KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

CVIČENÍ Z PARADIGMAT PROGRAMOVÁNÍ I

Lekce 12: Čistě funkcionální interpret Scheme

Učební materiál k přednášce 21. prosince 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 12: Čistě funkcionální interpret Scheme

Obsah lekce: V této lekci se nejprve budeme zabývat automatickým přetypováním a generickými procedurami. Pro generické procedury zavedeme jednoduchou metodu jejich aplikace prostřednictvím vyhledávání procedur pomocí vzorů uvedených v tabulkách. Ve zbytku lekce ukážeme implementaci čistě funkcionální podmnožiny jazyka Scheme. Zaměříme se na implementaci datové reprezentace elementů jazyka pomocí manifestovaných typů a na implementaci vyhodnocovacího procesu.

Klíčová slova: generická procedura, koerce, manifestovaný typ, přetypování, tabulka generických procedur.

12.1 Automatické přetypování a generické procedury

V úvodní sekci této lekce uděláme malou odbočku od hlavního zaměření lekce jímž bude konstrukce interpretu jazyka Scheme. V této sekci se budeme zabývat *generickými procedurami.* Většina procedur, které jsme v jazyku Scheme používali, byly těsně svázané s konkrétním datovým typem. Například procedura `append` prováděla spojování seznamů a nebylo ji možné použít s argumenty jiných typů než jsou seznamy. Toto chování je v mnoha situacích žádoucí.

Někdy je ale potřeba jednoduše používat tutéž proceduru s argumenty různých datových typů. Například procedura pro sčítání `+` je schopna pracovat s čísly v přesné reprezentaci (racionální zlomky) a s čísly v přibližné reprezentaci (čísla s pohyblivou desetinnou tečkou). Proceduru sčítání je dokonce možné použít s argumenty různých typů současně, to jest s některými argumenty (čísly) v přesné reprezentaci a s některými argumenty (čísly) v přibližné reprezentaci. Procedura pro sčítání tedy musí provádět dílčí operace, které jsou podmíněné typem elementů. V některých případech tedy musí provést před samotným součtem jistou „konverzi datových typů“, aby mohla aritmetickou operaci provést.

Procedurám, které svou činnost řídí podle typů svých argumentů, říkáme *generické procedury.* Přesněji řečeno, za generické procedury považujeme ty procedury, které podle typů argumentů provádějí aplikace jiných procedur a provádějí případně dodatečnou konverzi argumentů (elementů) na elementy vyžadovaných typů. Například procedura sčítání v jazyku Scheme je generická procedura. Pokud je sčítání aplikováno s racionálními argumenty, provede se sčítání přímo v přesná reprezentaci a výsledkem je opět číslo v přesné reprezentaci. Pokud by byl byt'jen jeden z argumentů v přibližné reprezentaci, pak procedura provede nejprve konverzi všech argumentů do přibližné reprezentace a provede sečtení v přibližné reprezentaci. Výsledkem takového sečtení je číslo v přibližné reprezentaci. To by nás nemělo překvapit, protože při práci s interpretem jazyka Scheme jsme si mohli všimnout následujícího chování `+`:

```
(+ 1/2 10)      ⟼  21/2
(+ 0.5 10)      ⟼  10.5
(+ 1/2 2/3)     ⟼  7/6
(+ 0.5 0.666)   ⟼  1.166
```

Automatickým konverzím na elementy jiných typů, ke kterým může docházet během používání generických procedur, se říká *koerce.* Koerce je tedy obecně řečeno *implicitní přetypování.* Skoro ve všech programovacích jazycích jsou k dispozici nějaké prostředky pro *explicitní přetypování,* to jest programátorem vynucenou změnu typu elementu. Příkladem může být třeba *převod čísla na řetězec znaků.* V jazyku Scheme, jako ve většině programovacích jazyků, existuje datový typ „řetězec znaků“. S řetězci lze dělat běžné operace, jimiž se podrobně nebudeme zabývat, protože to není naším hlavním cílem (zájemce odkazuji na specifikaci R^{5}RS jazyka Scheme, viz [R5RS]). Jednou z operací s řetězci je jejich spojování. K tomu slouží procedura `string-append`. Spojování řetězců je demonstrováno následujícími příklady:

```
(string-append)                  ⟼  ""
(string-append "Ahoj" "svete")   ⟼  "Ahojsvete"
(string-append "Ahoj " "svete")  ⟼  "Ahoj svete"
(string-append "a" "b" "c")      ⟼  "abc"
```

Převod čísel na řetězce se provádí pomocí procedury `number->string`, které pro dané číslo vrací řetězec znaků obsahující *externí reprezentaci čísla.* Pro ilustraci viz následující ukázky použití procedury:

```
(number->string 10.2)          ⟹  "10.2"
(number->string (/ -1 2))      ⟹  "-1/2"
(number->string (sqrt -1))     ⟹  "0+1i"
```

Předchozí operaci bychom de facto mohli chápat jako *explicitní přetypování*. Na programátorovu žádost byl element číslo „převeden" na element řetězec, se kterým se dál může pracovat. Naproti tomu výše uvedené *implicitní přetypování* (*koerci*) provádějí automaticky některé (generické) procedury. Je zajímavé, že koerce je někdy chápána jako potenciální zdroj chyb a některé jazyky koerci vůbec neumožňují. Jedním z takových jazyků je například funkcionální jazyk ML.

Ve zbytku této sekce si ukážeme modelový příklad, jak lze naprogramovat uživatelsky definované generické procedury. Vyjdeme z nám dobře známé operace sčítání čísel a obohatíme ji tak, aby mohla sloužit i ke spojování řetězců. Umožníme navíc, abychom mohli tuto novou verzi generického sčítání používat s argumenty různých typů (tedy s čísly i s řetězci). Toto zobecnění sčítání je ve skutečnosti docela praktické. Umožňuje nám snadno vytvářet textový výstup, v němž jsou některé hodnoty dopočtené, viz ukázku:

```
(let ((x 10)
      (y 20))
  (+ "Soucin " x " a " y
     " je " (* x y) "."))  ⟹  "Soucin 10 a 20 je 200."
```

Jako první vytvoříme pomocný predikát `match-type?` sloužící k testování typů argumentů. Predikát je

Program 12.1. Porovnávání datového typu argumentu se vzorem.

```
(define match-type?
  (lambda (preds args)
    (or (and (null? preds) (null? args))
        (and (pair? preds)
             (pair? args)
             ((car preds) (car args))
             (match-type? (cdr preds) (cdr args))))))
```

uveden v programu 12.1. Prvním argumentem procedury `match-type?` je seznam predikátů a druhým argumentem je seznam elementů. Výsledek aplikace `match-type?` pro dané argumenty je „pravda", právě když jsou oba dva seznamy předané jako argumenty stejně dlouhé a elementy z druhého seznamu odpovídají po řadě typům daným predikáty v prvním seznamu. Použití `match-type?` je demonstrováno následujícím příkladem.

```
(match-type? `(,number? ,number?) '(10.2 "Ahoj"))  ⟹  #f
(match-type? `(,number? ,string?) '(10.2 "Ahoj"))  ⟹  #t
(match-type? `(,string? ,number?) '(10.2 "Ahoj"))  ⟹  #f
(match-type? `(,string? ,string?) '(10.2 "Ahoj"))  ⟹  #f
```

Predikát `match-type?` bude použit při testování shody argumentů se vzorem v tabulce určující chování generické procedury. Pro každou generickou proceduru budeme vždy uvažovat *tabulku metod*[17], což bude tabulka ve speciálním tvaru určujícím vzory a procedury pro aplikaci, pokud argumenty budou odpovídat danému vzoru.

Pokud se nyní pro ilustraci zaměříme pouze na dva argumenty, můžeme nově implementované generické sčítání popsat tabulkou uvedenou v programu 12.2. Po vyhodnocení výrazu z programu 12.2 dojde k navázání seznamu párů na symbol `table-generic`. Seznam se skládá z párů jejichž první prvek lze chápat jako identifikátor typu konkrétní operace a druhý prvek je samotná operace, která se má provést. Například první řádek bychom mohli číst tak, že v případě sčítání dvou čísel je použita původní operace „sčítání čísel"

[17] Pojem „metoda" je často používán v objektovém programování. My budeme za metody považovat procedury spolu se vzorem jejich použití, které budou součástí tabulky určující činnost generické procedury.

Program 12.2. Konkrétní tabulka metod generické procedury.

```
(define table-generic
  (let ((+ +))
    `(((+ ,number? ,number?) . ,+)
      ((+ ,number? ,string?) . ,(lambda (x y)
                                  (string-append (number->string x) y)))
      ((+ ,string? ,number?) . ,(lambda (x y)
                                  (string-append x (number->string y))))
      ((+ ,string? ,string?) . ,string-append))))
```

(všimněte si vazby `+` v `let`-výrazu, pro připomenutí viz lekci 3). Na druhém řádku tabulky je uvedeno, že v případě sčítání čísla a řetězce se provede konverze čísla na řetězec a výsledek se spojí s předaným řetězcem. Při aplikaci poslední jmenované procedury vlastně z hlediska uživatele generické procedury (kterou vytvoříme dále) dochází ke koerci (přetypování čísla na řetězec).

Program 12.3. Vyhledání příslušné operace v tabulce metod generické procedury.

```
(define table-lookup
  (lambda (table op args)
    (cond ((null? table) #f)
          ((not (equal? (caaar table) op)) (table-loopkup (cdr table) op args))
          ((match-type? (cdaar table) args) (cdar table))
          (else (table-lookup (cdr table) op args)))))
```

Procedura `table-lookup` uvedená v programu 12.3 má na starost vyhledávat příslušnou operaci v tabulce metod generické procedury. Procedura `table-lookup` bere jako argumenty tabulku metod generické procedury, symbol identifikující generickou proceduru (v jedné tabulce metod mohou být záznamy pro více generických operací), a seznam argumentů, podle kterých se v tabulce vyhledává. Procedura iterativně prochází tabulku a při nalezení první shody vzoru metody s argumenty (zde se používá predikát `match-type?`) je vrácena procedura jež je součástí metody.

```
(table-lookup table-generic '+ '(10 20))          ⟼  primitivní procedura „sčítání čísel“
(table-lookup table-generic '+ '(10 "svete"))     ⟼  uživ. def. procedura z 2. řádku tabulky
(table-lookup table-generic '+ '("Ahoj" 20))      ⟼  uživ. def. procedura z 3. řádku tabulky
(table-lookup table-generic '+ '("Ahoj" "svete")) ⟼  primitivní procedura „spojení řetězců“
(table-lookup table-generic '+ '(10 #t))          ⟼  #f
```

Aplikace generických procedur bude prováděna pomocí procedury `apply-generic`, viz program 12.4. Procedura `apply-generic` provede vyhledání metody v tabulce navázané na `table-generic` podle vzoru

Program 12.4. Aplikace generické procedury.

```
(define apply-generic
  (lambda (op . args)
    (let ((proc (table-lookup table-generic op args)))
      (if proc
          (apply proc args)
          (error "No method for these types")))))
```

a provede následnou aplikaci procedury jež je součástí shodující se metody. Pro větší pohodlí při aplikaci generické procedury provedeme navázání nově vytvořené procedury na symbol `+`. Kód provádějící tuto vazbu je uveden v programu 12.5. Pomocná procedura `foldr1` pracuje stejně jako `foldr` s jedním

Program 12.5. Generická procedura pro sčítání.

```
(define foldr1
  (lambda (f term l)
    (cond ((null? l) term)
          ((null? (cdr l)) (car l))
          (else (f (car l) (foldr1 f term (cdr l)))))))

(define +
  (lambda args
    (foldr1 (lambda (x y)
              (apply-generic '+ x y))
            0
            args)))
```

seznamem, pouze s tím rozdílem, že hodnota navázaná na symbol `term` je vrácena pouze v případě, že seznam předaný `foldr1` jako třetí argument je prázdný. V případě, že seznam je jednoprvkový, je vrácen tento prvek – to je jediný rozdíl oproti původní verzi `foldr`. V programu 12.5 jsme použili `foldr1` k naprogramování nové procedury libovolných argumentů, která je posléze navázána na symbol `+`. Procedura `foldr1` je zde použita k zobecnění aplikace generické procedury ze dvou argumentů na libovolné množství argumentů. Touto problematikou jsme se již zabývali v sekci 7.3. Novou generickou proceduru `+` je nyní možné používat následujícími způsoby:

```
(+)              ⟼  0
(+ 10)           ⟼  10
(+ 10 20)        ⟼  30
(+ 10 20 30)     ⟼  60
(+ 10 "x" 30)    ⟼  "10x30"
(+ 10 20 30 "")  ⟼  "102030"
(+ 10 20 "" 30)  ⟼  "102030"
(+ 10 "" 20 30)  ⟼  "1050"
(+ "" 10 20 30)  ⟼  "60"
```

Všimněte si, že pokud uvádíme jako argumenty pouze čísla, generická procedura `+` se chová stejně jako původní procedura sčítání. Zbývá odpovědět na otázku, proč jsme při vytvoření generické procedury `+` použili `foldr1` místo klasického `foldr`. Je to z důvodu praktičnosti. Při použití `foldr` místo `foldr1` bychom totiž dostali nepřirozený výsledek v situaci, kdy je poslední ze sčítaných elementů řetězec:

```
(+ "")                          ⟼  "0"
(+ "Ahoj " "svete")             ⟼  "Ahoj svete0"
(+ "Faktorial " 4 " je " 24 ".")  ⟼  "Faktorial 4 je 24.0"
```

Naproti tomu verze se `foldr1` se chová přirozeně:

```
(+ "")                          ⟼  ""
(+ "Ahoj " "svete")             ⟼  "Ahoj svete"
(+ "Faktorial " 4 " je " 24 ".")  ⟼  "Faktorial 4 je 24."
```

12.2 Systém manifestovaných typů

Jelikož nám v této lekci jde o konstrukci interpretu jazyka Scheme, jako jeden z prvních problémů musíme vyřešit, jak budeme *reprezentovat jednotlivé elementy jazyka*: čísla, pravdivostní hodnoty, symboly, páry,

procedury (primitivní a uživatelsky definované), speciální formy a další. V této sekci nejprve naznačíme obecnou reprezentaci elementů při níž užijeme tak zvanou *manifestaci typů*. Každý element se bude skládat ze dvou základních částí:

(i) *identifikátor typu elementu,*

(ii) *data charakterizující element.*

Identifikátor typu bude sloužit k jednoznačnému určení o jaký element se jedná. Pomocí tohoto identifikátoru tedy rozlišíme, zda-li například daný element reprezentuje pár nebo proceduru. Jako identifikátory typů budeme používat symboly jejichž jména budou typ elementu označovat. Druhou částí každého elementu jsou data představující „hodnotu elementu".

Poznámka 12.1. (a) Pokud si například představíme dvě čísla `-13` a `27`, pak jejich vnitřní reprezentace (v našem konstruovaném interpretu Scheme) budou obsahovat shodný identifikátor typu „číslo". Oba elementy budou mít ale různou datovou část – v prvním případě bude datová část uchovávat číselnou hodnotu -13, v druhém případě 27.

(b) Otázkou je, proč v elementech potřebujeme identifikátory jejich typů a zda-li bychom se bez nich mohli obejít. Identifikátory skutečně potřebujeme, abychom mohli správně rozlišovat datové typy elementů (a v důsledku abychom byli schopni správně vyhodnocovat elementy). V některých případech totiž pouze na základě znalosti „datové části elementu" nejsme schopni určit jeho typ. Vezměme si například tečkový pár `(2 . 3)`. Mohli bychom se na něj dívat jako na dvojici číselných hodnot. Ta může reprezentovat třeba zlomek $\frac{2}{3}$, nebo komplexní číslo $2 + 3i$. Z pohledu syntaxe a sémantiky jazyka, viz sekci 1.2, je identifikátor typu *sémantická informace* určující *význam datové části elementu* (datová část elementu by sama o sobě neměla žádný význam).

Pod pojmem *manifestovaný typ* máme tedy na mysli přítomnost identifikátoru typu „v datech". Při reprezentaci elementů budeme používat manifestaci typů, jak jsme to naznačili v úvodu této sekce.

Pro práci s elementy a manifestovanými typy si vytvoříme sadu pomocných procedur, které jsou uvedeny v programu 12.6. Procedura `curry-make-elem` slouží k vytváření *konstruktorů elementů* s manifestovaným

Program 12.6. Systém manifestovaných typů.

```
(define curry-make-elem
  (lambda (type-tag)
    (lambda (data)
      (cons type-tag data))))

(define get-type-tag car)

(define get-data cdr)

(define curry-scm-type
  (lambda (type)
    (lambda (elem)
      (equal? type (get-type-tag elem)))))
```

typem. Procedura `curry-make-elem` bere jako argument symbol (formální argument `type-tag`) označující daný typ. Symbolům označujícím typy se někdy říká „visačky" nebo „tagy" (z anglického *tags*). Procedura `curry-make-elem` vrací konstruktor jímž je procedura jednoho argumentu. Tento argument představuje hodnotu, kterou chceme vložit do datové části elementu. Na fyzické úrovni je každý element reprezentován párem, jehož první složka je visačka a druhá složka je tvořena hodnotou elementu. Všimněte si, že procedura `curry-make-elem` pouze rozkládá proceduru dvou argumentů `cons` na dvě procedury jednoho argumentu, tento princip jsme popsali v sekci 2.4 jako *currying*. Procedury `get-type-tag` a `get-data` pro daný element

vrací jeho visačku respektive jeho datovou složku. Poslední procedura v programu 12.6 je `curry-scm-type`. Tato procedura pro daný identifikátor typu vrací predikát testující zda-li daný element je tohoto typu či nikoliv. Viz následující příklad použití.

Nejprve vytvoříme konstruktory a predikáty testující typ pro dva různé datové typy (racionální a komplexní čísla):

```
(define make-frac (curry-make-elem 'fraction))
(define make-cplx (curry-make-elem 'complex-number))
(define frac? (curry-scm-type 'fraction))
(define cplx? (curry-scm-type 'complex-number))
```

Předchozí procedury můžeme používat následujícím způsobem:

```
(define a (make-frac '(2 . 3)))
(define b (make-cplx '(2 . 3)))
a             ⟼  (fraction 2 . 3)
b             ⟼  (complex-number 2 . 3)
(get-data a)  ⟼  (2 . 3)
(get-data b)  ⟼  (2 . 3)
(frac? a)     ⟼  #t
(frac? b)     ⟼  #f
(cplx? a)     ⟼  #f
(cplx? b)     ⟼  #t
```

Elementy jazyka Scheme budeme reprezentovat jako hodnoty s *manifestovaným typem.* Typ elementu je manifestován pomocí *visačky*, což je identifikátor typu. Typ elementu je potřeba rozeznávat, protože samotná datová část elementu jednoznačně neurčuje typ (sémantiku) elementu. V další sekci uvidíme, že manifestace typu nám umožní rozlišovat od sebe například vnitřní reprezentaci párů a uživatelsky definovaných procedur. Samotný princip manifestace typů je samozřejmě použitelný i při řešení jiných problémů než je reprezentace elementů jazyka Scheme.

12.3 Datová reprezentace elementů jazyka Scheme

V této sekci ukážeme implementaci jednotlivých elementů jazyka Scheme, které budeme potřebovat při vytvoření jeho interpretu. Budeme postupovat od nejjednodušších elementů ke složitějším.

Před tím, než začneme, je potřeba udělat několik terminologických poznámek. Jelikož se zabýváme implementací interpretu jazyka Scheme *v jazyku Scheme*, pracujeme vlastně se dvěma interprety současně. Prvním z interpretů je pro nás ten interpret, který používáme při vývoji programu. Tímto programem je (druhý) interpret jazyka Scheme. V této sekci se tedy budeme zabývat reprezentací elementů nově vytvářeného interpretu Scheme, nikoliv reprezentací elementů v interpretu, který při vytváření používáme. Abychom zjednodušili terminologii, zavedeme nyní pojmy *metainterpret* a *interpret* jazyka Scheme:

- *metainterpret jazyka Scheme* je již existující interpret, který používáme pro vytváření dalších programů (mimo jiné našeho nového *interpretu* jazyka Scheme),
- *interpret jazyka Scheme* je (meta)program pro metainterpret jazyka Scheme, který provádí interpretaci jisté podmnožiny jazyka Scheme.

Podobně jako rozlišujeme pojmy metainterpret a interpret můžeme odlišovat další pojmy, se kterými jsme se doposud setkali. Tak třeba *metajazyk* (jazyk interpretovaný *metainterpretem*) a *jazyk* (jazyk interpretovaný *interpretem*), *metaelement* (element *metajazyka*) a *element* (element *jazyka*), *metaprogram* (program v *metajazyku*) a *program* (program v *jazyku*). Abychom situaci ještě zjednodušili, budeme někdy předponu „meta“ vynechávat, a to v případě, kdy bude jasné, že se bavíme o původním interpretu jazyka Scheme nebo o souvisejících pojmech.

Pokud budeme hovořit o „metapojmech“, budeme tím mít vždy na mysli pojmy vztažené k metainterpretu jazyka Scheme, to jest k interpretu, který používáme k vývoji našeho nového interpretu podmnožiny jazyka Scheme. Na programy, které dosud vytváříme se taky musíme dívat ze dvou úhlů pohledu. Metaprogramy jsou programy pro výchozí metainterpret, tedy náš nový interpret jazyka Scheme je sám o sobě metaprogram. Programy pro nově vytvářený interpret nazýváme v souladu s předchozí úmluvou pouze „programy“.

Nyní již obraťme naši pozornost k reprezentaci elementů jazyka Scheme. Mezi nejjednodušší elementy patří bezpochyby *čísla*. Při jejich implementaci využijeme toho, že se snažíme vytvořit „Scheme ve Scheme“, tedy nový interpret Scheme v již existujícím metainterpretu Scheme. Díky tomu můžeme de facto převzat veškeré aritmetické (meta) procedury, jak uvidíme dále. Při implementaci také nebudeme rozlišovat jednotlivé typy čísel (přesnou a nepřesnou reprezentaci čísel, racionální čísla, komplexní čísla a tak dále). Pro čísla tedy zavedeme pouze jejich konstruktor a predikát testující typ „číslo“ následovně:

```
(define make-number (curry-make-elem 'number))
(define scm-number? (curry-scm-type 'number))
```

Analogicky jednoduchá bude reprezentace *symbolů*. Hodnotou symbolu (jakožto elementu jazyka) je pro nás jeho „jméno“, které můžeme ztotožnit s řetězcem znaků. Abychom situaci ještě více zjednodušili, nebudeme k označení jmen symbolů používat řetězce znaků, ale metasymboly dostupné v metainterpretu jazyka Scheme. Konstruktory symbolů a predikát testující typ tedy zavedeme:

```
(define make-symbol (curry-make-elem 'symbol))
(define scm-symbol? (curry-scm-type 'symbol))
```

Nyní se zaměříme na speciální elementy jazyka, které se vyhodnocovaly na sebe sama. Jednalo se o *pravdivostní hodnoty*, *prázdný seznam*, a element zastupující *nedefinovanou hodnotu*. Pro tyto elementy nepotřebujeme vytvářet konstruktory, protože pravdivostní hodnoty jsou pouze dvě, prázdný seznam je pouze jeden a stejně tak pouze jeden je element zastupující nedefinovanou hodnotu.

V případě pravdivostních hodnot tedy vytvoříme dva nové elementy zastupující nepravdu a pravdu. Tyto elementy navážeme na symboly `scm-false` a `scm-true`. Dále vytvoříme predikát testující typ „pravdivostní hodnota“. Viz následující kód.

```
(define scm-false ((curry-make-elem 'boolean) #f))
(define scm-true  ((curry-make-elem 'boolean) #t))
(define scm-boolean? (curry-scm-type 'boolean))
```

Prázdný seznam bude nově vytvořený element navázaný na `the-empty-list`:

```
(define the-empty-list ((curry-make-elem 'empty-list) '()))
(define scm-null? (lambda (elem) (equal? elem the-empty-list)))
```

A konečně stejným způsobem vytvoříme i element zastupující nedefinovanou hodnotu:

```
(define the-undefined-value ((curry-make-elem 'undefined) '()))
(define scm-undefined? (lambda (elem) (equal? elem the-undefined-value)))
```

Všimněte si toho, že předchozí predikáty `scm-null?` a `scm-undefined?` využívají toho, že prázdný seznam a nedefinovaná hodnota jsou unikátní elementy daného typu. V tomto případě je tedy možné naprogramovat predikáty testující typ jako predikáty testující rovnost s daným elementem.

Nyní se budeme zabývat tečkovými páry. Nejprve podotkněme, že pro to, abychom v našem interpretu mohli uvažovat seznamy, není nutné (a ani vhodné) vytvářet elementy jazyka typu „seznam“. Plně si vystačíme s dále navrženými páry a prázdným seznamem, který jsme definovali výše. To je zcela v souladu s tím, jak jsme zavedli seznamy pomocí párů v lekci 5. Páry pro nás tedy budou speciální elementy typu „pár“, jejichž datovou složkou budou tvořit dva elementy v pevně daném pořadí. Fyzicky budeme páry reprezentovat pomocí metapárů ve tvaru

`(pair . (`⟨*první*⟩ `.` ⟨*druhý*⟩`))`,

kde ⟨*první*⟩ je element představující první složku páru a ⟨*druhý*⟩ je element představující druhou složku páru. Pro páry vytvoříme jejich konstruktor, dva selektory a predikát testující typ „pár“. V programu 12.7 je uveden konstruktor páru `make-pair`. Jedná se o proceduru dvou argumentů, která vznikla v prostředí

Program 12.7. Reprezentace tečkových párů.

```
(define make-pair
  (let ((make-physical-pair (curry-make-elem 'pair)))
    (lambda (head tail)
      (make-physical-pair (cons head tail)))))

(define scm-pair? (curry-scm-type 'pair))

(define pair-car
  (lambda (pair)
    (if (scm-pair? pair)
        (car (get-data pair))
        (error "CAR: argument must be a pair"))))

(define pair-cdr
  (lambda (pair)
    (if (scm-pair? pair)
        (cdr (get-data pair))
        (error "CDR: argument must be a pair"))))
```

v němž je na symbol `make-physical-pair` navázána procedura vytvářející element s manifestovaným typem „pár“. Samotná procedura `make-pair` provádí pouze jednu aplikaci `make-physical-pair` při níž jsou obě dvě složky spojeny do metapáru pomocí `cons`. Opět jsme tedy zvolili strategii, že k reprezentaci párů nám slouží metapáry a konstruktor páru `make-pair` je vytvořen pomocí konstruktoru metapáru `cons`. Pro objasnění uveďme následující příklady použití `make-pair`:

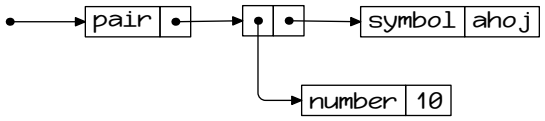
```
(define a (make-number 1))
(define b (make-number 2))
(make-pair a b)                  ⟹  (pair (number . 1) number . 2)
(make-pair a the-empty-list)     ⟹  (pair (number . 1) empty-list)
(make-pair the-empty-list b)     ⟹  (pair (empty-list) number . 2)
```

V programu 12.7 je dále uveden predikát `scm-pair?` testující, zda-li je daný element typu „pár“. Dále jsou zde uvedeny selektory `pair-car` a `pair-cdr` sloužící k přístupu k první, případně druhé, složce párů. Jejich implementace je přímočará.

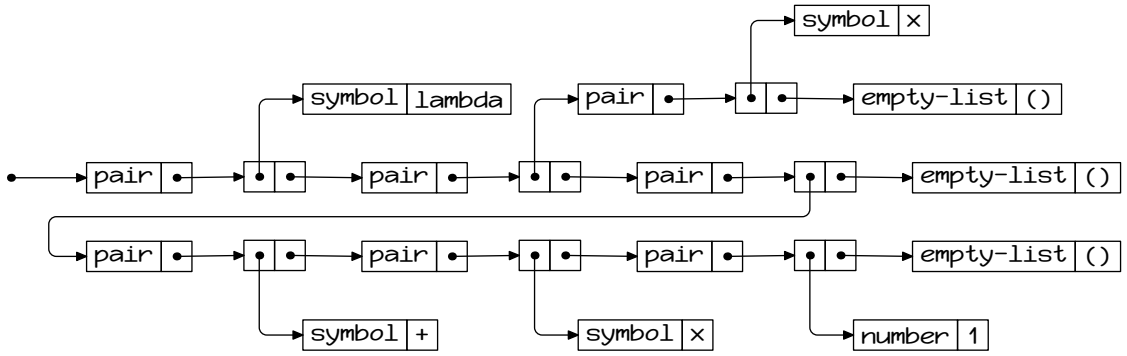
Příklad 12.2. Nyní si již můžeme udělat základní představu o tom, jak bude vypadat reprezentace složitějších elementů jazyka pomocí metaelementů. Například pár `(10 . ahoj)`, to jest pár, jehož první složkou je číslo a druhou složkou je symbol bude v metainterpretu fyzicky reprezentován metaelementem vyobrazeným na obrázku 12.1. Dále například λ-výraz `(10 . ahoj)` bude v metainterpretu reprezentován tak, jak ukazuje obrázek 12.2. Jak je na první Pohled zřejmé, reprezentace tohoto relativně malého seznamu je dost velká. To je jakási daň, kterou musíme zaplatit za visačky jednoznačně určující typy elementů.

Další elementy jazyka, jejichž reprezentaci popíšeme, jsou *prostředí.* Koncept prostředí byl představen již v lekci 1 a dále zpřesněn v lekci 2. Prostředí potřebujeme k udržování vazeb mezi symboly a elementy a kvůli schůdné implementaci lexikálního rozsahu platnosti: každé prostředí, kromě globálního, má ukazatele na svého lexikálního předka (prostředí svého vzniku). Datová část prostředí tedy musí obsahovat jednak tabulku vazeb mezi symboly a elementy a jednak (ukazatel na) další prostředí. Elementy typu prostředí tedy budeme reprezentovat metapáry ve tvaru

Obrázek 12.1. Fyzická reprezentace páru `(10 . ahoj)` pomocí metaelementů.



Obrázek 12.2. Fyzická reprezentace seznamu `(lambda (x) (+ x 1))` pomocí metaelementů.



`(environment .` `(`⟨*předek*⟩ `.` ⟨*tabulka*⟩`))`,

kde ⟨*předek*⟩ je buďto element prostředí nebo element „nepravda“ (dané prostředí nemá předka) a ⟨*tabulka*⟩ je tabulka vazeb mezi symboly a elementy udržovaná jako interní reprezentace seznamu ve tvaru

`(`⟨$symbol_1$⟩ ⟨$element_1$⟩`)`
`(`⟨$symbol_2$⟩ ⟨$element_2$⟩`)`
⋮
`(`⟨$symbol_n$⟩ ⟨$element_n$⟩`)))`.

Pro práci s prostředími budeme potřebovat několik základních procedur. V první řadě to bude konstruktor prostředí `make-env` a dva selektory `get-pred` a `get-table`, které pro dané prostředí vracejí prostředí předka nebo tabulku vazeb. Tyto procedury jsou uvedeny v programu 12.8. V tomto programu je dále uveden predikát testující typ elementu „prostředí“. Všimněte si, že základní konstruktory a selektory prostředí jsou de facto stejné jako konstruktory a selektory párů, viz program 12.7. V případě prostředí ale platí, že jeho datové složky nejsou libovolné elementy, ale přesně vymezené elementy (první element je předek, tedy „nepravda“ nebo opět prostředí a druhý element je vždy tabulka vazeb).

Pro pohodlnou manipulaci s prostředím zavedeme další procedury. V našem novém interpretu totiž musíme mít nějak definováno prostředí počátečních vazeb, musíme mít tedy k dispozici procedury, kterými prostředí vytvoříme. Další prostředí již budou vznikat při aplikaci uživatelsky definovaných procedur tak, jak jsme to vysvětlili v lekci 2. V programu 12.9 je uvedeno několik procedur, které využijeme při definici počátečních prostředí. Predikát `global?` je pro daný element pravdivý, právě když je element prostředí, které nemá předka. Jedná se tedy o predikát testující, zda-li je předaný element globální prostředí. Pomocná procedura `assoc->env` provede převod *asociačního metaseznamu* na tabulku vazeb realizovanou *asociačním seznamem*. Při bližším pohledu je vidět, že `assoc->env` je v podstatě jen jednoduchá rekurzivní procedura, která převádí metaseznam metapárů na seznam párů ve vnitřní reprezentaci (nového interpretu), přitom také provádí vytváření nových symbolů z metasymbolů. Pro objasnění ještě uveďme příklad jejího použití:

```
(define s `((a . ,(make-number 10)) (b . ,(make-number 20))))
(assoc->env s)  ⟹  (pair (pair (symbol . a) number . 10)
                         pair (pair (symbol . b) number . 20)
                              empty-list)
```

Konečně procedura `make-global-env` z programu 12.9 slouží k vytvoření globálního prostředí. Její použití uvidíme v jedné z dalších sekcí.

Program 12.8. Reprezentace prostředí.

```
(define make-env
  (let ((make-physical-env (curry-make-elem 'environment)))
    (lambda (pred table)
      (make-physical-env (cons pred table)))))

(define scm-env? (curry-scm-type 'environment))

(define get-table
  (lambda (elem)
    (if (scm-env? elem)
        (cdr (get-data elem))
        (error "GET-TABLE: argument must be an environment"))))

(define get-pred
  (lambda (elem)
    (if (scm-env? elem)
        (car (get-data elem))
        (error "GET-PRED: argument must be an environment"))))
```

Při implementaci vyhodnocovacího procesu budeme potřebovat hledat vazby v prostředích. K tomuto účelu naprogramujeme dvě pomocné procedury. Procedura `scm-assoc` z programu 12.10 provádí prakticky totéž co standardní procedura `assoc` (viz standard R^{5}RS jazyka Scheme [R5RS]). Jediný rozdíl je v tom, že `scm-assoc` pracuje s asociačními seznamy (a nikoliv metaseznamy) v jejich interní reprezentaci. Procedura pro daný klíč a asociační seznam prohledává daný asociační seznam a vrací první pár, jehož první složka odpovídá klíči. Pokud žádný takový pár neexistuje, je vrácena „nepravda“. Proceduru `scm-assoc` tedy můžeme použít k vyhledání vazby v tabulce vazeb nějakého prostředí. Druhá procedura v programu 12.10 je procedura `lookup-env`, což je komplexnější procedura, která vyhledává vazbu symbolů v prostředí nebo vrací element navázaný na symbol `not-found`, pokud není vazba nalezena. Formální argument pojmenovaný `search-nonlocal?` slouží jako přepínač, kterým lze ovlivňovat, co se má stát, pokud vazba není nalezena v tabulce aktuálního prostředí. Pokud je při aplikaci `lookup-env` na `search-nonlocal?` navázána hodnota „pravda“, pak se při nenalezení vazby v tabulce lokálního prostředí postupuje k nadřazenému prostředí. V případě, že je na `search-nonlocal?` navázána „nepravda“, pak je při nenalezení vazby v tabulce lokálního prostředí okamžitě vrácena hodnota navázaná na `not-found`. Procedura `lookup-env` bude používána k hledání vazeb symbolů přímo během vyhodnocování elementů.

Nyní se budeme zabývat reprezentací procedur a speciálních forem. Reprezentace *primitivních procedur*, tedy procedur, které budou přímo zabudované v našem novém interpretu, bude jednoduchá. Vytvoříme jejich konstruktor a predikát testující typ „primitivní procedura“ následovně:

```
(define make-primitive (curry-make-elem 'primitive))
(define scm-primitive? (curry-scm-type 'primitive))
```

Datovou složkou primitivní procedury bude *metaprocedura*, tedy nějaká procedura, která je přímo součástí konstruovaného interpretu a která bude při aplikaci prováděna metainterpretem jazyka Scheme. V tuto chvíli připomeňme, že už v první lekci, kdy jsme primitivní procedury zavedli, jsem upozornili na fakt, že se nebudeme zabývat tím, jak jsou vytvořené. Obecně můžeme říct, že primitivní procedury jsou vždy vytvořeny pomocí nějakých metaprocedur. Primitivní metaprocedury (primitivní procedury v metainterpretu) jsou rovněž vytvořeny pomocí nějakých „metametaprocedur“, které jsou, v případě interpretu jazyka Scheme vzniklého kompilací, zapsané v kódu stroje (a zdrojové kódy těchto „metametaprocedur“ budou naprogramovány nejspíš v nějakém vyšším programovacím jazyku, třeba v jazyku C, což je oblíbený jazyk pro tvorbu interpretů a překladačů).

Program 12.9. Konstruktor pro globální prostředí.

```
(define assoc->env
  (lambda (l)
    (if (null? l)
        the-empty-list
        (make-pair (make-pair (make-symbol (caar l))
                              (cdar l))
                   (assoc->env (cdr l))))))

(define make-global-env
  (lambda (alist-table)
    (make-env scm-false (assoc->env alist-table))))

(define global?
  (lambda (elem)
    (and (scm-env? elem)
         (equal? scm-false (get-pred elem)))))
```

At' tak či onak, nyní se *musíme* zabývat tím, jak primitivní procedury vytvářet, protože se zabýváme konstrukcí interpretu jazyka. Některé primitivní procedury budeme programovat jako *uživatelsky definované metaprocedury*. Řadu důležitých primitivních procedur, například aritmetické procedury, ale můžeme vytvořit jednoduchým „trikem" a to tak, že pouze zabalíme metaproceduru, která je protějškem dané procedury, do pomocného programu, který z daných elementů vyzvedne jejich datovou část, aplikuje metaproceduru s takto získanými hodnotami a nakonec výsledkem aplikace (tedy metaelement) převede do interní reprezentace. Na tuto problematiku se blíže podíváme v dalších sekcích.

V lekci 2 jsme uvedli, že *uživatelsky definované procedury* chápeme jako trojice hodnot $\langle\langle parametry\rangle, \langle tělo\rangle, \mathcal{P}\rangle$, kde $\langle parametry\rangle$ je seznam formálních argumentů, $\langle tělo\rangle$ je element reprezentující tělo procedury a $\mathcal{P}$ je prostředí vzniku procedury. V lekci 3 jsme rozšířili procedury tak, že jsme umožnili, aby v jejich těle bylo přítomno víc výrazů. Toto rozšíření jsme učinili z důvodu pohodlného zavedení interních definic. Jelikož se ale jedná o rys, který není čistě funkcionální (při vytváření definic dochází k vedlejšímu efektu jímž je modifikace prostředí), budeme se dále zabývat konceptem procedur tak, jak jsme jej představili v lekci 2 (jeden výraz v těle). Uživatelsky definovaná procedura je tedy element jazyka, který v sobě agreguje tři hodnoty (seznam formálních argumentů, prostředí a tělo). Reprezentaci uživatelsky definovaných procedur jakožto elementů našeho jazyka můžeme tedy provést zcela přímočaře tak, jak je to ukázáno v programu 12.11. Procedura `make-procedure` je konstruktor uživatelsky definovaných procedur akceptující tři argumenty: prostředí, seznam argumentů a tělo. Tyto tři položky jsou v elementu fyzicky uloženy do tříprvkového metaseznamu. Dále máme k dispozici tři selektory `procedure-environment`, `procedure-arguments` a `procedure-body` vracející prostředí, seznam argumentů a tělo pro daný element typu „uživatelsky definovaná procedura". Nakonec jsme opět zavedli predikát testující daný typ. Jelikož primitivní procedury a uživatelsky definované procedury chápeme souhrnně jako procedury, vytvoříme navíc dodatečný predikát testující, zda-li je element procedura (primitivní nebo uživatelsky definovaná):

```
(define scm-procedure?
  (lambda (elem)
    (or (scm-primitive? elem)
        (scm-user-procedure? elem))))
```

Posledním typem uvažovaných elementů jazyka budou speciální formy. Speciální formy budou implementované opět pomocí metaprocedur. Pro každou speciální formu, kterou bude náš interpret obsahovat, budeme vytvářet speciální uživatelsky definovanou metaproceduru. Zatím tedy vytvoříme pouze základní reprezentaci elementů typu „speciální forma":

Program 12.10. Vyhledávání vazeb v prostředí.

```
(define scm-assoc
  (lambda (key alist)
    (cond ((scm-null? alist) scm-false)
          ((equal? key (pair-car (pair-car alist))) (pair-car alist))
          (else (scm-assoc key (pair-cdr alist))))))

(define lookup-env
  (lambda (env symbol search-nonlocal? not-found)
    (let ((found (scm-assoc symbol (get-table env))))
      (cond ((not (equal? found scm-false)) found)
            ((global? env) not-found)
            ((not search-nonlocal?) not-found)
            (else (lookup-env (get-pred env) symbol #t not-found))))))
```

Program 12.11. Reprezentace uživatelsky definovaných procedur.

```
(define make-procedure
  (let ((make-physical-procedure (curry-make-elem 'procedure)))
    (lambda (env args body)
      (make-physical-procedure (list env args body)))))

(define procedure-environment (lambda (proc) (car (get-data proc))))
(define procedure-arguments (lambda (proc) (cadr (get-data proc))))
(define procedure-body (lambda (proc) (caddr (get-data proc))))

(define scm-user-procedure? (curry-scm-type 'procedure))
```

```
(define make-specform (curry-make-elem 'specform))
(define scm-specform? (curry-scm-type 'specform))
```

Nyní se dostáváme do bodu, kdy si můžeme dovolit malou epistemickou úvahu. Je zajímavé, že při našem exkurzu programováním v jazyku Scheme jsme postupovali od procedur vyšších řádů směrem k párům. U párů pro uvedli, že je můžeme plně vyjádřit pomocí uživatelsky definovaných procedur vyšších řádů. Nyní postupujeme zdánlivě obráceně. Uživatelsky definované procedury máme úplně reprezentované pomocí (meta) párů.

12.4 Vstup a výstup interpretu

Konstruovaný interpret jazyka Scheme musí mít k dispozici základní vstupní a výstupní části, konkrétně *reader* (proceduru realizující načítání vstupních symbolických výrazů a jejich převod do interní reprezentace) a *printer* (proceduru, která se stará o vytištění externí reprezentace elementů).

Pro načtení symbolického výrazu můžeme použít primitivní proceduru `read`, se kterou jsme se již setkali v lekci 5. Po načtení však musíme ještě provést konverzi hodnoty získané aplikací `read` do naší interní reprezentace. To jest všechny načtené symboly, čísla a seznamy je potřeba převést na příslušné elementy „symbol“, „číslo“ a „páry“ tak, jak jsme je představili v předchozí sekci. Tuto konverzi pro nás bude provádět nově vytvořená procedura `expr->intern`, viz program 12.12. Samotný reader je již možné naprogramovat

Program 12.12. Převod do interní reprezentace a implementace readeru.

```
(define expr->intern
  (lambda (expr)
    (cond ((symbol? expr) (make-symbol expr))
          ((number? expr) (make-number expr))
          ((and (boolean? expr) expr) scm-true)
          ((boolean? expr) scm-false)
          ((null? expr) the-empty-list)
          ((pair? expr) (make-pair (expr->intern (car expr))
                                   (expr->intern (cdr expr))))
          ((eof-object? expr) #f)
          (else (error "READER: Syntactic error.")))))

(define scm-read
  (lambda ()
    (expr->intern (read))))
```

pomocí `read` a procedury `expr->intern` tak, jak je to uvedeno v programu 12.12. Dodejme, že pokud bychom neměli v jazyku Scheme k dispozici proceduru `read`, museli bychom rovněž naprogramovat samotné načítání vstupních výrazů. V případě jazyka Scheme by to nebylo příliš obtížné, ale tématicky tento problém spadá do jiných kurzů. Laskavého čtenáře tímto odkazujeme na kurzy *formální jazyky a automaty* a *překladače*. V následující ukázce je uvedeno použití procedury `expr->intern`.

```
(expr->intern 1)          ⟼  (number . 1)
(expr->intern '+)         ⟼  (symbol . +)
(expr->intern '(1 . 2))   ⟼  (pair (number . 1) number . 2)
(expr->intern '(- 13))    ⟼  (pair (symbol . -) pair (number . 13) empty-list)
  ⋮
```

Printer se používá především k *vypisování výsledků vyhodnocení*. Při implementaci printeru musíme oproti readeru naprogramovat opačnou konverzi. Tedy konverzi elementu v interní reprezentaci na čitelnou reprezentaci, jenž může být vytištěna na obrazovku, zapsána do souboru, a tak dále. Pochopitelně, že naprogramovat printer je obecně vždy jednodušší než naprogramovat reader (konverze řetězce znaků na strukturovaná data je mnohem obtížnější než konverze strukturovaných dat na řetězec znaků[18]). Jedna z možností, jak naprogramovat printer je uvedena v programu 12.13. Printer bychom samozřejmě mohli realizovat i mnohem jednodušeji, například následující procedurou, která provede pouze vytištění interní reprezentace elementu na obrazovku:

```
(define scm-print
  (lambda (elem)
    (display elem)))
```

V tomto případě by ale výpis některých elementů nebyl přehledný (uvidíme dále).

12.5 Implementace vyhodnocovacího procesu

V této sekci rozebereme implementaci vyhodnocovacího procesu včetně aplikace procedur a speciálních forem. Postup bude kopírovat teorii, kterou jsme probrali v prvních dvou lekcích tohoto textu. Nejprve při-

[18] Toto pozorování by pro nás mělo být vlastně malým poučením. Při programování čehokoliv se vždy vyplatí reprezentovat data v co možná nejvíc strukturované podobě. Nikdy tím nemůžeme nic ztratit (snad kromě větší paměťové náročnosti na jejich uložení) a přidaná hodnota může být opravdu velká. Není náhodou, že metodám (automatického) strukturování velkých dat se v informatice věnuje řada disciplín.

Program 12.13. Převod do externí reprezentace Implementace printeru.

```
(define intern->expr
  (lambda (expr)
    (cond ((scm-pair? expr) (cons (intern->expr (pair-car expr))
                                  (intern->expr (pair-cdr expr))))
          ((scm-env? expr) "#<environment>")
          ((scm-primitive? expr) "#<primitive-procedure>")
          ((scm-user-procedure? expr) "#<user-defined-procedure>")
          ((scm-specform? expr) "#<special-form>")
          ((scm-undefined? expr) "#<undefined>")
          (else (get-data expr)))))

(define scm-print
  (lambda (elem)
    (display (intern->expr elem))))
```

pomeňme, že aplikace primitivních procedur a speciálních forem bude řešena pomocí aplikace uživatelsky definovaných metaprocedur.

Jak jsme již předeslali v předchozí sekci, v některých případech je možné vytvořit primitivní procedury s využitím primitivních metaprocedur pomocí jejich „pouhého zabalení“ do pomocné procedury. Pomocná procedura sloužící jako jakási obálka se stará o konverzi elementů na metaelementy (před aplikací metaprocedury) a opačně o konverzi metaelementů na elementy (po aplikaci metaprocedury). Na provedení tohoto zabalení „metaprocedury“ můžeme vytvořit překvapivě jednoduchou proceduru `wrap-primitive`, která je zobrazena v programu 12.14. Použití této procedury uvidíme v dalších sekcích. Princip `wrap-primitive`

Program 12.14. Reprezentace primitivních procedur.

```
(define wrap-primitive
  (lambda (proc)
    (make-primitive
     (lambda arguments
       (expr->intern (apply proc (map get-data arguments)))))))
```

si nejlépe uvědomíme na následujícím příkladu, ve kterém nejprve na symbol `p` navážeme element jímž je primitivní procedura vzniklá z metaprocedury sčítání. Datovou složkou tohoto elementu je tedy metaprocedura vzniklá vyhodnocením vnitřního λ-výrazu uvedeného v těle procedury z programu 12.14. Viz příklad:

```
(define p (wrap-primitive +))
p  ⟼  (primitive . „metaprocedura realizující proceduru sčítání čísel“)
```

Primitivní proceduru bychom nyní mohli aplikovat následovně:

```
((cdr p) (make-number 10) (make-number 20))  ⟼  (number . 30)
```

Během předchozí aplikace byla provedena extrakce metačísel z elementů reprezentujících čísla 10 a 20. Dále byla aplikována primitivní metaprocedura sčítání a jejím výsledkem je metačíslo 30. To bylo nakonec zkonvertováno na element pomocí procedury `expr->intern`, kterou jsme představili v programu 12.12 na straně 301. Výše uvedenou metodou převezmeme v našem novém interpretu další aritmetické procedury.

Jelikož bude během vyhodnocování občas potřeba konvertovat metaseznamy (seznamy složené z metapárů) na seznamy (seznamy složené z párů) a obráceně, vytvoříme si pomocný konstruktor `convert-list`,

viz jeho kód v programu 12.15. Procedura `convert-list` je de facto obecný konstruktor seznamů a me-

Program 12.15. Obecný konvertor seznamu na seznam ve vnitřní reprezentaci a obráceně.

```
(define convert-list
  (lambda (a-null? a-car a-cdr b-cons b-nil f l)
    (if (a-null? l)
        b-nil
        (b-cons (f (a-car l))
                (convert-list a-null? a-car a-cdr b-cons b-nil f
                              (a-cdr l))))))
```

taseznamů. Pomocí něj můžeme vytvořit řadu konvertorů těchto datových struktur a metastruktur. Ty nejdůležitější z nich jsou uvedeny v programu 12.16. Procedura `scm-list->list` konvertuje seznamy na

Program 12.16. Konverze seznamů na metaseznamy o obráceně.

```
(define scm-list->list
  (lambda (scm-l)
    (convert-list scm-null? pair-car pair-cdr cons '() (lambda (x) x) scm-l)))

(define list->scm-list
  (lambda (l)
    (convert-list null? car cdr make-pair the-empty-list (lambda (x) x) l)))

(define map-scm-list->list
  (lambda (f scm-l)
    (convert-list scm-null? pair-car pair-cdr cons '() f scm-l)))
```

metaseznamy. Naopak procedura `list->scm-list` konvertuje seznamy na metaseznamy. Pomocí procedury `map-scm-list->list` je rovněž možné převést seznam na metaseznam, ale během zpracování prvků výchozího seznamu se ještě používá procedura jednoho argumentu k modifikaci prvků (analogicky jako u standardní procedury `map`).

Nyní se již můžeme podívat na implementaci vyhodnocovacího procesu. Nejprve se budeme zabývat implementací procedury provádějící vyhodnocení elementů. V ní použijeme několik procedur, které objasníme dále. K vyhodnocování elementů bude sloužit procedura `scm-eval`, která je uvedena včetně jednoduchých komentářů v programu 12.17 na straně 304. Všimněte si, že `scm-eval` má dva argumenty, první z nich je element, který vyhodnocujeme, a druhým je aktuální prostředí, ve kterém tento element vyhodnocujeme. To koresponduje s tím jak jsme zavedli $\mathrm{Eval}[E, \mathcal{P}]$ v definici 2.7 na straně 47. V těle procedury `scm-eval` je jeden `cond`-výraz, ve kterém se rozhoduje o způsobu vyhodnocení elementu na základě jeho typu. Zde uplatníme manifestaci typů a predikáty testující typ elementu, které jsme doposud zavedli.

V prvním případě je vyřešena situace, kdy je daný element symbol. V tomto případě je hledá jeho vazba (počínaje předaným aktuálním prostředím) pomocí procedury `lookup-env` z programu 12.10 na straně 300.

Druhá větev `cond`-výrazu ošetřuje případ, kdy je daný element seznam nebo lépe řečeno, kdy je daný element *pár*[19]. V tomto případě, je nejprve v daném prostředí vyhodnocen první prvek páru. Všimněte

[19]Uvědomte si, že test toho, zda-li daný element seznam nelze provést v konstantním čase. Z důvodu efektivity by tedy bylo nešťastné definovat vyhodnocování *seznamů*, ale mnohem jednodušší je pracovat přímo s *páry*, které daný seznam tvoří. Tak je tomu i v tomto případě. Navíc nám tento postup umožní zapisovat program pomocí párů v tečkové notaci (i když to zřejmě není příliš užitečné a ani přehledné).

Program 12.17. Implementace vlastního vyhodnocovacího procesu.

```
;; vyhodnot vyraz v danem prostredi
(define scm-eval
  (lambda (elem env)

    ;; vyhodnocovani elementu podle jejich typu
    (cond

     ;; symboly se vyhodnocuji na svou aktualni vazbu
     ((scm-symbol? elem)
      (let* ((binding (lookup-env env elem #t #f)))
        (if binding
            (pair-cdr binding)
            (error "EVAL: Symbol not bound"))))

     ;; vyhodnoceni seznamu
     ((scm-pair? elem)

      ;; nejprve vyhodnotime prvni prvek seznamu
      (let* ((first (pair-car elem))
             (args (pair-cdr elem))
             (f (scm-eval first env)))

        ;; podle prvniho prvku rozhodni o co se jedna
        (cond

         ;; pokud se jedna o proceduru, vyhodnot argumenty a aplikuj
         ((scm-procedure? f)
          (scm-apply f (map-scm-list->list
                        (lambda (elem)
                          (scm-eval elem env))
                        args)))

         ;; pokud se jedna o formu, aplikuj s nevyhodnocenymi argumenty
         ((scm-specform? f)
          (scm-specform-apply env f (scm-list->list args)))

         ;; na prvnim miste stoji nepripustny prvek
         (error "EVAL: First element did not eval. to procedure"))))

     ;; vse ostatni (cisla, boolean, ... se vyhodnocuje na sebe sama)
     (else elem))))
```

si, že zde dochází k *rekurzivní aplikaci* `scm-eval`. Výsledek vyhodnocení je navázán v lokálním metaprostředí na metasymbol `f`. Dále vyhodnocování postupuje dvěma směry podle toho jakého typu je element navázaný na `f`. Pokud je to procedura (primitivní nebo uživatelsky definovaná), provedeme aplikaci metaprocedury `scm-apply`, kterou si záhy popíšeme. Tato metaprocedura má na starosti provedení aplikace procedury `f` s danými argumenty. Všimněte si, že argumenty jsou předány ve formě metaseznamu, jehož prvku jsou vyhodnocené elementy ze seznamu argumentů (zde opět provádíme rekurzivní aplikaci `scm-eval`). V případě, kdy je na `f` navázána speciální forma je provedena její aplikace pomocí metaprocedury `scm-specform-apply`. Zde si všimněme toho, že argumenty jsou `scm-specform-apply` předány bez vyhodnocení a navíc jako jeden z argumentů pro `scm-specform-apply` předáváme aktuální prostředí. To je nezbytné k tomu, aby mohla každá speciální forma provádět další vyhodnocování a aby věděla, ve kterém prostředí má vyhodnocování provádět.

Konečně v poslední větvi `cond`-výrazu (`else`-větev) obsaženého v těle `scm-eval` je vyřešeno vyhodnocování všech ostatních elementů (tedy čísel, pravdivostních hodnot, prázdného seznamu, nedefinované hodnoty, procedur, speciálních forem a prostředí): tyto elementy se *vyhodnocují na sebe sama.*

Všimněte si, že procedura `scm-eval` de facto implementuje postup, který jsme uvedli v definici 2.7 na straně 47. Procedura `scm-eval` se od tohoto postupu liší jen v technických drobnostech. Například implementace separátního bodu (A) z definice 2.7 není přítomna, protože je řešena již v rámci bodu (D), viz `else`-větev v proceduře `scm-eval`. Nyní se tedy můžeme konečně „prakticky přesvědčit“, že vyhodnocovací proces skutečně funguje tak, jak jsme jej původně uvedli.

Abychom dokončili vyhodnocovací proces, musíme vytvořit metaprocedury provádějící aplikaci procedur a speciálních forem. Aplikace speciálních forem je elementární. Jelikož každá forma řídí vyhodnocování svých argumentů jiným způsobem, necháváme při aplikaci speciální formy veškerý průběh vyhodnocování na metaproceduře, která je datovou složkou speciální formy. Proceduru `scm-specform-apply` použitou ve `scm-eval` bychom tedy mohli naprogramovat takto:

```
(define scm-specform-apply
  (lambda (env form args)
    (cond ((scm-specform? form) (apply (get-data form) env args))
          (else (error "APPLY: Expected special form")))))
```

Na předchozím kódu si opět všimněte, že metaproceduře realizující speciální formu je předáno prostředí jako první z argumentů. Implementaci jednotlivých speciálních forem ukážeme v další sekci.

V případě aplikace procedur je situace složitější. Musíme jednak rozlišit mezi primitivními procedurami a uživatelsky definovanými procedurami. Situace v případě primitivních procedur bude podobně jednoduchá jako v případě speciálních forem, jak záhy uvidíme. V případě aplikace uživatelsky definovaných procedur však musíme provést kroky popsané v definici 2.12 na straně 49. To jest, musíme vytvořit nové lokální prostředí s vazbami a v něm vyhodnotit tělo procedury.

K vytvoření tabulky vazeb mezi formálními argumenty a předanými argumenty, která je nezbytnou součástí nově vytvářeného lokálního prostředí, bude sloužit procedura `make-bindings` z programu 12.18. Procedura `make-bindings` akceptuje dva argumenty, prvním s nich je seznam formálních argumentů a druhým je metaseznam elementů (hodnot), které se mají na dané formální argumenty „navázat“. Výsledkem aplikace `make-bindings` je tabulka vazeb, která je posléze použita jako součást nového prostředí.

Samotná aplikace procedur je prováděna pomocí `scm-apply`, viz program 12.19. Ještě před tím, že probereme `scm-apply`, se budeme zabývat obecnější pomocnou metaprocedurou `scm-env-apply`, která je rovněž uvedena v programu 12.19. Metaprocedura `scm-env-apply` bere jako argumenty proceduru, prostředí a argumenty s nimiž má být procedura aplikována. Pokud jde o primitivní proceduru, její aplikace je provedena prostou aplikací metaprocedury. V tomto případě nehraje prostředí předané `scm-env-apply` žádnou roli. V případě, že je vyžadována aplikace uživatelsky definované procedury se vytvoří nová tabulka vazeb mezi formálními argumenty této procedury a hodnotami, se kterými proceduru aplikujeme. Dále je pomocí této tabulky a předaného prostředí (navázaného na `env`) vytvořeno nové prostředí a v něm je vyhodnoceno tělo

Program 12.18. Vytvoření tabulky vazeb mezi formálními a skutečnými argumenty.

```
(define make-bindings
  (lambda (formal-args args)
    (cond ((scm-null? formal-args) the-empty-list)
          ((scm-symbol? formal-args)
           (make-pair (make-pair formal-args (list->scm-list args))
                      the-empty-list))
          (else (make-pair
                 (make-pair (pair-car formal-args) (car args))
                 (make-bindings (pair-cdr formal-args) (cdr args)))))))
```

Program 12.19. Implementace aplikace procedur.

```
(define scm-env-apply
  (lambda (proc env args)
    (cond ((scm-primitive? proc) (apply (get-data proc) args))
          ((scm-user-procedure? proc)
           (scm-eval (procedure-body proc)
                     (make-env env
                               (make-bindings (procedure-arguments proc)
                                              args))))
          (else (error "APPLY: Expected procedure")))))

(define scm-apply
  (lambda (proc args)
    (cond ((scm-primitive? proc) (scm-env-apply proc #f args))
          ((scm-user-procedure? proc)
           (scm-env-apply proc (procedure-environment proc) args))
          (else (error "APPLY: Expected procedure")))))
```

procedury. Jinými slovy, `scm-env-apply` v případě uživatelsky definovaných procedur provádí vyhodnocení jejich těla v prostředí, jehož předek je prostředí navázané na `env`. Tím je `scm-env-apply` obecnější než tradiční `apply`, kde je předek nově vzniklého prostředí dán prostředím vzniku procedury.

Naprogramovat `scm-apply` pomocí `scm-env-apply` je již velmi jednoduché. V programu 12.19 vidíme, že v případě aplikace primitivních procedur je voláno `scm-env-apply` s prostředním argumentem nastaveným na „nepravda“ (hodnota tohoto argumentu může být jakákoliv, protože jak jsme již zjistili, nebude k ničemu použita). V případě aplikace uživatelsky definovaných procedur je opět voláno `scm-env-apply`, tentokrát je ale prostředí předka nastaveno na prostředí vzniku procedury (což je prostředí uložené v datové složce elementu reprezentujícího uživatelsky definovanou proceduru).

12.6 Počáteční prostředí a cyklus REPL

Nyní nadefinujeme počáteční prostředí našeho interpretu. Jelikož se snažíme vytvářet čistě funkcionální interpret Scheme, tedy interpret, ve kterém žádná procedura ani speciální forma *nemá vedlejší efekt*, náš interpret nebude obsahovat speciální formu `define`, jejímž vedlejším efektem je modifikace aktuálního prostředí. V důsledku budeme muset v našem interpretu vytvářet rekurzivní procedury pomocí y-kombinátorů.

Dalším zajímavým rysem vašeho interpretu bude, že na počátku vyhodnocování nebudeme uvažovat pouze jediné prostředí, ale hned několik prostředí, které mezi sebou budou mít určité vazby. Konkrétně

budeme rozlišovat dvě prostředí:

- *prostředí primitivních definic* (anglický název *toplevel environment*) – prostředí, které nemá předka (přísně vzato je to tedy *globální prostředí*), v němž jsou symboly navázány na primitivní procedury, speciální formy a případně další elementy vyjma uživatelsky definovaných procedur,
- *prostředí odvozených definic* (anglický název *midlevel environment*) – prostředí, jehož předkem je prostředí primitivních definic, v němž jsou symboly navázány na uživatelsky definované procedury.

Proč vůbec potřebujeme vytvořit dvě prostředí? Přísně vzato bychom nemuseli, ale tím pásem bychom v počátečním prostředí nemohli mít na žádný symbol navázanou netriviální uživatelsky definovanou proceduru. Každá uživatelsky definovaná procedura má totiž v sobě obsaženu informaci o prostředí svého vzniku. Jelikož nemáme k dispozici prostředky pro „změnu (mutaci) prostředí“, nemůžeme vytvořit proceduru, která by byla navázána v prostředí svého vlastního vzniku. Kromě prostředí primitivních definic tedy musíme mít k dispozici i *nové prostředí*, v němž mohou být definice uživatelsky definovaných procedur jejichž prostředím vzniku je právě prostředí primitivních definic. Takovým novým prostředím je právě prostředí odvozených definic[20]. Podotkněme, že uživatelsky definované procedury navázané v prostředí odvozených definic se „vzájemně nevidí“, protože prostředím vzniku všech procedur je prostředí primitivních definic. Kdybychom chtěli, aby některé z uživatelsky definovaných procedur mohly používat jiné, museli bychom vytvořit nové „prostředí odvozených definic II.“ jehož předkem by bylo existující prostředí odvozených definic. V tomto případě by uživatelsky definované procedury navázané na symboly v prostředí odvozených definic II. mohly používat procedury navázané ve výchozím prostředí odvozených definic. Dále bychom mohli dle potřeby vytvářet další „prostředí odvozených definic III., IV., . . . “.

Nyní si tedy rozebereme obsah prostředí primitivních definic a prostředí odvozených definic. Začneme *prostředím primitivních definic*. V našem interpretu bude prostředí zavedeno pomocí konstruktoru globálního prostředí `make-global-env` z programu 12.9 na straně 299.

```
(define scheme-toplevel-env
  (make-global-env
   `(
     ⋮
      )))
```

Výpustka uvedená v předchozím kódu značí místo, kam budou uváděny definice vazeb symbolů, kterými se budeme zabývat ve zbytku této sekce. Nejprve ukážeme definice několika základních speciálních forem. V programu 12.20 je ukázána definice speciální formy `if`. Tato definice (a každá další uvedená) je ve tvaru

Program 12.20. Definice speciální formy `if` v globálním prostředí.

```
(if . ,(make-specform
        (lambda (env condition expr . alt-expr)
          (let ((result (scm-eval condition env)))
            (if (equal? result scm-false)
                (if (null? alt-expr)
                    the-undefined-value
                    (scm-eval (car alt-expr) env))
                (scm-eval expr env))))))
```

páru `(`⟨*symbol*⟩ `.` ⟨*element*⟩`)`, kde `symbol` je metasymbol určující „jméno symbolu“ a ⟨*element*⟩ je konkrétní element navázaný na symbol. V našem případě je metasymbolem `if` a element navázaný na příslušný symbol v prostředí primitivních definic vznikne vyhodnocením výrazu `(make-specform` · · · . Zde by nás nemělo překvapit uvedení „`,`“ před výrazem, protože si musíme uvědomit, že celý pár `(`⟨*symbol*⟩ `.` ⟨*element*⟩`)`

[20] Některé uživatelsky definované procedury bychom navázané mít mohli, třeba procedury vracející konstantní číselnou hodnotu nebo projekce. Tyto procedury totiž ve svém těle kromě formálních argumentů nepoužívají žádné další symboly. Tím pádem může být prostředí jejich vzniku nastaveno na „nepravda“.

je vložen do předchozího výrazu na místě výpustky, která je v kvazikvotovaném seznamu (viz seznam navázaný na `scheme-toplevel-env`).

Prohlédneme-li si výraz v programu 12.20 definující speciální formu `if` vidíme, že odpovídá formálnímu popisu `if` z definice 1.31 na straně 36. Všimněte si, že speciální forma je realizována metaprocedurou, jejíž první argument je *prostředí* a další argumenty odpovídají argumentům speciální formy `if`. Prostředí je předáváno kvůli tomu, abychom mohli v těle metaprocedury realizující speciální formu provádět vyhodnocování, viz aplikaci speciálních forem v programu 12.17. V těle metaprocedury je nejprve vyhodnocen argument `condition` a výsledek je navázán na symbol `result`. Podle výsledné hodnoty je rozhodnuto, zda-li se vyhodnotí `expr` nebo nepovinný poslední argument `alt-expr`. Ošetřen je i případ vrácení nedefinované hodnoty v případě, že `condition` se vyhodnotilo na „nepravda“ a v `if`-výrazu chybí náhradník, viz definici 1.31 na straně 36. Všimněte si, že pro definici speciální formy `if` jsme, mimo jiné, použili speciální metaformu `if`.

Analogicky jako speciální formu `if` bychom mohli zavést speciální formu `and` jejíž definici nalezneme v programu 12.21. Při naprogramování speciální formy jsme opět postupovali tak, že se chová jako `and`

Program 12.21. Definice speciální formy `and` v globálním prostředí.

```
(and . ,(make-specform
          (lambda (env . exprs)
            (let and-eval ((exprs exprs))
              (cond ((null? exprs) scm-true)
                    ((null? (cdr exprs)) (scm-eval (car exprs) env))
                    (else (let ((result (scm-eval (car exprs) env)))
                            (if (equal? result scm-false)
                                scm-false
                                (and-eval (cdr exprs)))))))))))
```

představená v definici 2.22 na straně 64. Metaprocedura ve svém těle používá iterativní proceduru, která postupně prochází a vyhodnocuje jednotlivé elementy předané speciální formě. Pokud se některý z elementů vyhodnotí na „nepravda“, je iterace ukončena a vrácena je pravdivostní hodnota „nepravda“. Pokud již zbývá jen poslední element, je vrácena hodnota vzniklá jeho vyhodnocením. Pokud byly zpracovány všechny prvky, je výsledek aplikace speciální formy pravdivostní hodnota „pravda“.

Kromě `and` můžeme vytvořit i primitivní proceduru `not` následovně:

```
(not . ,(make-primitive
          (lambda (elem)
            (if (equal? elem scm-false)
                scm-true
                scm-false))))
```

Jelikož je `not` procedura a nikoliv speciální forma, není ji předáváno prostředí a předaný argument je již ve své vyhodnocené podobě. Pomocí `not` a `and` můžeme vyjadřovat i disjunktivní podmínky (viz komentář v sekci 2.7). Samozřejmě, že z programátorského hlediska by bylo dobré naprogramovat rovněž i speciální formu `or`. Realizace speciální formy `or` je jedním z řešených příkladů na konci této sekce, proto ji nyní uvádět nebudeme.

V programu 12.22 jsou uvedeny definice speciálních forem `lambda`, `the-environment` a `quote`. Jak je vidět, realizace těchto forem je velmi jednoduchá. Speciální forma `lambda` vytvoří nový element typu „uživatelsky definovaná procedura“ na základě předaného prostředí, seznamu argumentů a těla. Speciální forma `the-environment` je realizována metaprocedurou, která pouze vrací aktuální prostředí (tato metaprocedura je vlastně *identita*). Analogicky speciální forma `quote` vrací předaný element (bez jeho vyhodnocení), metaprocedura realizující `quote` je tedy *projekce.*

Pro uživatelsky definované procedury můžeme vytvořit sadu selektorů, viz program 12.23. Se třemi

Program 12.22. Definice speciálních forem `lambda`, `the-environment` a `quote` v globálním prostředí.

```
(lambda . ,(make-specform
             (lambda (env args body)
              (make-procedure env args body))))

(the-environment . ,(make-specform (lambda (env) env)))

(quote . ,(make-specform (lambda (env elem) elem)))
```

Program 12.23. Definice selektorů uživatelsky definovaných procedur v globálním prostředí.

```
(procedure-environment . ,(make-primitive procedure-environment))
(procedure-arguments . ,(make-primitive procedure-arguments))
(procedure-body . ,(make-primitive procedure-body))

(environment-parent . ,(make-primitive get-pred))

(environment->list . ,(make-primitive
                        (lambda (elem)
                          (if (equal? elem scm-false)
                              scm-false
                              (get-table elem)))))
```

procedurami `procedure-environment`, `environment-parent` a `environment->list` jsme se už setkali v lekci 6. Tyto procedury vrací prostředí vzniku procedury, nadřazené prostředí daného prostředí a poslední procedura převádí tabulku na čitelný asociační seznam. V programu 12.23 máme navíc selektor `procedure-arguments`, který pro danou proceduru vrací seznam jejich argumentů. Selektor `procedure-body` pro danou proceduru vrací výraz, který je tělem procedury.

Nyní můžeme popsat, jak do prostředí primitivních definic zabudujeme primitivní procedury `apply` a `eval`. Jelikož chceme, aby `apply` pracoval s libovolným počtem argumentů, to jest ve tvaru

(`apply` ⟨*procedura*⟩ ⟨arg_1⟩ ⟨arg_2⟩ ⋯ ⟨arg_n⟩ ⟨*seznam*⟩),

zavedeme nejprve pomocnou metaproceduru `apply-collect-arguments`, která ze všech předaných argumentů ve tvaru ⟨arg_1⟩ ⋯ ⟨arg_n⟩ ⟨*seznam*⟩ sestaví (jediný) *seznam všech argumentů*. Kód této metaprocedury je v programu 12.24. V programu 12.25 jsou pak uvedeny definice procedur `eval` (procedura dvou argu-

Program 12.24. Sestavení seznamu argumentů pro obecný typ volání procedury `apply`.

```
(define apply-collect-arguments
  (lambda (args)
    (cond ((null? args) (error "APPLY: argument missing"))
          ((and (not (null? args)) (null? (cdr args)))
           (scm-list->list (car args)))
          (else (cons (car args) (apply-collect-arguments (cdr args)))))))
```

mentů z nichž druhý – reprezentující prostředí – je vždy povinný), `apply` a `env-apply` (zobecněná verze `apply`, které je jako druhý argument předáno prostředí, viz program 12.19 na straně 306).

Program 12.25. Definice `eval`, `apply` and `env-apply` v globálním prostředí.

```
(eval . ,(make-primitive
          (lambda (elem env)
            (scm-eval elem env))))

(apply . ,(make-primitive
           (lambda (proc . rest)
            (scm-apply proc (apply-collect-arguments rest)))))

(env-apply . ,(make-primitive
               (lambda (proc env . rest)
                 (scm-env-apply proc
                                env
                                (apply-collect-arguments rest)))))
```

Ostatní procedury v prostředí primitivních vazeb mohou být vytvořeny rutinně, nebudeme je tedy všechny vypisovat. Tímto čtenáře odkazujeme na zdrojové kódy interpretu jazyka Scheme, který je dodáván spolu s tímto učebním textem. Naznačme ale zhruba, jak definice vypadají. Budeme předpokládat, že v prostředí primitivních vazeb budeme mít na symbol `pi` navázánu číselnou hodnotu čísla π. Tuto vazbu bychom provedli třeba takto:

```
(pi . ,(make-number (* 4 (atan 1))))
```

Aritmetické procedury (sčítání, odčítání, zaokrouhlování a podobně) můžeme vytvořit pomocí metaprocedury `wrap-primitive`, kterou jsme již popsali v programu 12.14 na straně 302. Při definici aritmetických procedur tedy budeme postupovat následovně:

```
(* . ,(wrap-primitive *))
(+ . ,(wrap-primitive +))
(- . ,(wrap-primitive -))
(/ . ,(wrap-primitive /))
(< . ,(wrap-primitive <))
(<= . ,(wrap-primitive <=))
(= . ,(wrap-primitive =))
 ⋮
(tan . ,(wrap-primitive tan))
(truncate . ,(wrap-primitive truncate))
(zero? . ,(wrap-primitive zero?))
```

Rovněž definice konstruktorů a selektorů párů je jednoduchá. Pouze převedeme metaprocedury z programu 12.7 na straně 296 na primitivní procedury a uvedeme vazby na příslušné symboly.

```
(cons . ,(make-primitive make-pair))
(car . ,(make-primitive pair-car))
(cdr . ,(make-primitive pair-cdr))
```

Predikát testující prázdnost seznamu vytvoříme pomocí porovnání daného elementu s elementem reprezentujícím prázdný seznam:

```
(null? . ,(make-primitive
           (lambda (l)
             (expr->intern (equal? l the-empty-list)))))
```

Nyní se můžeme začít věnovat prostředí odvozených definic. Toto prostředí vytvoříme analogicky jako prostředí primitivních definice. Jediným rozdílem bude, že prostředí odvozených definic již bude mít

jako svého předka nastaveno jiné prostředí, konkrétně právě prostředí primitivních definice. Následující fragment kódu ukazuje tvar, v jakém můžeme prostředí odvozených definic zavést.

```
(define scheme-midlevel-env
  (make-env
   scheme-toplevel-env
   (assoc->env
    `(
      ⋮
      (sgn . ,(make-procedure
               scheme-toplevel-env
               (expr->intern '(x))
               (expr->intern '(if (= x 0)
                                  0
                                  (if (> x 0)
                                      1
                                      -1)))))
      ⋮
     ))))
```

V předchozím kódu jsou opět uvedeny výpustky a je zde uveden příklad definice uživatelsky definované procedury `sgn`. Element navázaný na symbol `sgn` v tomto prostředí je skutečně uživatelsky definovaná procedura, protože se jedná o element vytvořený aplikací `make-procedure`, viz program 12.11 na straně 300. Při vytvoření této uživatelsky definované procedury jsme jako prostředí vzniku předali prostředí primitivních vazeb (navázané na `scheme-toplevel-env`). Seznam argumentů procedury signum jsme vytvořili převedením metaseznamu `(x)` do jeho interní formy. Stejným způsobem jsme zapsali tělo procedury.

Ze vztahu obou prostředí je patrné, že ani v prostředí odvozených definic nemůžeme vytvářet rekurzivní procedury bez použití y-kombinátoru, protože procedura není navázána na symbol v prostředí svého vzniku. Při definici rekurzivních procedur si tedy musíme pomoci y-kombinátorem, viz sekci 9.2. Příklady rekurzivních procedur definovaných v prostředí odvozených definic najdeme v programech 12.26 (výpočet délky seznamu) a 12.27 (mapování přes jeden seznam).

Program 12.26. Procedura `length` v prostředí odvozených definic.

```
(length . ,(make-procedure
            scheme-toplevel-env
            (expr->intern '(l))
            (expr->intern
             '((lambda (y)
                 (y y l))
               (lambda (length l)
                 (if (null? l)
                     0
                     (+ 1 (length length (cdr l)))))))))
```

Poslední věcí, kterou musíme vyřešit, je implementace cyklu REPL, ve kterém poběží samotné vyhodnocování. REPL bude realizován jednoduchou iterativní metaprocedurou, která simuluje činnost vyhodnocovacího cyklu tak, jak jsme jej popsali v sekci 1.5. Viz kód uvedený v programu 12.28. Nejprve je vytvořeno nové prostředí, jehož předkem je prostředí odvozených vazeb. Toto prostředí je navázáno na symbol `init-env`. Dále se opakuje cyklus, ve kterém je vždy načten symbolický výraz, poté je převeden do interní reprezentace, vyhodnocen v prostředí navázaném na `init-env`, výsledek je vytištěn a celý cyklus se opakuje dokud není vyčerpán vstup (nebo nedojde k chybě). Na posledním řádku metaprogramu (interpretu) tedy

Program 12.27. Procedura `map` v prostředí odvozených definic.

```
(map . ,(make-procedure
          scheme-toplevel-env
          (expr->intern '(f l))
          (expr->intern
           '((lambda (y)
               (y y l))
             (lambda (map l)
               (if (null? l)
                   ()
                   (cons (f (car l)) (map map (cdr l))))))))))
```

Program 12.28. Implementace cyklu REPL.

```
(define scm-repl
  (lambda ()
    (let ((init-env (make-env scheme-midlevel-env the-empty-list)))
      (let loop ()
        (display "]=> ")
        (let ((elem (scm-read)))
          (if (not elem)
              'bye-bye
              (let ((result (scm-eval elem init-env)))
                (newline)
                (scm-print result)
                (newline)
                (newline)
                (loop)))))))))
```

spustíme metaproceduru `(scm-repl)`, která dále řídí průběh vyhodnocování. Tím jsme završili vývoj první verze našeho interpretu (další vylepšení ukážeme v dalším díle tohoto učebního textu).

Zdrojový kód našeho interpretu (včetně komentářů) nepřesahuje 600 řádků, z pohledu velikosti se tedy jedná o *velmi malý program*. Velké programy běžně přesahují stovky tisíc i miliony řádků. Například jádra operačních systémů mívají kolem pěti milionů řádků, stejně tak kancelářské balíky. Mezi „největší programy“ patří bezpečnostní software pro řízení leteckého provozu a raketové systémy. I přes to, že náš program je pozoruhodně malý, jedná se o implementaci interpretu Turingovsky úplného programovacího jazyka, tedy jazyka, který je z hlediska své vyjadřovací síly stejně silný jako běžně používané programovací jazyky (například C, C++, LISP, Pascal, . . .).

12.7 Příklady použití interpretu

V této sekci ukážeme příklady použití nově vytvořeného interpretu. Příklady budeme komentovat pouze stručně, protože všechny konstrukce jsou již čtenářům důvěrně známé. Výsledné hodnoty zobrazujeme stejně jako je výstup našeho interpretu.

Nejprve ukážeme použití a chování speciální formy `quote`:

```
quote          ⟼  Specform: #<special-form>
(quote blah)   ⟼  Symbol: blah
'blah          ⟼  Symbol: blah
''blah         ⟼  Pair: (quote blah)
```

Další příklad ukazuje kvotování seznamu:

```
(+ 1 2 3)        ⟼  Number: 6
'(+ 1 2 3)       ⟼  Pair: (+ 1 2 3)
(+ 1 (* 2 3))    ⟼  Number: 7
'(+ 1 (* 2 3))   ⟼  (+ 1 (* 2 3))
```

Speciální elementy se vyhodnocují na sebe sama, jako obvykle:

```
()    ⟼  Empty-list: ()
'()   ⟼  Empty-list: ()
#t    ⟼  Boolean: #t
'#t   ⟼  Boolean: #t
#f    ⟼  Boolean: #f
'#f   ⟼  Boolean: #f
```

Speciální formu `if` používáme obvyklým způsobem. Z posledního příkladu je vidět, že `if` se skutečně chová jako speciální forma a nikoliv jako procedura:

```
if                    ⟼  Specform: #<special-form>
(if 1 2 3)            ⟼  Number: 2
(if #f 1 2)           ⟼  Number: 2
(if #f 1)             ⟼  Undefined: #<undefined>
(if #t 1 blah-blah)   ⟼  Number: 1
```

Následující příklad ukazuje použití speciální formy `and`:

```
(and)             ⟼  Boolean: #t
(and 10)          ⟼  Number: 10
(and #f)          ⟼  Boolean: #f
(and 0 2 4)       ⟼  Number: 4
(and 3 (= 1 1))   ⟼  Boolean: #t
```

Pomocí speciální formy `lambda` vytváříme procedury:

```
lambda                       ⟼  Specform: #<special-form>
(lambda (x) (+ x 1))         ⟼  Procedure: #<user-defined-procedure>
((lambda (x) (+ x 1)) 10)    ⟼  Number: 11
```

Náš interpret umožňuje práci se speciálními formami jako s elementy prvního řádu. V následujícím příkladu je speciální forma předána jako argument proceduře. Podobnou konstrukci by nám drtivá většina interpretů jazyka Scheme vůbec neumožnila (speciální formy nejsou v existujících interpretech jazyka Scheme chápány jako elementy prvního řádu).

```
(((lambda (procedura)
    (procedura (x) (+ x 1)))
  lambda)
 10)  ⟼  Number: 11
```

Rekurzivní procedury můžeme definovat pomocí y-kombinátoru, jako třeba:

```
((lambda (y)
   (y y 6))
 (lambda (fak n)
   (if (= n 0)
       1
       (* n (fak fak (- n 1))))))  ⟼  Number: 720
```

Na symbol `map` je navázána uživatelsky definovaná procedura:

```
map                              ⟹  Procedure: #<user-defined-procedure>
(procedure-environment map)      ⟹  Environment: #<environment>
(procedure-arguments map)        ⟹  Pair: (f l)
(procedure-body map)             ⟹  Pair: ((lambda (y) (y y l)) (lambda (map ···
```

Tato uživatelsky definovaná procedura funguje standardně:

```
(map - '(1 2 3 4))                            ⟹  Pair: (-1 -2 -3 -4)
(map (lambda (x) (cons x ())) '(a b c d))     ⟹  Pair: ((a) (b) (c) (d))
(map even? '(0 1 2 3 4 5 6))                  ⟹  Pair: (#t #f #t #f #t #f #t)
(map (lambda (x) (<= x 3)) '(0 1 2 3 4 5 6))  ⟹  Pair: (#t #t #t #t #f #f #f)
```

Následující příklad ukazuje použití `map` a rekurzivní procedury počítající faktoriál:

```
(map
 (lambda (n)
   ((lambda (y)
      (y y n))
    (lambda (fak n)
      (if (= n 0)
          1
          (* n (fak fak (- n 1)))))))
 '(0 1 2 3 4 5 6 7 8 9))  ⟹  Pair: (1 1 2 6 24 120 720 5040 40320 362880)
```

Následující dvě ukázky demonstrují použití libovolných a nepovinných argumentů.

```
((lambda list (map - list)) 1 2 3 4 5 6)          ⟹  Pair: (-1 -2 -3 -4 -5 -6)
((lambda (x y . list)
   (cons x (cons y (map - list)))) 1 2 3 4 5 6)   ⟹  Pair: (1 2 -3 -4 -5 -6)
```

Pomocí `procedure-environment` můžeme získat prostředí vzniku procedury:

```
(procedure-environment
 ((lambda (x)
    (lambda (y) (+ x y)))
  10))  ⟹  Environment: #<environment>
```

Vazby v prostředí můžeme vypsat pomocí `environment->list`:

```
(environment->list
 (procedure-environment
  ((lambda (x)
     (lambda (y) (+ x y)))
   10)))  ⟹  Pair: ((x . 10))
```

V následujícím příkladu je získáno a vypsáno počáteční prostředí (jeho tabulka vazeb je prázdná). V druhém případě je pak vypsána tabulka vazeb v předchůdci aktuálního prostředí, což je prostředí odvozených definic.

```
(the-environment)                        ⟹  Environment: #<environment>
(environment->list (the-environment))    ⟹  Empty-list: ()
(environment->list
 (environment-parent
  (the-environment)))  ⟹  Pair: ((sgn . #<user-defined-procedure>)
                                 (length . #<user-defined-procedure>)
                                 (map . #<user-defined-procedure>))
```

Procedura `eval` je možné používat pouze se dvěma argumenty:

```
(eval '(+ 1 2) (the-environment))  ⟹  Number: 3
```

V následující ukázce je vyhodnocen seznam `(length (quote (a b c)))` ve třech různých prostředích. V posledním případě dojde při vyhodnocení k chybě, protože v prostředí primitivních definic není procedura `length` definovaná.

```
(eval '(length '(a b c)) (the-environment))        ⟹  Number: 3
(eval '(length '(a b c)) (environment-parent
                          (the-environment)))      ⟹  Number: 3
(eval '(length '(a b c)) (environment-parent
                          (environment-parent
                           (the-environment))))    ⟹  Error (symbol not bound)
```

Další příklad ukazuje vyhodnocení seznamu `'(* x x)` v prostředí vzniku procedury:

```
(eval '(* x x)
      (procedure-environment
       ((lambda (x)
          (lambda (y) (+ x y)))
        10)))  ⟹  Number: 100
```

Proceduru `apply` je možné použít i s více jak dvěma argumenty:

```
(apply + '(1 2 3 4))           ⟹  Number: 10
(apply + 1 2 '(3 4))           ⟹  Number: 10
(apply + 1 2 3 4 '())          ⟹  Number: 10
(apply map - '((1 2 3 4)))     ⟹  Pair: (-1 -2 -3 -4)
(apply map - '(1 2 3 4) '())   ⟹  Pair: (-1 -2 -3 -4)
```

V dalším příkladu využijeme faktu, že na symbol `pi` máme navázanou hodnotu čísla π:

```
pi  ⟹  Number: 3.141592653589793
```

Při vyhodnocení následujícího výrazu nehraje globální vazba `pi` roli, protože interpret používá lexikální rozsah platnosti:

```
(((lambda (pi)
    (lambda (x)
      (+ x pi)))
  10)
 20)  ⟹  Number: 30
```

Totéž platí pro explicitní aplikaci:

```
(apply ((lambda (pi)
          (lambda (x)
            (+ x pi)))
        10)
       20
       '())  ⟹  Number: 30
```

V následující ukázce jsme provedli aplikaci uživatelsky definované procedury, přitom jsme „dočasně nastavili“ předka této procedury na lokální prostředí v němž je na `y` navázaná hodnota `100`:

```
(env-apply (lambda (x) (+ x y))
           ((lambda (y) (the-environment)) 100)
           20 '())  ⟹  Number: 120
```

Shrnutí

Zabývali jsme se automatickým přetypováním a generickými procedurami. Pro generické procedury jsme zavedli jednoduchou metodu jejich aplikace prostřednictvím vyhledávání metod pomocí vzorů uvedených v tabulkách. Na konkrétním příkladu generických procedur jsme ukázali jejich praktické použití. Dále

jsme se seznámili s konceptem manifestovaných typů. Pomocí manifestovaných typů jsme implementovali reprezentaci elementů jazyka Scheme. S jejich využitím jsme dále naprogramovali vyhodnocovací proces. Nakonec jsme zkompletovali interpret čistě funkcionální podmnožiny jazyka Scheme.

Pojmy k zapamatování

- přetypování, implicitní/explicitní přetypování,
- automatické přetypování, koerce,
- generická procedura, tabulka generických procedur,
- manifestované typy, visačky, tagy,
- metajazyk, metainterpret, metaelement, metaprocedura.

Nově představené prvky jazyka Scheme

- procedury `number->string` a `string-append`

Kontrolní otázky

1. *Co jsou to generické procedury?*
2. *Jaké jsou výhody a nevýhody automatického přetypování?*
3. *Co jsou to manifestované typy?*
4. *Jak jsme v naší implementaci jazyka Scheme reprezentovali jednotlivé elementy jazyka?*
5. *Kolik jsme uvažovali počátečních prostředí a proč?*
6. *Jaký je rozdíl mezi jazykem/interpretem a metajazykem/metainterpretem?*

Cvičení

1. Bez použití interpretu jazyka Scheme zjistěte, jak se vyhodnotí následující výrazy používající generické `+` tak, jak jsme jej představili v sekci 12.1:
```
(+ 2/3 "." 0.5)                          ⟼
(+ (- 1) "x" (- 2))                      ⟼
(+ 1 2 "+" 3 4)                          ⟼
(+ 2 (+ 2 "3") 10)                       ⟼
(+ 2 (log (exp 2)))                      ⟼
(apply + '("a" "b" (+ 1 2) 2))           ⟼
(apply + (map number->string '(1 2 3)))  ⟼
```
2. Obohaťte vytvořený interpret jazyka Scheme o primitivní proceduru `foldr`. To jest `foldr` by měla být ve vytvořeném interpretu k dispozici přímo v počátečním prostředí a mělo by se jednat o primitivní proceduru, tedy nikoliv o uživatelsky definovanou proceduru. Naprogramujte tuto primitivní proceduru tak, aby mohla pracovat s libovolným množstvím seznamů (vždy však alespoň s jedním).
3. Analogicky proveďte obohacení interpretu o primitivní proceduru `foldl`.
4. Obohaťte vytvořený interpret jazyka Scheme o speciální formu `or`. Tuto speciální formu naprogramujte tak, aby se chovala přesně jako speciální forma `or` v jazyku Scheme, viz [R5RS].
5. Obohaťte vytvořený interpret jazyka Scheme o speciální formu `let` (nepojmenovanou verzi).
6. Obohaťte vytvořený interpret jazyka Scheme o pojmenovaný `let`. Jelikož v jazyku nemáme k dispozici `define`, musíme si při programování pojmenovaného `let` pomocí y-kombinátoru. Speciální formu ale vytvořte tak, abychom při rekurzivní aplikaci nemuseli předávat proceduru prostřednictvím jejího prvního argumentu. Vytvořený pojmenovaný `let` by se tedy z uživatelského pohledu měl chovat jako pojmenovaný `let` popsaný ve [R5RS].
7. Obohaťte interpret jazyka Scheme o speciální formu `dyn-apply`, která bude provádět aplikaci procedur podobně jako `apply`, ale při aplikaci procedur pomocí `dyn-apply` se budou při vyhodnocování těla procedury hledat vazby symbolů ve smyslu dynamického rozsahu platnosti, tedy v prostředí aplikace procedury. Na následujícím příkladu je vidět rozdíl použití `apply` a `dyn-apply`:

```
((lambda (x)
   (apply ((lambda (x)
             (lambda (y)
               (+ x y))) 100)
          '(20)))
 1000)  ⟼  120

((lambda (x)
   (dyn-apply ((lambda (x)
                 (lambda (y)
                   (+ x y))) 100)
               '(20)))
   1000)  ⟼  1020
```

8. Obohaťte interpret o novou speciální formu `delta`, která je po syntaktické stránce shodná s formou `lambda`. Vyhodnocením δ-výrazu vznikne speciální typ uživatelsky definované procedury. Při aplikaci procedury vzniklé vyhodnocením δ-výrazů je uplatněn dynamický rozsah platnosti. Zavedení δ-výrazů do jazyka nám tedy umožní vytvářet uživatelsky definované procedury, jejichž vyhodnocování probíhá v souladu s dynamickým rozsahem platnosti. Viz příklad použití a rozdíl v použití speciálních forem `lambda` a `delta`.

```
((lambda (x)
   (((lambda (x)
       (lambda (y)
         (+ x y))) 100) 20)) 1000)  ⟼  120

((lambda (x)
   (((lambda (x)
       (delta (y)
         (+ x y))) 100) 20)) 1000)  ⟼  1020
```

Při implementaci využijte svých znalostí ze sekce 2.6.

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

```
(environment->list
 ((lambda (x) (the-environment)) 10))  ⟼

((lambda (x)
   (eval '(- x) (the-environment)))
 100)  ⟼

(eval '(+ x 1)
      ((lambda (x) (the-environment)) 10))  ⟼

((lambda (x)
   ((lambda (x)
      (eval '(cons x (quote x)) (procedure-environment x)))
    (lambda (x) 'blah)))
 1000)  ⟼

(environment->list
 (procedure-environment
  ((lambda (y)
     (lambda (x)
       (+ x 1)))
   100)))  ⟼
```

Úkoly k textu

1. Vytvořte systém generických procedur `+`, `-`, `*`, `modulo`, `quotient` a `=` sloužících pro práci s čísly a polynomy. Polynomy (jedné proměnné) reprezentujte seznamy jejich koeficientů tak, jak jsme to dělali v sekci 8.6. Základní aritmetické procedury `+`, `-`, `*` slouží k provádění aritmetických operací s čísly a polynomy. Například tedy pro `*` by to mělo pokrývat součin čísel, součin polynomů a součin čísla s polynomem. Procedury `modulo` a `quotient` by měly vracet podíl čísel nebo polynomů a zbytek po dělení číslem nebo polynomem. Predikát `=` by měl sloužit k porovnávání čísel a polynomů. Při implementaci si pomozte tím, že čísla jsou de facto konstantní polynomy. Kde lze, provádějte automatické přetypování.

2. Po dokončení předchozího úkolu naprogramujte procedury pro sčítání a násobení matic. Pokud jsme provedli správnou implementaci v předchozím bodě, měla by vaše implementace operací s maticemi umožňovat pracovat s maticemi jejichž prvky jsou čísla nebo polynomy. Důkladně vaši implementaci otestujte.

3. Obohaťte interpret jazyka Scheme o speciální formy `cond` a `let*`. Obě speciální formy naprogramujte tak, aby se chovaly stejně, jak jsme je představili v lekcích 2 a 3. Implementace obou speciálních forem důkladně otestujte a přesvědčte se o jejich správnosti.

4. Obohaťte interpret jazyka Scheme o speciální formu `named-lambda`, která je po syntaktické stránce totožná se speciální formou `lambda`. Speciální forma `named-lambda` slouží k vytváření procedur stejně jako speciální forma `lambda`, navíc však v těle procedury vytvořené pomocí `named-lambda` je vždy na symbol `self` navázaná samotná procedura. Pomocí `self` je tedy možné provádět rekurzivní volání, viz následující příklad použití.

```
(map (named-lambda (n)
       (if (= n 0)
           1
           (* n (self (- n 1)))))
     '(0 1 2 3 4 5 6 7 8))  ⟹  (1 1 2 6 24 120 720 5040 40320)
```

5. V předchozích příkladech na procvičení jsme viděli dva přístupy k aplikaci procedur řídící se dynamickým rozsahem platnosti. První metodou bylo použití explicitní aplikace pomocí `dyn-apply`. Druhým způsobem bylo zavedení δ-výrazů, pomocí nichž vznikaly procedury aplikované vždy ve smyslu dynamického rozsahu platnosti. Ani jedno z těchto řešení není úplně ideální, protože v některých případech by se mohlo hodit, abychom v těle jedné procedury uvažovali většinu symbolů ve smyslu lexikálního rozsahu platnosti (zajímat se budeme o lexikální vazby symbolů) a některé symboly ve smyslu dynamického rozsahu platnosti (zajímat se budeme o dynamické vazby symbolů).

   Obohaťte interpret o speciální formu `dynamic`, jejímž jediným argumentem bude symbol. Výsledkem vyhodnocení `(dynamic x)` (v daném prostředí) je dynamická vazba symbolu `x`, tedy vazba, která je hledána v dynamicky nadřazených prostředích (pokud není nalezena v lokálním prostředí), viz sekci 2.6. Následující ukázka demonstruje použití `dynamic`.

```
((lambda (x)
   (((lambda (x)
       (lambda (y)
         (+ x y))) 100) 20)) 1000)  ⟹  120

((lambda (x)
   (((lambda (x)
       (lambda (y)
         (+ (dynamic x) y))) 100) 20)) 1000)  ⟹  1020
```

Řešení ke cvičením

1. `"2/3.0.5","-1x-2","12+7","22310",4.0,Error: No method for these types,"123"`
2. Vytvoříme pomocnou proceduru `scm-foldr` následovně:
```
(define scm-foldr
  (lambda (f basis . lists)
    (if (scm-null? (car lists))
        basis
        (scm-apply
         f
         (append (map pair-car lists)
                 (list (apply
                        scm-foldr
                        f basis (map pair-cdr lists))))))))
```
Dále je nutné do tabulky vazeb prostředí „toplevel“ přidat záznam:
```
(foldr . ,(make-primitive scm-foldr))
```
3. Vytvoříme pomocnou proceduru `scm-foldl` následovně:
```
(define scm-foldl
  (lambda (f basis . lists)
    (let iter ((lists lists)
               (accum basis))
      (if (scm-null? (car lists))
          accum
          (iter (map pair-cdr lists)
                (scm-apply f (append (map pair-car lists)
                                     (list accum))))))))
```
Dále je nutné do tabulky vazeb prostředí „toplevel“ přidat záznam:
```
(foldl . ,(make-primitive scm-foldl))
```
4. Vytvoříme pomocnou proceduru `scm-or` následovně:
```
(define scm-or
  (lambda (env . exprs)
    (let or-eval ((exprs exprs))
      (cond ((null? exprs) scm-false)
            ((null? (cdr exprs)) (scm-eval (car exprs) env))
            (else (let ((result (scm-eval (car exprs) env)))
                    (if (equal? result scm-false)
                        (or-eval (cdr exprs))
                        result)))))))
```
Dále je nutné do tabulky vazeb prostředí „toplevel“ přidat záznam:
```
(or . ,(make-specform scm-or))
```
5. Vytvoříme pomocnou proceduru `map-scm-list` pro mapování přes seznam ve vnitřní reprezentaci:
```
(define map-scm-list
  (lambda (f l)
    (convert-list scm-null? pair-car pair-cdr
                  make-pair the-empty-list f l)))
```
Dále vytvoříme pomocnou proceduru `scm-let`:
```
(define scm-let
  (lambda (env bindings body)
    (let ((proc (make-procedure
                 env
                 (map-scm-list pair-car bindings)
                 body)))
      (scm-apply proc
                 (map-scm-list->list
```

```
                (lambda (elem)
                  (scm-eval (pair-car (pair-cdr elem)) env))
                bindings)))))
```

Dále je nutné do tabulky vazeb prostředí „toplevel“ přidat záznam:

```
(let . ,(make-specform scm-let))
```

6. Vytvoříme nejprve pomocný konstruktor seznamu v interní reprezentaci:

```
(define make-list
  (lambda args
    (if (null? args)
        the-empty-list
        (make-pair (car args)
                   (apply make-list
                          (cdr args))))))
```

Pro programování pomocné procedury, která bude v konstruovaném interpretu realizovat speciální formu „pojmenovaný `let`“ si musíme uvědomit, že kód ve tvaru

```
(let ⟨jméno⟩ ((⟨symbol_1⟩ ⟨výraz_1⟩)
              (⟨symbol_2⟩ ⟨výraz_2⟩)
                 ⋮
              (⟨symbol_n⟩ ⟨výraz_n⟩))
  ⟨tělo⟩)
```

stačí nahradit kódem ve tvaru

```
((lambda (y)
   (y y ⟨výraz_1⟩ ⟨výraz_2⟩ ··· ⟨výraz_n⟩))
 (lambda (⟨jméno⟩ ⟨symbol_1⟩ ⟨symbol_2⟩ ··· ⟨symbol_n⟩)
   ((lambda (⟨jméno⟩) ⟨tělo⟩)
    (lambda (⟨symbol_1⟩ ⟨symbol_2⟩ ··· ⟨symbol_n⟩)
      (⟨jméno⟩ ⟨jméno⟩ ⟨symbol_1⟩ ⟨symbol_2⟩ ··· ⟨symbol_n⟩))))))
```

Podle tohoto postupu naprogramujeme pomocnou proceduru `scm-named-let`:

```
(define scm-named-let
  (lambda (env . args)
    (if (not (scm-symbol? (car args)))
        (apply scm-let env args)
        (let* ((name (car args))
               (bindings (cadr args))
               (body (caddr args))
               (y (make-symbol 'y))
               (y-comb (make-procedure
                        env
                        (make-list y)
                        (make-pair y
                          (make-pair y
                            (map-scm-list
                             (lambda (elem)
                               (scm-eval (pair-car (pair-cdr elem)) env))
                             bindings)))))
               (proc (make-procedure
                      env
                      (make-pair
                       name
                       (map-scm-list pair-car bindings))
                      (make-list
                       (make-list (make-symbol 'lambda)
                                  (make-list name)
                                  body)
```

```
                        (make-list (make-symbol 'lambda)
                                   (map-scm-list pair-car bindings)
                                   (make-pair name
                                     (make-pair name
                                         (map-scm-list pair-car bindings))))))))
          (scm-apply y-comb (list proc))))))
```

Dále je nutné do tabulky vazeb prostředí „toplevel“ přidat záznam:

```
(let . ,(make-specform scm-named-let))
```

7. Vytvoříme pomocnou proceduru `scm-dyn-apply` následovně:

```
(define scm-dyn-apply
  (lambda (env proc . rest)
    (scm-eval
     (make-pair (make-symbol 'env-apply)
                (make-pair
                 proc
                 (make-pair
                  env
                  (let iter ((ar rest))
                    (if (null? (cdr ar))
                        (make-pair (car ar) the-empty-list)
                        (make-pair (car ar)
                                   (iter (cdr ar))))))))
     env)))
```

Dále je nutné do tabulky vazeb prostředí „toplevel“ přidat záznam:

```
(dyn-apply . ,(make-specform scm-dyn-apply))
```

8. Nejprve vytvoříme datovou reprezentaci nového typu elementu:

```
(define make-dyn-procedure
  (let ((make-physical-procedure (curry-make-elem 'dyn-procedure)))
    (lambda (args body)
      (make-physical-procedure (list args body)))))

(define dyn-procedure-arguments (lambda (proc) (car (get-data proc))))
(define dyn-procedure-body (lambda (proc) (cadr (get-data proc))))

(define scm-user-dyn-procedure? (curry-scm-type 'dyn-procedure))
```

Upravíme `scm-procedure?` tak, aby brala ohled i na nový typ procedury:

```
(define scm-procedure?
  (lambda (elem)
    (or (scm-primitive? elem)
        (scm-user-procedure? elem)
        (scm-user-dyn-procedure? elem))))
```

Dále přidáme do tabulky vazeb prostředí „toplevel“ záznam:

```
(delta . ,(make-specform
           (lambda (env args body)
             (make-dyn-procedure args body))))
```

Upravíme `scm-eval` tak, aby se při volání `scm-apply` předávalo aktuální prostředí jako nový poslední argument. Kritický fragment kódu by měl vypadat takto:

```
  ⋮
((scm-procedure? f)
 (scm-apply f
            (map-scm-list->list
             (lambda (elem)
               (scm-eval elem env))
             args)
```

```
            env))
              ⋮
```

Dále upravíme `scm-env-apply` a `scm-apply` následovně tak, že do nich přidáme větve ošetřující průběh aplikaci v případě procedur vzniklých vyhodnocením δ-výrazů.

```
(define scm-env-apply
  (lambda (proc env args)
    (cond ((scm-primitive? proc) (apply (get-data proc) args))
          ((scm-user-procedure? proc)
           (scm-eval (procedure-body proc)
                     (make-env env
                               (make-bindings (procedure-arguments proc)
                                              args))))
          ((scm-user-dyn-procedure? proc)
           (scm-eval (dyn-procedure-body proc)
                     (make-env env
                               (make-bindings (dyn-procedure-arguments proc)
                                              args))))
          (else (error "APPLY: Expected procedure")))))

(define scm-apply
  (lambda (proc args . env)
    (cond ((scm-primitive? proc) (scm-env-apply proc #f args))
          ((scm-user-procedure? proc)
           (scm-env-apply proc (procedure-environment proc) args))
          ((scm-user-dyn-procedure? proc)
           (scm-env-apply proc (car env) args))
          (else (error "APPLY: Expected procedure")))))
```

9. `((x . 10)), -100, 11, (1000 . x), ((y . 100))`

Reference

[SICP] Abelson H., Sussman G. J.: *Structure and Interpretation of Computer Programs.*
http://mitpress.mit.edu/sicp/full-text/book/book.html
The MIT Press, Cambridge, Massachusetts, 2nd edition, 1986. ISBN 0–262–01153–0.

[BW88] Bird R., Wadler P.: *Introduction to Functional Programming.*
Prentice Hall, Englewood Cliffs, New Jersey, 1988. ISBN 0–13–484197–2.

[Ch36] Church A.: An unsolvable problem of elementary number theory.
American Journal of Mathematics **58**(1936), 345–363.

[Ch41] Church A.: *The Calculi of Lambda Conversion.*
Princeton University Press, Princeton (NJ, USA), 1941.

[Di68] Dijkstra E. W.: Go To Statement Considered Harmful.
Communications of the ACM **11**(3)(1968), 147–148.

[Dy96] Dybvig R. K.: *The Scheme Programming Language.*
http://www.scheme.com/tspl3/
The MIT Press, Cambridge, Massachusetts, 3rd edition, 2003. ISBN 0–262–54148–3.

[DC06] Dybvig K., Clinger W. a kol.: *R^{6}RS Status Report.*
http://www.schemers.org/Documents/Standards/Charter/

[F3K01] Felleisen M., Findler R. B., Flatt M., Krishnamurthi S.:
How to Design Programs: An Introduction to Computing and Programming.
http://www.htdp.org/
The MIT Press, Cambridge, Massachusetts, 2001.

[FF96a] Friedman D. P., Felleisen M.: *The Little Schemer.*
MIT Press, 4th edition, 1996.

[FF96b] Friedman D. P., Felleisen M.: *The Seasoned Schemer.*
MIT Press, 1996.

[Gi97] Giloi W. K.: Konrad Zuse's Plankalkül: The First High-Level "non von Neumann" Programming Language. *IEEE Annals of the History of Computing* **19**(2)1997, 17–24.

[Ho01] Hopcroft J. E. a kol.: *Introduction to Automata Theory, Languages, and Computation,*
Addison-Wesley, 2001.

[R5RS] Kelsey R., Clinger W., Rees J. (editoři):
Revised^5 Report on the Algorithmic Language Scheme,
http://www.schemers.org/Documents/Standards/R5RS/
Higher-Order and Symbolic Computation **11**(1)1998, 7–105;
ACM SIGPLAN Notices **33**(9)1998, 26–76.

[Ko97] Kozen D. C.: *Automata and Computability,*
Springer, 1997

[Kr06] Krishnamurthi S.: *Programming Languages: Application and Interpretation.*
http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/

[Ll87] Lloyd, J. W.: *Foundations of Logic Programming*.
Springer-Verlag, New York, 2nd edition, 1987.

[ML95] Manis V. S., Little J. J.: *The Schematics of Computation.*
Prentice Hall, Englewood Cliffs, New Jersey, 1995. ISBN 0–13–834284–9.

[MC60] McCarthy J.: Recursive Functions of Symbolic Expressions and Their Computation by Machine.
Communications of the ACM **3**(4)1960, 184–195.

[MC67] McCarthy J.: A Basis for a Mathematical Theory of Computation.
Obsaženo ve: Braffort P., Hirschberg D. (editoři), *Computer Programming and Formal Systems.*
North-Holland, 1967.

[vN46] von Neumann J.: The principles of large-scale computing machines.
Ann. Hist. Comp **3**(3)1946.

[NS97] Nerode A., Shore A. R.: *Logic for Applications.*
Springer-Verlag, New York, 2nd edition, 1997.

[NM00] Nilson U., Maluszynski J.: *Logic, programming and Prolog.*
http://www.ida.liu.se/~ulfni/lpp/

[PeSa58] Perlis A. J., Samelson K.: Preliminary Report – International Algorithmic Language.
Communications of the ACM **1**(12)1958, 8–22.

[Pe81] Perlis A. J.: The American side of the development of ALGOL.
Obsaženo ve: Wexelblat R. L. (editor): *History of Programming Languages.* Academic Press, 1981.

[Qu96] Queinnec C.: *Lisp in Small Pieces.*
Cambridge University Press, 1996.

[Ro63] Robinson J. A.: Theorem-Proving on the Computer.
Journal of the ACM **10**(2)(1963), 163–174.

[Ro65] Robinson J. A.: A Machine-Oriented Logic Based on the Resolution Principle.
Journal of the ACM **12**(1)(1965), 24–41.

[Ro00] Rojas R. a kol.: *Plankalkül: The First High-Level Programming Language and its Implementation*.
http://www.zib.de/zuse/Inhalt/Programme/Plankalkuel/Plankalkuel-Report/Plankalkuel-Report.htm
Institut für Informatik, Freie Universität Berlin, Technical Report B-3/2000.

[SS86] Shapiro E., Sterling S.: *The Art of Prolog.*
The MIT Press, Cambridge, Massachusetts, 1986.

[Si04] Sitaram D.: *Teach Yourself Scheme in Fixnum Days.*
http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html
text je autorem postupně aktualizován, 1998–2004.

[SF94] Springer G., Friedman D. P.: *Scheme and the Art of Programming.*
The MIT Press, Cambridge, Massachusetts, 1994. ISBN 0–262–19288–8.

[St76] Steele G. L.: *LAMBDA: The Ultimate Declarative*
http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-379.pdf
AI Memo **379**, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1976.

[SS76] Steele G. L., Sussman G. J.: *LAMBDA: The Ultimate Imperative*
http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-353.pdf
AI Memo **353**, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1976.

[SS78] Steele G. L., Sussman G. J.: *The Revised Report on SCHEME: A Dialect of LISP.*
http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-452.pdf
AI Memo **452**, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1978.

[SS75] Sussman G. J., Steele G. L.: *SCHEME: An Interpreter for Extended Lambda Calculus.*
http://repository.readscheme.org/ftp/papers/ai-lab-pubs/AIM-349.pdf
AI Memo **349**, MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, 1975.

[Wec92] Wechler W.: *Universal Algebra for Computer Scientists.*
Springer-Verlag, Berlin Heidelberg, 1992.

[Zu43] Zuse K.: *Ansätze einer Theorie des allgemeinen Rechnens unter besonderer Berücksichtigung des Aussagenkalküls und dessen Anwendung auf Relaisschaltungen.*
Nepublikovaný rukopis, Zuse Papers 045/018, 1943.

[Zu72] Zuse K.: Der Plankalkül.
http://www.zib.de/zuse/English_Version/Inhalt/Texte/Chrono/40er/Pdf/0233.pdf
Berichte der Gesellschaft für Mathematik und Datenverarbeitung. Nr. 63, Sankt Augustin, 1972.

A Seznam vybraných programů

B Seznam obrázků