KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

CVIČENÍ Z PARADIGMAT PROGRAMOVÁNÍ I

Lekce 2: Vytváření abstrakcí pomocí procedur

Učební materiál k přednášce 12. října 2006
(pracovní verze textu určená pro studenty)

JAN KONEČNÝ, VILÉM VYCHODIL

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

Olomouc 2006

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 vznikaní 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, 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)` $\longmapsto$ 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

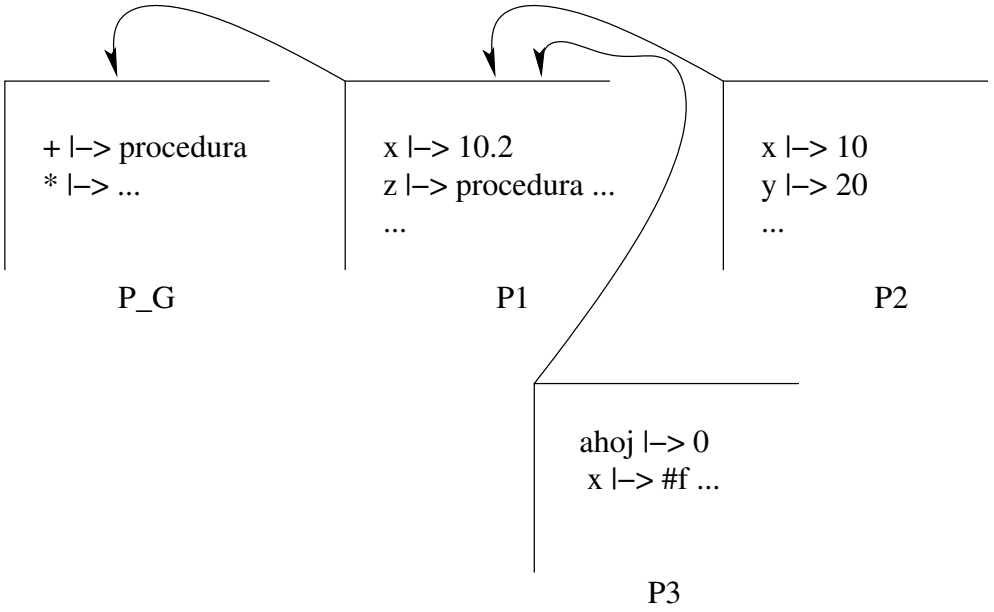
V předchozí sekci jsme nastínili problém potřeby pracovat s více prostředími. Jedno prostředí je počáteční prostředí, ve kterém jsou definovány počáteční vazby symbolů. Dalšími prostředími jsou lokální prostředí procedur, ve kterých jsou vázány na formální argumenty hodnoty se kterými byly procedury 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 (NACRT OBRAZKU)



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 $\mathrm{Eval}[E] := \mathrm{Apply}[F_1, E_2, \ldots, E_n]$.

(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 32, 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 je 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 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 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 $\langle \textit{parametry} \rangle$ je *seznam formálních argumentů* (to jest seznam po dvou různých symbolů), $\langle \textit{tělo} \rangle$ 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

$$\texttt{(lambda (}\langle \textit{param}_1 \rangle\ \langle \textit{param}_2 \rangle \cdots \langle \textit{param}_n \rangle\texttt{)}\ \langle \textit{tělo} \rangle\texttt{)}$$

v prostředí $\mathcal{P}$ vznikne procedura $\langle \texttt{(}\langle \textit{param}_1 \rangle\ \langle \textit{param}_2 \rangle \cdots \langle \textit{param}_n \rangle\texttt{)}, \langle \textit{tělo} \rangle, \mathcal{P} \rangle$. ■

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 $\langle \texttt{(}\langle \textit{param}_1 \rangle \cdots\ \langle \textit{param}_m \rangle\texttt{)}, \langle \textit{tělo} \rangle, \mathcal{P} \rangle$, pak hodnotu $\mathrm{Apply}[E, E_1, \ldots, E_n]$, definujeme takto:

(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.

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))))
```

(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:

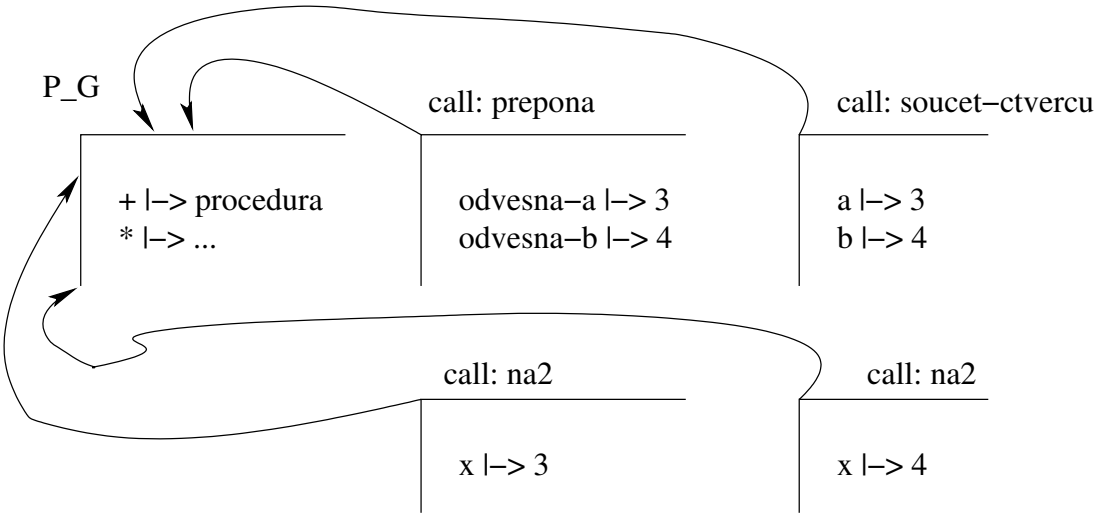
`((lambda (+) (- +)) 10)` $\quad\Longmapsto\quad$ `-10`,

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.

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

Obrázek 2.2. Vznik prostředí během aplikace procedur z programu 2.1 (NACRT OBRAZKU)



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 k 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` $\mapsto$ `3`) a `16` (v případě prostředí, kde `x` $\mapsto$ `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. ■

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. Formální

Program 2.2. Procedura provádějící infixovou aplikaci procedury dvou argumentů.

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

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 12. 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, zda-li 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+`:

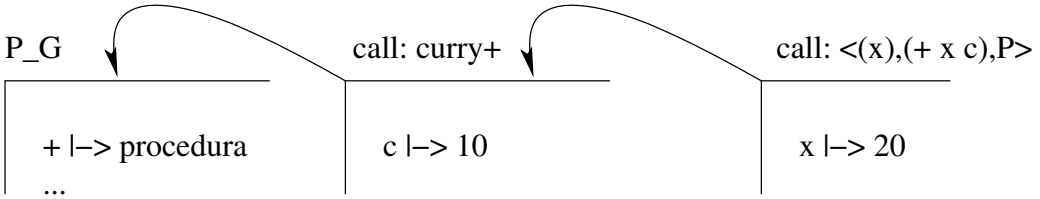
```
(define f (curry+ 10))
f                  ⟼  „procedura, která hodnotu svého argumentu přičte k hodnotě 10“
(f 20)             ⟼  30
```

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 (NACRT OBRAZKU)



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, 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 hodnotou `#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,
```

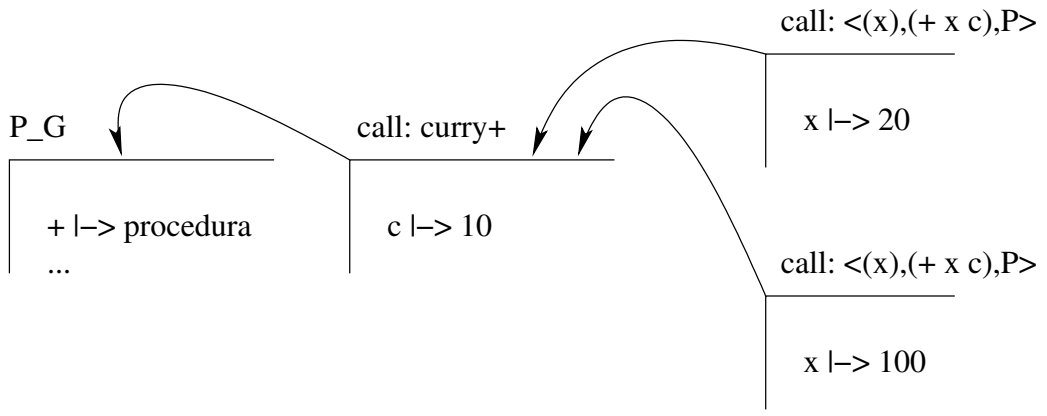
protože jsme symbol `+` v globálním prostředí redefinovali na projekci prvního argumentu ze dvou a tím je hodnota navázaná na symbol `x` (hodnota `#f` navázaná v předcházejícím prostředí na `c` tedy nebude hrát roli), viz program 2.3 a obrázek 2.3.

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 (NACRT OBRAZKU)



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[2]. 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.

Jak již bylo řečeno, vazby v daném prostředí překrývají definice vazeb v prostředích nadřazených. V našem příkladě z obrázku 2.4 to má ten (chtěný) efekt, že 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
```

[2]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.

V druhém kroku je zavedena vazba `c` $\mapsto_{\mathcal{P}_G}$ `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, zda-li 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 50 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

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

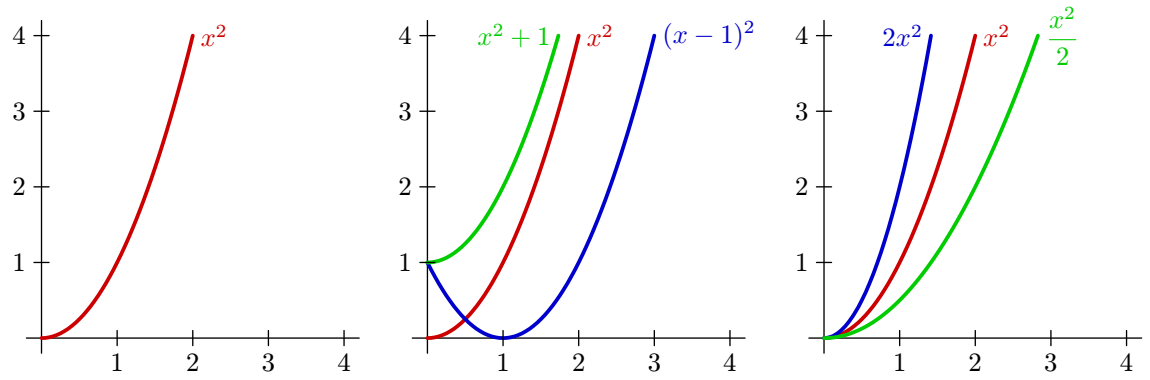
Funkce $f_{X,k}$ reprezentuje funkci f posunutou o k podél osy x a funkce $f_{Y,k}$ reprezentuje funkci f posunutou o k podél osy y. Na obrázku 2.5 (vlevo) je zobrazena část grafu funkce f dané předpisem $f(x) = x^2$ ($x \in \mathbb{R}$). Na témže obrázku uprostřed máme ukázky částí grafů funkcí $f_{X,1}$ 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“ běžně používané ve středoškolské matematice zajímavé, protože můžeme snadno naprogramovat procedury, které budou z procedur reprezentující tato zobrazení vytvářet 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

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)))))
```

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`“

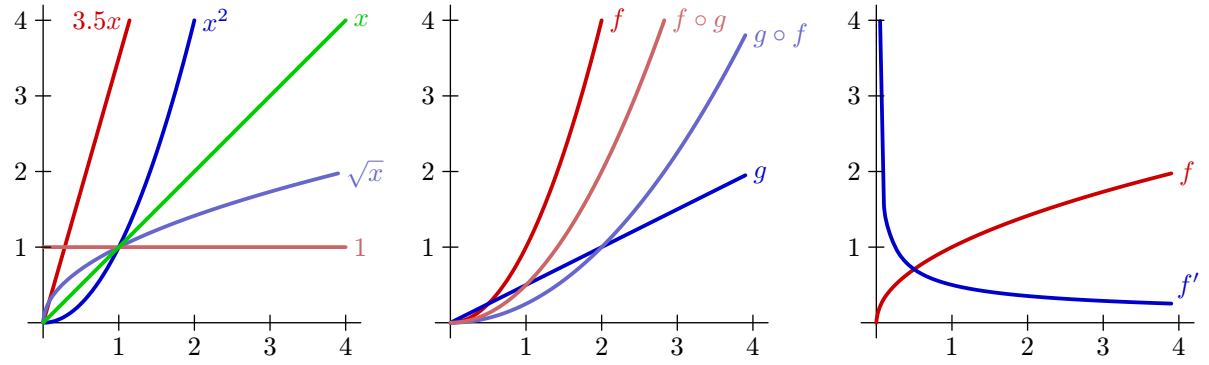
Takto vytvořené procedury bychom dále mohli třeba pojmenovat, nebo s nimi přímo pracovat:

```
((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
  ⋮
```

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

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



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

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^{5}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, 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 sami dokázat. 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).

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

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)))))
```

```
(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 sekce 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 52 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, zda-li 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.* Vezměme-li kód z příklady 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 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 49 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 47, v bodech (C.1) a (C.2) tak, aby se při každé aplikaci předávala informace o prostředí $\mathcal{P}_a$, ve kterém aplikaci provádíme. Tot' vše.

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 stále neplatí „zákon dvojí negace“ (dvojicí negací výchozí pravdivostní hodnoty získáme tutéž pravdivostní hodnotu), 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 $\wedge$):

$$\texttt{\#t} \wedge p = p \wedge \texttt{\#t} = p.$$

To jest výraz

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

je ekvivalentní výrazu:

$$p_1 \wedge p_2 \wedge \cdots \wedge p_n \wedge \texttt{\#t}.$$

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, zda-li 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*⟩.

Na závěr povídání o `and` si řekněme, 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, zda-li 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, zda-li 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 okamžimu, 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 33. ■

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, zda-li 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 (⟨test_1⟩ ⟨důsledek_1⟩)
      (⟨test_2⟩ ⟨důsledek_2⟩)
        ⋮
      (⟨test_n⟩ ⟨důsledek_n⟩)
      (else ⟨náhradník⟩)),
```

nahradit výrazem

```
(if ⟨test_1⟩
    ⟨důsledek_1⟩
    (if ⟨test_2⟩
        ⟨důsledek_2⟩
        (if ···
            (if ⟨test_n⟩
                ⟨důsledek_n⟩
                ⟨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, zda-li 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“[3]). 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,
- kompozice procedur, monoidální operace,
- nadřazené prostředí, lexikálně nadřazené prostředí, dynamicky nadřazené prostředí,

[3] 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 ze funkčnost svých výtvorů.

- 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 pracovat 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 59 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, zda-li 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.

4. Naprogramujte procedury vyšších řádů `rotate`, `x-reflect` a `y-reflect`, které budou sloužit k vytváření procedur reprezentující funkce jejichž grafy budou oproti výchozím funkcím otočené o daný úhel (ve stupních) v kladném smyslu, zrcadlené kolem osy x a zrcadlené kolem osy y. Využijte vhodně goniometrických funkcí dostupných v jazyku R^{5}RS Scheme, viz dokumentaci [R5RS].

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 -))
    ```