (a jeho druhým prvkem je číslo 2). Aplikací procedury `car` dostaneme toto číslo.

(b) Výsledkem vyhodnocení výrazu `(cdr par2)` bude číslo 3. Na symbol `par2` je navázán pár, jehož druhý prvek je číslo 3, které získáme aplikací procedury `cdr`. Výsledkem vyhodnocení výrazu `(car par2)` bude pár obsahující jako svůj první prvek jedničku a jako druhý prvek dvojku. Z příkladu je myslím jasné, že prvky párů mohou být obecně jakékoliv elementy, tedy i další páry.

(c) Vyhodnocení výrazu `(car (car par2))` bude vypadat následovně: Jelikož je na symbol `car` navázána procedura, vyhodnotíme předaný argument – tedy seznam `(car par2)`. Vyhodnocením tohoto seznamu dostaneme pár, který má první prvek číslo `1` a druhý prvek je číslo 2 – vyhodnocení tohoto podvýrazu bylo již popsáno v bodě (b). Aplikací procedury `car` na tento pár dostaneme číslo `1`. Obdobně číslo 2 dostaneme jako výsledek vyhodnocení `(cdr (car par2))`.

(d) Při vyhodnocení výrazu `(cdr (cdr par2))` dostaneme chybu „CHYBA: Argument není pár“, protože výsledek vyhodnocení výrazu `(cdr par2)` je číslo 3, tedy druhá (vnější) aplikace `cdr` selže, viz komentář v definici 4.4.

Při složitějších strukturách párů se může stát, že sekvence volání `car` a `cdr` budou dlouhé a nepřehledné, proto jsou ve Scheme k disposici složeniny. Například místo `(car (cdr p))` můžeme psát jen `(cadr p)`.

Ve Scheme jsou složeniny až do hloubky čtyři. Například na symboly `caaaar`, `cadadr` a `cddaar` budou ještě navázány složené selektory, ale na třeba na `cddadar` již ne. Složenin je tedy celkem 28.

Před tím než uvedeme další příklad podotkněme, že složeniny bychom mohli vytvářet sami:

```
(define caar (lambda (p) (car (car p))))
(define cadr (lambda (p) (car (cdr p))))
(define cdar (lambda (p) (cdr (car p))))
(define cddr (lambda (p) (cdr (cdr p))))
(define caaar (lambda (p) (car (caar p))))
  ⋮
```

Příklad 4.7. Uvažujme pár, který navážeme na symbol `par` vyhodnocením následujícího výrazu:

```
(define par (cons (cons 10 (cons 20 30)) 40)).
```

(a) Výraz `(caar par)` se vyhodnotí na číslo deset. Jde o totéž jako bychom psali `(car (car par))`.

(b) Výraz `(cdar par)` se vyhodnotí na číslo 2.

(c) Vyhodnocení výrazu `(caaar par)` skončí chybou „CHYBA: Argument není pár“.

(d) V předchozích příkladech jsme vždy používali selektory na páry, které jsme předtím navázali na symboly pomocí speciální formy `define`. Samozřejmě, že `define` se samotnými páry nic společného nemá a selektory bychom mohli použít přímo na páry vytvořené konstruktorem `cons` bez provádění jakýchkoliv vazeb, například:

```
(caar (cons (cons 10 20) (cons 30 40)))  ⟼  10
(cdar (cons (cons 10 20) (cons 30 40)))  ⟼  20
(cadr (cons (cons 10 20) (cons 30 40)))  ⟼  30
(cddr (cons (cons 10 20) (cons 30 40)))  ⟼  40
```

Než si ukážeme několik příkladů na použití tečkových párů, popíšeme, jak vypadá *externí reprezentace tečkových párů*, a také si ukážeme, jak se páry znázorňují graficky, pomocí tak zvané *boxové notace*.

Externí reprezentace páru majícího první prvek $\langle A \rangle$ a druhý prvek $\langle B \rangle$ je následující: `(`$\langle A \rangle$ `.` $\langle B \rangle$`)`. Z této notace pro externí reprezentaci tečkových párů je asi jasné, proč se o nich říká, že jsou „*tečkové*“. Viz příklad:

```
(cons 1 2))  ⟼  (1 . 2)
```

V případě, že druhý prvek páru je zase pár, se používá *zkrácený zápis*. V tomto zkráceném zápisu se vynechávají závorky náležející tomu vnitřnímu páru a tečka náležející vnějšímu. Třeba pár `(`$\langle A \rangle$ `.` `(`$\langle B \rangle$ `.` $\langle C \rangle$`))` můžeme zkráceně zapsat jako pár `(`$\langle A \rangle$ $\langle B \rangle$ `.` $\langle C \rangle$`)`.

Příklad 4.8. V tomto příkladu vidíme externí reprezentace párů:

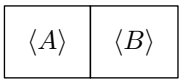
```
(cons 10 20)                          ⟼  (10 . 20)
(cons (cons 10 20) 30)                ⟼  ((10 . 20) . 30)
(cons 10 (cons 20 30))                ⟼  (10 . (20 . 30)) = (10 20 . 30)
(cons (cons 10 (cons 20 30)) 40)      ⟼  ((10 . (20 . 30)) . 40) = ((10 20 . 30) . 40)
(cons 10 (cons (cons 20 30) 40))      ⟼  (10 . ((20 . 30) . 40)) = (10 (20 . 30) . 40)
(cons (cons 10 20) (cons 30 40))      ⟼  ((10 . 20) . (30 . 40)) = ((10 . 20) 30 . 40)
```

Naopak následující výrazy nejsou externí reprezentace tečkových párů:

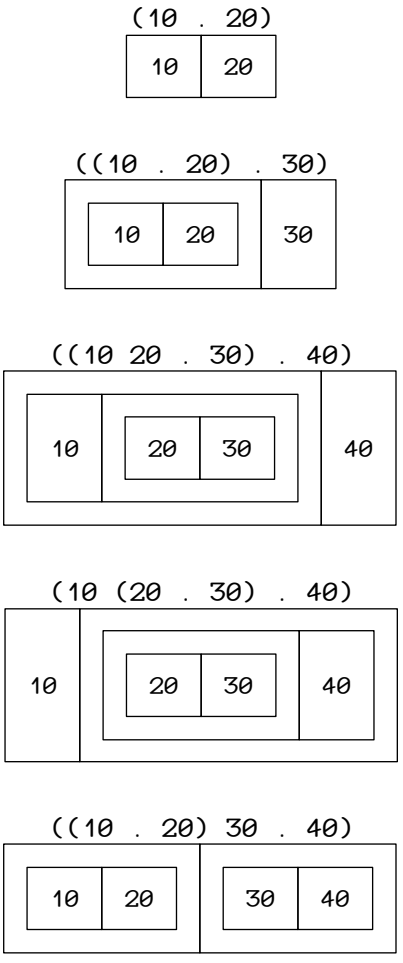
`(10 . ),( . 20),(10 . 20 . 30)` a `(10 . 20 30)`.

Nyní si ukážeme grafické znázornění tečkových párů – *boxovou notaci*. Pár je zobrazen jako obdélníček (box) rozdělený na dvě části. Do levé části boxu se zakresluje první prvek páru a do pravé druhý prvek páru. Tedy pár `(`$\langle A \rangle$ `.` $\langle B \rangle$`)` vypadá v boxové notaci tak, jak je znázorněno na obrázku 4.1.

Obrázek 4.1. Boxová notace tečkového páru.



Obrázek 4.2. Tečkové páry z příkladu 4.8 v boxové notaci



Příklad 4.9. Tečkové páry z příkladu 4.8 v boxové notaci najdeme na obrázku 4.2.

Příklad 4.10. Přirozeně nemusíme konstruovat páry jen z čísel a párů, jako v doposud uvedených příkladech. Můžeme vytvářet páry z jakýchkoli elementů:

```
(cons + -)            ⟼  („procedura sčítání“ . „procedura odčítání“)
(cons and not)        ⟼  („speciální forma and“ . „procedura negace“)
(cons (if #f #f) #t)  ⟼  („nedefinovaná hodnota“ . #f)
```

> Páry jsou elementy prvního řádu. Pokud se vrátíme k definici 2.17 na straně 56, pak můžeme jasně vidět, že páry je možno pojmenovat, předávat jako argumenty procedurám, vracet jako výsledky aplikace procedur a mohou být obsaženy v hierarchických datových strukturách (páry mohou být opět obsaženy v jiných párech).

Poslední, co zbývá k tečkovým párům říct, je to, jak vlastně páry zapadají do Read a Eval částí REPL cyklu. Existenci párů jsme v lekci 1 při definici symbolických výrazů zatajili. Ve skutečnosti i externí reprezentace řady tečkových párů je čitelná readerem [5]. V tuto chvíli můžeme obohatit množinu přípustných symbolických výrazů jazyka Scheme o speciální výrazy, které se shodují s externí reprezentací některých tečkových párů. Rozšíříme definici 1.6 ze strany 19 o následující nový bod:

- Jsou-li $e, f, e_1, e_2, \ldots, e_n$ symbolické výrazy $(n \geq 0)$, pak

  $(e\ \ e_1\ \ e_2\ \ \cdots\ \ e_n\ .\ f)$

  je symbolický výraz. Pokud je $n = 0$, symbolický výraz $(e\ .\ f)$ nazveme *pár.*

Teď nám ale dochází k nepříjemné dvojznačnosti: S-výraz $(e\ .\ f)$ teď můžeme považovat za tříprvkový seznam obsahující výraz e, symbol „ . “ a výraz f, a také za tečkový pár obsahující výrazy e a f. Aby k této dvojznačnosti nedocházelo, je potřeba zpřísnit definici symbolu o následující rys: Samostatný znak „ . “ *není symbol.* S-výraz $(e\ .\ f)$ tak může být jedině pár.

Další nejednoznačnost je v terminologii. Musíme si ujasnit, že i když nazýváme *symbolický výraz pár* i *element pár* shodně, to jest „pár“, jedná se o dvě různé věci. Pár jako *symbolický výraz* určuje jak může vypadat část programu (je to syntaktický pojem), kdežto pár jako element reprezentuje konkrétní hodnoty, se kterými pracuje interpret při vyhodnocování, jedná se tedy o sémantický pojem těsně spjatý s (abstraktním) interpretem jazyka Scheme.

Řekli jsme si tedy, že páry jsou čitelné readerem. Co jsme ale ještě neřekli je, jak se vyhodnocují. Podle toho, jak jsme nadefinovali Eval v definici 1.21 na straně 27, by se měl pár podle bodu (D) vyhodnotit sám na sebe. Podotkněme nyní, že z praktického hlediska se interprety jazyka Scheme dávají na vyhodnocování párů jinak. Zatím vyhodnocování párů ponecháme stranou a popíšeme je až v další sekci.

Na konci této sekce uvádíme několik zajímavých příkladů:

Příklad 4.11. V následujícím kódu vytváříme páry, které potom navazujeme na symboly. Nejprve na symbol `a` navážeme jeden pár a pomocí něj potom vytvoříme druhý pár a navážeme jej na symbol `b`. Pokud provedeme změnu vazby symbolu `a`, pár navázaný na symbolu `b` se (pochopitelně) nezmění:

```
(define a (cons 10 20))
(define b (cons (car a) 30))
b  ⟼  (10 . 30)
(define a (cons 5 -5))
b  ⟼  (10 . 30)
```

To vyplývá ze sémantiky speciální formy `define`, viz definici 1.26 na straně 1.26 a z faktu, že `cons` je procedura. Při vyhodnocení výrazu `(cons (car a) 30)` je aplikována procedura `cons` na vyhodnocení výrazů `(car a)` a `30` – to jest na čísla `10` a `30`. Výsledkem aplikace procedury `cons` je pak pár `(10 . 30)`, a je navázán na symbol `b`. Předefinováním hodnoty navázané na symbol `a` pomocí `define` se pak pár navázaný na symbol `b` nezmění.

[5] Někdy tomu tak ovšem být nemusí, vezměme si třeba tečkové páry z příkladu 4.10. Jejich externí reprezentace bude pro reader nečitelná, protože páry obsahují elementy s nečitelnou reprezentací (procedury, speciální formy a nedefinovanou hodnotu).

Příklad 4.12. Nadefinujeme proceduru, která vrací pár, jehož prvním prvkem je právě tato procedura:

```
(define a (lambda () (cons a #f)))
(a)                ⟼  („procedura“ . #f)
(car (a))          ⟼  „procedura“
((car (a)))        ⟼  („procedura“ . #f)
(car ((car (a))))  ⟼  „procedura“
  ⋮
```

Příklad 4.13. V lekci 2 jsme v příklady 2.11 na straně 69 definovali proceduru vyššího řádu `extrem` na vytváření procedur hledání extrémní hodnoty ze dvou argumentů vzhledem k nějakému uspořádání (viz program 2.11). Nyní můžeme vytvořit proceduru která vytváří dvě procedury současně – jednu na nalezení jednoho extrému a druhou na nalezení opačného extrému – a vrací je v páru:

```
(define extrem-pair
  (lambda (p?)
    (cons (extrem p?)
          (extrem (lambda (a b) (not (p? a b)))))))
```

Aplikací procedury `extrem` jednou na výchozí predikát a jednou na predikát vytvořený z výchozího pomocí negace podmínky dané tímto predikátem, jsme tedy dostali dvě procedury, ze kterých jsme pomocí `cons` vytvořili tečkový pár. Procedury `min` a `max` pomocí procedury `extrem-pair` můžeme definovat třeba takto:

```
(define extrems (extrem-pair <))
(define min (car extrems))
(define max (cdr extrems))
```

Poznámka 4.14. Vrátíme-li se nyní k motivačnímu příkladu s řešením kvadratické rovnice ze začátku této sekce, pak snadno nahlédneme, že bychom mohli chybějící tělo v proceduře `koreny` doplnit výrazem `(cons koren-1 koren-2)`. To by byl úplně nejjednodušší způsob, jak vrátit „dvě hodnoty současně.“ Z hlediska čistoty programu by již toto řešení bylo diskutabilní, jak uvidíme v další sekci.

4.3 Abstrakční bariéry založené na datech

V předchozí lekci jsme mimo jiné mluvili o vrstvách programu a o abstrakčních bariérách založených na procedurách. Abstrakční bariéry založené na datech jsou obecnější než abstrakční bariéry pomocí procedur, protože pro nás jsou procedury data (lze s nimi zacházet jako s daty). Procedury hrají ale i při datové abstrakci důležitou roli a tou je odstínění fyzické reprezentace dat od jejich uživatele.

Při vytváření abstrakčních bariér v programech bychom se tedy měli soustředit jako na procedury tak na data. Jednostranné zaměření na to či ono skoro nikdy nepřinese nic dobrého. Účelem datové abstrakce je oddělit data na nižší úrovni od abstraktnějších dat a mít pro ně vytvořené separátní konstruktory a selektory (pomocí nichž je abstrakční bariéra de facto vytvořena).

Opět zdůrazněme, že při vývoji programu je (i z pohledu dat) vhodné používat styl *bottom-up*, tedy pracovat a přemýšlet nad programem jako nad postupným obohacováním jazyka. V případě datové abstrakce jde o *rozšiřování jazyka* o *nové datové typy.* Pro ilustraci si uveďme následující příklad.

Příklad 4.15. Jako příklad použití abstraktních bariér uvádíme program 4.1 na výpočet kořenů kvadratické rovnice. Tento krátký program je rozdělen do tří vrstev, které jsou znázorněny na obrázku 4.3. Nejnižší vrstvu tvoří vrstva jazyka Scheme a konkrétní implementace tečkových párů. To jak tato implementace vypadá nás ve vyšších vrstvách nezajímá, pro nás jsou důležité konstruktory a selektory párů, které nám tato vrstva nabízí. Hned nad touto nejnižší vrstvou je vrstva implementace dvojice kořenů. Dvojici kořenů implementujeme pomocí tečkových párů, to pro nás ale v dalších vrstvách není důležité. Co je důležité jsou procedury `vrat-koreny`, `prvni-koren` a `druhy-koren` sloužící jako konstruktor a selektory pro dvojici kořenů. Dále již pracujeme jen s nimi. Všimněte si, že v procedurách `najdi-koreny` a `dvojnasobny?` neuvažujeme dvojice kořenů jako páry.

Například definici procedury `najdi-koreny` by bylo chybou namísto

```
(vrat-koreny ···
```

napsat

```
(cons ···,
```

protože by tím došlo k porušení pomyslné abstrakční bariéry. Na symboly `vrat-koreny` a `cons` je sice navázaná stejná procedura, ale při změně vrstvy „práce s kořeny“ (viz obrázek 4.3) bychom museli měnit i vyšší vrstvu. Třeba hned v úvodu další sekce ukážeme, jak bychom mohli zacházet s kořeny, kdybychom neměli k dispozici tečkové páry. Obdobným způsobem bychom mohli napsat i proceduru `vrat-koreny`.

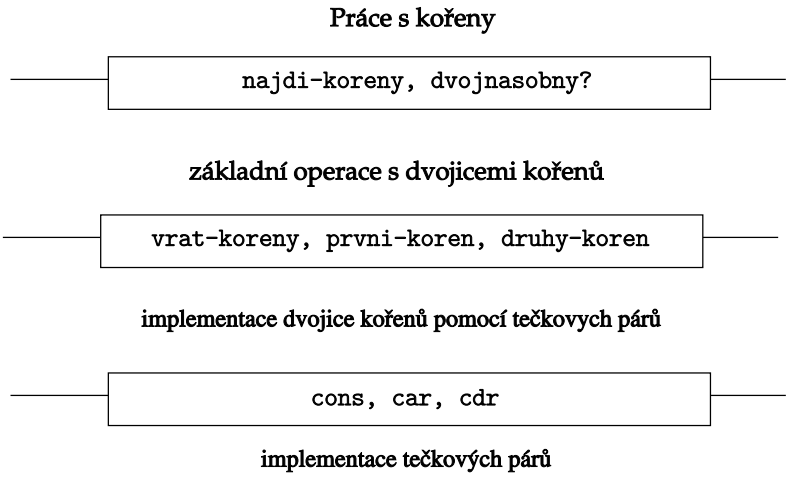
Program 4.1. Příklad abstrakční bariéry: výpočet kořenů kvadratické rovnice.

```
(define vrat-koreny cons)
(define prvni-koren car)
(define druhy-koren cdr)

(define najdi-koreny
  (lambda (a b c)
     (let ((diskr (- (* b b) (* 4 a c))))
       (vrat-koreny (/ (- (- b) (sqrt diskr)) 2 a)
                    (/ (+ (- b) (sqrt diskr)) 2 a)))))

(define dvojnasobny?
  (lambda (koreny)
    (= (prvni-koren koreny)
       (druhy-koren koreny))))
```

Obrázek 4.3. Schéma abstrakčních bariér



Pro každá data, která se v programu vyskytují, bychom měli vytvořit sadu nezávislých konstruktorů a selektorů. To vede ke snadné údržbě kódu a k snadným případným změnám implementace. Vždy je potřeba najít vhodné ekvilibrium mezi množstvím abstrakčních bariér a efektivitou programu (optima-

lizací programu týkající se jeho rychlosti, spotřeby systémových zdrojů a podobně). Tyto dvě kvality programu jdou leckdy proti sobě.

4.4 Implementace tečkových párů pomocí procedur vyšších řádů

V této sekci se budeme zabývat tím, zda-li je nutné k vytvoření elementů zapouzdřující dvojice hodnot uvažovat nové elementy jazyka (tečkové páry) tak, jak jsme to dělali doposud, nebo jestli není možné je modelovat s tím, co už máme v jazyku Scheme k dispozici. V této sekci ukážeme, že páry lze plně implementovat pouze s využitím procedur vyšších řádů a lexikálního rozsahu platnosti.

Nejdříve problematiku demonstrujeme na příkladu 4.1 s kořeny kvadratické rovnice z úvodu lekce. Poté tuto myšlenku zobecníme tak, že pomocí procedur vyšších řádů naimplementujeme vlastní tečkové páry.

Zopakujme, že chceme vytvořit proceduru, která by pro kvadratickou rovnici

$$ax^2 + bx + c = 0$$

zadanou jejími koeficienty a, b a c spočítala její dva kořeny a předpokládejme při tom, že nemáme k dispozici `cons`, `car` a `cdr` o kterých jsme doposud hovořili. Kód z příkladu 4.1 doplníme tak, jak je to ukázáno v programu 4.2. Do těla `let*`-bloku (které v kódu z úvodu lekce chybělo) jsme dali výraz:

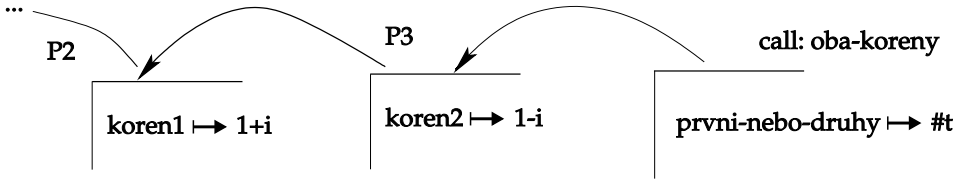
Program 4.2. Implementace procedury `koreny` pomocí procedur vyšších řádů.

```
(define koreny
  (lambda (a b c)
      (let* ((diskr (- (* b b) (* 4 a c)))
             (koren1 (/ (+ (- b) (sqrt diskr)) (* 2 a)))
             (koren2 (/ (- (- b) (sqrt diskr)) (* 2 a))))
        (lambda (prvni-nebo-druhy)
          (if prvni-nebo-druhy koren1 koren2)))))
```

```
(lambda (prvni-nebo-druhy)
  (if prvni-nebo-druhy
      koren1
      koren2))
```

Ten je při aplikaci procedury `koreny` vyhodnocen v posledním prostředí $\mathcal{P}_3$ (viz obrázek 4.4) vytvořeném vyhodnocením `let*`-bloku. Tedy v prostředí, ve kterém jsou známy vazby na symboly `koren1` a `koren2`. V tomto prostředí $\mathcal{P}_3$ nám tedy vzniká procedura, která na základě pravdivostní hodnoty, na kterou je aplikovaná, vrací buďto element navázaný na symbol `koren1` nebo na `koren2`.

Obrázek 4.4. Vznik prostředí při aplikaci procedury z příkladu 4.2




```
(koreny 1 -2 2)
```
$\Longmapsto$ procedura

```
(define oba-koreny (koreny 1 -2 2))
```

```
(oba-koreny #t)                    ⟹  1+i
(oba-koreny #f)                    ⟹  1-i
```

Při aplikaci procedury navázané na symbol `oba-koreny` s různými pravdivostními hodnotami jako argumenty tedy dostáváme jednotlivé kořeny. Je tomu skutečně tak, protože v prostředí vzniku procedury jsou symboly `koren1` a `koren2` navázané na hodnoty vypočtené již při aplikaci procedury `koreny`. Podle toho také můžeme vytvořit selektory `prvni-koren` a `druhy-koren`. Ty budou proceduru zapouzdřující oba kořeny aplikovat na pravdivostní hodnoty, podle toho, který z nich chceme:

```
(define prvni-koren (lambda (oba) (oba #t)))
(define druhy-koren (lambda (oba) (oba #f)))
(prvni-koren oba-koreny)  ⟹  1+i
(druhy-koren oba-koreny)  ⟹  1-i
```

Nyní z tohoto příkladu vytáhneme základní myšlenku – možnost uchovat vazby dvou symbolů v prostředí, které mohou reprezentovat „dvě hodnoty pohromadě“, a možnost přístupu k těmto prvkům pomocí procedur vyšších řádů. Ukážeme tedy, jak naimplementovat tečkové páry pouze s využitím procedur vyšších řádů a s využitím vlastností lexikálního rozsahu platnosti. Bez něj by dále uvedené úvahy o implementaci párů neměly význam (při použití dynamického rozsahu platnosti bychom velmi brzy narazili na fatální problémy). Naše implementace nám zároveň dá odpověď na fundamentální otázku, zda jsou páry definovatelné pomocí procedur vyšších řádů (odpověď bude kladná).

Pár v naší reimplementaci tedy bude procedura, která pro různé argumenty vrací různé prvky páru. Nově vytvářený `cons` bude procedurou vyššího řádu, která bude náš pár vytvářet. Podívejme se na následující implementaci procedury `cons`:

```
(define cons
  (lambda (x y)
    (lambda (k)
      (if k x y))))
```

Při její aplikaci bude vytvořeno prostředí $\mathcal{P}$, ve kterém budou existovat vazby na symboly `x` a `y`. Toto prostředí pak bude prostředím vzniku procedury $\langle$`(k)`, `(if k x y)`, $\mathcal{P}\rangle$. Výsledkem aplikace procedury `cons` tedy bude procedura, která v závislosti na jejím jednom argumentu `k` bude vracet buďto hodnotu navázanou na symbol `x` nebo `y`. Přitom `x` a `y` nejsou mezi formálními argumenty vrácené procedury, ta má pouze formální argument `k`. To jest vazby symbolů `x` a `y` se budou při vyhodnocování těla vrácené procedury hledat v nadřazeném prostředí a to je právě prostředí jejího vzniku, tedy $\mathcal{P}$. Při aplikaci této procedury s různými pravdivostními hodnotami dostáváme jednotlivé složky páru.

```
((cons 1 2) #t)  ⟹  1
((cons 1 2) #f)  ⟹  2
```

Selektory `car` a `cdr` zavolají takto vzniklou proceduru s různým argumentem. Jde vlastně o totéž jako v programu 4.4. Přehledně je to znázorněno na obrázku 4.5.

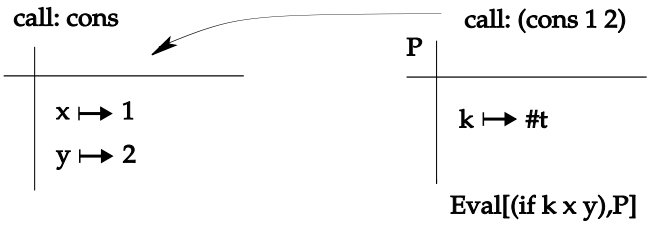
```
(define car (lambda (p) (p #t)))
(define cdr (lambda (p) (p #f)))

(define p (cons 2 3))
p  ⟹  „procedura“

(car p)  ⟹  2
(cdr p)  ⟹  3
```

Můžeme udělat ještě jinou reimplementaci, která je z hlediska funkcionálního programovaní čistější. Provedeme vlastně totéž, ale *s použitím procedur projekce*. Z lekce 2 známe procedury projekce, vracející jeden z jejich argumentů. Pomocí těchto procedur můžeme jednoduše implementovat konstruktor a selektory párů, viz program 4.3. Výsledkem aplikace procedury `cons` bude procedura, v jejímž prostředí vzniku budou na

Obrázek 4.5. Prostředí vznikající při použití vlastní implementace párů



Program 4.3. Implementace tečkových párů pomocí procedur vyšších řádů.

```
(define cons
  (lambda (x y)
     (lambda (proj)
       (proj x y))))

(define 1-z-2 (lambda (x y) x))
(define 2-z-2 (lambda (x y) y))

(define car (lambda (p) (p 1-z-2)))
(define cdr (lambda (p) (p 2-z-2)))
```

symboly `x` a `y` navázané argumenty se kterými byla `cons` aplikována což jsou, stejně jako v předchozím případě, elementy z nichž chceme „vytvořit pár“. Tato procedura nebude brát jako argument pravdivostí hodnotu, nýbrž projekci. Selektory `car` a `cdr` pak opětně ve svém těle volají tuto proceduru s různými projekcemi.

Opět se můžeme vrátit k abstrakčním bariérám založeným na datech. V předchozí podkapitole jsme uvedli, že bariéry založené na datové abstrakci jsou de facto obecnější než bariéry založené na procedurální abstrakci. Když si nyní uvědomíme, že hierarchická data lze plně vyjádřit pouze pomocí procedur vyšších řádů, ihned dostaneme, že bariéry založené na datové abstrakci jsou „stejně silné“ jako bariéry založené na procedurální abstrakci. Podotkněme, že toto pozorování vlastně říká, že nemá příliš smysl odlišovat od sebe procedurální a datovou abstrakci – můžeme se bez újmy bavit pouze o jediné abstrakci (datové = procedurální). V drtivé většině programovacích jazyků si však takový „luxus“ dovolit nemůžeme, protože v nich nelze s programy (respektive s procedurami) zacházet jako s daty ani tak nelze činit obráceně. Právě popsaný pohled na datovou a procedurální abstrakci je silným rysem řady funkcionálních jazyků, především pak dialektů LISPu k nimž patří i jazyk Scheme.

4.5 Symbolická data a kvotování

Doposud jsme všechna složená data, která jsme používali, konstruovali až na výjimky pomocí párů z čísel. Na čísla se lze dívat jako na *primitivní*, nedělitelná neboli *nehierarchická data*. V této sekci rozšíříme vyjadřovací sílu našeho jazyka uvedením možnosti pracovat s libovolnými *symboly* jako s daty. Nejprve řekněme, k čemu by nám to mohlo být dobré. Symboly slouží při programování jako „jména“ a v programech je leckdy potřeba pracovat se jmény jako s daty. Pomocí symboly můžeme například označovat některé části hierarchických datových struktur (uvidíme v dalších lekcích) nebo mohou sloužit přímo jako zpracovávaná data. Například symboly `Praha`, `New-York`, `Prostejov` a další můžeme chápat jako „jména měst“, symboly `red`, `green`, `blue`, . . . můžeme v programech chápat jako symbolická označení barev a tak dále.

Se symboy můžeme pracovat jako s daty díky speciální formě `quote`, která zabraňuje vyhodnocování svého (jediného) argumentu.

Definice 4.16 (speciální forma `quote`)**.** Speciální forma `quote` se používá s jedním argumentem

`(quote` $\langle arg \rangle$`)`.

Výsledkem aplikace této speciální formy je pak $\langle arg \rangle$. Jinými slovy, speciální forma `quote` vrací svůj argument v nevyhodnocené podobě. Přesněji, označíme-li Q speciální formu navázanou na `quote`, pak

$$\mathrm{Apply}(Q, \langle arg \rangle) := \langle arg \rangle.$$ ■

Zde máme příklady aplikace speciální formy `quote`.

```
(quote 10)        ⟼  10
(quote blah)      ⟼  blah
(quote (a . b))   ⟼  (a . b)
```

Poznámka 4.17. (a) Je naprosto zřejmé, že na symbol `quote` musí být navázána speciální forma. Kdyby to byla procedura, muselo by dojít k vyhodnocení jejího argumentu, jak vyplývá z definice 1.21.

(b) Všimněte si, že je zbytečné kvotovat čísla. Stejně tak je zbytečné kvotovat jakýkoli jiný element, který se vyhodnocuje sám na sebe. V takovém případě se totiž vyhodnocený element rovná nevyhodnocenému. Konkrétně třeba výraz `(quote 10)` se vyhodnotí na číslo `10`, ale číslo `10` se vyhodnotí samo na sebe – tedy na číslo `10`.

(c) Slovo „quote“ znamená v angličtině „uvozovka“. Použití kvotování je analogické s použitím uvozovek v přirozeném jazyce. Porovnejte například věty: (i) Zobraz x; a (ii) Zobraz „x“.

Pro zkrácení zápisu je ve Scheme možné namísto `(quote` $\langle arg \rangle$`)` psát jen `'`$\langle arg \rangle$. Tedy můžeme psát:

```
'blah      ⟼  blah
'10        ⟼  10
'(a . b)   ⟼  (a . b)
```

Poznámka 4.18. Apostrof `'` je takzvaný *syntaktický cukr* pro `quote` a neměli bychom jej chápat ani jako samostatný symbol ani jako rozšíření syntaxe jazyka Scheme[6]. Z našeho pohledu je to v podstatě jen metasyntaktická značka (je „nad“ syntaxí jazyka Scheme), kterou zkracujeme delší výraz. *Syntaktický cukr* je *terminus technicus* užívající se pro rozšíření zápisu programu, které jej dělají snadnější („sladší“) pro použití člověkem. Syntaktický cukr dává člověku alternativní způsob psaní kódu, který je praktičtější tím, že je stručnější nebo tím, že je podobný nějaké známé notaci. V případě jednoduché uvozovky ve Scheme vlastně jde o obojí. Zápis je stručnější a podobný použití uvozovek v přirozeném jazyce.

Pozor! Je opět nutné rozlišovat symbol samotný a element navázaný na symbol. Jde o dvě zcela odlišné věci, jak již dobře víme z předchozích lekcí. Nyní se tento kontrast ale zvětšuje, protože symboly jsou pro nás nejen jedny ze symbolických výrazů, ale díky `quote` již může pracovat se symboly jako s elementy jazyka. Například v následujícím kódu navážeme na symbol `plus` různé elementy – jednou proceduru navázanou na symbol `+`:

```
(define plus +)
(plus 1 2)  ⟼  3
plus        ⟼  „procedura sčítání čísel“
```

a podruhé, s použitím kvotování, symbol `+`:

```
(define plus '+)
(plus 1 2)  ⟼  „CHYBA: + není procedura ani speciální forma.“
plus        ⟼  +
```

[6] Při konstrukci konkrétního interpretu jazyka Scheme je však potřeba tento symbol načítat a brát jej v potaz. Reader musí být uzpůsoben k tomu, aby mohl zkrácené výrazy nahrazovat seznamy s `quote`. Takže někdo by v tuto chvíli mohl namítat, že apostrof *je* součástí syntaxe jazyka Scheme. U konkrétních interpretů tomu tak je. Z pohledu abstraktního interpretu Scheme však budeme apostrof uvažovat „mimo jazyk“ a tím pádem nebudeme muset opět rozšiřovat definici symbolických výrazů.

Příklad 4.19. Pomocí `define` můžeme zavést symboly, které se vyhodnocují na sebe sama:

```
(define blah 'blah)
blah  ⟹  blah
```

4.6 Implementace racionální aritmetiky

V této sekci naimplementujeme vlastní racionální aritmetiku. Jedná se o komplexní příklad demonstrující práci s páry, datovou abstrakcí a abstrakčními bariérami.

Nejprve vytvoříme konstruktor `make-r` a selektory `numer` a `denom`. První verze konstruktoru bude vypadat jednoduše – prostě ze dvou argumentů představujících jmenovatele a čitatele vytvoříme tečkový pár.

```
(define make-r (lambda (x y) (cons x y)))
```

Podle toho pak také budou vypadat selektory: `numer` bude vracet první prvek tohoto páru a `denom` jeho druhý prvek.

```
(define numer (lambda (x) (car x)))
(define denom (lambda (x) (cdr x)))
```

Nyní naimplementujeme procedury pro základní aritmetické operace s racionálními čísly: procedury `r+` a `r-` pro sčítání a odčítání. Další procedury `r*` a `r/` budou součástí úkolů na konci lekce. Procedury jsou přímým přepisem předpisu

$$\frac{a}{b} \pm \frac{c}{d} = \frac{ad \pm bc}{bd}$$

Všimněte si, že v kódu už nebudeme používat procedury `cons`, `car` a `cdr`, protože se nacházíme za abstrakční bariérou a nepracujeme již s páry (s konkrétní reprezentací), ale s racionálními čísly (s abstrakcí).

```
(define r+
  (lambda (x y)
    (make-r (+ (* (numer x) (denom y))
               (* (denom x) (numer y)))
            (* (denom x) (denom y)))))

(define r-
  (lambda (x y)
    (make-r (- (* (numer x) (denom y))
               (* (denom x) (numer y)))
            (* (denom x) (denom y)))))
```

Též můžeme naimplementovat celou řadu predikátů. Protože by se jejich kódy téměř nelišily, vytvoříme je pomocí procedury vyššího řádu:

```
(define make-rac-pred
  (lambda (p?)
    (lambda (a b)
      (p? (* (numer a) (denom b))
          (* (numer b) (denom a))))))
```

Touto procedurou vytvoříme vlastní predikáty na porovnávání racionálních čísel:

```
(define r< (make-rac-pred <))
(define r> (make-rac-pred >))
(define r= (make-rac-pred =))
(define r<= (make-rac-pred <=))
(define r>= (make-rac-pred >=))
```

```
(r< (make-r 1 2) (make-r 2 3))  ⟼  #t
(r> (make-r 1 2) (make-r 2 3))  ⟼  #f
```

Dále naprogramujeme procedury `r-max` a `r-min` jako analogie procedur `max` a `min` z lekce 2. Můžeme k tomu směle využít proceduru vyššího řádu `extrem`. Tuto proceduru jsme definovali v programu 2.11 na straně 69.

```
(define r-max (extrem r>))
(define r-min (extrem r<))
```

Máme tedy procedury na výběr většího či menšího racionálního čísla:

```
(r-max (make-r 1 2) (make-r 2 3))  ⟼  (2 . 3)
(r-min (make-r 1 2) (make-r 2 3))  ⟼  (1 . 2)
```

Nyní uděláme změnu v konstruktoru racionálního čísla, tak, aby zlomek při vytváření automaticky krátil. Tedy čitatele i jmenovatele vydělí jejich největším společným jmenovatelem. Zbytek kódu (ostatní procedury) se nezmění.

```
(define make-r
  (lambda (x y)
    (let ((g (gcd x y)))
      (cons (/ x g) (/ y g)))))
```

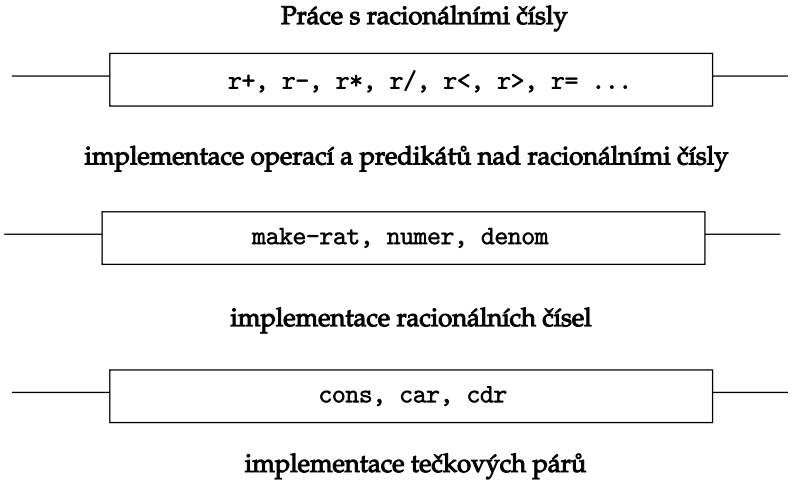
Teď provedeme reimplementaci `cons`, `car` a `cdr` tak, jak je to v programu 4.3. Dále pro vypisování doděláme proceduru `r->number`. To proto, že po reimpelemtaci budou racionální čísla procedury, jejichž externí reprezentace nám neprozradí jejich složení.

```
(define r->number
  (lambda (x)
    (/ (numer x) (denom x))))
```

Všimněte si, že zbytek programu bude fungovat, aniž bychom jej museli měnit. Modifikovali jsme nejspodnější vrstvu programu (viz obrázek 4.6) bez nutnosti měnit vrstvy, které jsou nad ní. Ve vyšších vrstvách je změna neznatelná. Například procedury `r-max` a `r-min` budou pracovat bez jakékoli změny:

```
(r->number (r-max (make-r 1 2) (make-r 2 3)))  ⟼  2/3
(r->number (r-min (make-r 1 2) (make-r 2 3)))  ⟼  1/2
```

Obrázek 4.6. Vrstvy v implementaci racionální aritmetiky



A teď ještě jednou změníme procedury navázané na symboly `cons`, `car` a `cdr`. Tentokrát tak, že zaměníme pořadí prvků v páru. Takže například výraz `(cons 1 2)` se nevyhodnotí na pár `(1 . 2)`, ale na pár `(2 . 1)`. Tedy opět změníme nejspodnější vrstvu programu.

```
(define kons cons)
(define kar car)
(define kdr cdr)

(define cons (lambda (x y) (kons y x)))

(define car (lambda (p) (kdr p)))
(define cdr (lambda (p) (kar p)))
```

Díky dodržování abstrakčních bariér se však tato změna neprojeví ve vyšších vrstvách:

```
(r->number (r-max (make-r 1 2) (make-r 2 3)))  ⟹  2/3
(r->number (r-min (make-r 1 2) (make-r 2 3)))  ⟹  1/2
```

Shrnutí

V této lekci jsme se zabývali vytvářením abstrakcí pomocí dat. Představili jsme si tečkové páry, pomocí nichž vytváříme hierarchické datové struktury. Rozšířili jsme symbolické výrazy o tečkové páry a doplnili Eval. Ukázali jsme rovněž, že tečkové páry bychom mohli plně definovat pouze pomocí procedur vyšších řádů. Dále jsme se zabývali kvotováním a jeho zkrácenou notací, jenž nám umožní chápat symboly jako data. Jako příklad datové abstrakce jsme uvedli implementaci racionální aritmetiky.

Pojmy k zapamatování

- datová abstrakce
- konstruktor
- kvotování
- selektor
- symbolická data
- tečkové páry
- syntaktický cukr
- konkrétní datová reprezentace

Nově představené prvky jazyka Scheme

- procedury `cons`, `car`, `cdr`
- speciální forma `quote`

Kontrolní otázky

1. *Co je datová abstrakce?*
2. *Co jsou páry?*
3. *Jak vypadá externí reprezentace párů?*
4. *Co je boxová notace?*
5. *Jak jsme změnili definici S-výrazu?*
6. *Co je to kvotování?*
7. *Jaká je zkrácená notace kvotování?*
8. *Je možné naimplementovat páry pomocí procedur vyšších řádů? Jak?*

Cvičení

1. Napište bez použití interpretru vyhodnocení následujících výrazů:

```
cons                                ⟹
(cons 1 2)                          ⟹
(cons car cdr)                      ⟹
(cdr (cons cdr cdr))                ⟹
'cons                               ⟹

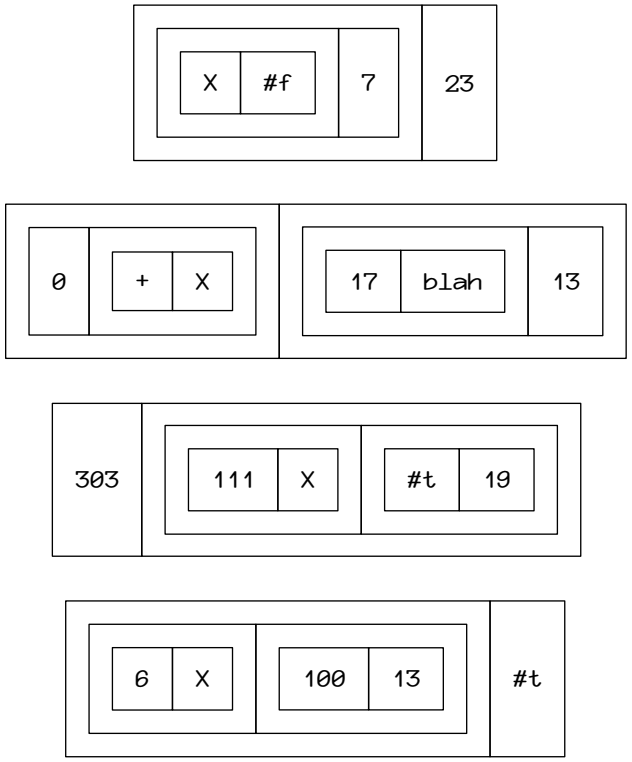
(cons '(cons . cons) car)           ⟹
(cons lambda 'x)                    ⟹
('+ 1 2 3 4)                        ⟹
'10                                 ⟹
'(20 . 40)                          ⟹

(caar (cons (cons 1 2) 3))          ⟹
(cadr '((1 . 2) . 4))               ⟹
(car 'cons)                         ⟹
(cdr 1 2)                           ⟹
(define 'a 10)                      ⟹

(if 'symbol 10 20)                  ⟹
((car (cons and or)))               ⟹
```

2. Mějme hierarchická data znázorněná v boxové notaci na obrázku 2. Napište výrazy, jejichž vyhodnocením vzniknou.

Obrázek 4.7. Boxová notace tečkových párů – zadání ke cvičení



3. Napište páry z obrázku 2 v tečkové notaci.
4. Napište sekvenci procedur `car` a `cdr` (respektive složeniny), kterými se v hierarchických strukturách z obrázku 2 dostaneme k prvku `X`.

5. Napište proceduru `map-pair`, která bere jako argumenty proceduru o jednom argumentu ⟨*proc*⟩ a pár ⟨*pár*⟩ a vrací pár výsledků aplikací procedury ⟨*proc*⟩ na prvky páru ⟨*pár*⟩. Příklady aplikace:
   `(map-pair - '(1 . 2))` ⟹ `(-1 . -2)`
   `(map-pair (lambda (op) (op 3)) (cons + -))` ⟹ `(3 . -3)`
6. Upravte konstruktor racionálních čísel tak, aby jmenovatel byl vždy přirozené číslo. Například:
   `(define r (make-r 3 -4))`
   `(numer r)` ⟹ `-3` `(denom r)` ⟹ `4`
   nikoli
   `(numer r)` ⟹ `3` `(denom r)` ⟹ `-4`
7. Napište procedury `r*` a `r/` na násobení a dělení racionálních čísel.

Úkoly k textu

1. Podobným způsobem jako jsme naimplementovali páry pomocí procedur vyšších řádů (tedy bez použití procedur `cons`, `car`, `cdr`) naimplementujte uspořádané trojice.
2. Pomocí procedur `kons`,`kar` a `kdr` vytvořte obdoby párů z obrázku a nakreslete, jak vypadají vzniklá hierarchie prostředí a vazby v nich.
3. Zamyslete se nad potřebou zpracovávat složená data. Uveďte jiné příklady, než jsou ukázány v úvodu této lekce, ze kterých tato potřeba vyplývá.

Řešení ke cvičením

1. procedura cons, (1 . 2), (procedura car . procedura cdr), procedura cdr, cons, ((cons . cons) procedura car), (specialni forma lambda . x), chyba, 10, (20 . 40) 1, chyba, chyba, chyba, chyba 10, #t
2. ```
   (cons (cons (cons 'X #f) 7) 23)

   (cons
    (cons 0 (cons '+ 'X))
    (cons (cons 17 'blah) 13))

   (cons 303
         (cons (cons 111 'X)
               (cons #t 19)))

   (cons (cons (cons 6 'X)
               (cons 100 13))
         #t)
   ```
3. ```
   (((X . #f) . 7) . 23)
   ((0 + . X) (17 . blah) . 13)
   (303 (111 . X) #t . 19)
   (((6 . X) 100 . 13) . #t)
   ```
4. `caaar`, `cddar`, `cdadr`, `cdaar`
5. ```
   (define map-pair
     (lambda (f p)
       (cons (f (car p)) (f (cdr p)))))
   ```
6. ```
   (define make-rac
     (lambda (n d)
       (let ((div ((if (< d 0) - +) (gcd n d))))
         (cons (/ n div) (/ d div)))))
   ```

7.
```
(define r*
  (lambda (a b)
    (make-r (* (numer a) (numer b))
            (* (denom a) (denom b)))))
```

```
(define r/
  (lambda (a b)
    (make-r (* (numer a) (denom b))
            (* (denom a) (numer b)))))
```

Lekce 5: Seznamy

Obsah lekce: V této lekci se budeme zabývat seznamy, což jsou speciální hierarchické datové struktury konstruované pomocí tečkových párů. Ukážeme několik typických procedur pro manipulaci se seznamy: procedury pro konstrukci seznamů, spojování seznamů, obracení seznamů, mapovací proceduru a další. Vysvětlíme, jaký je vztah mezi seznamy chápanými jako data a seznamy chápanými jako symbolické výrazy. Dále představíme typový systém jazyka Scheme a poukážeme na odlišnost abstraktního interpretu od skutečných interpretů jazyka Scheme, které musí řešit správu paměti.

Klíčová slova: mapování, reverze seznamu, seznam, spojování seznamů, správa paměti, typový systém.

5.1 Definice seznamu a příklady

V minulé lekci jsme uvedli složenou strukturu, kterou jsme nazývali *tečkový pár*. Tečkové páry jsou elementy jazyka, které v sobě agregují dva další elementy. Jelikož páry jsou samy o sobě elementy jazyka, můžeme konstruovat páry jejichž (některé) prvky jsou opět páry. Tímto způsobem lze vytvářet libovolně složité hierarchické struktury postupným „vnořováním párů“ do sebe. V této lekci se zaměříme na speciální případ takových hierarchických dat konstruovaných z tečkových párů, na takzvané *seznamy*.

Než přikročíme k definici seznamu, potřebujeme zavést nový speciální element jazyka. Tímto elementem je *prázdný seznam* (nazývaný též *nil*). Tento element bude, intuitivně řečeno, reprezentovat „seznam neobsahující žádný prvek“ (zpřesnění toho pojmu uvidíme dále). Externí reprezentace prázdného seznamu je `()`. Dále platí, že externí reprezentace prázdného seznamu je *čitelná readerem*. Element „prázdný seznam“ se dle bodu (D) definice vyhodnocování 2.7 uvedené na straně 48 bude *vyhodnocovat sám na sebe*. Máme tedy:

```
()  ⟼  ()
```

Element *prázdný seznam* je pouze jeden, nemá se tedy smysl bavit o „různých prázdných seznamech“. Připomeňme, že z předchozího výkladu již známe několik dalších elementů určených ke speciálním účelům. Jsou to elementy *pravda*, *nepravda* a element *nedefinovaná hodnota*.

Poznámka 5.1. V některých interpretech není pravdou, že se prázdný seznam vyhodnocuje sám na sebe a vyhodnocení výrazu `()` skončí chybou „CHYBA: Pokus o vyhodnocení prázdného seznamu“. Element *prázdný seznam* můžeme v takových interpretech získat pomocí kvotování tím, že potlačíme vyhodnocení výrazu `()`, viz příklady:

```
(quote ())  ⟼  ()
'()         ⟼  ()
```

Ve všech programech v této a v následujících lekcích budeme uvádět prázdný seznam vždy ve kvotovaném tvaru. Dodejme, že v některých dialektech jazyka LISP je prázdný seznam navázaný na symbol `nil`.

Nyní můžeme zavést pojem samotný seznam.

Definice 5.2. *Seznam* je každý element L jazyka Scheme splňující právě jednu z následujících podmínek:

1. L je prázdný seznam (to jest L je element vzniklý vyhodnocením `'()`), nebo
2. L je pár ve tvaru $(E\ .\ L')$, kde E je libovolný element a L' je seznam. V tomto případě se element E nazývá *hlava seznamu* L a seznam L' se nazývá *tělo seznamu* L (řidčeji též *ocas seznamu* L).

Předpokládejme, že seznam L je ve tvaru $(E\ .\ L')$. Pod pojmem *prvek seznamu* L rozumíme element E (*první prvek seznamu* L) a dále prvky seznamu L'. Počet všech prvků seznamu se nazývá *délka seznamu*. Prázdný seznam `()` *nemá žádný prvek*, to jest má *délku* 0. ■

Před tím, než uvedeme příklady seznamů, si popíšeme jejich externí reprezentaci. Jelikož je každý seznam buďto párem, nebo prázdným seznamem, souhlasí externí reprezentace seznamů s reprezentacemi těchto elementů. Reprezentaci tečkových párů, kterou jsme si ukázali v předchozí lekci ještě upravíme. Řekli jsme si, že v případě, kdy je druhým prvkem páru opět pár, používáme zkrácený zápis, viz podrobný popis v sekci 4.2. Zkrácení zápisu spočívalo v odstranění tečky náležející reprezentaci vnějšího páru a závorek náležejících vnitřnímu páru. V případě, že druhým prvkem páru je prázdný seznam, zkracujeme stejným způsobem: odstraníme tečku z reprezentace páru a *obě závorky* z reprezentace prázdného seznamu – reprezentace prázdného seznamu tak nebude zobrazena vůbec. Viz příklad následující příklad.

Příklad 5.3. V následujících ukázkách jsou zobrazeny zkrácené externí reprezentace seznamů (uprostřed) a jejich slovní popis (vpravo):

`()`	`= ()`	prázdný seznam
`(a . ())`	`= (a)`	jednoprvkový seznam obsahující symbol `a`
`(a . (b . ()))`	`= (a b)`	dvouprvkový seznam obsahující symboly `a` a `b`
`(1 . (2 . (3 . ())))`	`= (1 2 3)`	tříprvkový seznam
`(1 . ((20 . 30) . ()))`	`= (1 (20 . 30))`	dvouprvkový seznam jehož druhý prvek je pár
`((1 . ()) . ((2 . ()) . ()))`	`= ((1) (2))`	dvouprvkový seznam obsahující jednoprvkové seznamy

Upozorněme na fakt, že i prázdný seznam je element, tedy prázdný seznam sice nemůže obsahovat žádné prvky, ale může být prvek jiných (neprázdných) seznamů. Viz následující příklady:

`(() . ())`	`= (())`	jednoprvkový seznam obsahující prázdný seznam
`(() . (b . ()))`	`= (() b)`	dvouprvkový seznam s prázdným seznamem jako prvním prvkem
`(a . (() . ()))`	`= (a ())`	dvouprvkový seznam s prázdným seznamem jako druhým prvkem
`(() . (() . ()))`	`= (() ())`	dvouprvkový seznam jehož oba prvky jsou prázdné seznamy

Jak je již z předchozích ukázek patrné, zkrácená externí reprezentace seznamů je ve tvary symbolických výrazů tak, jak jsme je zavedli v první lekci v definici 1.6 na straně 19. Korespondence mezi symbolickými výrazy (seznamy) a elementy jazyka (seznamy) se budeme ještě podrobněji zabývat.

Neprázdné seznamy jsou tedy páry, a tak pro ně platí vše, co bylo řečeno v předchozí lekci o párech. Z pohledu seznamů můžeme pohlížet na konstruktor páru `cons` a selektory `car` a `cdr` následovně:

konstruktor `cons` bere dva argumenty, libovolný element E a seznam L (v tomto pořadí), a vrací seznam který vznikne přidáním elementu E na začátek seznamu L.

selektor `cdr` bere jako argument neprázdný seznam a vrací seznam, který vznikne z původního seznamu odebráním prvního prvku. Jinými slovy, `cdr` pro daný seznam vrací jeho *tělo.*

selektor `car` bere jako argument neprázdný seznam a vrací jeho první prvek. Jinými slovy, `car` pro daný seznam vrací jeho *hlavu.*

Příklad 5.4. Příklady konstrukce seznamů použitím procedury `cons`.

`(cons 'a '())`	$\longmapsto$	`(a . ()) = (a)`
`(cons 'a (cons 'b '()))`	$\longmapsto$	`(a . (b . ())) = (a b)`
`(cons 1 (cons 2 (cons 3 '())))`	$\longmapsto$	`(1 . (2 . (3 . ()))) = (1 2 3)`
`(cons 1 (cons (cons 20 30) '()))`	$\longmapsto$	`(1 . ((20 . 30) . ())) = (1 (20 . 30))`
`(cons (cons 10 20) (cons 3 '()))`	$\longmapsto$	`((10 . 20) . (3 . ())) = ((10 . 20) 3)`
`(cons (cons 1 '()) (cons 2 '()))`	$\longmapsto$	`((1 . ()) . (2 . ())) = ((1) 2)`
`(cons (cons 1 '()) (cons (cons 2 '()) '()))`	$\longmapsto$	`((1 . ()) . ((2 . ()) . ())) = ((1) (2))`

V následujících ukázkách vidíme konstrukci seznamů obsahujících prázdné seznamy jako svoje prvky:

`(cons '() '())`	$\longmapsto$	`(() . ()) = (())`
`(cons '() (cons 'b '()))`	$\longmapsto$	`(() . (b . ())) = (() b)`
`(cons 'a (cons '() '()))`	$\longmapsto$	`(a . (() . ())) = (a ())`
`(cons '() (cons '() '()))`	$\longmapsto$	`(() . (() . ())) = (() ())`

Následující hierarchické struktury vzniklé použitím `cons` *nejsou seznamy*:

```
(cons 10 (cons 20 30))            ⟼  (10 . (20 . 30)) = (10 20 . 30)
(cons 'a (cons 'b (cons 'c 'd)))  ⟼  (a . (b . (c . d))) = (a b c . d)
```

Pomocí procedur `cons`, `car` a `cdr` můžeme vytvářet další odvozené konstruktory a selektory pro práci se seznamy. Například můžeme definovat pomocné procedury pro přístup k prvním několika prvkům seznamu.

```
(define first car)
(define second cadr)
(define third caddr)
```

Nebo jednoduché konstruktory seznamů. Třeba procedury tvořící jedno-, dvou- nebo tříprvkové seznamy:

```
(define jednoprvkovy
  (lambda (x)
    (cons x '())))

(define dvouprvkovy
  (lambda (x y)
    (cons x (cons y '()))))

(define triprvkovy
  (lambda (x y z)
    (cons x (cons y (cons z '())))))
```

Analogicky bychom samozřejmě mohli vyrábět procedury na vytváření víceprvkových seznamů.

Příklad 5.5. Upozorněme znovu na fakt, že ne každý seznam je pár. Konkrétně *prázdný seznam není pár.* Vyhodnocení následujícího kódu proto skončí chybou.

```
(car '())  ⟼  „CHYBA: Argument není pár“
(cdr '())  ⟼  „CHYBA: Argument není pár“
```

Poznámka 5.6. V předchozím textu jsme neprázdné seznamy zavedli jako páry jejichž druhým prvkem byl seznam. To byla v podstatě jakési naše „úmluva“ jak reprezentovat lineární datovou strukturu (seznam) pomocí zanořených párů. Samozřejmě, že principiálně nám nic nebrání v tom, abychom udělali jinou úmluvu. Například bychom mohli zaměnit význam prvního a druhého prvku a seznam bychom mohli nadefinovat jako libovolný element L splňující právě jednu z následujících podmínek:

1. L je prázdný seznam, nebo
2. L je pár ve tvaru $(L' \,.\, E)$, kde L' je seznam a E je libovolný element.

Pak bychom například výsledek vyhodnocení výrazu

```
(cons 1 (cons 2 (cons 3 '())))  ⟼  (1 . (2 . (3 . ())))
```

nepovažovali za seznam, ale vyhodnocení následujícího výrazu ano:

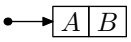
```
(cons (cons (cons '() 1) 2) 3)  ⟼  ((() . 1) . 2) . 3)
```

Neformálně řečeno, jediný rozdíl v obou chápáních seznamu je ve „směru“ v jakém se do sebe tečkové páry zanořují. Ve zbytku textu budeme vždy uvažovat zavedení seznamu tak, jak jsme jej uvedli v definici 5.2 na začátku této lekce.

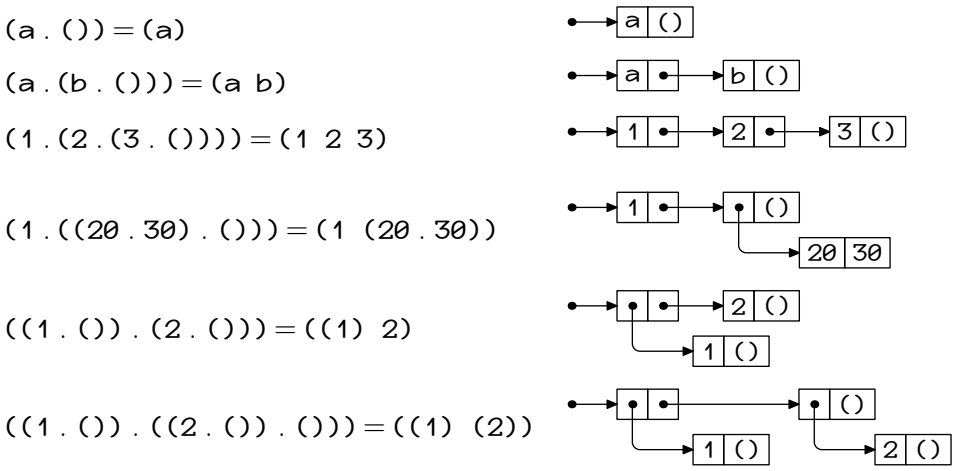
Mimo tečkové notace párů jsme v lekci 4 také představili *boxovou notaci* párů. Tu teď trochu upravíme: Za samotný tečkový pár budeme považovat pouze „tečku“, která obsahuje *ukazatel* na určité místo v paměti, kde je uložen první a druhý prvek páru. Pro ilustraci viz obrázek 5.1.

Naše úprava původní boxové notace má dvě výhody. První výhodou je, že tato notace je přehlednější. Člověk se v nákresu hierarchické struktury v původní notaci snadno ztratí kvůli množství do sebe vnořených boxů.

Obrázek 5.1. Boxová notace tečkového páru používající ukazatel



Obrázek 5.2. Seznamy z příkladu 5.4 v boxové notaci



V nové notaci je každý pár zakreslen zvlášť. Pokud je některý pár obsažen v jiném páru, v diagramu se to promítne tak, že první nebo druhou složkou páru je pouze tečka, ze které vede ukazatel do obsaženého páru. Příklady jsou uvedeny v obrázku 5.2. Další výhodou upravené notace je její větší korespondence s fyzickým uložením párů v pamětích počítačů.

V této lekci jsme vlastně poprvé nahlédli poněkud blíž k fyzické reprezentaci hierarchických dat v počítačích i když, na druhou stranu, pořád se o ní bavíme na dost abstraktní úrovni, která nám zaručuje jisté „pohodlí“. Ve většině interpretů funkcionálních jazyků jsou seznamy reprezentovány dynamickými datovými strukturami, které na sebe vzájemně ukazují pomocí *ukazatelů* – *pointerů* (anglicky *pointers*).

5.2 Program jako data

V definici 1.6 na straně 19 jsme zavedli pojem seznam, jako speciální případ symbolického výrazu. Symbolický výraz je čistě syntaktický pojem, který jsme zavedli proto, abychom mohli formálně popsat programy v jazyku Scheme – programy jsou posloupnosti symbolických výrazů. Jinými slovy, *seznam chápaný jako symbolický výraz* je tedy *část programu.*

V této lekci jsme však doposud pracovali s jiným pojmem „seznam“. Konkrétně jsme pracovali se seznamy, které jsme chápali jako speciální elementy jazyka (speciální hierarchická data). Nabízí se přirozená otázka, jaký je vztah mezi seznamy chápanými jako symbolické výrazy a seznamy chápanými jako elementy jazyka? Z předchozích ukázek je patrné, že se externí reprezentace elementů typu seznam (ve své zkrácené podobě) shoduje s tím, jak jsme zavedli seznamy jako symbolické výrazy. Zde je samozřejmě nutnou podmínkou, aby měl element seznam externí reprezentaci čitelnou readerem, tedy jeho prvky by měly být pouze čísla, symboly a páry (to zahrnuje i další seznamy). Například seznam vzniklý vyhodnocením

```
(cons 'a (cons (lambda () 10) '()))
```
$\Longmapsto$ `(a` „konstantní procedura“)

nemá čitelnou externí reprezentaci, protože jeho druhým prvkem je procedura. Externí reprezentace takto vzniklého seznamu tedy není symbolický výraz.

Nyní se zaměříme na opačný problém. V okamžiku, kdy reader přečte vstupní seznam (symbolický výraz), tak jej převede do jeho interní reprezentace. V první lekci, kde jsme se o symbolických výrazech a jejich

externích reprezentacích bavili, jsme záměrně neřekli, jak interní reprezentace seznamů vypadají. Udělali jsme to proto, že až teď je možné tvar interní reprezentace zcela pochopit. *Interní reprezentací symbolického výrazu seznam* je právě hierarchická struktura –seznam– zavedená v definici 5.2 v této lekci. Reader při čtení vstupního symbolického výrazu podle něj konstruuje odpovídající hierarchickou strukturu, která je uložena v paměti interpretu. Dál se již pracuje pouze s touto hierarchickou strukturou, která je interní reprezentací načítaného symbolického výrazu. Zároveň taky platí, že externí reprezentace této hierarchické struktury se shoduje s výchozím symbolickým výrazem. Uvědomte si, že z tohoto nového pohledu je vlastně *program totéž co data.*

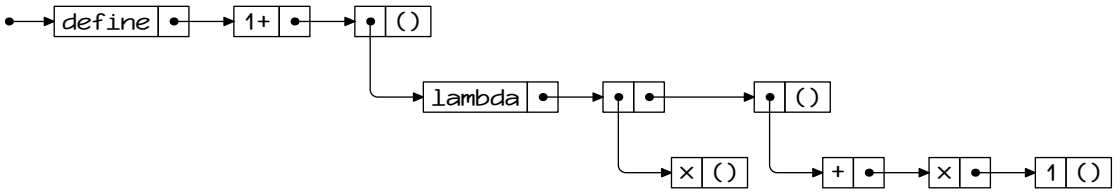
V dialektech jazyka LISP, k nimž patří i jazyk Scheme, se programy chápou jako totéž co data, protože programy se skládají se seznamů. V této lekci navíc dál uvidíme, že zkonstruované seznamy během výpočtu budeme schopni vyhodnocovat. Chápání programů jako dat je doménou prakticky pouze jen programovacích jazyků z rodiny dialektů LISPu. V ostatních programovacích jazycích je datová reprezentace programů buďto skoro nemožná kvůli neprakticky složité syntaxi jazyka nebo je těžkopádná.

Objasněme si tuto problematiku na příkladu. Uvažujme následující program, který je de facto složen z jediného symbolického výrazu, kterým je seznam:

```
(define 1+
  (lambda (x)
    (+ x 1)))
```

Reader předchozí seznam načte a vytvoří podle něj hierarchickou strukturu (element seznam), která je zobrazena v obrázku 5.3.

Obrázek 5.3. Program `(define 1+ (lambda (x) (+ x 1)))` jako data.



Nyní se můžeme vrátit k vyhodnocování párů. V předchozí lekci jsme uvedli, že se zatím vyhodnocováním párů nebudeme zabývat. Nyní, jelikož víme, že seznamy jsou konstruované z párů, se nabízí rozšířit vyhodnocovací proces i pro páry. Při vyhodnocování tečkového páru rozlišíme dvě situace. Konkrétně bude důležité, zda je tento pár seznam, nebo ne. V případě, že ano, vyhodnocujeme tento pár = seznam, jako dříve. V opačném případě skončí vyhodnocování chybou. Vyhodnocení elementu E

Definice 5.7 (vyhodnocení elementu $\boldsymbol{E}$ v prostředí $\boldsymbol{\mathcal{P}}$)**.**
Výsledek vyhodnocení elementu E v prostředí $\mathcal{P}$, značeno $\mathrm{Eval}[E, \mathcal{P}]$, je definován:

(A) Pokud je E *číslo*, pak $\mathrm{Eval}[E, \mathcal{P}] := E$.

(B) Pokud je E *symbol*, mohou nastat tři situace:

(B.1) Pokud $E \mapsto_{\mathcal{P}} F$, pak $\mathrm{Eval}[E, \mathcal{P}] := F$.

(B.2) Pokud E *nemá vazbu v* $\mathcal{P}$ a pokud $\mathcal{P}' \prec \mathcal{P}$, pak $\mathrm{Eval}[E, \mathcal{P}] := \mathrm{Eval}[E, \mathcal{P}']$.

(B.e) Pokud E *nemá vazbu v* $\mathcal{P}$ a pokud $\mathcal{P}$ *je globální prostředí*, pak ukončíme vyhodnocování hlášením „CHYBA: Symbol E nemá vazbu.“.

(C) Pokud je E *tečkový pár* tvaru $(E_1 \ . \ E_a)$, pak mohou nastat dvě situace:

(C.α) E_a je seznam ve tvaru $(E_2 \ E_3 \cdots E_n)$. Pak $F_1 := \mathrm{Eval}[E_1, \mathcal{P}]$. Dále rozlišujeme tři situace:

(C.1) Pokud F_1 je *procedura*, pak se v nespecifikovaném pořadí vyhodnotí $E_2, \ldots, E_n$:

$$\begin{aligned} F_2 &:= \mathrm{Eval}[E_2, \mathcal{P}], \\ F_3 &:= \mathrm{Eval}[E_3, \mathcal{P}], \\ &\vdots \quad\quad \vdots \\ F_n &:= \mathrm{Eval}[E_n, \mathcal{P}]. \end{aligned}$$

Potom položíme $\mathrm{Eval}[E, \mathcal{P}] := \mathrm{Apply}[F_1, F_2, \ldots, F_n]$.

(C.2) Pokud F_1 je *speciální forma*, pak $\mathrm{Eval}[E] := \mathrm{Apply}[F_1, E_2, \ldots, E_n]$.

(C.e) Pokud F_1 není *procedura ani speciální forma*, skončíme vyhodnocování hlášením „CHYBA: Nelze provést aplikaci: E se nevyhodnotil na proceduru ani na speciální formu.“.

(C.β) E_a není seznam. Pak skončíme vyhodnocování chybovým hlášením „CHYBA: Nelze provést aplikaci: E_a není seznam argumentů.“.

(D) Ve všech ostatních případech klademe $\mathrm{Eval}[E, \mathcal{P}] := E$. ■

Poznámka 5.8. Protože seznamy jako S-výrazy jsou interně reprezentovány jako hierarchické struktury konstruované z párů, mohli bychom programy psát přímo jako tečkové páry. Například jednoduchý program `(+ (* 7 3) 5)` bychom též mohli psát třeba těmito způsoby:

```
(+ (* 7 3) . (5))                          ⟹  26
(+ (* 7 3) . (5 . ()))                     ⟹  26
(+ (* 7 . (3)) 5)                          ⟹  26
(+ . ((* . (7 . (3 . ()))) . (5 . ())))    ⟹  26
```

Při praktickém programování se však výše uvedené vyjadřování kódu přímo pomocí párů nepoužívá.

V sekci 4.5 jsme si ukázali speciální formu `quote` a kvotování. Stejně tak, jako jsme kvotovali čísla, symboly a páry, můžeme samozřejmě také kvotovat i seznamy:

```
(quote ())                          ⟹  ()
(quote (+ 1 2))                     ⟹  (+ 1 2)
'(+ 1 2)                            ⟹  (+ 1 2)
'(1 2 3 4)                          ⟹  (1 2 3 4)
'(abbe blangis curval durcet)       ⟹  (abbe blangis curval durcet)
'(1 (2 (3)) 4)                      ⟹  (1 (2 (3)) 4)
```

Poznámka 5.9. Kdybychom se pokusili kvotovat dvojnásobně – třeba takto:

```
''blah  ⟹  (quote blah)
```

obdržíme jako výsledek skutečně seznam `(quote blah)`. To by nás ale nemělo překvapit. Když přepíšeme výraz `''blah` bez použití apostrofů (pokud odstraníme syntaktický cukr), dostáváme výraz `(quote (quote blah))`. Tento výraz se vyhodnocuje následujícím způsobem. Na symbol `quote` je navázána speciální forma, které je při aplikaci předán jako argument seznam `(quote blah)` v nevyhodnocené podobě. Speciální forma `quote` tento seznam bez dalšího vyhodnocování vrací. Máme tedy:

```
(quote (quote blah))  ⟹  (quote blah)
```

což je stejný výsledek, který dostaneme při použití apostrofů místo `quote`.

U problematiky procedur jako dat ještě zůstaňme. V první lekci jsme hovořili o readeru jako o části interpretu, která slouží k převodu symbolických výrazů do jejich interních forem. Reader je nám v jazyku k dispozici i jako procedura. Procedura `read` je procedura jednoho argumentu (podle standardu R^{5}RS je to procedura s volitelným druhým parametrem, ale my jím zabývat nebudeme), která čeká na uživatelův vstup a po jeho dokončení vrací interní reprezentaci uživatelem zadaného výrazu. Uvažujme například následující kód:

```
(+ 1 (read))
```

Po jeho vyhodnocení bude uživatel interpretem vyzván k zadání vstupu. Pokud uživatel na vstup zadá například `100`, pak reader převede tento řetězec snaků na element „číslo 100“, které bude přičteno k jedničce a výsledkem je tedy `101`. Kdybychom například vzali výraz

```
(let ((x (read)))
  (cons '+ (cons 1 x)))
```

a kdybychom uvažovali, že uživatel po vyzvání zadá na vstup řetězec znaků „`(a (b 10) c)`“, pak jej reader načte jako seznam a převede jej do interní formy. Tento seznam je dále navázán v lokálním prostředí na `x` a v tomto prostředí bude vyhodnoceno tělo `let`-bloku. Výsledkem vyhodnocení by tedy byl pětiprvkový seznam `(+ 1 a (b 10) c)`. Pomocí `read` tedy můžeme načítat data pomocí uživatelského vstupu. Obzvlášť zajímavé to je v případě, kdy data chápeme jako program. O tom se budeme bavit v následující lekci a k použití `read` se ještě vrátíme.

5.3 Procedury pro manipulaci se seznamy

V této sekci si představíme několik užitečných procedur pro vytváření seznamů a práci se seznamy. Každou proceduru podrobně popíšeme, uvedeme příklady použití a v případě některých procedur rovněž ukážeme, jak bychom je mohli implementovat, kdyby nebyly v interpretu k dispozici.

První z těchto procedur je procedura (konstruktor) `list` vytvářející seznam podle výčtu jeho prvků v daném pořadí. Proceduru lze použít s libovolným (i nulovým) počtem argumentů ve tvaru:

`(list` $\langle arg_1 \rangle$ $\langle arg_2 \rangle \cdots \langle arg_n \rangle$`)`

Výsledkem její aplikace je pak seznam těchto argumentů `(`$\langle arg_1 \rangle$ $\langle arg_2 \rangle \cdots \langle arg_n \rangle$`)`.

```
(list)                    ⟹  ()
(list 1 2 3)              ⟹  (1 2 3)
(list + 1 2)              ⟹  („procedura sčítání“ 2 3)
(list '+ 1 2)             ⟹  (+ 1 2)
(list (+ 1 2))            ⟹  3
(list (list 1) (list 2))  ⟹  ((1) (2))
```

Poznámka 5.10. Je důležité uvědomit si rozdíl mezi konstrukcí seznamu vznikající vyhodnocením

`(list` $\langle arg_1 \rangle$ $\langle arg_2 \rangle \cdots \langle arg_n \rangle$`)`

a zdánlivě ekvivalentním použitím speciální formy `quote`:

`(quote (`$\langle arg_1 \rangle$ $\langle arg_2 \rangle \cdots \langle arg_n \rangle$`))`

Speciální forma `quote` zabrání vyhodnocování svého argumentu. Zatímco `list` je procedura a během vyhodnocování prvního z uvedených výrazů jsou před samotnou aplikací `list` vyhodnoceny všechny předané argumenty. Ukažme si rozdíly na příkladech:

(a) V prvním příkladě získáme stejné seznamy, uvažované prvky se totiž vyhodnocují samy na sebe:

```
(list 1 2 #t ())      ⟹  (1 2 #t ())
(quote (1 2 #t ()))   ⟹  (1 2 #t ())
```

(b) V druhém příkladě budeme uvažovat dva prvky – seznam a symbol s navázanou hodnotou:

```
(define x 10)
(list (+ 1 2) x)      ⟹  (3 10)
(quote ((+ 1 2) x))   ⟹  ((+ 1 2) x)
```

V případě procedury `list` došlo k vyhodnocení argumentů, kdežto při použití `quote` nikoliv.

Jiným konstruktorem seznamů je procedura `build-list`. Ta se používá se dvěma argumenty. Prvním z nich je číslo n určující délku konstruovaného seznamu a druhým argumentem je tak zvaná *tvořící procedura* F, což je procedura jednoho argumentu. Tvořící procedura má jediný účel a to pro daný index vracet prvek, který chceme na dané pozici do seznamu vložit. Výsledkem aplikace procedury `build-list` je tedy seznam o zadané délce n, jehož prvky jsou výsledky aplikací tvořící procedury na čísla $0, 1, \ldots, n-1$. Formálně zapsáno, výsledkem aplikace `build-list` je seznam

$$(\mathrm{Apply}[F, 0] \;\; \mathrm{Apply}[F, 1] \cdots \mathrm{Apply}[F, n-1]).$$

Viz následující vysvětlující příklady:

```
(build-list 5 (lambda (x) x))   ⟹  (0 1 2 3 4)
(build-list 5 (lambda (x) 2))   ⟹  (2 2 2 2 2)
(build-list 0 (lambda (x) x))   ⟹  ()
(build-list 5 -)                ⟹  (0 -1 -2 -3 -4)
(build-list 5 list)             ⟹  ((0) (1) (2) (3) (4))
```

Podotkněme, že procedura `build-list` je jednou z procedur, které ukazují praktičnost použití procedur vyšších řádů. Procedura `build-list` je procedura vyššího řádu, protože bere jako jeden ze svých dvou argumentů proceduru. Tato předaná procedura slouží programátorům k vyjádření toho, jaké prvky mají být v seznamu na daných pozicích. Procedura `build-list` je vlastně nejobecnějším konstruktorem seznamů dané délky s důležitou vlastností: hodnoty prvků obsažených v těchto seznamech závisí pouze na jejich pozici.

Poznámka 5.11. Všimněte si, že při popisu `build-list` jsme neuvedli, v jakém pořadí bude tvořící procedura na jednotlivé indexy aplikována – uvedli jsme jen pozici výsledků aplikací ve výsledném seznamu. Z pohledu čistého funkcionálního programování je přísně vzato jedno, v jakém pořadí aplikace proběhly – – libovolné pořadí aplikací vždy povede na stejný seznam. Z pohledu abstrakčních bariér se na `build-list` můžeme opět dívat jako na primitivní proceduru jejíž implementace je nám (zatím skryta). Známe pouze její vstupní argumenty a víme, jak bude vypadat výsledek aplikace.

Další užitečnou procedurou, kterou si představíme, je procedura `length` sloužící ke zjišťování délky seznamu. Procedura akceptuje jeden argument jímž je seznam:

`(length` ⟨*seznam*⟩`)`

a vrací jeho délku, neboli počet jeho prvků.

Příklad 5.12. (a) Délka tříprvkového seznamu je přirozeně tři. `(length '(1 2 3))` ⟹ `3`

(b) Pro prázdný seznam je procedurou `length` vrácena nula. `(length '())` ⟹ `0`

(c) Pokud by seznam l jako svůj prvek obsahoval další seznam l', pak se celý seznam l' počítá jako jeden prvek v seznamu nehledě na počet prvků v l'. Tento rozdíl si nejlépe uvědomíme na následujících příkladech:

```
(length '(a b c d))     ⟹  4
(length '(a (b c) d))   ⟹  3
(length '(a (b c d)))   ⟹  2
(length '((a b c d)))   ⟹  1
```

V sekci 5.1 jsme definovali procedury `first`, `second` a `third` pro přístup k prvnímu, druhému a třetímu prvku seznamu. Často ale potřebujeme přistupovat k nějakému prvku seznamu, jehož pozici v době psaní kódu neznáme, a vypočítáme ji až během výpočtu (to je ostatně spíš pravidlem než výjimkou). V takovém případě můžeme použít proceduru `list-ref`. Používá se se dvěma argumenty ⟨*seznam*⟩ a ⟨*pozice*⟩. Její aplikací je pak vrácen prvek seznamu ⟨*seznam*⟩ na pozici ⟨*pozice*⟩. Pozice jsou přitom počítány od nuly – – první prvek je tedy na pozici nula, druhý na pozici jedna, a tak dále (číslování pozic prvků v seznamu se shoduje s tím, co používá i procedura `build-list`).

```
(list-ref '(a (b c) d) 0)   ⟹  a
(list-ref '(a (b c) d) 1)   ⟹  (b c)
(list-ref '(a (b c) d) 2)   ⟹  d
```

```
(list-ref '(a (b c) d) 3)   ⟼   „CHYBA: prvek na pozici 3 neexistuje“
(list-ref '() 0)            ⟼   „CHYBA: prvek na pozici 0 neexistuje“
```

Další procedurou, kterou budeme někdy potřebovat, je procedura `reverse`, která akceptuje jeden argument, kterým je seznam, a vrací nový seznam obsahující tytéž prvky jako výchozí seznam, ale v opačném pořadí:

```
(reverse '(a b c d))   ⟼   (d c b a)
(reverse '())          ⟼   ()
```

Poznámka 5.13. Všimněte si, že procedura `reverse` nemodifikuje původní seznam, ale vytváří (konstruuje) podle něj nový. Původní seznam zůstává zachován. V následujícím kódu převracíme seznam navázaný na symbol `sez`.

```
(define sez '(1 2 3))
(reverse sez)          ⟼   (3 2 1)
sez                    ⟼   (1 2 3)
```

Jak je z příkladu patrné, seznam navázaný na `sez` se nezměnil. To je zcela v souladu s funkcionálním duchem našeho jazyka. Nové seznamy jsou konstruovány na základě jiných seznamů.

Manipulaci se seznamy je principiálně možné provádět dvojím způsobem: *konstruktivně* a *destruktivně*. Konstruktivní přístup je vlastní funkcionálnímu programování a spočívá pouze ve vytváření nových seznamů na základě jiných, přitom nedochází k žádné modifikaci již existujících seznamů. Byl-li seznam jednou vytvořen, již jej nelze fyzicky změnit. Destruktivní přístup se používá v procedurálních jazycích a spočívá právě v modifikaci (již existujících) seznamů. Destruktivní modifikace jakýchkoliv struktur je vždy nebezpečná, protože těsně souvisí s *vedlejším efektem*. V dalším díle tohoto textu ukážeme, že i v jazyku Scheme máme k dispozici aparát pro destruktivní práci se seznamy, nyní však budeme se seznamy pracovat pouze konstruktivně.

S tím, co už známe (konkrétně s procedurami `length`, `build-list` a `list-ref`), můžeme naprogramovat vlastní obracení seznamu. To znamená, že proceduru `reverse` bychom v interpretu jazyka Scheme nemuseli mít jako primitivní, ale mohli bychom ji zavést jako uživatelsky definovanou proceduru. Její implementaci nalezneme v programu 5.1. Základní myšlenka programu je následující. Nejprve pomocí procedury `length`

Program 5.1. Obracení seznamu pomocí `build-list`.

```
(define reverse
  (lambda (l)
    (let ((len (length l)))
      (build-list len
                  (lambda (i)
                    (list-ref l (- len i 1)))))))
```

zjistíme délku původního seznamu. Výsledný (převrácený) seznam, který chceme zkonstruovat, bude mít stejný počet prvků. Zjištěná délka n vstupního seznamu bude tedy vstupním argumentem pro konstruktor `build-list`. Tvořící procedura předaná `build-list` pak bude pro argument i (pozice v konstruovaném seznamu) vracet prvek, který je v původním seznamu na pozici $n-i-1$. Tedy na stejné pozici, ale počítáno od konce. Přesvědčte se, že procedura funguje jak má a to včetně mezního případu, kdy je jejím vstupem prázdný seznam[7].

Další představenou procedurou bude `append`. Procedura `append` slouží ke spojování seznamů. Jako argumenty bere dva seznamy

[7] Podotkněme, že implementace `reverse` uvedená v programu 5.1 není příliš efektivní. Stejně tomu bude i u implementací dalších procedur v této sekci. Daleko efektivnější implementace těchto procedur si ukážeme v dalších lekcích. Nyní se spíše než na efektivitu soustředíme na funkcionální styl práce se seznamy.

`(append` $\langle seznam_1 \rangle$ $\langle seznam_2 \rangle$`)`

a vrací tyto dva seznamy spojené v jeden seznam tak, že za posledním prvkem prvního seznamu následuje první prvek druhého seznamu. Viz příklady použití procedury `append`:

```
(append '(a b c) '(1 2))        ⟹  (a b c 1 2)
(append '(a (b c)) '(1 2))      ⟹  (a (b c) 1 2)
(append '(a (b) c) '((1 2)))    ⟹  (a (b) c (1 2))
(append '() '(1 2))             ⟹  (1 2)
(append '(a b c) '())           ⟹  (a b c)
(append '() '())                ⟹  ()
```

Všimněte si, že například na druhém řádku předchozí ukázky jsme spojovali seznamy `(a (b c))` a `(1 2)`, to jest první z těchto seznamů měl jako druhý prvek seznam `(b c)`. Ve výsledném spojení je opět druhým prvkem seznam `(b c)`, nedochází tedy ke vkládání jednotlivých prvků ze seznamu `(b c)` do výsledného seznamu. Z předchozí ukázky je rovněž patrné, že operace „skládání seznamů“, která je prováděna při aplikaci `append` se chová neutrálně vůči prázdnému seznamu. Spojení seznamu l s prázdným seznamem (nebo obráceně) dá opět seznam l. Jako důsledek získáváme, že spojením prázdného seznamu se sebou samým je prázdný seznam, viz poslední řádek ukázky.

Podotkněme, že provedení spojení dvou seznamů si lze představit jako sérii postupných aplikací konstruktoru `cons`. Výsledek spojení dvou seznamů je právě to, co bychom získali, kdybychom postupně všechny prvky $\langle prvek_1 \rangle, \langle prvek_2 \rangle, \ldots, \langle prvek_n \rangle$ prvního seznamu $\langle seznam_1 \rangle$ připojili na začátek seznamu druhého seznamu takto:

`(cons` $\langle prvek_1 \rangle$ `(cons` $\langle prvek_2 \rangle \cdots$ `(cons` $\langle prvek_n \rangle$ $\langle seznam_2 \rangle$`)` $\cdots$ `))`.

Například v případě seznamů `(a b c)` a `(1 2)` máme:

```
(append '(a b c) '(1 2))              ⟹  (a b c 1 2)
(cons 'a (cons 'b (cons 'c '(1 2))))  ⟹  (a b c 1 2)
```

Také proceduru pro spojování dvou seznamů už budeme schopni naprogramovat. Její naprogramování lze provést mnoha způsoby. Nejefektivnější z nich je založen právě předchozí násobné aplikaci `cons`. My si však zatím ukážeme méně efektivní verzi spojení dvou seznamů (nazvanou `append2`) uvedenou v programu 5.2. Pomocí procedury `build-list` vytvoříme seznam, jehož délka bude součtem délek původních seznamů,

Program 5.2. Spojení dvou seznamů pomocí `build-list`.

```
(define append2
  (lambda (l1 l2)
    (let ((len1 (length l1))
          (len2 (length l2)))
      (build-list (+ len1 len2)
                  (lambda (i)
                    (if (< i len1)
                        (list-ref l1 i)
                        (list-ref l2 (- i len1)))))))))
```

protože nám jde o to zkonstruovat jejich spojení. Tvořící procedura pak pomocí procedury `list-ref` vybírá prvky z původních seznamů. V těle tvořící procedury je potřeba rozlišit dva případy (všimněte si podmínky formulované ve speciální formě `if`). Konkrétně musíme rozlišit případ, kdy ještě vybíráme prvky z prvního seznamu a kdy je již vybíráme z druhého (v případě druhého seznamu je navíc potřeba přepočítat index, zdůvodněte si podrobně proč).

Spojování seznamů je asociativní operace. To znamená, že třeba vyhodnocení výrazu

```
(append (append '(1 2) '(3 4)) '(5 6))  ⟹  (1 2 3 4 5 6)
```

má stejný výsledek jako vyhodnocení výrazu

```
(append '(1 2) (append '(3 4) '(5 6)))  ⟹  (1 2 3 4 5 6).
```

Jelikož jsme v předchozí části viděli, že prázdný seznam se chová vůči spojování seznamů neutrálně, můžeme říct, že *operace spojování seznamů* je *monoidální operace* (viz sekci 2.5) na množině všech seznamů. Máme tedy další důležitý příklad monoidální operace. Podotkněme, že spojování pochopitelně není komutativní, protože spojením dvou seznamů v opačném pořadí nemusíme získat stejný výsledek.

Procedura `append` je ve skutečnosti obecnější, než jak jsme na v úvodu předeslali. Procedura `append` bere libovolný počet argumentů jimiž jsou seznamy:

`(append` ⟨*seznam*$_1$⟩ ⟨*seznam*$_2$⟩ $\cdots$ ⟨*seznam*$_n$⟩`)`,

kde $n \geq 0$. Následující ukázky ukazují použití `append` s různým počtem argumentů.

```
(append '(a b c) '(1 2) '(#t #f))  ⟹  (a b c 1 2 #t #f)
(append '(a b c) '() '(#t #f))     ⟹  (a b c #t #f)
(append '() '() '())               ⟹  ()
(append '(a b c))                  ⟹  (a b c)
(append)                           ⟹  ()
(let ((s '(a b c)))
  (append s s s s))                ⟹  (a b c a b c a b c a b c)
```

Při aplikaci bez argumentu vrací `append` prázdný seznam. Pokud vás to překvapuje, připomeňme, se například i procedury `+` a `*` se chovaly tak, že pro prázdný seznam argumentů vracely neutrální prvek.

Nyní obrátíme naši pozornost na další proceduru vyššího řádu pracující se seznamy. Často je potřeba z daného seznamu vytvořit nový seznam stejné délky, jehož prvky jsou nějakým způsobem „změněny“. Takové činnosti se říká *mapování procedury přes seznam*. K provedení této operace a ke konstrukci nového seznamu tímto způsobem nám ve Scheme slouží procedura `map`. Procedura `map` bere dva argumenty. Prvním z nich je procedura ⟨*proc*⟩ jednoho argumentu. Druhým argumentem je seznam. Proceduru ⟨*proc*⟩ budeme ve zbytku kapitoly nazývat *mapovaná procedura*. Aplikace `map` tedy probíhá ve tvaru

`(map` ⟨*proc*⟩ ⟨*seznam*⟩`)`

Výsledkem této aplikace je seznam výsledků aplikace procedury ⟨*proc*⟩ na každý prvek seznamu ⟨*seznam*⟩. Označme prvky seznamu ⟨*seznam*⟩ postupně ⟨*prvek*$_1$⟩, ⟨*prvek*$_2$⟩, …, ⟨*prvek*$_n$⟩. Aplikací procedury `map` tedy dostáváme seznam

`(`Apply(⟨*proc*⟩, ⟨*prvek*$_1$⟩) Apply(⟨*proc*⟩, ⟨*prvek*$_2$⟩) $\cdots$ Apply(⟨*proc*⟩, ⟨*prvek*$_n$⟩)`)`.

Viz následující příklady použití procedury `map`:

```
(map (lambda (x) (+ x 1)) '(1 2 3 4))   ⟹  (2 3 4 5)
(map - '(1 2 3 4))                      ⟹  (-1 -2 -3 -4)
(map list '(1 2 3 4))                   ⟹  ((1) (2) (3) (4))
(map (lambda (x) x) '(a (b c) d))       ⟹  (a (b c) d)
(map (lambda (x) #f) '(1 2 3))          ⟹  (#f #f #f)
(map (lambda (x) #f) '())               ⟹  ()
(map (lambda (x) (<= x 3)) '(1 2 3 4))  ⟹  (#t #t #t #f)
(map even? '(1 2 3 4))                  ⟹  (#f #t #f #t)
```

Mapování procedury přes jeden seznam jsme schopni implementovat. Kód procedury je uveden v programu 5.3. V tomto programu jsme pomocí procedury `build-list` vytvořili nový seznam o stejné délce jako je seznam původní. Tvořící procedura nám pro daný index vybere prvek z původního seznamu na odpovídající pozici (pomocí `list-ref`) a pak na něj aplikuje mapovanou proceduru.

Proceduru `map` lze použít rovněž s více než se dvěma argumenty. Seznam nemusí být jen jeden, ale může jich být jakýkoli kladný počet k:

`(map` ⟨*proc*⟩ ⟨*seznam*$_1$⟩ ⟨*seznam*$_2$⟩ $\cdots$ ⟨*seznam*$_k$⟩`)`

Všechny seznamy ⟨*seznam*$_i$⟩ $(1 \leq i \leq k)$ musí mít stejnou délku. Mapovací procedura ⟨*proc*⟩ musí být procedurou k argumentů. Jinými slovy: musí být procedurou tolika argumentů, kolik je seznamů. Výsledkem je pak seznam výsledků aplikací procedury ⟨*proc*⟩ na ty prvky seznamů, které se v předaných seznamech nacházejí na stejných pozicích. Pro vysvětlení viz příklady aplikací:

Program 5.3. Mapovací procedura pracující s jedním seznamem pomocí `build-list`.

```
(define map1
  (lambda (f l)
    (build-list (length l)
                (lambda (i)
                  (f (list-ref l i))))))
```

```
(map (lambda (x y) (+ x y)) '(1 2 3) '(10 20 30))  ⟼  (11 22 33)
(map + '(1 2 3) '(10 20 30))                       ⟼  (11 22 33)
(map cons '(a b) '(x y))                           ⟼  ((a . x) (b . y))
(map cons '(a b #t) '(x y #f))                     ⟼  ((a . x) (b . y) (#t . #f))
(map <= '(1 2 3 4 5) '(5 4 3 2 1))                 ⟼  (#t #t #t #f #f)
(map (lambda (x y z)
       (list x (+ y z))) '(a b) '(1 2) '(10 20))   ⟼  ((a 11) (b 22))
```

Proceduru `map` pro více než jeden seznam zatím vytvořit neumíme. Ale vrátíme se k této problematice v příštích lekcích. Opět (obdobně jako v případě procedury `build-list`) jsme zatajili v jakém pořadí se bude mapovaná procedura na jednotlivé prvky seznamu aplikovat. Z čistě funkcionálního pohledu na programování, o který nám v této části textu jde, to asi není důležité.

5.4 Datové typy v jazyku Scheme

V této sekci si nejprve shrneme všechny typy elementů, které jsme si do této chvíle představili. Typ elementy budeme stručně nazývat *datový typ*, protože na elementy se můžeme dívat jako na *data* (*hodnoty*), nad kterými provádíme různé výpočty. Ukážeme predikáty, kterými můžeme identifikovat typ daného elementu jazyka. Dále si ukážeme predikát `equal?`, kterým je možné „porovnávat“ libovolné dva elementy.

Z předchozích lekcí již známe tyto základní typy elementů jazyka: *čísla*, *pravdivostní hodnoty*, *symboly*, *tečkové páry*, *prázdný seznam*, *procedury* (*primitivní procedury* a *uživatelsky definované procedury*). K těmto základním typů elementů bychom ještě mohli přidat element *nedefinovaná hodnota* i když standard R^{5}RS jazyka Scheme nedefinovanou hodnotu chápe jinak, viz [R5RS]. V této sekci jsme rovněž představili odvozený typ elementu *seznam*[8].

Při vytváření programů v jazyku Scheme je leckdy potřeba řídit výpočet podle typu argumentu nebo více argumentů, které jsou předané procedurám. Při psaní procedur je tedy vhodné mít k dispozici predikáty, kterými můžeme pro daný element rozhodnout o jeho typu. V jazyku Scheme máme k dispozici následující predikáty, kterými můžeme kvalifikovat typy elementů:

`boolean?`	*pravdivostní hodnota*,
`list?`	*seznam*,
`null?`	*prázdný seznam*,
`number?`	*číslo*,
`pair?`	*tečkový pár*,
`procedure?`	*procedura* (*primitivní* nebo *uživatelská*),
`symbol?`	*symbol*.

V levé části výpisu máme uvedena jména symbolů, na které jsou predikáty navázány a v pravém sloupci je uveden popis jejich významu.

Poznámka 5.14. Pokud nepočítáme predikát `list?`, jsou uvedené predikáty vzájemně disjunktní, to jest: při jejich aplikaci na jakýkoli element nám *nejvýše jeden z nich vrátí pravdu.* Pokud predikát `list?` vrací pro

[8] Seznam je skutečně „odvozený typ elementu“, protože přísně vzato je každý seznam buďto prázdný seznam (speciální element) nebo je to tečkový pár ve speciálním tvaru.

nějaký element pravdu, pak vrací pravdu i právě jeden z predikátů `pair?` nebo `null?` (to plyne ze zavedení seznamu, viz definici 5.2 na straně 115). Viz následující příklady:

```
(list? '(10 20 30))     ⟹  #t
(list? '(10 20 . 30))   ⟹  #f
(pair? '(10 20 30))     ⟹  #t
(pair? '(10 20 . 30))   ⟹  #t
```

Nyní se můžeme opět vrátit k rozdílu mezi symbolem a hodnotou, která je na něm navázána, o kterém jsme již několikrát mluvili. Opět je nutné dobře si uvědomovat tento rozdíl. Uvažujme následující příklad:

```
(define x 10)
(symbol? x)          ⟹  #f
(symbol? 'x)  ⟹  #t
```

V této ukázce máme v globálním prostředí navázáno číslo `10` na symbol `x`. Při prvním použití predikátu `symbol?` je vrácena pravdivostní hodnota `#f`, protože symbol `x` se vyhodnotí na svou vazbu (číslo `10`), což není symbol. V druhém případě je vráceno `#t`, protože `'x` se vyhodnotí na symbol `x`.

Predikáty uvedené v této sekci můžeme použít ke spouště účelů. Můžeme pomocí nich například vytvořit „bezpečnou verzi“ selektorů `car` a `cdr`, které se odlišují od původních selektorů tím, že při aplikaci na prázdný seznam nezpůsobí chybu. Namísto toho vrací prázdný seznam `()`[9]:

```
(define safe-car
  (lambda (l)
    (if (null? l)
        '()
        (car l))))

(define safe-cdr
  (lambda (l)
    (if (null? l)
        '()
        (cdr l))))
```

V obou případech jsme nejdříve použitím predikátu `null?` zjistili, zda není na formální argument `l` navázán prázdný seznam. Pokud ano, je výsledkem vyhodnocení prázdný seznam `()`, jinak na hodnotu navázanou na `l` aplikujeme původní selektor.

Další ukázkou použití predikátů rozhodujících typ je jednoduchý predikát `singleton-list?`. Ten zjišťuje, jestli je jeho argument jednoprvkový seznam:

```
(define singleton-list?
  (lambda (l)
    (and (pair? l) (null? (cdr l)))))
```

Program je vlastně přímým přepisem tvrzení, že jednoprvkový seznam je pár a jeho druhý prvek je prázdný seznam `()`.

Ve Scheme existuje predikát `equal?`, kterým je možné porovnávat libovolné elementy:

Definice 5.15 (predikát `equal?`)**.** Procedura `equal?` se používá se dvěma argumenty

`(equal?` ⟨*element*$_1$⟩ ⟨*element*$_2$⟩`)`

a vrací `#t`, právě když pro ⟨*element*$_1$⟩ a ⟨*element*$_2$⟩ platí:

- oba ⟨*element*$_1$⟩ a ⟨*element*$_2$⟩ jsou buď `#t` nebo `#f`;
- oba ⟨*element*$_1$⟩ a ⟨*element*$_2$⟩ jsou stejné symboly;

[9] Takto fungují selektory `car` a `cdr` v jazyce Common LISP

- oba $\langle element_1 \rangle$ a $\langle element_2 \rangle$ jsou čísla, jsou obě buď v přesné nebo v nepřesné reprezentaci a obě čísla jsou si numericky rovny, to jest aplikací `=` na $\langle element_1 \rangle$ a $\langle element_2 \rangle$ bychom získali `#t`;
- oba $\langle element_1 \rangle$ a $\langle element_2 \rangle$ jsou prázdné seznamy;
- oba $\langle element_1 \rangle$ a $\langle element_2 \rangle$ jsou nedefinované hodnoty;
- oba $\langle element_1 \rangle$ a $\langle element_2 \rangle$ jsou stejné primitivní procedury nebo stejné uživatelsky definované procedury, nebo stejné speciální formy;
- oba $\langle element_1 \rangle$ a $\langle element_2 \rangle$ jsou páry ve tvarech $\langle element_1 \rangle = (E_1 \ . \ F_1)$ a $\langle element_2 \rangle = (E_2 \ . \ F_2)$ a platí:
  - výsledek aplikace `equal?` na E_1 a E_2 je `#t`
  - výsledek aplikace `equal?` na F_1 a F_2 je `#t`;

a vrací `#f` ve všech ostatních případech. ■

Při použití predikátu `equal?` na čísla je potřeba, aby byla obě buď v přesné nebo v nepřesné reprezentaci. To je rozdíl vzhledem k predikátu `=`, u kterého rozhoduje jen numerická rovnost. Viz následující příklad:

```
(equal? 2 2.0)   ⟹ #f
(= 2 2.0)        ⟹ #t
(equal? 2 2)     ⟹ #t
(equal? 2.0 2.0) ⟹ #t
```

Ačkoliv „rovnost procedur“ budeme testovat jen zřídka, zdali vůbec, objasněme co znamená, že dvě uživatelsky definované procedury jsou „stejné“. Každý uživatelsky definovaná procedura je z našeho pohledu „stejná“ jen sama se sebou. I kdybychom vzali dvě procedury vzniklé vyhodnocením téhož λ-výrazu, z pohledu predikátu `equal?` se bude jednat o různé elementy, viz následující ukázku. Máme:

```
(equal? (lambda (x) (* x x)) (lambda (x) (* x x)))  ⟹  #f
```

Ale na druhou stranu:

```
(define na2 (lambda (x) (* x x)))
(equal? na2 na2)                    ⟹  #t
```

Rovněž upozorněme na fakt, že `equal?` se hodí k porovnávání jakýchkoliv elementů, tedy i hierarchických struktur. Z popisu `equal?` vidíme, že tímto predikátem můžeme snadno testovat rovnost dvou tečkových párů, což nemůžeme provést pomocí predikátu `=` (rovnost čísel):

```
(equal? '(a b) '(a b))   ⟹ #t
(= '(a b) '(a b))        ⟹ „CHYBA: Argumenty nejsou čísla“
```

Poznámka 5.16. Pozorní čtenáři si jistě všimli, že jsme neuvedli žádné predikáty, kterými by bylo možné identifikovat speciální formy. Neuvedli jsme je proto, že ve specifikaci jazyka R^{5}RS, viz dokument [R5RS], takové predikáty nejsou. Standard jazyka Scheme R^{5}RS totiž nechápe speciální formy jako elementy jazyka. Stejně tak jsme mohli vytvořit dva predikáty, které testují zda-li je daná procedura primitivní nebo uživatelsky definovaná. Představený systém predikátů tedy mohl být mnohem bohatší. Pro naše účely si ale plně vystačíme s předchozími procedurami, některé nové predikáty určující typy elementů představíme v poslední lekci tohoto dílu textu.

Na závěr této sekce o datových typech v jazyku Scheme podotkněme, že jiné programovací jazyky mají k datovým typům odlišný přístup. Teorie datových typů je jedna z řady disciplín, které jsou v informatice dodnes hluboce studovány a výzkum v této oblasti postupuje pořád dál. Každý jazyk lze z hlediska použitého modelu typů rozdělit do několika kategorií. První rozdělení je založeno na tom, v jakém okamžiku je možné provést kontrolu typu elementů:

Staticky typované jazyky. Jedná se o programovací jazyky, pro které je z principu možné udělat kontrolu typů již před interpretací nebo během překladu programu, pouze na základě znalosti jeho syntaktické struktury.

Dynamicky typované jazyky. U dynamicky typovaných jazyků platí, že pouhá struktura kódu (vždy) nestačí ke kontrole typů a typy elementů musí být kontrolovány až za běhu programu. U dynamicky typových jazyků je rovněž možné, že jedno jméno (symbol nebo proměnná) může během života programu nést hodnoty různých typů.

Z toho co víme o jazyku Scheme je jasné, že se jedná o *dynamicky typovaný jazyk*, protože když uvážíme třeba výraz

```
(let ((x (read)))
  (number? x)),
```

tak je zcela zřejmé, že o hodnotě jeho vyhodnocení, která závisí na kontrole typu elementu navázaného na `x` budeme moci rozhodnout až za běhu programu, protože na `x` bude navázán vstupní výraz zadaný uživatelem. Naproti tomu třeba jazyk C je staticky typový, typ všech proměnných je navíc potřeba explicitně *deklarovat*, to jest „dát překladači na vědomí“ před jejich prvním použitím v programu.

Další úhel pohledu na vlastnosti typového systému je jeho „síla“. Některé programovací jazyky umožňují používat procedury (operace) jen s argumenty přesně daného typu a pokud je vstupní argument jiného typu, provedení operace selže. Tak se chová i jazyk Scheme. Naopak jiné programovací jazyky definují sadu pravidel pro „převod“ mezi datovými typy (příkladem takového jazyka je třeba jazyk PERL). Proto rozlišujeme dva typy „síly“ typového systému:

Silně typované jazyky. Silně typovaný jazyk má pro každou operaci přesně vymezený datový typ argumentů. Pokud je operace použita s jiným typem argumentů než je přípustné, dochází k chybě.

Slabě typované jazyky. Slabě typované jazyky mají definovánu sadu konverzních pravidel, která umožňují, pokud je to nutné, převádět mezi sebou data různých typů. Pokud je provedena operace s argumenty, které jí typově neodpovídají, je provedena konverze typů tak, aby byla operace proveditelná.

Poslední vlastností je typová bezpečnost:

Bezpečně typované jazyky. Tyto jazyky se chovají korektně z hlediska provádění operací mezi různými typy elementů. To znamená, že pokud je operace pro daný typ elementů proveditelná, její provedení nemůže způsobit havárii programu.

Nebezpečně typované jazyky. Nebezpečně typované jazyky jsou jazyky, které nejsou bezpečně typované. Výsledek operace mezi různými typy tedy může vést k chybě při běhu programu. Takovým jazykem je třeba C, kde chyba tohoto typu může být způsobena například přetečením paměti nebo dereferencí ukazatele z ukazujícího na místo paměti nepatřící programu.

Jazyk Scheme je z tohoto pohledu bezpečně typovaný jazyk.

5.5 Implementace párů uchovávajících délku seznamu

V této sekci si ukážeme implementaci „párů“, které si pamatují délku seznamu. To povede na zvýšení efektivity při práci se seznamy – délku seznamu bude možné zjistit k konstantním čase, protože bude přímo kódovaná v seznamu. Přestože uvádíme, že ukážeme novou implementaci pár, bude se vlastně jednat o trojice, jejichž prvky budou ⟨*hlava*⟩, ⟨*tělo*⟩ a ⟨*délka*⟩. A právě v posledním jmenovaném prvku budeme uchovávat délku seznamu.

Seznam délky n budeme pomocí těchto párů definovat následovně:

- pro $n = 0$ je to prázdný seznam `()`;
- pro $n \in N$ je to pár jehož prvek *hlava* je libovolný element, prvek *tělo* je seznam délky $n - 1$ a prvek *délka* je číslo n.

Kdykoli budeme v této sekci mluvit o seznamech, budeme tím myslet seznam ve smyslu výše uvedené definice. V dalších sekcích ale vrátíme k původní reprezentaci párů, která je součástí interpretu jazyka Scheme.

Náš nový pár bude opět reprezentován procedurou, stejně jako v programu 4.3 na straně 4.3. Svazujeme jí ale tři hodnoty: dvě z nich jsou vstupní argumenty `a` a `b`. Třetí hodnota je pak délka seznamu `b` zvýšená o 1. Konstruktor nového páru bude vypadat následovně:

```
(define cons
  (lambda (hlava telo)
    (define delka (+ 1 (fast-length telo)))
    (lambda (proj)
      (proj (cons hlava telo delka)))))
```

Všimněte si, že v konstruktoru jsme použili proceduru `fast-length` což je nový selektor vracející pro daný seznam jeho délku. Při připojení nového prvku k již existujícímu seznamu zjistíme pomocí tohoto selektoru jeho délku a do aktuálního páru zakódujeme délku zvětšenou o 1 (přidali jsme jeden prvek). Implementaci selektoru `fast-length` uvedeme posléze.

Selektory `car` a `cdr` můžeme naprogramovat klasicky pomocí projekcí:

```
(define 1-ze-3 (lambda (x y z) x))
(define 2-ze-3 (lambda (x y z) y))

(define car
  (lambda (par)
     (par 1-ze-3)))

(define cdr
  (lambda (par)
     (par 2-ze-3)))
```

Zjišťování délky seznamu se bude trochu lišit. Nejdříve ověříme, zda se nejedná o prázdný seznam `()`. V tom případě vracíme nulu – tedy délku prázdného seznamu. V opačném případě vracíme prvek *délka*, což je informace o délce, která je součástí každého páru. Viz následující kód:

```
(define 3-ze-3 (lambda (x y z) z))

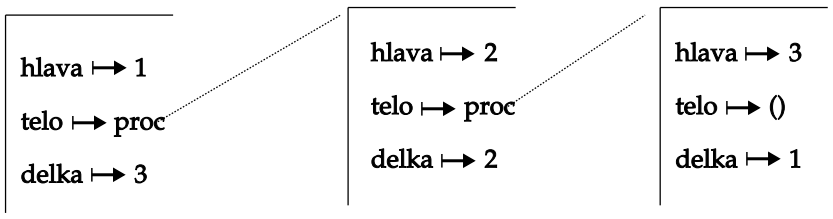
(define fast-length
  (lambda (par)
     (if (null? par)
         0
         (par 3-ze-3))))
```

Příklad 5.17. Podívejme se nyní na použití nové implementace párů:

```
(define seznam (cons 1 (cons 2 (cons 3 '()))))
```

Vytvořili jsme tedy seznam nikoli z tečkových párů, ale z procedur vyšších řádů, které v sobě zapouzdřují tři hodnoty – hlavu, tělo a délku seznamu. Tělo seznamu je přitom zase seznamem. Obrázek 5.4 ukazuje prostředí, která reprezentují seznam. Přerušovaná čára ukazuje prostředí vzniku procedur reprezentujících seznam.

Obrázek 5.4. Procedury a prostředí u párů uchovávajících délku seznamu



Nyní uvedeme použití selektorů: Výraz `(car seznam)` se vyhodnotí na číslo `1`. Na seznam je navázána procedura o jednom argumentu. Tuto proceduru aplikuje selektor `car` ve svém těle na projekci prvního

prvku navázanou na symbol `1-ze-3`. Tuto projekci aplikuje procedura `seznam` na elementy navázané na symboly `hlava`, `telo` a `delka` v prostředí jejího vzniku $\mathcal{P}_1$. Výsledkem je tedy číslo `1`.

Výraz `(car (cdr seznam))` se vyhodnotí takto: vyhodnocením podvýrazu `(cdr seznam)` (podobným způsobem jako v předchozím příkladě) dostáváme hodnotu navázanou na symbol `telo` v prostředí $\mathcal{P}_1$, což je opět (neprázdný) seznam – procedura, která bere jako argument projekci, a vrací výsledek její aplikace na hodnoty navázané na symboly `hlava`, `telo` a `delka` v prostředí $\mathcal{P}_2$. A tedy dostáváme číslo `2`. Podobně vyhodnocením výrazu `(fast-length (cdr seznam))` bude číslo `2` – hodnota navázaná na symbol `delka` v prostředí $\mathcal{P}_2$.

Poznámka 5.18. Všimněte si, že naše nová procedura `cons` nemůže být použita ke konstrukci obecných párů, to jest párů, jejichž druhý prvek není ani další pár ani prázdný seznam. Jako argument `telo` musí předán seznam; v těle konstruktoru `cons` totiž zjišťujeme délku seznamu navázaného na `telo`.

5.6 Správa paměti během činnosti interpretu

Při našem exkursu funkcionálním paradigmatem jsem se až doposud výhradně bavili o činnosti abstraktního interpretu jazyka Scheme. Skutečné interprety, tedy ty se kterými reálně pracujeme, jsou obvykle výrazně složitější. Vyhodnocovací proces je v nich implementován zhruba tak, jako v našem abstraktním interpretu, jejich složitost však spočívá v tom, že interprety musí řešit řadu otázek souvisejících se správou a organizací paměti.

V této sekci malinko nahlédneme pod pokličku skutečného interpretu jazyka Scheme a zaměříme se na jeden specifický problém správy paměti související s tečkovými páry. Každý interpret jazyka Scheme musí nějak reprezentovat elementy jazyka: čísla, seznamy, procedury, páry a tak dále. Pro reprezentaci párů se obvykle používá struktura obsahující ukazatel na dvojici dalších elementů – elementů „obsažených v páru“. To je v souladu s upravenou boxovou notací uvedenou v této lekci. Během činnosti programu vznikají nové páry konstrukcí a časem může dojít k situaci, že již některý pár, která je uložen v paměti, není potřeba. To znamená, že neexistuje metoda, jak v programu tento pár získat vyhodnocením nějakého výrazu. Demonstrujme si tuto situaci příkladem:

```
(define s '(a b c))
s                    ⟹  (a b c)
(define s #f)
s                    ⟹  #f
```

Nejprve jsme na `s` navázali seznam jehož interní reprezentaci pomocí párů musí udržovat interpret Scheme v paměti. Na třetím řádku jsme na symbol `s` navázali hodnotu `#`. Tím pádem jsem ztratili jakoukoliv vazbu na zkonstruovaný seznam `(a b c)`. Logika věci říká, že interpret jazyka Scheme by to měl nějak poznat a odstranit interní reprezentaci seznamu z paměti[10]. Někoho by nyní mohlo napadnout, že testovat „viditelnost“ seznamů je jednoduché: nabízí se pouze „smazat každý seznam“, který byl dříve navázaný na symbol, který jsme právě redefinovali (to odpovídá předchozí situaci). Není to, bohu žel, tak jednoduché, viz následující příklad:

```
(define s '(a b c))
s                    ⟹  (a b c)
(define r s)
(define s #f)
s                    ⟹  #f
r                    ⟹  (a b c)
```

Zde jsme provedli skoro totéž, co v předchozím případě, provedli jsme ale navíc jednu definici. Před redefinicí vazby `s` jsme na `r` definovali aktuální vazbu `s`. To bylo v okamžiku, kdy na `s` byl ještě navázán seznam `(a b c)`. I když po provedení redefinice `s` již na `s` tento seznam navázaný není, je stále navázaný

[10] Interpret si nemůže dovolit pouze plnit pamět' novými páry, uvědomte si, že třeba při každém použití `cons` je vytvořen jeden nový pár, při konstrukci n prvkového seznamu je to n nových párů. Kdyby interpret nevěnoval správě paměti pozornost, činnost většiny programů by byla záhy násilně ukončena operačním systémem z důvodu vypotřebování paměti.

na r. Předchozí navržená metoda „vymazávání seznamů po redefinici“ by tedy nebyla ani účinná a ani korektní (vedla by k havárii interpretu a tím i interpretovaného programu).

Užitečné je taky uvědomit si, že seznamy obecně nelze jen tak likvidovat celé. Někdy může být ještě část seznamu „dostupná“. Opět si uveďme demonstrační příklad:

```
(define r '(x y))
(define s (cons 100 r))
r                       ⟼  (x y)
s                       ⟼  (100 x y)
(define s #f)
r                       ⟼  (x y)
s                       ⟼  #f
```

Zde jsme zkonstruovali seznam a navázali jej na r. Poté jsme na s navázali seznam vzniklý z předchozího přidáním prvku na první pozici. Pokud redefinujeme vazbu s tak, jak je v příkladu ukázáno, dostaneme se do situace, kdy již není dostupný seznam `(100 x y)`, ale je pořád ještě dostupný dvouprvkový seznam `(x y)` tvořící jeho tělo. To znamená, že není možné celou reprezentaci seznamu `(100 x y)` odstranit z paměti, protože tělo tohoto seznamu je stále dostupné pomocí symbolu r. Zároveň je ale jasné, že pár jehož prvním prvkem je číslo 100 a druhým prvkem je (ukazatel na) dvouprvkový seznam `(x y)`, již smazat můžeme.

Jak tedy interprety provádějí správu paměti? Obecně je to tak, že každý interpret má v sobě zabudovaný podprogram, tak zvaný *garbage collector*, který se během vyhodnocování jednou za čas spustí (například, když rychle klesne volná paměť), projde všechny elementy reprezentované v paměti a smaže z nich ty, které již nejsou nijak dostupné. V praxi se k úklidu paměti často používá algoritmus označovaný jako *mark & sweep* (česky by se dalo nazvat „označ a zamet“), jehož činnost si nyní zevrubně popíšeme.

Nejprve podotkněme, že paměť v níž jsou uloženy reprezentace elementů, se nazývá *halda*. Pojem *halda* se v informatice používá ještě v jiném významu (speciální kořenový strom) a čtenářům bude asi důvěrně známý z kursu algoritmické matematiky. V našem kontextu je však „halda“ pouze název pro úsek paměti. Název algoritmu napovídá, že algoritmus má dvě části. První část, zvaná *mark*, spočívá v tom, že se projde přes dosažitelné elementy a to tak, že se začne elementy jež mají vazbu v globálním prostředí (ty jsou zcela jistě dosažitelné). Pokud právě označený element ukazuje na další elementy, i ty budou dále zpracovány (protože jsou tím pádem taky dosažitelné). To se týká například *tečkových párů* (každý pár se odkazuje na první a druhou složku) a *uživatelsky definovaných procedur* (každá uživatelsky definovaná procedura ukazuje na seznam argumentů, tělo a prostředí). Takto se postupuje, dokud je co označovat. V druhém kroku nastupuje fáze *sweep*. Garbage collector projde celou haldu a pro každý element provede následující:

- pokud má element značku, značka je smazána a pokračuje se dalším elementem,
- pokud element značku nemá, pak je element odstraněn z paměti.

Po dokončení cesty haldou jsme v situaci, kdy byly nedostupné elementy smazány a všechny značky byly odstraněny. Mazání značek bylo provedeno jako inicializace pro další spuštění garbage collectoru.

5.7 Odvozené procedury pro práci se seznamy

V této sekci uvedeme řadu procedur pro vytváření a manipulaci se seznamy. Na implementaci těchto procedur pak ukážeme použití seznamů a základních procedur pro práci s nimi, které jsme představili v předchozích lekcích.

Prvními dvěma procedurami, které uvedeme, budou procedury pro zdvojení prvního prvku seznamu (procedura dupf) a zdvojení posledního prvku seznamu (procedura dupl). Bude se jednat o procedury o jednom argumentu, kterým bude seznam. Vracet budou seznam, obsahující duplikovaný první prvek seznamu v případě dupf, respektive duplikovaný poslední prvek seznamu v případě dupf. Takže například:

```
(dupf '(a))         ⟼  (a a)
(dupf '(1 2 3))     ⟼  (1 1 2 3)
```

```
(dupf '((1 2) 3))  ⟹  ((1 2) (1 2) 3)

(dupl '(a))        ⟹  (a a)
(dupl '(1 2 3))    ⟹  (1 2 3 3)
(dupl '(1 (2 3)))  ⟹  (1 (2 3) (2 3))
```

Duplikace prvního prvku seznamu vlastně odpovídá konstrukci hlavy seznamu na samotný seznam. Proceduru `dupf` tedy jednoduše naprogramujeme takto:

```
(define dupf
  (lambda (l)
    (cons (car l) l)))
```

Všimněte si, že v těle předchozího λ-výrazu používáme jen procedury `cons` a `car`, které známe z předchozí lekce. Jde nám o připojení prvního prvku na začátek celého seznamu. První prvek získáme aplikací procedury `car` a připojíme jej na začátek seznamu použitím procedury `cons`.

Nyní zdvojení posledního prvku. Postupujeme podle následující úvahy: Chceme-li se dostat k poslednímu prvku, obrátíme seznam pomocí procedury `reverse` a vezmeme první prvek z tohoto obráceného seznamu:

```
(reverse '(a b c))        ⟹  (c b a)
(car (reverse '(a b c)))  ⟹   a
```

Poslední prvek zdvojíme tak, že zdvojíme první prvek obráceného seznamu, a výsledek pak ještě jednou obrátíme:

```
(dupf (reverse '(a b c)))            ⟹  (c c b a)
(reverse (dupf (reverse '(a b c))))  ⟹  (a b c c)
```

Celá procedura tedy bude vypadat takto:

```
(define dupl
  (lambda (l)
    (reverse (dupf (reverse l)))))
```

Kdybychom chtěli vytvořit proceduru pro generování seznamu čísel od a do b, můžeme to udělat následujícím způsobem pomocí procedury `build-list`. Seznam čísel od a do b bude mít délku $b - a + 1$. Tvořící procedura pak bude pro index i vracet číslo $i + a$, které bude vloženo na i-tou pozici v seznamu (počítáno od 0). Přepsáno do Scheme:

```
(define range
  (lambda (a b)
    (build-list (+ 1 (- b a))
                (lambda (i) (+ i a)))))
```

Další dvě procedury budou také využitím procedury `build-list`. Konkrétně se bude jednat o vypuštění n-tého prvku seznamu a vsunutí prvku na n-tou pozici.

```
(define remove
  (lambda (l n)
    (build-list (- (length l) 1)
                (lambda (i)
                  (if (< i n)
                      (list-ref l i)
                      (list-ref l (+ i 1)))))))
```

Pro ilustraci viz následující příklady použití procedury:

```
(define s '(a b c d e))
(remove s 0)  ⟹  (b c d e)
(remove s 2)  ⟹  (a b d e)
(remove s 4)  ⟹  (a b c d)
```

Při definici procedury `remove` jsme použili procedury `build-list`, abychom s ní vytvořili seznam, jehož délka je o jedna kratší. Tvořící procedura pak aplikací procedury `list-ref` vybírá prvky z původního seznamu. Tyto prvky bereme z odpovídajících indexů, v případě, že jde o indexy menší, než je pozice odstraňovaného prvku `n`, nebo z indexů o jedna vyšších.

Analogickou procedurou je procedur `insert`, vkládající prvek na danou pozici:

```
(define insert
  (lambda (l n elem)
    (build-list (+ (length l) 1)
                (lambda (i)
                  (cond ((< i n) (list-ref l i))
                        ((= i n) elem)
                        (else (list-ref l (- i 1))))))))
```

Proceduru `insert` používáme takto:

```
(define s '(a b c d e))
(insert s 0 666)  ⟹  (666 a b c d e)
(insert s 1 666)  ⟹  (a 666 b c d e)
(insert s 3 666)  ⟹  (a b c 666 d e)
(insert s 5 666)  ⟹  (a b c d e 666)
```

Další dvě procedury, které napíšeme pomocí procedur představených v sekci 5.3 budou procedury nahrazování prvků v seznamu. Jejich společným rysem je použití procedury `map`. Mapovaná procedura bude vypadat v obou případech podobně – vznikne vyhodnocením *lambda*-výrazu:

`(lambda (x) (if` ⟨*podmínka*⟩ `new x))`.

Procedura tedy podle podmínky ⟨*podmínka*⟩ buďto nahrazuje původní prvek prvkem `new` nebo vrací původní prvek `x`. Nahrazování na základě splnění predikátu `prop?` by mohlo být implementováno takto:

```
(define replace-prop
  (lambda (sez new prop?)
    (map (lambda (x) (if (prop? x) new x))
         sez)))
```

Druhou procedurou nahrazení je nahrazování podle vzoru reprezentovaného seznamem pravdivostních hodnot určujících zda-li se má prvek na dané pozici zachovat nebo nahradit novým elementem. Ukažme si nejprve použití této procedury:

```
(replace-pattern '(1 2 3 4) '(#f #f #f #f) 'x)  ⟹  (1 2 3 4)
(replace-pattern '(1 2 3 4) '(#t #f #t #f) 'x)  ⟹  (x 2 x 4)
(replace-pattern '(1 2 3 4) '(#f #t #t #t) 'x)  ⟹  (1 x x x)
(replace-pattern '(1 2 3 4) '(#t #t #t #t) 'x)  ⟹  (x x x x)
```

Proceduru `replace-pattern` můžeme naprogramovat takto:

```
(define replace-pattern
  (lambda (l pattern new)
    (map (lambda (x y)
           (if y new x))
         l pattern)))
```

Další odvozenou procedurou je procedura `list-pref` vracející n-prvkový prefix seznamu. Připomeňme, že n-prvkovým prefixem seznamu rozumíme seznam obsahující prvních n prvků seznamu:

```
(define list-pref
  (lambda (l n)
    (build-list n
                (lambda (i)
                  (list-ref l i))))))
```

V předchozím řešení jsme vytvořili procedurou `build-list` seznam o délce n, prvky do něj pak vybíráme z původního seznamu pomocí procedury `list-ref` z odpovídajících indexů. Podobným způsobem můžeme napsat proceduru vracející n-prvkový suffix (posledních n prvků seznamu) seznamu:

```
(define list-suff
  (lambda (l n)
    (let ((len (length l)))
      (build-list n
                  (lambda (i)
                    (list-ref l (- len (- n i))))))))
```

Další procedury budou procedury *rotací seznamu*. Rotace seznamu doleva je procedurou o jednom argumentu. Tímto argumentem je seznam a procedura vrací seznam, který má oproti původnímu seznamu prvních n prvků na konci a ostatní posunuté vlevo. Tedy například:

```
(rotate-left (1 2 3 4) 1)  ⟼  (2 3 4 1)
(rotate-left (1 2 3 4) 2)  ⟼  (3 4 1 2)
(rotate-left (1 2 3 4) 3)  ⟼  (4 1 2 3)
(rotate-left (1 2 3 4) 4)  ⟼  (1 2 3 4)
```

Analogicky pak bude fungovat procedura rotace seznamu doprava:

```
(rotate-right (1 2 3 4) 1)  ⟼  (4 1 2 3)
(rotate-right (1 2 3 4) 2)  ⟼  (3 4 1 2)
(rotate-right (1 2 3 4) 3)  ⟼  (2 3 4 1)
(rotate-right (1 2 3 4) 4)  ⟼  (1 2 3 4)
```

Rotaci seznamu doleva `rotate-left` můžeme implementovat následujícím způsobem. Pomocí procedury `build-list` vytvoříme nový seznam stejné délky, jako je ten původní, na jednotlivé pozice vybíráme prvky z původního seznamu použitím procedury `list-ref`, a to z pozice vždy o `n` vyšší. Přetečení prvního prvku na konec seznamu je zajištěno použitím procedury `modulo` (vrácení celočíselného zbytku po dělení).

```
(define rotate-left
  (lambda (l n)
    (let ((len (length l)))
      (build-list len
                  (lambda (i)
                    (list-ref l (modulo (+ i n) len)))))))
```

Procedura rotace seznamu doprava bude velice podobná:

```
(define rotate-right
  (lambda (l n)
    (let ((len (length l)))
      (build-list len
                  (lambda (i)
                    (list-ref l (modulo (- i n) len)))))))
```

Předchozí dvě procedury – `rotate-left` a `rotate-right` – můžeme sjednotit pomocí proceduru vyššího řádu *selekce*. Ta bere dva argumenty, prvním z nich je seznam $(E_0\ \ E_1\ \cdots\ E_{n-1})$, tak jako u procedur rotace, a druhým je procedura reprezentující zobrazení $f : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$. Procedura selekce vrací seznam ve tvaru

$$(E_{f(0,n)}\ \ E_{f(1,n)}\ \cdots\ E_{f(n-1,n)}),$$

to jest seznam vzniklý z výchozího výběrem elementů určeným procedurou reprezentující f. Obecný kód procedury `select` by tedy mohl vypadat následovně:

```
(define select
  (lambda (l sel-f)
    (let ((len (length l)))
      (build-list len
                  (lambda (i)
```

```
      (list-ref l (modulo (sel-f i len) len)))))))
```

Pomocí `select` můžeme vytvořit procedury rotace seznamu, které jsme uvedli výše:

```
(define rotate-left
  (lambda (l n)
    (select l (lambda (i len) (+ i n)))))

(define rotate-right
  (lambda (l n)
    (select l (lambda (i len) (- i n)))))
```

K implementaci `rotate-left` a `rotate-right` stačilo pouze provést specifikaci procedury reprezentující zobrazení $f : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}_0$. V obou případech byla f definována tak, že $f(i, j) = i \pm k$, kde k počet prvků o kolik se má rotace provést (druhý argument j se v tomto případě neuplatnil).

Pomocí procedury `select` můžeme vytvořit ale například i proceduru `reverse`, kterou jsme představili v sekci 5.3:

```
(define reverse
  (lambda (l)
    (select l (lambda (i len) (- len i 1)))))
```

Nebo třeba proceduru na vytvoření seznamu náhodně vybraných prvků z původního seznamu:

```
(define random-select
  (lambda (l n)
    (select l (lambda (i len) (random len)))))
```

V posledním případě již přísně vzato druhý argument předaný `select` při aplikaci nebyla procedura reprezentující zobrazení. To ale v zásadě ničemu nevadí, viz diskusi v sekci 2.5 na straně 57.

5.8 Zpracování seznamů obsahujících další seznamy

V této sekci si ukážeme použití seznamů obsažených v seznamech. A to na dvou konkrétních problémových doménách – na práci s datovými tabulkami a na počty s maticemi.

V první části této sekce použijeme seznamy obsahující další seznamy k reprezentaci datových tabulek. Seznam seznamů o stejných délkách můžeme považovat za tabulku s informacemi. Předvedeme si implementaci několika základních operací s datovými tabulkami. Tabulkou tedy budeme rozumět seznam řádků. A řádky budou seznamy o stejné délce. Jako příklad uvádíme tabulku `mesta`:

```
(define mesta
  '((Olomouc 120 3)
    (Prostejov 50 2)
    (Praha 1200 8)))
```

Proceduru vracející n-tý řádek tabulky napíšeme velice jednoduše. Jelikož tabulka je seznam řádků, jde nám o n-tý prvek tohoto seznamu. Ten získáme pomocí procedury `list-ref`:

```
(define n-th-row
  (lambda (table n)
     (list-ref table n)))
```

Což bychom mohli napsat také stručněji následujícím způsobem:

```
(define n-th-row list-ref)
```

O něco složitější by bylo naprogramovat proceduru vracející „n-tý sloupec“ reprezentovaný jako seznam hodnot v tomto sloupci. Z každého řádku umíme vybrat n-tý prvek pomocí procedury `list-ref`. A jelikož máme všechny řádky v jednom seznamu – tabulce – můžeme přes tento seznam mapovat proceduru, která vznikne vyhodnocením λ-výrazu

```
    (lambda (row) (list-ref row n)).
```

Tím dostaneme n-tou hodnotu z každého řádku, tedy n-tý sloupec.

```
(define n-th-column
  (lambda (table n)
    (map (lambda (row)
           (list-ref row n))
         table)))
```

Pro datové tabulky existuje celá řada operací, které mají význam v relační algebře. Jednou z nich je *projekce tabulky*, což je vytvoření nové tabulky z tabulky výchozí výběrem některých jejích sloupců. To jest, výsledkem aplikace projekce bude nová tabulka sestavená z některých sloupců původní tabulky. Proceduru provádějící projekci bychom mohli vytvořit například následujícím způsobem:

```
(define projection
  (lambda (table id-list)
    (map (lambda (row)
           (map (lambda (n)
                  (list-ref row n))
                id-list))
         table)))
```

Viz příklad použití projekce na tabulku navázanou na `mesta`:

```
(projection mesta '(0 2))  ⟹  ((Olomouc 3)
                               (Prostejov 2)
                               (Praha 8))
```

Při použití projekce jsme tedy kromě samotné tabulky ještě předávají seznam sloupců, které budeme chtít v nové tabulce zachovat. Podotkněme, že při implementaci procedury realizující projekci bychom též mohli použít proceduru `n-th-column`, kterou již máme naprogramovanou.

Druhá problémová doména, na které ukážeme využití seznamů a procedur na manipulaci s nimi, budou problematika týkající se číselných matice a operací nad nimi. Matici budeme reprezentovat podobně jako datovou tabulku. Tedy matice bude seznam řádků a řádek přitom bude seznam. Na rozdíl od datové tabulky však tyto řádky budou obsahovat pouze čísla.

První procedurou bude obdoba konstruktoru `build-list`. Chceme vytvořit konstruktor `build-matrix`, který bere jako parametry dvě čísla, představující rozměry matice, a jednu „tvořící“ proceduru, tentokrát však o dvou argumentech. To proto, že vstupem pro tvořící proceduru nebude jen jeden index, tedy pozice v seznamu, ale číslo řádku a číslo sloupce. Procedura `build-matrix` bude vypadat následovně:

```
(define build-matrix
  (lambda (rows cols f)
    (build-list rows
                (lambda (i)
                  (build-list cols
                              (lambda (j)
                                (f i j)))))))
```

V těle předchozí procedury jsme použili konstruktor `build-list` na vytvoření seznamu řádků. Každý z těchto řádků jsme vytvořili opět konstruktorem `build-list`. Viz příklad použití procedury:

```
(build-matrix 2 3 (lambda (x y) (- x y)))  ⟹  ((0 -1 -2) (1 0 -1))
```

Dále vytvoříme procedury pro operace sčítání a odčítání matic. Budou to procedury dvou argumentů, kterými budou matice. Vracet budou novou matici, jejíž prvky budou součty prvků sčítaných matic na stejných pozicích. Implementovány jsou pomocí procedury `map`.

```
(define matrix-+
  (lambda (m1 m2)
```

```
    (map (lambda (row1 row2)
           (map + row1 row2))
         m1 m2)))

(define matrix--
  (lambda (m1 m2)
    (map (lambda (row1 row2)
           (map - row1 row2))
         m1 m2)))
```

Předchozí dvě procedury se odlišovaly jen v použité proceduře při výpočtu součtu nebo rozdílu hodnot. Nabízí se tedy sjednotit obě procedury do jediné procedury vyššího řádu `matrix-map`. Ta bude, oproti procedurám na sčítání a odčítání matic, brát jeden argument navíc. Tímto argumentem bude procedura, která se bude aplikovat na prvky na stejné pozici.

```
(define matrix-map
  (lambda (m1 m2 f)
    (map (lambda (row1 row2)
           (map f row1 row2))
         m1 m2)))
```

Procedury sčítání a odčítání matic můžeme pomocí procedury `matrix-map` nadefinovat například takto:

```
(define matrix-+
  (lambda (m1 m2)
    (matrix-map m1 m2 +)))

(define matrix--
  (lambda (m1 m2)
    (matrix-map m1 m2 -)))
```

Shrnutí

V této lekci jsme zavedli seznamy jako struktury konstruované pomocí párů a speciálního elementu jazyka – – prázdného seznamu. Upřesnili jsme externí reprezentaci seznamů. Vysvětlili jsme, jaký je vztah mezi seznamy chápanými jako data a seznamy chápanými jako symbolické výrazy. Interní reprezentací seznamů (symbolických výrazů) jsou právě seznamy konstruované z párů. Naopak, externí reprezentace seznamů (elementů jazyka), které mají čitelnou externí reprezentaci, je ve tvaru symbolických výrazů (seznamů). Ukázali jsme několik typických procedur pro manipulaci se seznamy. Zaměřili jsme se především na konstruktory seznamů, procedury pro spojování seznamů, obracení seznamů, mapovací proceduru a další. Pozornost jsme věnovali i typovému systému jazyka Scheme. Vysvětlili jsme, jaký je rozdíl mezi statickým a dynamickým typováním; silně a slabě typovanými jazyky; a bezpečným a nebezpečným typováním. Věnovali jsme se i vybraným technických aspektům souvisejícím s konstrukcí interpretů jazyka Scheme. Konkrétně jsme se zabývali problémem automatické správy paměti. V závěru lekce jsme ukázali několik příkladů demonstrující praktickou práci se seznamy.

Pojmy k zapamatování

- seznam, prázdný seznam, hlava seznamu, ocas seznamu,
- kvotování seznamu,
- mapování procedury přes seznam, reverze seznamu, spojování seznamů,
- správa paměti, garbage collector, algoritmus mark & sweep,
- typový systém,
- silně/slabě typovaný jazyky,
- staticky/dynamicky typovaný jazyky,

- bezpečně/nebezpečně typovaný jazyky,

Nově představené prvky jazyka Scheme

- procedury `append`, `build-list`, `list`, `map`, `read`, `reverse`
- predikáty `boolean?`, `list?`, `null?`, `number?`, `pair?`, `procedure?`, `symbol?`

Kontrolní otázky

1. *Co je prázdný seznam?*
2. *Jak jsou definovány seznamy?*
3. *Jak jsme změnili boxovou notaci?*
4. *Jak jsme změnili tečkovou notaci?*
5. *Jaký je ve Scheme vztah mezi programy a daty?*
6. *Co je to garbage collector?*
7. *Popište algoritmy práce garbage collectoru.*
8. *Jak jsme naprogramovali seznamy pamatující si vlastní délku?*
9. *Jaký je rozdíl mezi silně a slabě typovaným jazykem?*
10. *Jaký je rozdíl mezi staticky a dynamicky typovaným jazykem?*
11. *Jaký je rozdíl mezi bezpečně a nebezpečně typovaným jazykem?*

Cvičení

1. Bez použití interpretu vyhodnoťte následující výrazy:

```
(cons 1 (cons (cons 2 '()) (cons 3 '())))          ⟼
(cons (cons '() '()) '())                          ⟼
(caddr '(1 2 3))                                   ⟼

(list '1 2 3)                                      ⟼
(list '(1 2 3))                                    ⟼
(list list)                                        ⟼
(list (+ 1 2))                                     ⟼
(list '+ 1 2)                                      ⟼
(list + 1 2)                                       ⟼
(list '(+ 1 2))                                    ⟼

(reverse '(1 2 3))                                 ⟼
(reverse '((1 2 3)))                               ⟼
(reverse '(1 (2 3) 4))                             ⟼

(build-list 5 -)                                   ⟼
(build-list 3 list)                                ⟼
(build-list 0 (lambda(x) nedef-symbol))            ⟼
(build-list 1 (lambda (x) '()))                    ⟼

(map number? '(1 (2 3) 4))                         ⟼
(map + '(1 2 3))                                   ⟼
(map + '(2 4 8) '(1 3 9))                          ⟼
(map (lambda (x) (cons 1 x)) '(2 4 8) '(1 3 9))    ⟼
(map (lambda (x) '()) '(1 2 3))                    ⟼
(map (lambda (x) nenavazany-symbol) '())           ⟼

(append 1 2)                                       ⟼
```

```
(append 1 '(2))                                    ⟹
(append '(1) '(2))                                 ⟹
(append (append))                                  ⟹
(append '(1 2))                                    ⟹
(append)                                           ⟹
(append '())                                       ⟹
(append '(1 2) '(3 4) (list 5))                    ⟹
(append '(list 5))                                 ⟹

'(map (lambda (x) '()) 10)                         ⟹
(quote quote)                                      ⟹
(quote (quote (1 2 3)))                            ⟹
('quote (1 2 3))                                   ⟹
(car ''10)                                         ⟹
```

2. Naprogramujte proceduru o třech argumentech n, a, d, která vrací seznam prvních n členů aritmetické posloupnosti $\{a + nd\}_{n=0}^{\infty}$
3. Naprogramujte proceduru o třech číselných argumentech n, a a q, která vrací seznam prvních n členů geometrické posloupnosti $\{aq^n\}_{n=0}^{\infty}$.
4. Naprogramujte konstruktor diagonální čtvercové matice a pomocí něho konstruktor jednotkové matice.
5. Napište obdobu procedury `map`, která do mapované procedury dává nejen prvek seznamu ale i jeho index. Například:
   `(map cons '(a b c))` ⟹ `((a . 0) (b . 1) (c . 2))`
   `(map 2-ze-2 '(a b c))` ⟹ `(0 1 2)`

Úkoly k textu

1. Naimplementujte procedury `car`, `cdr` a `cons` pro číselné seznamy, které si pamatují svoji délku a počet prvků.

Řešení ke cvičením

1. (1 (2) 3), ((())), 3
   (1 2 3), ((1 2 3)), (*procedura*), (3), (+ 1 2), (*procedura* 1 2), ((+ 1 2))
   (3 2 1), ((1 2 3)), (4 (2 3) 1)
   (0 -1 -2 -3 -4), ((0) (1) (2)), (), (())
   (#t #f #t), (1 2 3), (3 7 17), (() () ()), (), (1 2), ()
   chyba, chyba, (1 2), (), (1 2), (), (), (1 2 3 4 5), (list 5)
   (map (lambda (x) '()) 10), quote, (quote (1 2 3)), chyba, quote
2. ```
   (define (arit first diff n)
     (build-list n (lambda (i) (+ first (* i diff)))))
   ```
3. ```
   (define (geom first quot n)
     (build-list n (lambda (i) (* first (expt quot i)))))
   ```
4. ```
   (define build-diag-matrix
     (lambda (diag-l)
       (let ((n (length diag-l)))
         (build-matrix n n
                       (lambda (x y)
                         (if (= x y)
                             (list-ref diag-l x)
                             0))))))
   ```
   ```

```
(define build-unit-matrix
  (lambda (n)
    (build-diag-matrix
     (build-list n (lambda (i) 1)))))

(define map-index
  (lambda (f l)
    (map f l (build-list (length l) +))))
```

Lekce 6: Explicitní aplikace a vyhodnocování

Obsah lekce: V předchozích lekcích jsme popsali aplikaci procedur jakožto jednu z částí abstraktního interpretu. Část interpretu odpovědnou za aplikaci jsme označovali „Apply“. Stejně tak jsme popsali část interpretu provádějící vyhodnocování elementů a označovali jsme ji jako „Eval“. V této lekci ukážeme, že „Apply“ a „Eval“ lze bez újmy chápat jako procedury abstraktního interpretu, které mohou programátoři kdykoliv využít k přímé aplikaci procedur na seznamy hodnot a k vyhodnocování libovolných elementů v daném prostředí. Dále ukážeme, že prostředí je možné chápat jako element prvního řádu.

Klíčová slova: aplikace procedur, procedura `apply`, procedura `eval`, prostředí, vyhodnocování elementů.

6.1 Explicitní aplikace procedur

Nyní se budeme věnovat aplikaci procedur. V lekci 1 jsme uvedli, že aplikace procedur probíhá v momentě, kdy se první prvek seznamu vyhodnotil na proceduru. Procedura vzniklá vyhodnocením prvního prvku seznamu je aplikována s argumenty jimiž jsou elementy vzniklé *vyhodnocením* všech dalších prvků seznamu. Této aplikaci procedur můžeme říkat *implicitní aplikace,* protože je prováděna implicitně během vyhodnocování elementů (seznamů). V některých případech bychom ale mohli chtít provést *explicitní aplikaci* dané procedury s argumenty, které máme k dispozici ve formě seznamu – tedy *argumenty již máme k dispozici* a nechceme je získávat vyhodnocením (nějakých výrazů).

Předpokládejme pro ilustraci, že na symbol `s` máme navázaný seznam čísel, který jsme získali jako výsledek aplikace nějakých procedur. Z nějakého důvodu bychom třeba mohli chtít provést součet všech čísel ze seznamu (navázaného na) `s`. Kdybychom mohli v programu zaručit, že tento seznam bude mít vždy pevnou velikost (vždy stejný počet prvků), pak bychom mohli jeho prvky sečíst pomocí

`(+ (car s) (cadr s) (caddr s) ···)`.

Co kdybychom ale délku seznamu dopředu neznali? Pak bychom zřejmě předchozí výraz nemohli přesně napsat, protože bychom neznali počet argumentů, které bychom měli předat proceduře sčítání při její aplikaci. Další problém předchozího kódu je jeho *neefektivita.* Pro seznam délky n potřebujeme provést celkem $\frac{n(1+n)}{2}$ aplikací procedur `car` a `cdr` k tomu, abychom vyjádřili všechny argumenty pomocí `car`, `cadr`, `caddr`, a tak dále. Předchozí řešení tedy není ani univerzální ani efektivní.

Předchozí problém by bylo možné snadno vyřešit, kdybychom měli v jazyku Scheme k dispozici Apply (viz definici 2.12 na straně 50) jako primitivní proceduru vyššího řádu, které bychom mohli předat

(i) *proceduru* E, kterou chceme aplikovat,

(i) *seznam argumentů* $E_1, \ldots, E_n$, se kterými chceme proceduru E aplikovat,

a která by vrátila hodnotu aplikace $\mathrm{Apply}[E, E_1, \ldots, E_n]$.

V jazyku Scheme je taková primitivní procedura skutečně k dispozici. Část interpretu „Apply“, která je odpovědná za aplikaci procedur, je tedy přístupná programátorovi pomocí primitivní procedury vyššího řádu (vyššího řádu proto, že jedním z předaných argumentů je samotná procedura, kterou chceme aplikovat). Tato primitivní procedura je navázána na symbol `apply` a nyní si ji popíšeme.

Definice 6.1 (primitivní procedura `apply`)**.** Primitivní procedura `apply` se používá s argumenty ve tvaru:

`(apply` ⟨*procedura*⟩ ⟨arg_1⟩ ⟨arg_2⟩ $\cdots$ ⟨arg_n⟩ ⟨*seznam*⟩`)`,

přitom ⟨arg_1⟩, …, ⟨arg_n⟩ jsou *nepovinné a mohou být vynechány.* Jelikož je `apply` procedura, je aplikována s argumenty v jejich vyhodnocené podobě. Při aplikaci procedura `apply` nejprve ověří, zda-li je element ⟨*procedura*⟩ primitivní nebo uživatelsky definovaná procedura. Pokud by tomu tak nebylo, ukončí se aplikace hlášením „CHYBA: První argument předaný `apply` musí být procedura.“ V opačném případě procedura `apply` *sestaví seznam hodnot* ze všech ostatních argumentů, takto: první prvek seznamu hodnot bude ⟨arg_1⟩, druhý prvek seznamu hodnot bude ⟨arg_2⟩, … n-tý prvek seznamu hodnot bude ⟨arg_n⟩ a další prvky seznamu hodnot budou tvořeny prvky ze seznamu ⟨*seznam*⟩. Pokud by poslední argument uvedený při

aplikaci `apply` (to jest argument ⟨*seznam*⟩) nebyl seznam, pak se aplikace ukončí hlášením „CHYBA: Poslední argument předaný `apply` musí být seznam.“ Výsledkem aplikace `apply` je hodnota vzniklá aplikací procedury ⟨*procedura*⟩ s argumenty jimiž jsou elementy ze *sestaveného seznamu hodnot*. ■

Z předchozí definice je tedy jasné, že `apply` musí být volána alespoň se dvěma argumenty, první z nich se musí vyhodnotit na proceduru (kterou aplikujeme) a poslední z nich se musí vyhodnotit na seznam argumentů, které chceme při aplikaci použít (seznam může být prázdný).

Nyní ukážeme několik vysvětlujících příkladů použití `apply`, praktické příklady budou následovat v dalších sekcích. Nejprve se zaměříme na použití `apply` pouze se dvěma argumenty, tedy bez $\langle arg_1 \rangle, \ldots, \langle arg_n \rangle$, viz definici 6.1. Vyhodnocení následujících výrazů vede na součet čísel (všimněte si použití `apply`):

```
(+ 1 2 3 4 5)                ⟼  15
(apply + (list 1 2 3 4 5))   ⟼  15
(apply + '(1 2 3 4 5))       ⟼  15
(apply + 1 2 3 4 5)          ⟼  „CHYBA: Poslední argument předaný apply musí být seznam.“
```

V posledním případě při aplikaci došlo k chybě, protože posledním argumentem předaným `apply` nebyl seznam (ale číslo `5`). Při použití `apply` se seznamem hodnot si ale musíme dát pozor na to, že hodnoty v seznamu (poslední argument `apply`) se používají jako argumenty při aplikaci a nejsou před aplikací vyhodnoceny. Všimněte si například rozdílu v následujících dvou aplikacích:

```
(apply + (list 1 (+ 1 2) 5))  ⟼  9
(apply + '(1 (+ 1 2) 5))      ⟼  „CHYBA: Pokus o sčítání čísla se seznamem.“
```

Uvědomte si, že v prvním případě v ukázce vznikl vyhodnocením výrazu `(list 1 (+ 1 2) 5)` tříprvkový seznam `(1 3 5)`, takže došlo k sečtení těchto hodnot. V druhém případě byla ale procedura sčítání aplikována se seznamem `(1 (+ 1 2) 5)` jehož druhým prvkem není číslo, takže aplikace sčítání na argumenty `1` (číslo), `(+ 1 2)` (seznam) a `5` (číslo) selhala.

Pokud se nyní vrátíme k našemu motivačnímu příkladu, tak bychom sečtení hodnot v seznamu navázaném na `s` provedli následovně:

```
(apply + s)
```

Použití `apply` je tedy vhodné v případě, kdy máme dány argumenty (se kterými chceme provést aplikaci nějaké procedury) v seznamu. Nemusíme přitom předávané argumenty nijak „ručně vytahovat“ ze seznamu (jak to bylo naznačeno na začátku této sekce). V případech, kdybychom neznali délku seznamu, protože by byla proměnlivá, by to stejně k uspokojivému řešení nevedlo.

Následující příklady ukazují další použití `apply`:

```
(apply +)                                  ⟼  „CHYBA: Chybí seznam hodnot.“
(apply + '())                              ⟼  0
(apply append '())                         ⟼  ()
(apply append '((1 2 3) () (c d) (#t #f))) ⟼  (1 2 3 c d #t #f)
(apply list '(1 2 3 4))                    ⟼  (1 2 3 4)
(apply cons (list 2 3))                    ⟼  (2 . 3)
(apply cons '(1 2 3 4))                    ⟼  „CHYBA: cons má mít dva argumenty.“
(apply min '(4 1 3 2))                     ⟼  1
```

Doposud jsme ukazovali pouze příklady použití `apply` se dvěma argumenty, to jest s procedurou a seznamem hodnot. Nyní si ukážeme příklady použití `apply` s více argumenty (viz definici 6.1). Nepovinné argumenty, které se nacházejí za předanou procedurou a před posledním seznamem (který musí být vždy přítomen) jsou při aplikaci předávány tak, jak jsou uvedeny. Následující příklad ukazuje několik možností sečtení čísel z daného seznamu za použití nepovinných argumentů `apply`:

```
(apply + 1 2 3 4 '())  ⟼  10
(apply + 1 2 3 '(4))   ⟼  10
(apply + 1 2 '(3 4))   ⟼  10
(apply + 1 '(2 3 4))   ⟼  10
(apply + '(1 2 3 4))   ⟼  10
```

V prvním případě byly nepovinné argumenty čtyři a povinný poslední seznam byl prázdný. V druhém případě byly nepovinné argumenty tři a poslední seznam byl jednoprvkový a tak dále. Uvědomte si, že nepovinné argumenty budou před aplikací `apply`, a tedy i před explicitní aplikací předané procedury, vyhodnoceny. Viz rozdíl mezi následujícími aplikacemi:

```
(apply + '((+ 1 2) 10))   ⟹   „CHYBA: Pokus o sčítání čísla se seznamem.“
(apply + (+ 1 2) '(10))   ⟹   13
```

Otázkou je, kdy použít tyto nepovinné argumenty. Používají se v případě, kdy potřebujeme aplikovat proceduru se seznamem hodnot, ale k tomuto seznamu ještě potřebujeme další hodnoty (zepředu) dodat. Kdybychom například chtěli sečíst všechny hodnoty v seznamu navázaném na `l` s číslem `1`, pak bychom to mohli udělat následujícími způsoby:

```
(define l '(10 20 30 40 50))
(apply + (cons 1 l))          ⟹   151
(+ 1 (apply + l))             ⟹   151
(apply + 1 l)                 ⟹   151
```

V prvním případě jsme ručně jedničku připojili jako nový první prvek seznamu hodnot. V druhém případě jsme přičtení jedničky provedli až po samotné aplikaci. V tomto případě to bylo možné, ale u některých procedur bychom to takto řešit nemohli (uvidíme dále). Poslední příklad byl nejkratší a nejčistší, využili jsme jeden nepovinný argument – přidání jedničky k seznamu argumentů za nás vyřeší procedura `apply`.

Následující příklad je o něco složitější na představivost, ale ukazuje praktičnost nepovinných argumentů, které lze předat proceduře `apply`. Uvažujme tedy aplikaci:

```
(apply map list '((a b c) (1 2 3)))   ⟹   ((a 1) (b 2) (c 3))
```

Při této aplikaci došlo k aplikování `map` jehož prvním argumentem byla procedura navázaná na `list`, druhý a třetí argument byly prvky seznamu `((a b c) (1 2 3))`. Předchozí aplikaci si tedy můžeme představit jako vyhodnocení následujícího výrazu:

```
(map list
     '(a b c)
     '(1 2 3))   ⟹   ((a 1) (b 2) (c 3))
```

Nyní by již výsledná hodnota měla být jasná. Procedura `map` má první argument jiného významu než ostatní argumenty, technicky bychom tedy nemohli provést trik jako byl v případě `+` proveden na třetím řádku v předchozí ukázce.

Na aplikaci procedur na seznamy argumentů se lze dívat jako na způsob *agregace elementů seznamu do jediné hodnoty.* Aplikaci pomocí `apply` používáme v případě, kdy máme vytvořený seznam hodnot, třeba nějakým předchozím výpočtem, a chceme na tento celý seznam aplikovat jednu proceduru. Při aplikaci (tímto způsobem) je potřeba pamatovat na to, že prvky v seznamu předaném `apply` jsou aplikované proceduře předány „jak leží a běží“, tedy bez dodatečného vyhodnocování.

6.2 Použití explicitní aplikace procedur při práci se seznamy

V minulé lekci jsme ukázali řadu procedur pro práci se seznamy. Pro některé z těchto procedur jsme ukázali, jak bychom je mohli naprogramovat, kdybychom je v jazyku Scheme implicitně neměli. Nyní budeme v této problematice pokračovat a zaměříme se přitom na využití `apply`.

Prvním problémem bude stanovení *délky seznamu.* V předchozí lekci jsme ukázali proceduru `length`, která pro daný seznam vrací počet jeho prvků. Nyní ukážeme, jak tuto proceduru naprogramovat. Předpokládejme, že máme na symbol `s` navázaný nějaký seznam, třeba:

```
(define s '(a b c d e f g))
s                             ⟹   (a b c d e f g)
```

Jak napsat výraz, jehož vyhodnocením bude délka seznamu navázaného na `s`? Jde nám přitom o to, nalézt obecné řešení. To jest, když změníme seznam `s`, vyhodnocením výrazu by měla být délka modifikovaného seznamu. Délku seznamu získáme tak, když *sečteme počet jeho prvků*. Při výpočtu délky by tedy zřejmě mělo hrát roli *sčítání*. Nejde však o sčítání elementů nacházejících se v seznamu (což nemusí být ani čísla, jako v našem případě), ale o jejich „počet“. V seznamu se na každé jeho pozici nachází *jeden* element. Na základě seznamu `s` tedy můžeme vytvořit seznam stejné délky tak, že každý prvek ze seznamu `s` zaměníme za `1`:

```
(map (lambda (x) 1) s)  ⟹  (1 1 1 1 1 1 1)
```

K záměně jsme použili mapování konstantní procedury vracející pro každý argument číslo `1`. Nyní již stačí aplikovat na vzniklý seznam proceduru sčítání. Takže délku seznamu navázaného na `s` spočítáme pomocí:

```
(apply + (map (lambda (x) 1) s))  ⟹  7
```

Zopakujme si ještě jednou použitou myšlenku: klíčem k vypočtení délky seznamu bylo „sečíst tolik jedniček, kolik bylo prvků ve výchozím seznamu“. Potřebovali jsme tedy udělat jeden mezikrok, kdy jsme z daného seznamu vytvořili seznam stejné délky obsahující samé jedničky, pak již jen stačilo aplikovat sčítání. Uvědomte si zde dobře roli `apply`. Procedura sčítání je aplikována na seznam „neznámé délky“ (délku seznamu se teprve snažíme zjistit) obsahující hodnoty, se kterými chceme proceduru sčítání aplikovat; například tedy výraz:

```
(+ (map (lambda (x) 1) s))  ⟹  „CHYBA: Nelze sčítat seznam.“
```

by tedy nebyl nic platný. Podle výše uvedeného postupu tedy můžeme naprogramovat proceduru pro výpočet délky seznamu jak je tomu v programu 6.1. Všimněte si, že procedura funguje skutečně stejně tak

Program 6.1. Výpočet délky seznamu.

```
(define length
  (lambda (l)
    (apply + (map (lambda (x) 1) l))))
```

jako procedura `length`, kterou jsme představili v předchozí lekci a to včetně mezního případu. To jest, délka prázdného seznamu je nula, viz následující příklady:

```
(length '(a b c d))      ⟹  4
(length '(a (b (c)) d))  ⟹  3
(length '())             ⟹  0
```

Na tomto příklady je dobře vidět, že pokud provedeme při programování správný myšlenkový rozbor problému, nemusíme se zabývat ošetřování okrajových případů, což často vede ke vzniku nebezpečných chyb, které nejsou vidět.

Pokud někteří čtenáři doposud pochybovali o užitečnosti definovat procedury jako jsou sčítání a násobení „bez argumentů“, tak nyní vidíme, že je to velmi výhodné. Kdyby nebylo sčítání definováno pro prázdný seznam (to jest `(+)` ⟹ `0`), tak bychom v proceduře `length` v programu 6.1 museli ošetřovat speciální případ, kdy bude na argument `l` navázán prázdný seznam. Kdybychom ošetření neprovedli, `length` by počítala pouze délky neprázdných seznamů. Při některých aplikacích `length` v programu by bylo nutné dělat dodatečné testy prázdnosti seznamu – což by vše jen komplikovalo kód.

Nyní ukážeme užitečnou a často používanou proceduru (vyššího řádu), která provádí *filtraci elementů v seznamu podle jejich vlastnosti* (predikát jednoho argumentu), která je dodaná spolu se seznamem formou argumentu. Pro objasnění si nejprve ukážeme několik příkladů, na kterých uvidíme, jak bychom chtěli proceduru používat:

```
(filter even? '(1 2 3 4 5 6))                    ⟼  (2 4 6)
(filter (lambda (x) (<= x 4)) '(1 2 3 4 5 6))    ⟼  (1 2 3 4)
(filter pair? '(1 (2 . a) 3 (4 . k)))            ⟼  ((2 . a) (4 . k))
(filter (lambda (x)
          (not (pair? x)))
        '(1 (2 . a) 3 (4 . k)))                  ⟼  (1 3)
(filter symbol? '(1 a 3 b 4 d))                  ⟼  (a b d)
```

V prvním případě procedura z daného seznamu čísel vyfiltrovala všechna sudá čísla, v druhém případě byla vyfiltrována čísla menší nebo rovna čtyřem, v třetím případě byly ze seznamu vyfiltrovány páry, v dalším případě vše kromě párů a v posledním případě symboly. Proceduru `filter` tedy chceme používat se dvěma argumenty: prvním je predikát jednoho argumentu a druhým je seznam. Chceme, aby procedura vrátila seznam, ve kterém budou právě ty prvky z výchozího seznamu, pro které je daný predikát pravdivý (přesněji: výsledek jeho aplikace je cokoliv kromě `#f`).

Při implementaci filtrace tedy musíme vyřešit problém, jak vynechávat prvky v seznamu. Zde bychom si mohli pomoci aplikací procedury pro *spojování seznamů*, protože při spojování nehrají roli „prázdné seznamy“, ve výsledku spojení jsou vynechány. Můžeme tedy říct, že výsledkem filtrace je spojení jednoprvkových seznamů obsahujících prvky z původního seznamu splňující danou vlastnost. V prvním kroku nám tedy z výchozího seznamu stačí vytvořit nový seznam, ve kterém budou: (i) všechny prvky splňující danou vlastnost obsaženy v jednoprvkových seznamech a (ii) místo ostatních prvků zde budou prázdné seznamy. K vytvoření tohoto seznamu můžeme použít `map`. Uvažujme následující seznam navázaný na `s` a vlastnosti reprezentovanou predikátem navázaným na `even?`:

```
(define s '(1 3 2 6 1 7 4 8 9 3 4))
s                                    ⟼  (1 3 2 6 1 7 4 8 9 3 4)
```

Pak následujícím mapováním získáme:

```
(map (lambda (x)
       (if (even? x)
           (list x)
           '()))
     s)  ⟼  (() () (2) (6) () () (4) (8) () () (4))
```

Na takto vytvořený seznam již jen stačí aplikovat `append`:

```
(apply append
       (map (lambda (x)
              (if (even? x)
                  (list x)
                  '()))
            s))  ⟼  (2 6 4 8 4),
```

což je výsledek filtrace sudých čísel z výchozího seznamu. Hlavní myšlenkou filtrace tedy bylo zřetězení jednoprvkových, případně prázdných, seznamů obsahující filtrované prvky (jednoprvkové seznamy obsahovaly právě prvky splňující vlastnost). Obecnou filtrační proceduru tedy můžeme naprogramovat zobecněním předchozího principu. Viz proceduru `filter` v programu 6.2. Podotkněme, že stejně jako tomu bylo v případě `length` máme naším přístupem opět automaticky vyřešen mezní případ filtrace prvků z prázdného seznamu. Viz příklady:

```
(filter even? '())            ⟼  ()
(filter (lambda (x) #f) '())  ⟼  ()
```

Filtrace je ve funkcionálních jazycích velmi oblíbená. Skoro každý funkcionální programovací jazyk je vybaven nějakou filtrační procedurou vyššího řádu. Pokud ne, lze ji snadno naprogramovat jako tomu bylo v programu 6.2. S pomocí filtrace lze naprogramovat celou řadu užitečných procedur. V programu 6.3 máme příklady dvou z nich. Procedura `remove` je vlastně jen „nepatrnou modifikací“ předchozí filtrační procedury, která spočívá v tom, že prvky splňující danou vlastnost jsou ze seznamu odstraňovány místo toho aby byly ponechávány. S pomocí `filter` již můžeme `remove` naprogramovat snadno – stačí, abychom

Program 6.2. Filtrace prvků seznamu splňujících danou vlastnost.

```
(define filter
  (lambda (f l)
    (apply append
           (map (lambda (x)
                  (if (f x)
                      (list x)
                      '()))
                l))))
```

Program 6.3. Odstraňování prvků seznamu a test přítomnosti prvku v seznamu.

```
(define remove
  (lambda (f l)
    (filter (lambda (x)
              (not (f x)))
            l)))

(define member?
  (lambda (elem l)
    (not (null? (filter
                 (lambda (x)
                   (equal? x elem))
                 l)))))
```

totiž aplikovali `filter` s vlastností reprezentující negaci vlastnosti pro `remove`, viz program 6.3. Druhou procedurou v programu 6.3 je predikát `member?` testující přítomnost daného prvku v seznamu. Myšlenka této procedury je založena na tom, že daný prvek E je obsažen v seznamu, pokud vyfiltrováním prvků vlastnosti „prvek *je roven* E“ vznikne neprázdný seznam (to jest musí být v něm aspoň jeden prvek *roven* E). Viz příklady použití:

```
(member? 'a '())           ⟹  #f
(member? 'a '(1 2 3 4))    ⟹  #f
(member? 'a '(1 2 a 3 4))  ⟹  #t
```

V předchozí lekci jsme ukázali proceduru `list-ref`, která pro daný seznam a danou pozici (číslo) vrací prvek na dané pozici. Nyní si můžeme ukázat, jak lze pomocí filtrace danou proceduru naprogramovat, viz program 6.4. Myšlenka je v tomto případě následující. Procedura `list-ref` si nejprve s použitím procedury

Program 6.4. Procedura vracející prvek na dané pozici v seznamu.

```
(define list-ref
  (lambda (l index)
    (let ((indices (build-list (length l) (lambda (i) i))))
      (cdar
       (filter (lambda (cell) (= (car cell) index))
               (map cons indices l))))))
```

`build-list` vytvoří pomocný seznam indexů `(0 1 2 3` $\cdots$, který je stejně dlouhý jako vstupní seznam.

Pomocí mapování je potom vytvořen pomocný seznam párů ve tvaru `(`⟨*index*⟩ `.` ⟨*prvek*⟩`)`, přitom ⟨*prvek*⟩ je právě prvek seznamu na pozici ⟨*index*⟩. Pak už jen stačí vyfiltrovat z tohoto seznamu prvky (respektive prvek, bude jediný) s vlastností „první prvek páru (to jest index) má hodnotu danou argumentem `index`“. Nakonec stačí jen z tohoto páru vybrat druhý prvek, což je element na dané pozici.

Další zajímavou aplikací filtrace by mohla být procedura `list-indices`, která vlastně provádí opak toho, co procedura `list-ref`. Procedura akceptuje jako argumenty seznam a element a vrací *seznam pozic* (*indexů*), *na kterých se daný element v seznamu vyskytuje.* Obecně je toto řešení lepší než vracet například jen jednu (první) pozici, protože prvek se může v seznamu vyskytovat na víc místech. Proceduru máme uvedenu v programu 6.5. Princip jejího vytvoření je podobný jako u procedury `list-ref`. Opět si vytvoříme seznam

Program 6.5. Procedura vracející všechny pozice výskytu daného prvku.

```
(define list-indices
  (lambda (l elem)
    (let ((indices (build-list (length l) (lambda (i) i))))
      (remove null?
              (map (lambda (x id)
                     (if (equal? x elem)
                         id
                         '()))
                   l
                   indices)))))
```

pomocných indexů a mapováním přes předaný seznam a seznam indexů vytváříme nový seznam, který bude obsahovat buď indexy (v případě že na dané pozici prvek je), nebo prázdné seznamy. Z tohoto meziproduktu již nám pak stačí odstranit prázdné seznamy a získáme tak seznam indexů reprezentujících pozice všech výskytů daného prvku. Viz příklady použití:

```
(list-indices '(a b b a c d a) 'a)  ⟹  (0 3 6)
(list-indices '(a b b a c d a) 'd)  ⟹  (5)
(list-indices '(a b b a c d a) 10)  ⟹  ()
(list-indices '() 'a)               ⟹  ()
```

V sekci 5.8 jsme implementovali procedury pro vytváření matic, a také procedury pro jejich sčítání a odčítání. Nyní tuto sadu rozšíříme o transpozici matice `matrix-transpose` a násobení dvou matic `matrix-*`.

```
(define matrix-transpose
  (lambda (m)
    (apply map list m)))
```

Proceduru map jsme aplikovali na matici, která je reprezentovaná jako seznam seznamů. Mapováním procedury `list` na seznamy představující řádky, dostáváme seznam seznamů s čísly ve stejných sloupcích. Tento seznam můžeme považovat za transponovanou matici.

Násobení matic bychom mohli implementovat následovně. Abychom pracovali jen s řádky, transponujeme druhou matici použitím `matrix-transpose`. Každý řádek `x` první matice skalárně vynásobíme s každým řádkem `y` transponované druhé matice a dostaneme prvek výsledné matice:

```
(define matrix-*
  (lambda (m1 m2)
    (let ((t (matrix-transpose m2)))
      (map (lambda (x)
             (map (lambda (y)
                    (apply + (map * x y)))
                  t))
           m1))))
```

V sekci 5.8 jsme implementovali i selekce a projekce nad databázovými tabulkami. Tyto tabulky byly reprezentovány pomocí seznamů, jejichž prvky byly stejně dlouhé seznamy – řádky. Řádky při selekci byly přitom voleny na základě indexů těchto řádků. Pomocí filtrování můžeme vybírat řádky na základě predikátu. Procedura pro selekci (výběr řádků) bude vytvořena s využitím procedury `filter`:

```
(define selection
  (lambda (table property)
    (filter (lambda (x)
              (apply property x))
            table)))
```

Procedura selekce tak bude fungovat pro libovolný počet sloupců. Viz příklad použití:

```
(define mesta
  '((Olomouc 120 3 stredni)
    (Prostejov 45 2 male)
    (Prerov 50 3 male)
    (Praha 1200 8 velke)))

(selection mesta
           (lambda (jmeno p-obyvatel p-nadrazi velikost)
             (and (>= p-obyvatel 50)
                  (not (equal? velikost 'male)))))
  ⟹  ((olomouc 120 3 stredni)
      (praha 1200 8 velke))
```

6.3 Procedury s nepovinnými a libovolnými argumenty

Řada primitivních procedur, se kterými jsme se doposud setkali, umožňovala mít při jejich aplikaci některé argument nepovinné. Například procedura `map` musela mít k dispozici jako argument proceduru a seznam a volitelně jako nepovinné argumenty ji mohly být předány ještě další seznamy. Některé primitivní procedury, jako například `+`, `*` a `append` mohly být aplikovány dokonce s libovolným počtem argumentů, včetně žádného argumentu. V této sekci si ukážeme, jak lze vytvářet uživatelsky definované procedury s nepovinnými argumenty nebo s libovolnými argumenty.

Nejprve ukážeme, jak je možné vytvořit procedury, které mají několik povinných argumentů, které musejí být vždy uvedeny, a kromě nich mohou být předány další nepovinné argumenty. Platí podmínka, že *nepovinné argumenty* lze uvádět až *za všemi povinnými*. Při psaní λ-výrazů jejichž vyhodnocením mají vzniknout procedury pracující s nepovinnými argumenty, píšeme místo tradiční specifikace seznamu argumentů

$$(\langle param_1\rangle\ \langle param_2\rangle \cdots \langle param_n\rangle)\,,$$

kterou jsme používali doposud, seznam argumentů ve tvaru

$$(\langle param_1\rangle\ \langle param_2\rangle \cdots \langle param_n\rangle\ .\ \langle zbytek\rangle)\,,$$

kde $\langle param_1\rangle, \langle param_2\rangle, \ldots, \langle param_n\rangle, \langle zbytek\rangle$ jsou vzájemně různé symboly. To jest kromě povinných formálních argumentů (zapsaných jako dosud), jsme pomocí tečky „ . “ oddělili poslední symbol $\langle zbytek\rangle$. Přísně vzato, struktura argumentů zapsaná v tomto tvaru již *není seznam*, protože druhý prvek jeho „posledního páru“ není prázdný seznam. Úkol argumentu $\langle zbytek\rangle$ je následující. Při aplikaci procedury vzniklé vyhodnocením λ-výrazu se hodnoty všech povinných argumentů naváží na symboly $\langle param_1\rangle, \ldots, \langle param_n\rangle$ (jako doposud). Pokud byly navíc při aplikaci použity další argumenty, pak je vytvořen seznam všech těchto dodatečných argumentů a při aplikaci procedury je tento seznam navázaný na symbol $\langle zbytek\rangle$. Pokud tedy žádné nepovinné argumenty nebyly předány, na $\langle zbytek\rangle$ bude navázaný prázdný seznam.

Následující příklad demonstruje použití nepovinných argumentů:

```
((lambda (x y . rest) (list x y rest)) 1 2)       ⟹  (1 2 ())
((lambda (x y . rest) (list x y rest)) 1 2 3)     ⟹  (1 2 (3))
```

```
((lambda (x y . rest) (list x y rest)) 1 2 3 4 5)  ⟼  (1 2 (3 4 5))
((lambda (x y . rest) (list x y rest)) 1)          ⟼  „CHYBA: Chybí argument.“
```

V předchozích případech jsme tedy definovali proceduru, která měla dva povinné argumenty (v proceduře reprezentované formálními argumenty `x` a `y`) a dále mohla mít nepovinné argumenty, jejichž seznam byl při aplikaci navázaný na symbol `rest`. V prvním případě byly předány právě dva povinné argumenty, takže seznam nepovinných argumentů byl prázdný. V druhém případě již seznam nepovinných argumentů obsahoval jeden prvek. V třetím případě bylo předáno celkem pět argumentů, takže seznam nepovinných argumentů obsahoval poslední tři z nich.

Příklad použití nepovinných argumentů je v programu 6.6. Procedura `find` provádí podobnou činnost

Program 6.6. Test přítomnosti prvku v seznamu s navrácením příznaku.

```
(define find
  (lambda (elem l . not-found)
    (cond ((member? elem l) elem)
          ((null? not-found) #f)
          (else (car not-found)))))
```

jako procedura `member?` z programu 6.3 na straně 147 (`find` je, jak vidíme, dokonce naprogramovaná pomocí `member?`). Procedura `find` má dva povinné argumenty, prvním z nich je element, druhým je seznam. Procedura slouží k rozhodování, zda-li se daný element nachází v seznamu hodnot. V případě nalezení je ale vrácen samotný element (to se může v některých případech hodit), v případě nenalezení je vráceno standardně `#f`. Co kdybychom ale chtěli v seznamu hledat element „nepravda“, to jest samotné `#f`? Pak bychom vždy tak jako tak dostali jako výsledek aplikace `#f` (v případě nalezení i nenalezení prvku v seznamu). Problém bychom mohli napravit tak, že proceduře budeme předávat nepovinný argument, jehož hodnota bude vrácena v případě, kdy prvek nalezen nebude. Pokud nepovinný argument nebude uveden, pak při nenalezení prvku vrátíme standardní `#f`. Viz ukázky použití procedury:

```
(find 'a '(a b c d))                    ⟼  a
(find 'x '(a b c d))                    ⟼  #f
(find 'a '(a b c d) 'prvek-nenalezen)   ⟼  a
(find 'x '(a b c d) 'prvek-nenalezen)   ⟼  prvek-nenalezen
(find #f '(a b c d))                    ⟼  #f
(find #f '(a b #f d))                   ⟼  #f
(find #f '(a b c d) 'prvek-nenalezen)   ⟼  prvek-nenalezen
(find #f '(a b #f d) 'prvek-nenalezen)  ⟼  #f
```

Všimněte si, že procedura `find` pracuje de facto pouze s jedním nepovinným argumentem. Zbytek nepovinných argumentů, které by byly při její aplikaci předány v seznamu navázaném na symbol `not-found`, je procedurou ignorován.

Nyní obrátíme naši pozornost na problematiku předávání libovolného počtu argumentů. V předchozí notaci musela mít každý procedura aspoň jeden povinný argument, protože výraz `( . rest)` by nebyl syntakticky správně. Co když ale potřebujeme definovat proceduru, která může mít jakýkoliv počet argumentů. Z praxe takové procedury známe a víme o tom, že „libovolné argumenty“ jsou užitečné (vzpomeňme například jen proceduru `append`).

Uživatelsky definované procedury, které mají mít libovolný počet argumentů, vznikají vyhodnocením λ--výrazů, ve kterých je místo seznamu formálních argumentů uveden *jediný symbol.* Na tento jediný symbol bude při aplikaci navázán *seznam všech argumentů.* Viz příklady pro ilustraci:

```
((lambda args (list 'predano args)) 1 2 3 4 5 6)  ⟼  (predano (1 2 3 4 5 6))
((lambda args (list 'predano args)))              ⟼  (predano ())
((lambda args (reverse args)) 1 2 3 4 5 6)        ⟼  (6 5 4 3 2 1)
```

V prvních dvou příkladech byla aplikována procedura, která jako výsledek vrátila dvouprvkový seznam: na jeho první pozici byl symbol `predano` a na druhé pozici byl seznam všech předaných argumentů. V třetím případě jsme viděli ukázku procedury, která dané argumenty vrátí v obráceném seznamu.

V programu 6.7 je uvedena procedura `+2m` provádějící součet čtverců přes libovolné argumenty. V druhé

Program 6.7. Součet druhých mocnin.

```
(define
  +2m
  (lambda values
    (apply + (map (lambda (x) (* x x)) values))))
```

lekci jsme v programu 2.1 na straně 51 definovali proceduru na výpočet součtu dvou čtverců jako jednu z prvních procedur vůbec. V programu 6.7 se tedy nachází její zobecnění pracující s libovolným počtem argumentů. Pomocí mapování je ze seznamu čísel vytvořen seznam jejich druhých mocnin a pomocí aplikace sčítání je získána výsledná hodnota. Vše opět funguje i v mezním případě, kdy je procedura `+2m` zavolána bez argumentu. Viz následující příklady:

```
(+2m)       ⟼  0
(+2m 2)     ⟼  4
(+2m 2 3)   ⟼  13
(+2m 2 3 4) ⟼  29
```

Následující procedura provádí spojení libovolně mnoha seznamů v opačném pořadí:

```
(define
  rev-append
  (lambda lists
    (reverse (apply append lists))))
```

Proceduru můžeme použít následovně:

```
(rev-append)                      ⟼  ()
(rev-append '(a b))               ⟼  (b a)
(rev-append '(a b) '(c d))        ⟼  (d c b a)
(rev-append '(a b) '(c d) '(1 2 3)) ⟼  (3 2 1 d c b a)
```

V předchozí lekci jsme ukázali konstruktor seznamu `list`. Nyní je ale jasné, že pokud máme k dispozici aparát pro předávání libovolného množství argumentů pomocí jejich seznamu, pak lze proceduru `list` snadno naprogramovat tak, jak je to uvedeno v programu 6.8. V tomto programu je procedura `list`

Program 6.8. Vytvoření konstruktoru seznamu.

```
(define list
  (lambda list list))
```

definována jako procedura akceptující libovolné argumenty, která vrací seznam těchto argumentů, což je přesně to, co provádí `list` představený v předchozí lekci.

Následující definice shrnuje, jak vypadá syntaxe λ-výrazů. V tomto ani v následující části učebního textu (týkající se imperativních rysů při programování) ji již nebudeme nijak rozšiřovat.

Definice 6.2 (**λ-výraz s nepovinnými a libovolnými formálními argumenty**)**.** Každý seznam ve tvaru

`(lambda (`$\langle param_1 \rangle\ \langle param_2 \rangle \cdots \langle param_m \rangle$`)` $\langle výraz_1 \rangle\ \langle výraz_2 \rangle \cdots \langle výraz_k \rangle$`)`, nebo
`(lambda (`$\langle param_1 \rangle\ \langle param_2 \rangle \cdots \langle param_n \rangle$ `.` $\langle zbytek \rangle$`)` $\langle výraz_1 \rangle\ \langle výraz_2 \rangle \cdots \langle výraz_k \rangle$`)`, nebo

`(lambda` $\langle parametry \rangle$ $\langle výraz_1 \rangle$ $\langle výraz_2 \rangle \cdots \langle výraz_k \rangle$`)`,

kde n, k jsou kladná čísla, m je nezáporné číslo, $\langle param_1 \rangle, \langle param_2 \rangle, \ldots, \langle param_n \rangle, \langle zbytek \rangle$ jsou vzájemně různé symboly, $\langle parametry \rangle$ je symbol, a $\langle výraz_1 \rangle, \ldots, \langle výraz_k \rangle$ jsou libovolné výrazy tvořící *tělo*, se nazývá λ-*výraz* (*lambda výraz*). Symboly $\langle param_1 \rangle, \ldots, \langle param_n \rangle$ se nazývají *formální argumenty* (někdy též *parametry*). Čísla m, n nazýváme *počet povinných formálních argumentů* (*parametrů*). Symbol $\langle zbytek \rangle$ se nazývá *formální argument* (*parametr*) *zastupující seznam nepovinných argumentů*. Symbol $\langle parametry \rangle$ se nazývá *formální argument* (*parametr*) *zastupující seznam všech argumentů*. ■

V jazyku Scheme je možné vytvářet uživatelsky definované procedury, které mají jakýkoliv počet argumentů, nebo mají některé argumenty povinné, vždy alespoň jeden, a ostatní argumenty jsou nepovinné. V obou případech jsou nepovinné argumenty předávány proceduře formou seznamu, který je navázaný na speciální formální argument.

Poznámka 6.3. Programovací jazyky mají různé způsoby, jak předat nepovinné argumenty. Jednou z oblíbených metod, kterou disponují například jazyky jako je Common LISP, PHP a další je předávání nepovinných argumentů, které jsou identifikovány svým jménem (tak zvaným *klíčem*).

6.4 Vyhodnocování elementů a prostředí jako element prvního řádu

Nyní se budeme věnovat primitivní proceduře, pomocí niž budeme schopni získat na žádost hodnotu vzniklou vyhodnocením elementu. Analogicky jako jsme v předešlých sekcích řekli, že „Apply“ je k dispozici programátorovi prostřednictvím primitivní procedury vyššího řádu `apply`, tak i „Eval“ bude uživateli k dispozici prostřednictvím primitivní procedury (vyššího řádu) `eval`. Pomocí této primitivní procedury budeme moci provádět *explicitní vyhodnocení elementů*. Veškeré doposud používané vyhodnocování bylo vždy *implicitní*.

Primitivní procedura `eval` je aplikována se dvěma argumenty z nichž druhý argument je nepovinný. Prvním (povinným) argumentem je *element*, který chceme vyhodnotit. Druhým nepovinným argumentem je *prostředí*, ve kterém chceme daný element vyhodnotit. Pokud není prostředí uvedeno, `eval` bude uvažovat vyhodnocení v globálním prostředí $\mathcal{P}_G$. Výsledkem aplikace `eval` pro dané argumenty je výsledek vyhodnocení elementu v prostředí. Z toho, co jsme teď řekli plyne, že argumenty předávané `eval` plně korespondují s argumenty pro „Eval“ tak, jak byla popsán v lekci 2, viz definici 2.7 na straně 48.

Uveďme si nyní nějaké příklady použití `eval`, zatím pouze s jedním argumentem jímž je element, který bude vyhodnocen:

```
(eval 10)          ⟼  10
(eval '+)          ⟼  „procedura sčítání čísel“
(eval '(+ 1 2))    ⟼  3
```

Předchozí tři příklady korespondují s body (A), (B) a (C) definice vyhodnocování, protože číslo se vyhodnotilo na sebe sama, symbol `+` se vyhodnotil na svou vazbu a seznam se vyhodnotil obvyklým postupem. Zde upozorněme na fakt, že `eval` je skutečně procedura, tedy před její aplikací jsou vyhodnoceny její argumenty. Proto jsme museli předat proceduře symbol `+` pomocí kvotování, stejně tak seznam `(+ 1 2)`. Kdybychom to neučinili, symbol `+` by se vyhodnotil na svou vazbu a proceduře `eval` by byla předána k vyhodnocení procedura. V tom případě by se dle bodu (D) procedura vyhodnotila na sebe sama:

```
(eval +)           ⟼  „procedura sčítání čísel“
((eval +) 1 2)     ⟼  3
```

V tomto bodu by nám asi mělo být jasné, proč jsme do definice vyhodnocování, viz definici 2.7 na straně 48, přidali bod (D). Doposud se během výpočtu vyhodnocovaly pouze elementy, které byly interními formami symbolických výrazů – těch, co jsme uvedli v programu. Pokud ale máme k dispozici evaluátor ve formě procedury `eval`, je možné mu předat libovolný element k vyhodnocení, tedy i element, který není interní formou žádného symbolického výrazu, jak je tomu například u procedur.

Pomocí `eval` je možné manipulovat s daty jako s programem. V předchozích lekcích jsme upozornili na fakt, že programy v jazyku Scheme lze chápat jako data. Interní formy seznamů jsou konstruovány pomocí párů. Pomocí `eval` tedy máme možnost vyhodnocovat datové struktury reprezentující „kusy programu“. To nám na jednu stranu dává obrovský potenciál, protože můžeme třeba *uživatelský vstup* transformovat na kód a spustit jej, což usnadňuje řadu operací. Na druhou stranu je použití `eval` krajně nebezpečné a mělo by být vždy odůvodněné.

V následujícím příkladu ukazujeme konstrukci dvou seznamů (data), která jsou použita „jako program“:

```
(eval (cons '+ (cons 1 (cons 2 '()))))  ⟹  3
(eval (cons + (cons 1 (cons 2 '()))))   ⟹  3
(cons + (cons 1 (cons 2 '())))          ⟹  („procedura sčítání čísel“ 1 2)
```

Všimněte si, že na druhém řádku byl zkonstruován seznam `(„procedura sčítání čísel“ 1 2)`, který začíná procedurou a dalšími prvky jsou dvě čísla. Oproti prvnímu řádku tedy nestojí na prvním místě seznamu symbol, ale přímo procedura. Této situace bychom nemohli dosáhnout, kdybychom nepoužívali `eval` explicitně.

V následující ukázce jsme vyhodnocením vytvořili proceduru a dále ji aplikovali. Jelikož `eval` s jedním argumentem vyhodnocuje elementy v globálním prostředí, bude prostředí vzniku této procedury právě globální prostředí:

```
(eval '(lambda (x) 10))                 ⟹  „konstantní procedura vracející 10“
((eval '(lambda (x) 10)) 20)            ⟹  10
(apply (eval '(lambda (x) 10)) 20 '())  ⟹  10
```

Vyhodnocení následujícího výrazu končí chybou

```
(let ((x 10))
  (eval '(+ x 1)))  ⟹  „CHYBA: Symbol x nemá vazbu.“,
```

protože seznam `(+ x 1)` byl vyhodnocen v globálním prostředí (ve kterém `x` nemá vazbu) a to i navzdory tomu, že `eval` jsme uvedli v `let`-bloku, kde měl symbol `x` vazbu.

Jako další příklad si uveďme následující proceduru vyššího řádu:

```
(define proc
  (lambda (c)
    (eval '(lambda (x) (+ x c)))))
```

Tato procedura při své aplikaci vrací novou proceduru, která byla ale vytvořena v globálním prostředí. To jest při aplikaci `proc` je sice předán argument navázaný na `c`, jeho vazba ale není viditelná z prostředí vzniku vrácené procedury. Kdybychom tuto vrácenou proceduru aplikovali, vazba symbolu `c` by byla hledána v globálním prostředí, viz ukázku:

```
((proc 10) 20)  ⟹  „CHYBA: Symbol c nemá vazbu.“
(define c 100)
((proc 10) 20)  ⟹  120
```

Kdybychom někdy potřebovali vyrábět proceduru vyhodnocením seznamu (třeba protože část procedury by byla dodána až během činnosti programu pomocí interakce s uživatelem), pak bychom mohli problém s vazbou volných symbolů vyřešit tak, že místo symbolů bychom do seznamu rovnou dosadili výsledky jejich vyhodnocení – jejich vazby v aktuálním prostředí, ve kterém `eval` aplikujeme. Viz následující ukázku:

```
(define proc
  (lambda (c)
    (eval (list 'lambda
                (list 'x)
                (list '+ 'x c)))))
```

V tomto případě procedura `proc` vrací proceduru, která vznikne vyhodnocením seznamu v globálním prostředí. V tomto případě jsem ale do seznamu místo symbolu `c` vložili hodnotu jeho vazby v lokálním prostředí procedury `proc`. Vytvořili jsme tak vlastně seznam ve tvaru

`(lambda (x) (+ x „hodnota c“))`

a ten byl vyhodnocen v globálním prostředí. Vzniklou proceduru již tedy můžeme používat bez nutnosti provádět globální definici a procedura má kýžený efekt:

`((proc 10) 20)` $\longmapsto$ `30`

> Z pohledu jazyka Scheme jsou *data* totéž co *program*. Program lze chápat jako data a data mohou být použita pomocí `eval` jako program. Při používání `eval` je však potřeba dbát velké obezřetnosti, protože jeho (nadměrné) používání Často znesnadňuje ladění programu. Chyby mohou vznikat za běhu programu, aniž by byly v programu „vidět“ na nějakém jeho konkrétním místě.

Našim dalším cílem bude naimplementovat procedury `forall` a `exists` reprezentující kvantifikátory $\forall$ (všeobecný kvantifikátor „pro každý . . . “) a $\exists$ (existenční kvantifikátor „existuje . . . “). Budou jako argument brát predikát o jednom argumentu ⟨*predikát*⟩ a seznam ⟨*seznam*⟩ a vracet pravdivostní hodnotu. V případě `forall` to bude pravda `#t`, pokud každý prvek seznamu ⟨*seznam*⟩ splňuje predikát ⟨*predikát*⟩ a jinak `#f`. Procedura pro existenční kvantifikátor bude vracet `#t`, pokud alespoň jeden prvek ze seznamu ⟨*seznam*⟩ splňuje predikát ⟨*predikát*⟩. V opačném případě bude vracet `#f`. Jelikož obě procedury si budou podobné, omezíme se v následujícím na rozbor procedury `forall` modeující všeobecný kvantifikátor.

Pomocí `map` a daného predikátu dostaneme z původního seznamu seznam pravdivostních hodnot určujících, zda-li prvek na dané pozici splňuje podmínku danou predikátem:

`(map` ⟨*predikát*⟩ ⟨*seznam*⟩`)`.

Nyní potřebujeme zjistit, jaké pravdivostní hodnoty seznam obsahuje. V případě, že by `and` byla procedura, mohli bychom toho dosáhnout pomocí procedury `apply`. Ale jelikož jde o speciální formu, obdržíme při případném pokusu o aplikaci chybu:

`(apply and '(#t #t #f))` $\longmapsto$ „CHYBA: Nesprávné použití speciální formy `and`“ .

Potřebujeme tedy mít „and“ a „or“ jako procedury libovolně mnoha argumentů. Budeme se teď zabývat tímto problémem. Proceduru pro „and“ – `and-proc` bychom mohli implementovat například takto:

```
(define and-proc
  (lambda args
    (null? (remove (lambda (x) x) args))))
```

Nejdříve jsme použitím procedury `remove` ze seznamu argumentů navázaného na symbol `args` odstranili všechny prvky různé od `#f`, použitím procedury `remove` s procedurou identity. Poté jsme otestovali, zda je výsledný seznam prázdný. Pokud ano, znamená to, že žádný argument procedury `and-proc` nebyl nepravda `#f`, a tedy výsledkem aplikace procedury `and-proc` bude pravda. V opačném případě bude výsledkem `and-proc` nepravda.

Procedura `and-proc` je tedy implementací operace logické konjunkce:

```
(and-proc)                      ⟼ #t
(and-proc 1 2 3)                ⟼ #t  rozdíl oproti and: (and 1 2 3) ⟼ 3
(and-proc #t (< 1 2) #t)        ⟼ #f
```

Na `and-proc` je navázána procedura, nikoli speciální forma. Důsledkem toho je, jakým způsobem se vyhodnocují její argumenty:

`(and-proc #f nenavazany-symbol #f)` $\longmapsto$ „CHYBA: Symbol `nenavazany-symbol` nemá vazbu“

Pro nás je důležitější ta skutečnost, že `and-proc` jako procedura může být použita třeba jako argument procedury `apply`:

`(apply and-proc '(#t #t #t))` $\longmapsto$ `#t`

K implementaci `and-proc` bychom také mohli chytře použít speciální formu `and` a proceduru `eval`. Máme-li seznam argumentů, můžeme na jeho začátek přidat symbol `and` a výsledný seznam explicitně vyhodnotit použitím `eval`.

```
(define and-proc
  (lambda args
    (eval (cons 'and args))))
```

Analogicky můžeme vytvořit proceduru, která bude obdobou speciální formy `or`:

```
(define or-proc
  (lambda args
    (eval (cons 'or args))))

(and-proc 1 2 3 #f 10)       ⟼ #f
(and-proc (+ 1 2))           ⟼ 3
(and-proc  (if #f #t #t))    ⟼ #t
```

Poznámka 6.4. Tato implementace ale není úplně korektní a bude pracovat správně jen do té doby, dokud výsledky vyhodnocení argumentů budou pravdivostní hodnoty, čísla nebo jiné elementy, které se vyhodnocují samy na sebe. Podívejme se na to na následujícím příkladě, kde dostáváme opačné hodnoty pro stejný argument.

```
(and-proc '(if #f #t #f))  ⟼  #f
(and '(if #f #t #f))       ⟼  (if #f #t #f) tedy hodnota považovaná za pravdu.
```

Speciální forma `and` nám vrací čtyřprvkový seznam `(if #f #t #f)`. Ten vznikne vyhodnocením výrazu `'(if #f #t #f)` a je vrácen jako výsledek aplikace této formy, protože se jedná o poslední argument. Při aplikaci procedury `and-proc` dochází k nepříjemnému efektu, který není na první pohled zřejmý. Protože se jedná o proceduru, jsou její argumenty implicitně vyhodnoceny. Do seznamu jejich vyhodnocení je přidán symbol `and` a pak je vyhodnocen použitím speciální formy `eval`. Při aplikaci formy `and`, jak je popsána v definici 2.22 na straně 66, se postupně *vyhodnocují* argumenty. Argumenty jsou tak vlastně vyhodnoceny dvakrát. Vyhodnocením seznamu `(if #f #t #f)` dostaneme `#f`. Odtud výsledná hodnota. S procedurou `or-proc` to samozřejmě bude podobné.

Pomocí procedur `and-proc` a `or-proc` můžeme konečně naprogramovat proceduru univerzálního kvantifikátoru `forall` a proceduru existenčního kvantifikátoru `exists`.

```
(define forall
  (lambda (f l)
    (apply and-proc (map f l))))

(define exists
  (lambda (f l)
    (apply or-proc (map f l))))
```

Procedura univerzálního kvantifikátoru `forall` tedy vrací pravdu, pokud predikát platí pro všechny prvky v seznamu:

```
(forall even? '(1 2 3 4 5))  ⟼  #f
(forall even? '(2 4))        ⟼  #t
(forall even? '(1 3 5))      ⟼  #f
(forall even? '())           ⟼  #t
```

Všimněte si posledního případu: každý prvek prázdného seznamu splňuje jakoukoliv vlastnost triviálně (souhlasí s vlastnostmi všeobecného kvantifikátoru). Analogicky procedura existenčního kvantifikátoru `exists` vrací pravdu, pokud predikát platí alespoň pro jeden prvek v seznamu:

```
(exists even? '(1 2 3 4 5))  ⟼  #t
(exists even? '(2 4))        ⟼  #t
(exists even? '(1 3 5))      ⟼  #f
(exists even? '())           ⟼  #f
```

Naše kvantifikátory můžeme rozšířit na procedury více argumentů. Podobně jako u procedury `map` pak vstupní predikát musí přijímat tolik argumentů, kolik je vstupních seznamů. Predikát je pak aplikován na prvky na stejných pozicích.

```
(define forall
  (lambda (f . lists)
    (apply and-proc (apply map f lists))))

(define exists
  (lambda (f . lists)
    (apply or-proc (apply map f lists))))
```

Všeobecný kvantifikátor `forall` pak zjišťuje, zda všechny prvky na stejných pozicích v seznamech splňují tento predikát.

```
(forall (lambda (x y) (<= x y)) '(10 20 30) '(11 22 33))  ⟹  #t
(forall (lambda (x y) (<= x y)) '(10 23 30) '(11 22 33))  ⟹  #f
```

V předchozí ukázce bychom přísně vzato nemuseli používat λ-výrazy, stačilo by pouze uvést:

```
(forall <= '(10 20 30) '(11 22 33))  ⟹  #t
(forall <= '(10 23 30) '(11 22 33))  ⟹  #f
```

A podobně existenční kvantifikátor `exists` pro více seznamů vrací pravdu `#t`, jestliže prvky na stejných pozicích splňují daný predikát.

```
(exists (lambda (x y) (> x y)) '(10 20 30) '(11 22 33)) ⟹ #f
(exists (lambda (x y) (> x y)) '(10 23 30) '(11 22 33)) ⟹ #t
```

Teď se budeme zabývat procedurou `eval` se dvěma argumenty. Jak již bylo řečeno, prvním argumentem je element k vyhodnocení, druhým argumentem je *prostředí*, ve kterém má k vyhodnocení dojít. Zde se vlastně dostáváme do zajímavého bodu, protože pokud chceme, abychom pomocí `eval` vyhodnocovali elementy relativně vzhledem k prostředí, pak musí být prostředí v jazyku Scheme *elementem prvního řádu.* Vskutku, prostředí jsou de facto tabulky obsahující symboly a jejich vazby plus ukazatele na svého předchůdce. Nic nám tedy nebrání abychom tyto „tabulky“ chápali jako elementy jazyka Scheme. *Prostředí* je pro nás tedy *nový element jazyka.*

Proto, abychom mohli pracovat s prostředím jako s elementem, potřebujeme mít k dispozici nějaké primitivní procedury nebo speciální formy, které budou nějaká prostředí vracet. Nejprve budeme uvažovat speciální formu `the-environment`:

Definice 6.5 (speciální forma `the-environment`)**.** Speciální forma `the-environment` se používá bez argumentu. Výsledkem její aplikace je prostředí, ve kterém byla aplikována (aktuální prostředí). ■

Před tím, než ukážeme příklad si uvědomme, proč je `the-environment` speciální forma a nikoliv procedura. Při aplikaci procedur nehraje roli prostředí, ve kterém byly procedury aplikovány, protože používáme lexikální rozsah platnosti. Procedury ani nemají možnost zjistit, v jakém prostředí byly aplikovány. Naproti tomu speciální formy řídí vyhodnocování svých argumentů, musí mít tedy prostředí své aplikace k dispozici. Speciální forma `the-environment` prostě udělá jen to, že toto prostředí vrátí jako výsledek. Viz příklady použití:

```
(the-environment)          ⟹  „globální prostředí“
((lambda (x)
   (the-environment))
 10)                       ⟹  „lokální prostředí procedury, kde x ↦ 10“
(let ((x 10))
  (the-environment))       ⟹  „lokální prostředí procedury, kde x ↦ 10“
```

V druhém případě si všimněte, že procedura vzniklá vyhodnocením λ-výrazu ve svém těle provede pouze aplikaci `the-environment`. Při aplikaci této procedury je vytvořeno lokální prostředí, v němž je na `x`

navázána hodnota 10 a toto prostředí je vráceno. V třetím případě se jedná o stejný případ, protože let--výraz je v podstatě jen zkrácením výrazu na druhém řádku.

Nyní již můžeme provést vyhodnocení výrazu v aktuálním prostředí:

```
(let ((x 10))
  (eval '(+ x 1) (the-environment)))  ⟹  11
```

Kdybychom v předchozím příkladu u eval neuvedli druhý argument, pak by byl výraz vyhodnocen v globálním prostředí a nastala by chyba. My jsme ale výraz vyhodnotili v aktuálním prostředí (v tele let-bloku), to jest v prostředí, kde je na x navázána hodnota.

Dále budeme v jazyku uvažovat proceduru environment-parent, která *pro dané prostředí vrátí jeho předka.* V případě, že je environment-parent aplikována na globální prostředí, které předka nemá, pak vrátí #f. Například použitím

```
(let* ((x 10)
       (y 20))
  (environment-parent (the-environment)))
```

bychom získali prostředí, ve kterém má vazbu symbol x, ale ve kterém nemá vazbu symbol y. Musíme si uvědomit, že speciální forma let* vytváří s každým symbolem nové prostředí a tato prostředí jsou do sebe zanořená, viz třetí lekci.

Procedura procedure-environment pro danou uživatelsky definovanou proceduru *vrátí prostředí jejího vzniku.* Například pomocí

```
(procedure-environment
 (let ((x 10))
   (lambda (y)
     (+ x y))))
```

získáme prostředí vzniku procedury vzniklé vyhodnocením uvedeného λ-výrazu. To je prostředí vytvořené pomocí let, ve kterém je na symbol x navázána hodnota 10. Máme-li k dispozici procedure-environment, pak bychom již nutně nemuseli mít the-environment, protože kdekoliv v programu pomocí

```
(procedure-environment (lambda () #f))
```

můžeme získat aktuální prostředí. Samozřejmě, že toto řešení je méně efektivní, protože při něm vždy vytvoříme novou proceduru jen proto, abychom posléze získali prostředí jejího vzniku.

Poslední pomocnou procedurou, kterou představíme, je procedura environment->list, která pro dané prostředí vrací seznam tečkových párů ve tvaru ((*⟨symbol⟩* . *⟨vazba⟩*)) obsahující všechny vazby v daném prostředí. Pomocí této procedury tedy budeme schopni „srozumitelně vypisovat“ obsah prostředí. Samotná prostředí nemají žádnou čitelnou externí reprezentaci. Viz příklad použití právě popsané procedury:

```
(environment->list
 (procedure-environment
  (let ((x 10)
        (z 20))
    (lambda (y)
      (+ x y)))))        ⟹  ((x . 10) (z . 20))

(environment->list
 ((lambda (x y)
    (the-environment))
  100 200))              ⟹  ((x . 100) (y . 100))
```

Použitím předchozích procedur spolu s env můžeme provádět vyhodnocování elementů v aktuálním prostředí i v prostředích vzniku daných procedur. Například následující ukázka demonstruje vyhodnocení seznamu v prostředí vzniku nějaké procedury.

```
(eval '(* x x)
      (procedure-environment
       (let ((x 10))
         (lambda (y) (+ x y)))))  ⟹  100
```

Procedura vzniklá vyhodnocením `(lambda (y) (+ x y))` není v předchozím příkladu vůbec aplikována.

Prostředí je v našem pojetí elementem prvního řádu. Na prostředí se taky můžeme dívat jako na speciální hierarchická data. Konstruktorem prostředí je aplikace uživatelsky definovaných procedur, protože při ní prostředí vznikají. Selektory prostředí jsou reprezentovány procedurami jako je `environment-parent` a podobně.

Pokud jsme o používání `eval` řekli, že je nebezpečné, pak bychom měli o používání `eval` s druhým argumentem (prostředím) říct, že je ještě mnohem víc nebezpečné a dát za to jeden velký tlustý vykřičník. Pomocí `eval` totiž můžeme vyhodnocovat elementy při jejichž vyhodnocení dojde k *vedlejšímu efektu*, například ke změně vazby nebo překrytí vazby v nějakém prostředí. Od toho okamžiku mohou začít některé procedury vykazovat „zvláštní chování“. Demonstrujme si vše na následujícím větším příkladu. Předpokládejme, že máme definovánu proceduru `aux-proc` následovně:

```
(define aux-proc
  (let ()
    (lambda (x)
      (+ x y))))
```

Procedura vznikla v prostředí, ve kterém nejsou žádné vazby, které vzniklo použitím `let` bez seznamu vazeb. Předkem tohoto prostředí je globální prostředí. Vyhodnocení následujícího výrazu pochopitelně končí chybou:

`(aux-proc 10)` ⟹ „CHYBA: Symbol `y` nemá vazbu.“

Vyhodnocením následujícího výrazu:

```
(eval '(define y 20)
      (procedure-environment aux-proc))
```

došlo ke vzniku vedlejšího efektu, jímž byla definice nové vazby symbolu `y`. Výraz způsobující tuto definici jsme ale nevyhodnotili v globálním prostředí, nýbrž v prostředí vzniku procedury `aux-proc`. Takže v globálním prostředí symbol `y` zůstává i nadále bez vazby, ale aplikace `aux-proc` již proběhne bez chyby:

```
y           ⟹  „Symbol y nemá vazbu“
(aux-proc 10)  ⟹  30
```

To co jsme teď provedli byl z programátorského hlediska „extrémně nečistý trik“ (slangově *hack*), kdy jsme lokálnímu prostředí procedury, které by za normálních podmínek nebylo z globálního prostředí nijak dosažitelné, „vnutili“ novou vazbu symbolu. Přitom tato vazba nadále z venčí není na první pohled vidět, globální prostředí je nezměněno. Podíváme-li se nyní na definici procedury `aux-proc`, nikde tam symbol `y` pochopitelně nevidíme. Externí pozorovatel, který by nevěděl o naší „černé magii“, by si myslel, že aplikace `aux-proc` bude končit chybou, stejně tak, jak to bylo v původním případě.

Při dalším předefinování `y` v prostředí vzniku procedury, by se `aux-proc` opět začala chovat jinak:

```
(eval '(define y 200)
      (procedure-environment aux-proc))
(aux-proc 10)  ⟹  210
```

Pokud to není nezbytně nutné, procedura `eval` by neměla být vůbec používána. Na druhou stranu, je-li její použití na místě a pokud může výrazně urychlit vývoj programu, pak jejímu použití nelze snad nic namítat. Proceduru bychom ale měli používat pokud možno tak, abychom vyhodnocováním výrazů co možná nejméně ovlivňovali činnost zbytku programu.

Podotkněme, že procedura `eval` je popsána v definici R^{5}RS jazyka Scheme, viz [R5RS]. V tomto reportu je popsána i verze `eval` se dvěma argumenty, ale pouze ve velmi omezené míře. Výše uvedená speciální forma `the-environment` a procedury `procedure-environment`, `environment-parent` a `environment->list` nejsou v R^{5}RS vůbec zahrnuty. Některé široce používané interprety jazyka Scheme ale podobnými procedurami skutečně disponují, například interpret Elk. V poslední lekci v této části učebního textu do paradigmat programování si ukážeme implementaci skutečného interpretu jazyka Scheme, ve kterém budeme mít všechny tyto speciální formy a procedury k dispozici.

Poznámka 6.6. Ve funkcionálních programovacích jazycích je procedura `eval`, nebo nějaký její ekvivalent, obvykle k dispozici. Totéž se nedá říct o jiných programovacích jazycích. V procedurálních jazycích se procedury provádějící evaluaci vyskytují jen minimálně. V těchto vyšších programovacích jazycích také v podstatě neplatí, že program a data jsou totéž (teoreticky to možná platí, ale prakticky nikoliv). Možnosti vyhodnocování jsou někdy mylně přičítány jen interpretům programovacích jazyků. Protipříkladem mohou být jazyk Common LISP, který je kompilovaný (i interpretovaný) a ve kterém je `eval` přítomen (prostředí zde ale není element prvního řádu).

Na závěr této sekce uvedeme několik příkladů použití procedury `apply` a speciální formy `eval` a odvozených procedur. Prvním z nich je přibližná rovnost vektorů. Vektory jsou representovány číselnými seznamy `v1` a `v2`. Argument `delta` je pak tolerance, ve které se může pohybovat rozdíl jednotlivých složek, aby ještě vektory byly uznány za rovné. To pomáhá předejít problémům se zaokrouhlovacími chybami, které vznikají při manipulaci s neexaktními čísly. Například na některých platformách může nastat situace, kde

```
(= (- 1.2 1) 0.2)  ⟹  #f.
```

Proceduru `vec-equal?` bychom mohli naimplementovat například takto:

```
(define vec-equal?
  (lambda (v1 v2 delta)
    (forall (lambda (x y)
              (< (abs (- x y)) delta))
            v1 v2)))
```

Teď uvádíme aplikace:

```
(vec-equal? '(0 1 3) '(0 1.2 3) 0.1)      ⟹  #f
(vec-equal? '(0 1 3) '(0 1.2 3.001) 0.3)  ⟹  #t
```

Asociativní pole je datová struktura, která se skládá z kolekce tak zvaných klíčů a kolekce hodnot. Každý klíč je přiřazen jedné hodnotě. Základní operace na této datové struktuře jsou: přiřazení hodnoty nějakému klíči (respektive změna stávajícího přiřazení), vyhledání hodnoty podle klíče a zrušení stávajícího přiřazení. My budeme asociativní pole reprezentovat seznamem přiřazení. Přiřazením přitom budeme rozumět pár, jehož první prvek je klíč a druhý prvek je přiřazená hodnota. Budeme implementovat první dvě zmíněné základní operace. Teď tedy přidání prvku do asociativního pole:

```
(define cons-assoc
  (lambda (key val assoc)
    (let ((cell (cons key val)))
      (if (exists (lambda (x) (equal? (car x) key))
                  assoc)
          (map (lambda (x)
                 (if (equal? (car x) key)
                     cell
                     x))
               assoc)
          (cons cell assoc)))))
```

Nejdříve jsme zjistili použitím procedury kvantifikátoru `exists`, jestli je k zadanému klíči `key` přiřazena hodnota. Jestliže ne, přidáme pár reprezentující přiřazení hodnoty ke klíči do asociativního pole pomocí

procedury `cons`. Pokud ale přiřazení k takovému klíči již v poli je, použijeme proceduru `map`, abychom nahradili původní přiřazení novým.

Druhou základní operací je selekce. Použitím procedury `filter` vybereme ty prvky seznamu reprezentující asociativní pole, které mají shodný klíč s požadovaným klíčem. Pokud výsledný je výsledný seznam prázdný, nebylo přiřazení nalezeno. V opačném případě vracíme hodnotu z tohoto přiřazení.

```
(define assoc-key
  (lambda (assoc key)
    (let ((found (filter (lambda (x)
                           (equal? (car x) key))
                         assoc)))
      (if (null? found)
          #f
          (cdar found)))))
```

Další příklad se týká opět datových tabulek. Je vlastně vylepšení reprezentace datových tabulek, kterou jsme zavedli v sekci 5.8, tak, aby každá tabulka měla „hlavičku“, ve které jsou jména atributů, které budeme používat při selekci a projekci. Toto je příklad takové tabulky:

```
(define mesta
  '((jmeno p-obyvatel p-nadrazi velikost)
    (Olomouc 120 3 stredni)
    (Prostejov 45 2 male)
    (Prerov 50 3 male)
    (Praha 1200 8 velke)))
```

Procedura selekce bude brát jako argumenty tabulku a výraz reprezentující podmínku. V tomto výrazu používáme jména z hlavičky tabulku. Příklad použití může být třeba tento:

```
(selection mesta
           '(and (>= p-obyvatel 50)
                 (not (equal? velikost 'male))))
```

Nejdůležitější rys použitý v této procedure je vytvoření λ-výrazu vyhodnocením výrazu

```
(list 'lambda head condition)
```

a jeho následné vyhodnocení procedurou `eval`. Tato procedura a tělo tabulky – to jest tabulka bez hlavičky – pak budou vstupními argumenty pro proceduru `filter`, která pak provede vyfiltrování požadovaných řádků.

```
(define selection
  (lambda (table condition)
    (let* ((head (car table))
           (body (cdr table))
           (property (eval (list 'lambda head condition))))
      (filter (lambda (x)
                (apply property x))
              body))))
```

Toto naše řešení má ale mouchu. A to právě v použití procedury `eval`. Jde o to, že tato procedura, je-li použita s jedním argumentem, vyhodnocuje tento svůj argument v globálním prostředí. A z toho důvodu nebude fungovat následující kód, kde se ve výrazu, který reprezentuje podmínku odvoláváme na lokálně vázaný symbol.

```
(let ((x 50))
  (selection mesta
             '(and (>= p-obyvatel x)
                   (not (equal? velikost 'male))))) ⟹ „CHYBA: Symbol x nemá vazbu.“
```

Tento problém bychom samozřejmě mohli vyřešit použitím procedury `eval` se dvěma argumenty. Druhým argumentem bude aktuální prostředí, to musíme předat jako další argument při spuštění selekce:

```
(define selection
  (lambda (table env condition)
    (let* ((head (car table))
           (body (cdr table))
           (property (eval (list 'lambda head condition) env)))
      (filter (lambda (x)
                (apply property x))
              body))))
```

Viz příklad použití:

```
(let ((x 50))
  (selection mesta
             (the-environment)
             '(and (>= p-obyvatel x)
                   (not (equal? velikost 'male)))))
```

V jednom z dalších dílů textu si ukážeme mnohem elegantnější řešení tohoto problému.

6.5 Reprezentace množin a relací pomocí seznamů

V této sekci ukážeme reprezentaci konečných množin a relací pomocí seznamů, a procedur pro manipulaci s nimi. Na nich demonstrujeme použití procedur `apply`, `eval` a procedur implementovaných v této sekci. Také na nich ukážeme filtraci a vytváření procedur s nepovinnými argumenty a s libovolným počtem argumentů.

Za množinu budeme považovat seznam bez duplicit. V tomto seznamu pro nás bude důležitý pouze výčet prvků, nikoli jejich pořadí. Prázdná množina je reprezentovaná prázdným seznamem:

```
(define the-empty-set '())
```

Počet prvků množiny (kardinalita) se bude shodovat s délkou seznamu:

```
(define card
  (lambda (set)
    (length set)))
```

Procedura `make-set` vytváří množinu výběrem těch prvků z množiny ⟨*universe*⟩, které mají vlastnost ⟨*prop?*⟩. Jedna se o jednoduché použití procedury `filter`:

```
(define make-set
  (lambda (prop? universe)
    (filter prop? universe)))
```

Přidání prvku do množiny. Potřebujeme nejdříve zkontrolovat, jestli přidávaný prvek už v množině není. V případě, že ne, přidáme prvek. Jinak je výsledkem původní množina. Tak se zabrání možnému vzniku duplicit.

```
(define cons-set
  (lambda (x set)
    (if (in? x set)
        set
        (cons x set))))
```

V proceduře tedy potřebujeme predikát `in?`, který by rozhodoval, zda je daný element prvkem v seznamu. To zjistíme tak, že vyfiltrujeme prvky, které jsou rovny (při porovnání pomocí `equal?`), a otestujeme, jestli je výsledný seznam neprázdný.

```
(define in?
  (lambda (x set)
    (not (null? (filter (lambda (y) (equal? x y)) set)))))
```

Též bychom mohli použít proceduru `exists` a s její pomocí zjišťovat, zda je alespoň jeden prvek množiny roven danému elementu. Kód by pak vypadal následně:

```
(define in?
  (lambda (x set)
    (exists (lambda (p) (equal? x p)) set)))
```

Další procedura `set?` bude testovat, jestli je její argument množina. Tedy ověří, že se jedná o seznam a pak pro každý prvek tohoto seznamu zjistíme, jestli je počet jeho výskytů roven jedné. Ke zjištění počtu výskytů je využita procedura `occurrences`, kterou napíšeme vzápětí.

```
(define set?
  (lambda (elem)
    (and (list? elem)
         (forall (lambda (x)
                   (= (occurrences x elem) 1))
                 elem))))
```

Teď tedy k pomocné proceduře `occurrences`, která se používá se dvěma argumenty – elementem $\langle elem \rangle$ a seznamem $\langle l \rangle$. Tato procedura vrací počet výskytů prvku $\langle elem \rangle$ v seznamu $\langle l \rangle$. Můžeme ji realizovat například tak, že vyfiltrujeme ze seznamu $\langle l \rangle$ všechny prvky, které jsou shodné s elementem $\langle elem \rangle$. Kód takové implementace by vypadal takto:

```
(define occurrences
  (lambda (elem l)
    (length (filter (lambda (x) (equal? x elem)) l))))
```

Jinou možností je vytvořit aplikací procedurou `map` zaměnit prvky shodné s elementem $\langle elem \rangle$ za jedničky a ostatní za nuly. Na takto vzniklý seznam pak aplikujeme proceduru sčítání čísel. Než se začneme zabývat množinovými operacemi, vytvoříme si dvě pomocné procedury `map-index` a `map-tail`. Jedná se o obdoby procedury `map` (pro jeden seznam). Procedura `map-index` předává vstupní proceduře nejen prvky zadaného seznamu, ale také jejich indexy. Mapovaná procedura tedy bude přijímat dva argumenty – prvním z nich bude prvek z původního seznamu, druhým index tohoto prvku.

```
(define map-index
  (lambda (f l)
    (let ((indices (build-list (length l) (lambda (i) i))))
      (map f l indices))))
```

Procedura `map-index` je naprogramována tak, že vytvoří seznam indexů pomocí `build-list`, podobně jako v programu 6.4, to jest jako v implementaci procedury `list-ref`. Poté namapujeme předanou proceduru na původní seznam a seznam indexů. Viz příklady použití:

```
(map-index cons '(a b c d))                ⟹  ((a . 0) (b . 1) (c . 2) (d . 3))
(map-index (lambda (x y) x) '(a b c d))    ⟹  (a b c d)
(map-index (lambda (x y) y) '(a b c d))    ⟹  (0 1 2 3)
```

Mapovací procedura `map-tail` bude předávat mapované proceduře místo jednotlivých prvků podseznamy. Místo prvního prvku, bude předán celý seznam, namísto druhého seznam bez prvního prvku, a tak dále. Až konečně namísto posledního prvku seznamu bude předán jednoprvkový seznam obsahující poslední prvek.

Tyto podseznamy budeme získávat pomocí procedury `list-tail`. Ta se používá se dvěma argumenty – – prvním je seznam, a druhým číslo $\langle i \rangle$. Výsledkem aplikace je pak seznam bez prvních $\langle i \rangle$ prvků. Viz příklady požití:

```
(list-tail '(1 2 3 4 5) 1)  ⟹  (2 3 4 5)
(list-tail '(1 2 3 4 5) 3)  ⟹  (4 5)
(list-tail '(1 2 3 4 5) 5)  ⟹  ()
(list-tail '(1 2 3 4 5) 7)  ⟹  „CHYBA: Seznam má příliš malý počet prvků.“
```

Pomocí této procedury a `map-index` je implementace procedury `map-tail` velmi přímočará:

```
(define map-tail
  (lambda (f l)
    (map-index (lambda (x i)
                 (f (list-tail l i)))
               l)))
```

Viz příklad použití procedury:

```
(map-tail (lambda (x) x) '(a b c d))  ⟹  ((a b c d) (b c d) (c d) (d))
```

Nyní uděláme proceduru `list->set`, která bude konvertovat seznam na množinu. Tento seznam bude jejím jediným argumentem a procedura z něj odstraní duplicitní prvky.

```
(define list->set
  (lambda (l)
    (apply append
           (map-tail (lambda (x)
                       (let ((head (car x))
                             (tail (cdr x)))
                         (if (in? head tail)
                             '()
                             (list head))))
                     l))))
```

Nyní se zaměříme na operace s množinami – na sjednocení množin (`union`), průnik množin (`intersect`) a množinový rozdíl. Implementace procedury pro sjednocení množin je velice jednoduchá. Množiny, tedy seznamy, spojíme aplikací procedury `append`. V takto vzniklém seznamu se ale mohou vyskytovat duplicitní prvky – ty odstraníme použitím `list->set`.

```
(define union
  (lambda (set-A set-B)
    (list->set (append set-A set-B))))
```

Průnik množin `intersect` bude pro nás aplikací `make-set`. Universem bude sjednocení množin vytvořené pomocí procedury `union`, požadovanou vlastností pak bude přítomnost prvku (zjištěná predikátem `in?`) v obou množinách:

```
(define intersect
  (lambda (set-A set-B)
    (make-set (lambda (x)
                (and (in? x set-A)
                     (in? x set-B)))
              (union set-A set-B))))
```

Tyto dvě uvedené množinové operace můžeme sjednotit do procedury vyššího řádu `set-operation`:

```
(define set-operation
  (lambda (prop)
    (lambda (set-A set-B)
      (filter (lambda (x) (prop x set-A set-B))
              (list->set (append set-A set-B))))))
```

Pomocí `set-operation` můžeme definovat operace sjednocení a průniku množin.

```
(define union (set-operation (lambda (x A B) (or (in? x A) (in? x B)))))
(define intersect (set-operation (lambda (x A B) (and (in? x A) (in? x B)))))
```

Nebo třeba operaci množinového rozdílu:

```
(define minus (set-operation (lambda (x A B) (and (in? x A) (not (in? x B))))))
```

```
(union '(10 20 30) '(20 30 40))      ⟼  (10 20 30 40)
(intersect '(10 20 30) '(20 30 40))  ⟼  (20 30)
(minus '(10 20 30) '(20 30 40))      ⟼  (10)
```

Dále předchozí reprezentace množin použijeme k reprezentaci binárních relací na množinách. Připomeňme, že binární relace na množině je podmnožinou druhé kartézské mocniny této množiny. Kartézská mocnina množiny A je množina uspořádaných dvojic $\langle a, b\rangle$ takových, že první prvek a i druhý prvek b patří do množiny A. My budeme uspořádanou dvojici reprezentovat tečkovým párem (což se přímo nabízí). Vytvoříme si proto separátní konstruktory a selektory pro uspořádanou dvojici:

```
(define make-tuple cons)
(define 1st car)
(define 2nd cdr)
```

Procedura `cartesian-square` bude pro množinu vracet její druhou kartézskou mocninu.

```
(define cartesian-square
  (lambda (set)
    (apply append
           (map (lambda (x)
                  (map (lambda (y)
                         (make-tuple x y))
                       set))
                set))))
```

Výraz `(map (lambda (y) (make-tuple x y)) set)` vytvoří seznam všech dvojic, jejichž prvním prvkem je hodnota navázaná na symbol `x` a durhým prvkem je prvek z množiny `set`. Vnějším použitím procedury `map`

```
(map (lambda (x)
  (map (lambda (y)
         (make-tuple x y))
       set))
     set)
```

dostáváme seznam obsahující pro každé x z množiny `set` seznam všech párů kódujících uspořádané dvojice $\langle x, y\rangle$, kde y patří do množiny `set`. A tyto seznamy spojíme použitím procedury `append`.

Nyní můžeme přikročit k definici konstruktoru relace. Procedura `make-relation` bude procedurou dvou argumentů; prvním z nich bude vlastnost reprezentovaná predikátem dvou argumentů a druhým množina nad kterou bude relace definovaná.

```
(define make-relation
  (lambda (prop? universe)
    (filter (lambda (x)
              (prop? (1st x) (2st x)))
            (cartesian-square universe))))
```

Vytvořili jsme tedy druhou kartézskou mocniny množiny `universe` a z ní jsme aplikaci procedury `filter` vybrali ty dvojice, které splňují vlastnost `prop?`. Následují příklady použití tohoto konstruktoru.

```
(define u '(0 1 2 3 4 5))
(make-relation (lambda (x y) #f) u)
  ⟼  ()
```

```
(make-relation (lambda (x y) (= x y)) u)
  ⟼  ((0 . 0) (1 . 1) (2 . 2) (3 . 3) (4 . 4) (5 . 5))
(make-relation (lambda (x y) (= (+ x 1) y)) u)
  ⟼  ((0 . 1) (1 . 2) (2 . 3) (3 . 4) (4 . 5))
(make-relation (lambda (x y) (= (modulo (+ x 1) (length u)) y)) u)
  ⟼  ((0 . 1) (1 . 2) (2 . 3) (3 . 4) (4 . 5) (5 . 0))
(make-relation (lambda (x y) (< x y)) u)
  ⟼  ((0 . 1) (0 . 2) (0 . 3) (0 . 4) (0 . 5) (1 . 2) (1 . 3) (1 . 4) (1 . 5) (2 . 3) (2 . 4)
       (2 . 5) (3 . 4) (3 . 5) (4 . 5))
```

Protože relace jsou speciální množiny, můžeme na ně aplikovat procedury, které jsme výše vytvářeli pro množiny bez jakýchkoli změn:

```
(define r1 (make-relation (lambda (x y) (= (modulo (+ x 1) (length u)) y)) u))
(define r2 (make-relation (lambda (x y) (< x y)) u))

(union r1 r2)
  ⟼  ((5 . 0) (0 . 1) (0 . 2) (0 . 3) (0 . 4) (0 . 5) (1 . 2) (1 . 3)
       (1 . 4) (1 . 5) (2 . 3) (2 . 4) (2 . 5) (3 . 4) (3 . 5) (4 . 5))
(intersect r1 r2)
  ⟼  ((0 . 1) (1 . 2) (2 . 3) (3 . 4) (4 . 5))
(minus r1 r2)
  ⟼  ((5 . 0))
(minus r2 r1)
  ⟼  ((0 . 2) (0 . 3) (0 . 4) (0 . 5) (1 . 3) (1 . 4) (1 . 5) (2 . 4) (2 . 5) (3 . 5))
```

Níže budeme potřebovat proceduru, která pro libovolné dvě relace vrátí jejich nejmenší společné univerzum. Použitím procedury `map` pro každou z těch dvou relací vytvoříme seznam všech prvků vyskytujících se v prvních prvcích dvojic a seznam všech prvků vyskytujících se v druhých prvcích dvojic. A tyto čtyři seznamy spojíme aplikací procedury `append`. Z výsledného seznamu pak vytvoříme množinu pomocí procedury `list->set`:

```
(define get-universe
  (lambda (rel-R rel-S)
    (list->set (append (map 1st rel-R) (map 2st rel-R)
                       (map 1st rel-S) (map 2nd rel-S)))))
```

Tuto proceduru můžeme zobecnit na proceduru libovolného počtu argumentů. `get-universe` tak bude vracet nejmenší společné univerzum pro jakýkoli počet relací. Procedura, která vznikne vyhodnocením λ-výrazu

```
(lambda (x) (append (map 1st x) (map 2nd x))),
```

vrací seznam prvků vyskytujících se v prvních prvcích dvojic v relaci, na kterou je aplikována. Tuto proceduru mapujeme na seznam relací `relations`. Na seznam seznamů, který bude výsledkem tohoto mapování, aplikujeme proceduru `append` a vytvoříme tak jeden seznam. Z toho vytvoříme množinu procedurou `list->set`, viz následující kód:

```
(define get-universe
  (lambda relations
    (list->set (apply append
                      (map (lambda (x)
                             (append (map 1st x) (map 2nd x)))
                           relations)))))
```

Zde uvádíme aplikace procedury `get-universe` na různý počet relací:

```
(get-universe)                                  ⟼  ()
(get-universe '())                              ⟼  ()
(get-universe '() '() '())                      ⟼  ()
```

```
(get-universe '() '((a . b)) '())                          ⟹  (a b)
(get-universe '((10 . 20) (20 . 30)) '((a . b)) '())       ⟹  (10 20 30 a b)
```

K relaci R můžeme uvažovat inverzní relaci R^{-1}, to jest relaci $\{\langle b, a\rangle \mid \langle a, b\rangle \in R\}$. Vytvoříme proceduru `invert-relation`, která k dané relaci vrací inverzní relaci:

```
(define invert-relation
  (lambda (rel-R)
    (map (lambda (x)
           (make-tuple (2nd x) (1st x)))
         rel-R)))
```

Použitím procedury `map` jsme na každý prvek seznamu, který reprezentuje relaci aplikovali proceduru, která převrací pořadí prvků v uspořádané dvojici. Následuje ukázka použití procedury `invert-relation`.

```
(invert-relation '())  ⟹  ()
(invert-relation r1)   ⟹  ((1 . 0) (2 . 1) (3 . 2) (4 . 3) (5 . 4) (0 . 5))
```

Relace lze také skládat. Máme-li binární relace R a S na množině M, jejich složením rozumíme takovou binární relaci $R \circ S$ na množině M, že platí $\langle x, y\rangle \in R \circ S$, právě když existuje prvek $z \in M$ tak, že máme: $\langle x, z\rangle \in R$ a $\langle z, y\rangle \in S$. Procedura `compose-relations` bere dva argumenty, kterými jsou relace, a vrací relaci jejich složení.

```
(define compose-relations
  (lambda (rel-R rel-S)
    (let ((universe (get-universe rel-R rel-S)))
      (make-relation
       (lambda (x y)
         (exists (lambda (z)
                   (and (in? (make-tuple x z) rel-R)
                        (in? (make-tuple z y) rel-R)))
                 universe))
       universe))))
```

Implementace této procedury je vlastně přímým přepisem uvedeného předpisu. Nejdříve jsme pomocí procedury `get-universe` získali univerzum M a nad tímto univerzem jsme vytvořili relaci aplikací procedury `make-relation`. Vstupním argumentem této procedury byl mimo univerza predikát realizující vlastnost „existuje $z \in M$: $\langle x, z\rangle \in R$ a $\langle z, y\rangle \in S$“. V něm jsme použili proceduru existenčního kvantifikátoru `exists` a predikát `in?`. Nyní tedy ukážeme aplikaci této procedury:

```
(compose-relations r1 r2)
   ⟹  ((0 . 2) (1 . 3) (2 . 4) (3 . 5) (4 . 0) (5 . 1))
(compose-relations r2 r1)
   ⟹  ((1 . 3) (1 . 4) (1 . 5) (2 . 4) (2 . 5) (3 . 5) (0 . 2) (0 . 3) (0 . 4) (0 . 5))
```

O binární relaci R na množině M říkáme, že je *reflexivní*, pokud pro každé $x \in M$ platí $\langle x, x\rangle \in R$. Predikát `reflexive?`, který zjišťuje zda je relace reflexivní bychom mohli napsat například tak, jak je uvedeno níže. Opětně jde o přímý přepis předpisu.

```
(define reflexive?
  (lambda (rel-R)
    (forall (lambda (x)
              (in? (make-tuple x x) rel-R))
            (get-universe rel-R))))
```

A zde uvádíme aplikace:

```
(reflexive? (make-relation (lambda (x y) #f) u))                          ⟹  #t
(reflexive? (make-relation (lambda (x y) (= x y)) u))                     ⟹  #t
(reflexive? (make-relation (lambda (x y) (= (+ x 1) y)) u))               ⟹  #f
(reflexive? (make-relation (lambda (x y) (= (modulo x 2) (modulo y 2))) u))  ⟹  #t
(reflexive? (make-relation (lambda (x y) (< x y)) u))                     ⟹  #f
```

Binární relace R na množině M je *symetrická*, pokud pro všechna $x, y \in M$ platí, že pokud $\langle x, y\rangle \in R$, pak $\langle y, x\rangle \in R$:

```
(define symmetric?
  (lambda (rel-R)
    (let ((universe (get-universe rel-R)))
      (forall (lambda (x)
                (forall (lambda (y)
                          (if (in? (cons x y) rel-R)
                              (in? (cons y x) rel-R)
                              #t))
                        universe))
              universe))))
```

Zase šlo o přímý přepis uvedené definice. Zde uvádíme použití:

```
(symmetric? (make-relation (lambda (x y) #f) u))                          ⟼  #t
(symmetric? (make-relation (lambda (x y) (= x y)) u))                     ⟼  #t
(symmetric? (make-relation (lambda (x y) (= (+ x 1) y)) u))               ⟼  #f
(symmetric? (make-relation (lambda (x y) (= (modulo x 2) (modulo y 2))) u))  ⟼  #t
(symmetric? (make-relation (lambda (x y) (< x y)) u))                     ⟼  #f
```

Poslední vlastností relací jíž se budeme zabývat je tranzitivita. Binární relace R na množině M je tranzitivní, pokud pro všechna $x, y, z \in M$ platí, že pokud $\langle x, y\rangle \in R$ a $\langle y, z\rangle \in R$, pak $\langle x, z\rangle \in R$:

```
(define transitive?
  (lambda (rel-R)
    (let ((universe (get-universe rel-R)))
      (forall (lambda (x)
                (forall (lambda (y)
                          (forall (lambda (z)
                                    (if (and (in? (cons x y) rel-R)
                                             (in? (cons y z) rel-R))
                                        (in? (cons x z) rel-R)
                                        #t))
                                  universe))
                        universe))
              universe))))
```

Použití:

```
(transitive? (make-relation (lambda (x y) #f) u))                         ⟼  #t
(transitive? (make-relation (lambda (x y) (= x y)) u))                    ⟼  #t
(transitive? (make-relation (lambda (x y) (= (+ x 1) y)) u))              ⟼  #f
(transitive? (make-relation (lambda (x y) (= (modulo x 2) (modulo y 2))) u))  ⟼  #t
(transitive? (make-relation (lambda (x y) (< x y)) u))                    ⟼  #t
```

Shrnutí

Zabývali jsme se explicitní aplikací procedur a vyhodnocováním elementů. Představili jsme proceduru `apply` pomocí niž je možné aplikovat procedury s danými argumenty. Explicitní aplikaci procedur lze použít k agregaci více elementů do jednoho výsledku pomocí aplikace procedury. Ukázali jsme, jak lze s pomocí `apply` naprogramovat některé často používané procedury. Typickou úlohou, kterou lze vyřešit pomocí `apply` je filtrace prvků seznamu. V další části jsme se zabývali problematikou vytváření uživatelsky definovatelných procedur s nepovinnými a s libovolnými argumenty. Provedli jsme rozšíření λ-výrazů, které již si ponecháme (rozšíření bylo definitivní). Dále jsme ukázali proceduru `eval` a další sadu speciálních forem a procedur pro manipulaci s prostředím. Uvedli jsme, že prostředí lze bez újmy chápat jako element

jazyka Scheme, dokonce jako element prvního řádu. Upozornili jsme na úskalí při používání `eval` související se vznikem těžko odhalitelných chyb v programech. Nakonec jsme ukázali reprezentaci množin a relací pomocí seznamů, na které jsme demonstrovali použití procedur `apply` a `eval`.

Pojmy k zapamatování

- implicitní aplikace procedury, explicitní aplikace procedury,
- implicitní vyhodnocení elementů, explicitní vyhodnocení elementů,
- filtrace,
- nepovinné argumenty, libovolné argumenty,
- prostředí jako element prvního řádu.

Nově představené prvky jazyka Scheme

- speciální forma `the-environment`,
- procedury `apply`, `eval`, `environment-parent`, `procedure-environment`, `environment->list`

Kontrolní otázky

1. *Jaký je rozdíl mezi implicitní a explicitní aplikací procedury?*
2. *Jaké argumenty může mít procedura* `apply` *a jaký mají význam?*
3. *V kterých případech je nutné použít* `apply`*?*
4. *Jak se zapisují formální argumenty procedur s nepovinnými argumenty?*
5. *Jak se zapisují formální argumenty procedur s libovolnými argumenty?*
6. *Jaké omezení platí při použití nepovinných argumentů?*
7. *Jaké znáte speciální formy a procedury pro práci s prostředím?*
8. *Jak se provádí explicitní vyhodnocování elementů?*

Cvičení

1. Naprogramujte proceduru na zpracovaní seznamu podle vzoru. Vzorem se myslí seznam o stejném počtu prvků, který obsahuje:
   - symbol `del`, pak je prvek na odpovídající pozici smazán
   - symbol `ins`, pak je prvek na odpovídající pozici ponechán
   - procedury, pak je prvek seznamu na odpovídající pozici nahrazen aplikací této procedury na tento prvek.

   Viz příklad volání:

   ```
   (format '(1 2 3 4 5) (list 'del even? even? 'ins (lambda (x) (+ x 1))))
      ⟹ (#t #f 4 5)
   ```
2. Naprogramujte proceduru vracející seznam mocnin čísla k (od 0-té až po $(n-1)$-té).
3. Naprogramujte proceduru konverze `binary->decimal`, které binární číslo, reprezentované číselným seznamem obsahujícím `0` a `1`, převede na číslo (Scheme-ovské).
4. Implementujte pro reprezentaci množin uvedenou v sekci 6.5:
   - operaci symetrického rozdílu
   - predikát rovnosti množin
   - predikát zjišťující, zda je zadaný seznam množinou

Úkoly k textu

1. Naprogramujte relace reprezentované páry $\langle$*univerzum*$\rangle$. $\langle$*vlastnost*$\rangle$. Kde $\langle$*univerzum*$\rangle$ seznam, a $\langle$*vlastnost*$\rangle$ je predikát představující funkci příslušnosti.
2. Implementujte reprezentace zobrazení jako speciálních relací.
3. Naprogramujte predikáty `antisymetric?`, `ireflexive?` a `complete?` pro zjištění antisymetrie, ireflexivity a úplnosti relace.

   Relace R je

   - antisymetrická, pokud pro každé x a y platí, že pokud $(x, y) \in R$ a $(y, x) \in R$, pak $x = y$.
   - ireflexivní, pokud pro každé x platí $(x, x) \notin R$
   - úplná, pokud pro každé x a y platí, $(x, y) \in R$ nebo $(y, x) \in R$.

Řešení ke cvičením

1.
```
(define format
  (lambda (l pattern)
    (apply append
           (map (lambda (atom pat)
                  (cond ((equal? pat 'del) '())
                        ((equal? pat 'ins) (list atom))
                        ((procedure? pat) (list (pat atom)))
                        (else #f)))
                l pattern))))

(format '(1 2 3 4 5) (list 'del even? even? 'ins (lambda (x) (+ x 1))))
```
2.
```
(define (power-list k n)
  (apply map *
         (map
          (lambda (r)
            (build-list n (lambda (i)
                            (if (<= i r)
                                1
                                k))))
          (build-list n (lambda (i) i)))))
```
3.
```
(define (binary->decimal bin)
  (apply + (map * (reverse bin) (power-list 2 (length bin)))))
```
4.
```
(define sminus
  (set-operation
   (lambda (x A B)
     (or (and (in? x A) (not (in? x B)))
         (and (in? x B) (not (in? x A)))))))

(define sminus
  (lambda (set-A set-B)
    (union (minus set-A set-B)
           (minus set-B set-A))))
```
- ```
  (define set-equal?
    (lambda (set-A set-B)
      (null? (sminus set-A set-B))))
  ```
  ```

- ```
  (define set?
    (lambda (elem)
      (apply and-proc
             (map-tail (lambda (x)
                         (not (in? (car x) (cdr x))))
                       elem))))

  (define set?
    (lambda (elem)
      (equal? (list->set elem) elem)))
  ```
  ```

Reference

[SICP] Abelson H., Sussman G. J.: *Structure and Interpretation of Computer Programs.*
http://mitpress.mit.edu/sicp/full-text/book/book.html
The MIT Press, Cambridge, Massachusetts, 2nd edition, 1986. ISBN 0–262–01153–0.

[BW88] Bird R., Wadler P.: *Introduction to Functional Programming.*
Prentice Hall, Englewood Cliffs, New Jersey, 1988. ISBN 0–13–484197–2.

[Ch36] Church A.: An unsolvable problem of elementary number theory.
American Journal of Mathematics **58**(1936), 345–363.

[Ch41] Church A.: *The Calculi of Lambda Conversion.*
Princeton University Press, Princeton (NJ, USA), 1941.

[Di68] Dijkstra E. W.: Go To Statement Considered Harmful.
Communications of the ACM **11**(3)(1968), 147–148.

[Dy96] Dybvig R. K.: *The Scheme Programming Language.*
http://www.scheme.com/tspl3/
The MIT Press, Cambridge, Massachusetts, 3rd edition, 2003. ISBN 0–262–54148–3.

[F3K01] Felleisen M., Findler R. B., Flatt M., Krishnamurthi S.:
How to Design Programs: An Introduction to Computing and Programming.
http://www.htdp.org/
The MIT Press, Cambridge, Massachusetts, 2001.

[FF96a] Friedman D. P., Felleisen M.: *The Little Schemer.*
MIT Press, 4th edition, 1996.

[FF96b] Friedman D. P., Felleisen M.: *The Seasoned Schemer.*
MIT Press, 1996.

[Gi97] Giloi W. K.: Konrad Zuse's Plankalkül: The First High-Level "non von Neumann" Programming Language. *IEEE Annals of the History of Computing* **19**(2)1997, 17–24.

[Ho01] Hopcroft J. E. a kol.: *Introduction to Automata Theory, Languages, and Computation,*
Addison-Wesley, 2001.

[R5RS] Kelsey R., Clinger W., Rees J. (editoři):
Revised^5 Report on the Algorithmic Language Scheme,
http://www.schemers.org/Documents/Standards/R5RS/
Higher-Order and Symbolic Computation **11**(1)1998, 7–105;
ACM SIGPLAN Notices **33**(9)1998, 26–76.

[KV08] Klir G. J., Vysocký P.: *Počítače z Loretánského náměstí: Život a dílo Antonína Svobody.*
Nakladatelství ČVUT, Praha, 2008. ISBN 978–80–01–03953–3.

[Ko97] Kozen D. C.: *Automata and Computability,*
Springer, 1997

[Kr06] Krishnamurthi S.: *Programming Languages: Application and Interpretation.*
http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/

[Ll87] Lloyd, J. W.: *Foundations of Logic Programming*.
Springer-Verlag, New York, 2nd edition, 1987.

[ML95] Manis V. S., Little J. J.: *The Schematics of Computation.*
Prentice Hall, Englewood Cliffs, New Jersey, 1995. ISBN 0–13–834284–9.

[MC60] McCarthy J.: Recursive Functions of Symbolic Expressions and Their Computation by Machine.
Communications of the ACM **3**(4)1960, 184–195.

[MC67] McCarthy J.: A Basis for a Mathematical Theory of Computation.
Obsaženo ve: Braffort P., Hirschberg D. (editoři), *Computer Programming and Formal Systems.*
North-Holland, 1967.

[vN46] von Neumann J.: The principles of large-scale computing machines.
Ann. Hist. Comp **3**(3)1946.

[NS97] Nerode A., Shore A. R.: *Logic for Applications.*
Springer-Verlag, New York, 2nd edition, 1997.

[NM00] Nilson U., Maluszynski J.: *Logic, programming and Prolog.*
http://www.ida.liu.se/~ulfni/lpp/

[PeSa58] Perlis A. J., Samelson K.: Preliminary Report – International Algorithmic Language.
Communications of the ACM **1**(12)1958, 8–22.

[Pe81] Perlis A. J.: The American side of the development of ALGOL.
Obsaženo ve: Wexelblat R. L. (editor): *History of Programming Languages.* Academic Press, 1981.

[Qu96] Queinnec C.: *Lisp in Small Pieces.*
Cambridge University Press, 1996.

[Ro63] Robinson J. A.: Theorem-Proving on the Computer.
Journal of the ACM **10**(2)(1963), 163–174.

[Ro65] Robinson J. A.: A Machine-Oriented Logic Based on the Resolution Principle.
Journal of the ACM **12**(1)(1965), 24–41.

[Ro00] Rojas R. a kol.: *Plankalkül: The First High-Level Programming Language and its Implementation*.
http://www.zib.de/zuse/Inhalt/Programme/Plankalkuel/Plankalkuel-Report/Plankalkuel-Report.htm
Institut für Informatik, Freie Universität Berlin, Technical Report B-3/2000.

[SS86] Shapiro E., Sterling S.: *The Art of Prolog.*
The MIT Press, Cambridge, Massachusetts, 1986.

[Si04] Sitaram D.: *Teach Yourself Scheme in Fixnum Days.*
http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html
text je autorem postupně aktualizován, 1998–2004.

[R6RS] Sperber M., Dybvig R., Flatt M., Van Straaten A., Findler R., Matthews J.:
Revised^6 Report on the Algorithmic Language Scheme,
http://www.r6rs.org/
Journal of Functional Programming **19**(S1)2009, 1–301.

[SF94] Springer G., Friedman D. P.: *Scheme and the Art of Programming.*
The MIT Press, Cambridge, Massachusetts, 1994. ISBN 0–262–19288–8.

[St76] Steele G. L.: *LAMBDA: The Ultimate Declarative*
http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-379.pdf
AI Memo **379**, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1976.

[SS76] Steele G. L., Sussman G. J.: *LAMBDA: The Ultimate Imperative*
http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-353.pdf
AI Memo **353**, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1976.

[SS78] Steele G. L., Sussman G. J.: *The Revised Report on SCHEME: A Dialect of LISP.*
http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-452.pdf
AI Memo **452**, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1978.

[SS75] Sussman G. J., Steele G. L.: *SCHEME: An Interpreter for Extended Lambda Calculus.*
http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-349.pdf
AI Memo **349**, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1975.

[Wec92] Wechler W.: *Universal Algebra for Computer Scientists.*
Springer-Verlag, Berlin Heidelberg, 1992.

[Zu43] Zuse K.: *Ansätze einer Theorie des allgemeinen Rechnens unter besonderer Berücksichtigung des Aussagenkalküls und dessen Anwendung auf Relaisschaltungen*.
Nepublikovaný rukopis, Zuse Papers 045/018, 1943.

[Zu72] Zuse K.: Der Plankalkül.
http://www.zib.de/zuse/English_Version/Inhalt/Texte/Chrono/40er/Pdf/0233.pdf
Berichte der Gesellschaft für Mathematik und Datenverarbeitung. Nr. 63, Sankt Augustin, 1972.

A Seznam vybraných programů

B Seznam obrázků