Vysoké učení technické v Brně

Fakulta informačních technologií

Teorie programovacích jazyků

Interprety a předávání parametrů

16. 6. 2003 **Luboš Lorenc**

Obsah

1 Interprety

V této kapitole postupně vytvoříme jednoduchý interpret používající základní sémantiku mnoha moderních programovacích jazyků. Začneme nejjednodušší možnou verzí, která dokáže interpretovat pouze literály, proměnné a aplikace. Potom budeme postupně v jednotlivých krocích přidávat další formy.

1.1 Jednoduchý interpret

Jednou z důležitých částí specifikace jakéhokoliv programovacího jazyka je množina hodnot, se kterými má daný jazyk manipulovat. Nás budou nyní zajímat dvě podmnožiny této množiny. A to množina *vyjadřovaných hodnot* a množina *označovaných hodnot*. Vyjadřované hodnoty označují možné výsledky výrazů a označované hodnoty jsou hodnoty vázané k proměnným. Vyjadřované hodnoty tedy obvykle představují například čísla, řetězce, znaky a podobně. Označované hodnoty pak obvykle chápeme jako proměnné, respektive jako paměťové buňky obsahující nějaké hodnoty.

Pro jednoduchost bude náš jazyk obsahovat pouze dva typy vyjadřovaných hodnot - celá čísla a procedury. V této chvíli nebudeme rozlišovat mezi označovanými a vyjadřovanými hodnotami. Můžeme tedy uvést vztahy:

Vyjadřovaná hodnota = Číslo + Procedura
Označovaná hodnota = Číslo + Procedura

Nyní potřebujeme ještě rozlišit dva druhy jazyků: *Definovaný jazyk* je jazyk, který specifikujeme vytvářeným interpretem a *definující jazyk* je jazyk, v němž píšeme (vytváříme) interpret. Než přistoupíme ke konstrukci interpretu, uvedeme si ještě konkrétní a abstraktní syntaxi definovaného jazyka:

```
<exp>      ::= <integer-literal>                          lit (datum)
             | <varref>
             | <operator> <operands>                      app (rator rands)
<operator> ::= <varref> | (<exp>)
<operands> ::= ()
             | (<operand>) {,<operand>}*)
<operand>  ::= <exp>
<varref>   ::= <var>                                      varref (var)
```

Abstraktní syntaktické stromy jsou poté vytvářeny ze záznamů, které obsahují definice typů a jsou založeny na abstraktních syntaktických jménech daných gramatikou (konkrétní syntaxí).

```
(define-record lit (datum))
(define-record varref (var))
(define-record app (rator rands))
```

V této chvíli již můžeme přistoupit ke konstrukci interpretu.

Příklad 1.1 - Jednoduchý interpret s větvením pro několik případů

```
(define eval-exp
  (lambda (exp)
    (variant-case exp
      (lit (datum) datum)
      (varref (var) (apply-env init-env var))
      (app (rator rands)
        (let ((proc (eval-exp rator)) (args (eval-rands rands)))
          (apply-proc proc args)))
      (else (error "Invalid abstract syntax: " exp)))))

(define eval-rands
  (lambda (rands)
    (map eval-exp rands)))
```

Symboly `lit`, `varref`, `app`, `rator` a `rand`(`s`) jsou zkratkami slov literál, odkaz na proměnnou (variable reference), aplikace, operátor a operand(y).

Pole `rands` záznamu `app` obsahuje seznam abstraktních syntaktických stromů pro operandy aplikace. Hlavní procedura `eval-exp` zpracuje abstraktní syntaktický strom výrazu, který dostane jeho parametr a vrátí jeho vypočtenou hodnotu. Podívejme se blíže na činnost procedury `eval-exp`. Pokud `exp` je literál, je navrácen tento literál. Pokud je `exp` uzel, který reprezentuje proměnnou, hodnota výrazu bude hodnota vázaná k této proměnné. Nyní ale potřebujeme rozhodnout, odkud chceme získat tuto hodnotu. K tomu potřebujeme poskytnout nějaké *prostředí*. Tedy, potřebujeme nějakou konečnou funkci, která vezme daný symbol a vrátí příslušnou hodnotu, kterou tento symbol popisuje v daném prostředí. V našem jazyce máme (pro tuto chvíli) pouze jediné prostředí, a to počáteční prostředí `init-env`.

Nyní nadefinujeme abstraktní datový typ (ADT) pro prostředí:

Příklad 1.2 - definice ADT prostředí

```
(define the-empty-env (create-empty-ff))
(define extend-env (extend-ff*))
(define apply-env apply-ff)
```

Pokud vyčíslujeme nějakou proměnnou, interpret potřebuje pouze najít její vazbu v `init-env`. Proto je tedy v našem interpretu uvedena následující větev:

```
(varref (var) (apply-env init-env var))
```

Tato větev umožňuje zpracování případu aplikace. Protože operátor může být jakýkoliv výraz, musíme ho nejprve vyčíslit, abychom získali proceduru, která má být zavolána. Toho je v našem interpretu jednoduše dosaženo pomocí rekurzivního volání (`eval-exp-rator`). Tak jak bývá obvyklé u většiny programovacích jazyků, i jazyk, který zpracovává náš interpret používá vyhodnocování v aplikačním pořadí - operandy každé operace jsou vyhodnoceny před voláním této procedury. Pro vyčíslení seznamu operandů a vytvoření seznamu argumentů (hodnot operandů) je volána pomocná procedura `eval-rands` používající `map` k rekurzivnímu vyvolávání `eval-exp` pro každý operand.

Jakým způsobem bude aplikace provedena závisí na reprezentaci procedur. Tato závislost byla izolována do procedury `apply-proc`, čímž bylo umožněno kompletní vytvoření procedury `eval-exp` bez ohledu na způsob reprezentace procedur.

Nyní se podíváme na reprezentaci procedur. Na začátku máme pouze jediný druh procedur: *Primitivní procedury*, které jsou podporovány přímo základní implementací. Abychom byli přesní, použijeme gramatiku v BNF pro specifikaci dat, se kterými má pracovat `apply-proc`:

```
<Procedure> ::= <prim-op>                    prim-proc (prim-op)
<prim-op>   ::= <addition>                   +
              | <subtraction>                -
              | <multiplication>             *
              | <increment>                  add1
              | <decrement>                  sub1
```

Tato specifikace určuje, že existují pouze primitivní procedury, jež mohou provádět právě jednu z pěti uvedených operací. Každá takováto procedura může být reprezentována pomocí záznamu `(define-record prim-proc (prim-op))`, což nám později umožní snadnější odlišení od jiných typů procedur. Jediná položka uložená v záznamu `prim-proc` je symbol označující použitý operátor. Procedury `make-prim-proc` a `apply-proc` definují abstraktní datový typ pro procedury.

Jediné, co v této chvíli vykonává procedura `apply-proc` je ověření, zda její první parametr je primitivní procedura, a následné vyvolání `apply-prim-op`.

Příklad 1.3 - Část interpretu - procedury *apply-proc* a *apply-prim-op*

```
(define apply-proc
  (lambda (proc args)
    (variant-case proc
      (prim-proc (prim-op) (apply-prim-op prim-op args))
      (else (error "Invalid procedure: " proc)))))

(define apply-prim-op
  (lambda (prim-op args)
    (case prim-op
      ((+) (+ (car args) (cadr args)))
      ((-) (- (car args) (cadr args)))
      ((/) (/ (car args) (cadr args)))
      ((add1) (+ (car args) 1))
      ((sub1) (- (car args) 1))
      (else (error "Invalid prim-op name: " prim-op)))))
```

Nyní nám ještě zbývá vytvořit počáteční prostředí definující symboly primitivních operátorů a máme vytvořený kompletní jednoduchý interpret.

Příklad 1.4 - Počáteční prostředí

```
(define prim-op-names '(+ - * add1 sub1))

(define init-env
  (extend-env prim-op-names (map make-prim-proc prim-op-names)
    the-empty-env))
```

1.2 Podmíněné vyhodnocování

V této a následujících kapitolách budeme do definovaného jazyka postupně přidávat nové vlastnosti. Pro každou vlastnost přidáme do gramatiky pravidlo pro nonterminál `<exp>`, specifikujeme abstraktní syntaxi pro toto pravidlo a nakonec vložíme příslušnou větev do `eval-exp` pro zpracování nového typu uzlu abstraktního syntaktického stromu. Nejdříve přidáme podmíněné vyhodnocování výrazů s následující konkrétní a abstraktní syntaxí:

```
<exp> ::= if <exp> then <exp> else <exp>                    if (test-exp then-exp else-exp)
```

Abyhom nemuseli pro boolovské výrazy definovat nový datový typ, budeme nulu považovat za nepravdivou hodnotu a všechny ostatní hodnoty za pravdivé. K tomu účelu využijeme proceduru `true-value?`, která zavádí potřebnou abstrakci.

Příklad 1.5 - Procedura *true-value?*

```
(define true-value?
  (lambda (x)
    (not (zero? x))))
```

Požadovaného chování interpretu dosáhneme pouhým přidáním následující větve pro zpracování podmíněných příkazů do našeho interpretu.

Příklad 1.6 - Větev vyhodnocující podmíněné příkazy

```
(if (test-exp then-exp else-exp)
  (if (true-value? (eval-exp test-exp))
                   (eval-exp then-exp)
                   (eval-exp else-exp)))
```

Z uvedeného kódu je jasně patrné, jak silně závisí definovaný jazyk na chování jazyka definujícího. Pokud bychom nepochopili chování definujícího jazyka, nejsme z tohoto kódu schopni pochopit ani definovaný jazyk.

1.3 Lokální vazby

V této chvíli dokáže náš interpret vyhodnocovat všechny výrazy pouze vzhledem k jedinému prostředí `init-env`. To je velmi významné omezení, které například nedovoluje definici lokálních proměnných uvnitř procedur a funkcí, jak bývá běžné ve většině programovacích jazyků. Abychom dosáhli nápravy, je nutné náš interpret poněkud upravit. Nejprve do našeho jazyka přidáme klíčová slova `let` a `in` sloužící k deklarování proměnných a jejich vázání v těle výrazu.

Konkrétní syntaxe let formy je následující:

```
<exp>   ::= let <decls> in <exp>                            let (decls body)
<decls> ::= <decl> {<decl>}*
<decl>  ::= <var> = <exp>                                   decl (var exp)
```

Ze zápisu konkrétní syntaxe je jasně patrné, že jednotlivé deklarace je možné zanořovat do sebe, což umožňuje unvnitř jednoho výrazu (procedury) deklarovat nové lokální proměnné pro jistý podvýraz. Odpovídající abstraktní syntaxe pak využívá následující typy záznamů:

```
(define-record let (decls body))
(define-record decl (var exp))
```

Pole `decls` záznamu `let` je asociováno s nonterminálem, který se může opakovat nula nebo vícekrát - tedy obsahuje seznam záznamů `decl`. Tělo `let` výrazu by mělo být vyhodnoceno v prostředí, ve kterém jsou deklarované proměnné vázány k výsledkům výrazů na pravé straně deklarací, zatímco ostatní vazby by měly být získány z prostředí, ve kterém je vyhodnocován celý výraz. Proto musíme upravit proceduru `eval-exp` tak, aby zpracovávala dva argumenty - výraz a prostředí. Nová procedura `eval-exp` musí zadaný výraz vyhodnocovat v zadaném prostředí.

Příklad 1.7 - Interpret s *let*

```
(define eval-exp
  (lambda (exp env)
    (variant-case exp
      (lit (datum) datum)
      (varref (var) (apply-env env var))
      (app (rator rands)
        (let ((proc (eval-exp rator env)) (args (eval-rands rands env)))
          (apply-proc proc args)))
      (if (test-exp then-exp else-exp)
        (if (true-value? (eval-exp test-exp env))
            (eval-exp then-exp env)
            (eval-exp else-exp env)))
      (let (decls body)
        (let ((vars (map decl->var decls)) (exps (map decl->exp decls)))
          (let ((new-env (extend-env vars (eval-rands exps env) env)))
            (eval-exp body new-env))))
      (else (error "Invalid abstract syntax: " exp)))))
```

Pokud je vyhodnocován `let` výraz, jsou nejprve vyhodnoceny podvýrazy na pravé straně jeho deklarací. Jelikož rozsah platnosti těchto deklarací je omezen pouze na tělo výrazu, podvýrazy na pravé straně deklarací jsou pomocí `eval-rands` vyhodnoceny v původním prostředí celého výrazu. Tělo výrazu je poté vyhodnoceno v novém prostředí získaném rozšířením původního prostředí o nově deklarované proměnné pomocí `extend-env`.

1.4 Procedury

Náš jazyk obsahuje stále pouze procedury obsažené v počátečním prostředí. Nyní proto doplníme možnost definovat vlastní procedury. Použijeme tuto konkrétní syntaxi:

```
<exp>     ::= proc <varlist> <exp>
<varlist> ::= () | (<vars>)
<vars>    ::= <var> {,<var>}*
```

Při vyhodnocování aplikace procedury je její tělo vyhodnoceno v prostředí, ve kterém jsou její formální parametry vázány k argumentům této aplikace. Vázání proměnných vyskytujících se v

těle procedury jako volné zůstávají zachována vůči původnímu prostředí. Procedury mohou být předávány jako parametry do jiných procedur, vráceny jako výsledky procedur a ukládány do datových struktur.

Aby mohla procedura zachovávat vázání svých volných proměnných tak, jak byly vázány v čase jejího vytvoření, musí tvořit *uzavřený celek* nezávislý na prostředí, v němž byla vytvořena. Takovýto celek nazýváme *uzávěr* a obsahuje tělo procedury, její formální parametry a vázání jejích volných proměnných. Často též říkáme, že procedura je uzavřena vůči prostředí, v němž byla vytvořena. Uzávěry budeme reprezentovat jako záznamy:

```
(define-record closure (formals body env))
```

Nyní musíme modifikovat proceduru `apply-proc` tak, aby kromě primitivních procedur neobsahujících uzávěry zohledňovala i uzávěry procedur nově definovaných. Proceduru můžeme popsat pomocí následující rovnice:

Procedura = primitivní procedura + uzávěr

`Apply-proc` tedy musí nejprve zkontrolovat, jaký typ procedury zpracovává. Pokud tato má uzávěr, jednoduše vyvolá tělo uzávěru v příslušně rozšířeném prostředí.

Příklad 1.8 - Interpret podporující uživatelem definované procedury

```
(define eval-exp
  (lambda (exp env)
    (variant-case exp
      (lit (datum) datum)
      (varref (var) (apply-env env var))
      (app (rator rands)
        (let ((proc (eval-exp rator env)) (args (eval-rands rands env)))
          (apply-proc proc args)))
      (if (test-exp then-exp else-exp)
        (if (true-value? (eval-exp test-exp env))
            (eval-exp then-exp env)
            (eval-exp else-exp env)))
      (let (decls body)
        (let ((vars (map decl->var decls)) (exps (map decl->exp decls)))
          (let ((new-env (extend-env vars (eval-rands exps env) env)))
            (eval-exp body new-env))))
      (proc (formals body)
        (make-closure formals body env))
     (else (error "Invalid abstract syntax: " exp)))))

(define apply-proc
  (lambda (proc args)
    (variant-case proc
      (prim-proc (prim-op) (apply-prim-op prim-op args))
      (closure (formals body env)
        (eval-exp body (extend-env formals args env)))
      (else (error "Invalid procedure: " proc)))))
```

1.5 Přiřazení proměnných

Dále potřebujeme do našeho interpretu přidat podporu pro přiřazování do proměnných pomocí operátoru přiřazení (`:=`). Nejdříve musíme změnit abstrakci našeho prostředí, neboť v současné době neposkytuje prostředky pro změnu hodnoty vázané k proměnné - každá proměnná je přímo vázaná ke své vyjadřované hodnotě. Označované hodnoty (hodnoty vazeb proměnných) a vyjadřované hodnoty (hodnoty výrazů) jsou tedy identické. Přímé vazby proměnných k jejich hodnotám nahradíme vazbami přes paměťové buňky. Proměnné již nebudou vázány přímo na své hodnoty, ale na příslušné paměťové buňky, jejichž obsah může být snadno modifikován pomocí přiřazení. Můžeme tedy uvést:

Popisovaná hodnota = buňka (vyjadřovaná hodnota).

Nyní provedeme modifikaci interpretu, aby jako označované hodnoty používal paměťové buňky. Tato operace vyžaduje dvě změny. První se týká procedury `apply-proc` v okamžiku vyvolání uzávěru - nové vazby musí být buňky obsahující argumenty. Namísto

```
(extend-env formals args env)
```

použijeme konstrukci

```
(extend-env formals (map make-cell args) env).
```

Druhá změna se týká procedury `eval-exp`. Pokud je ve výrazu použita proměnná, je nutné provést extrakci skutečné hodnoty z paměťové buňky, na kterou je tato proměnná vázána. Namísto

```
(varref (var) (apply-env env var))
```

použijeme

```
(varref (var) (cell-ref (apply-env env var))).
```

Tento přístup, kdy jsou hodnoty operandů procedury uloženy v paměti je též znám pod názvem *předávání hodnotou* (*call-by-value*).

Abychom do našeho interpretu mohli definitivně přidat operátor přiřazení, musíme ještě zavést jeho konkrétní snyntaxi

`<exp> ::= <var> := <exp>`	`varassign (var exp)`

a abstraktní syntaxi založenou na záznamu:

```
(define-record varassign (var exp)).
```

Pro implementaci přiřazení musíme přidat následující větev do `eval-exp`:

```
(varassign (var exp)
  (cell-set! (apply-env env var) (eval-exp exp env)))
```

Prvky, které jsou předávány do `cell-set!` jsou obecně označovány jako *L-hodnoty* (mohou stát na levé straně operátoru přiřazení).

Příklad 1.9 - Interpret podporující přiřazování do proměnných (call-by-value)

```
(define eval-exp
  (lambda (exp env)
    (variant-case exp
      (lit (datum) datum)
      (varref (var) (cell-ref (apply-env env var)))
      (app (rator rands)
        (let ((proc (eval-exp rator env)) (args (eval-rands rands env)))
          (apply-proc proc args)))
      (if (test-exp then-exp else-exp)
        (if (true-value? (eval-exp test-exp env))
            (eval-exp then-exp env)
            (eval-exp else-exp env)))
      (proc (formals body)
        (make-closure formals body env))
      (var-assign (var exp)
        (cell-set! (apply-env env var) (eval-exp exp env)))
     (else (error "Invalid abstract syntax: " exp)))))

(define apply-proc
  (lambda (proc args)
    (variant-case proc
      (prim-proc (prim-op) (apply-prim-op prim-op args))
      (closure (formals body env)
        (eval-exp body (extend-env formals (map make-cell args) env)))
      (else (error "Invalid procedure: " proc)))))
```

1.6 Rekurze

Poslední vlastností, kterou je ještě nutné implementovat do našeho jednoduchého interpretu, je rekurze. Jednou možností, kterou lze využít, je operátor pevného bodu. Tato možnost je však spíše pouze teoretická, neboť je většinou velmi neefektivní pro rutinní praktické použití.

Druhou možností je využití výrazu `letrec`. Výraz `letrec` je syntakticky transformován na výraz, který použije `let` k vytvoření prostředí s nadbytečnými vazbami proměnných. Rekurzivní procedury jsou vzhledem k tomuto prostředí uzavřeny. Díky tomu je následně možné využít nadbytečné vazby jejich asociovaných proměnných k uložení jejich uzávěrů v paměťových buňkách.

Příklad 1.10 - Transformace příkazu *letrec*

```
   (letrec ((var1 exp1) ... (varn expn))
     body)

⇒ (let ((var1 '*) ... (varn '*))
     (let ((v1 exp1) ... (vn expn))
       (set! var1 v1)
        ...
       (set! varn vn)
       'unspecified)
     body
```

$v_1, \ldots, v_n$ jsou nové proměnné, které se nevyskytují nikde mimo výraz `letrec`. `'Unspecified` je vloženo pouze proto, aby bylo zajištěno, že vnitřní `let` bude mít vždy nějaké tělo, i když nebude obsahovat žádné definice. Rekurzivní procedury jsou vytvářeny jejich uzavřením v prostředí obsahujícím proměnné, ke kterým budou vázány. Pokud bychom se pokusili o dereferenci kterékoliv proměnné var_i v příkazu exp_j před provedením příslušného příkazu `set!`, získáme nespecifikované hodnoty. Pokud ale každý výraz exp_i bude lambda, potom je zaručeno, že k takovéto situaci nikdy nedojde a pořadí vyhodnocování nebude podstané. Kromě některých speciálních případů (jako jsou například streamy) bývá tato podmínka prakticky vždy splněna.

Pro implementaci rekurze do našeho interpretu použijeme jistou variaci syntaxe příkazu `letrec`, která omezuje pravou stranu rekurzivních deklarací na výraz `proc`. Konkrétní a abstraktní syntaxe je popsána následující gramatikou:

```
<exp>        ::= letrecproc <procdecls>        letrecproc (prodecls
                   in <exp>                      body)
<procdecls> ::= <procdecl> {;<procdecl>}*
<procdecl>  ::= <var> <varlist> = <exp>       procdecl (var formals body)
```

Levou stranu rekurzivní deklarace tvoří jméno rekurzivní procedury a seznam jejích formálních parametrů. Pravá strana je tvořena tělem této procedury.

Abychom mohli začlenit `letrecproc` do našeho interpretu, musíme nejprve zavést do ADT prostředí nový operátor `extend-rec-env`, který rozšíří prostředí o množinu vzájemně rekurzivních procedur.

Příklad 1.11 - Abstraktní datový typ pro prostředí bez cyklických uzávěrů podporující rekurzi

```
(define-record extend-rec-env (vars vals old-env))

(define extend-rec-env
  (lambda (procdecls env)
    (make-extended-rec-env
      (map procdecl->var procdecls)
      (list->vector
        (map (lambda (procdecl)
               (make-proc
                 (procdecl->formals procdecl)
                 (procdecl->body procdecl)))
             procdecls))
      env)))

(define apply-env
  (lambda (env var)
  (variantcase env
    (extended-rec-env (vars vals old-env)
      (let ((p (ribassoc var vars vals '*fail*)))
        (if (eq? p '*fail*)
            (apply-env old-env var)
            (make-closure (map make-cell
                               ((proc->formals p) (proc->body p))) env))))
    ...)))
```

Nyní již můžeme `letrecproc` začlenit do našeho interpretu. Přidáme jej jako novou větev do `eval-exp`. Tato větev bude velice podobná větvi pro `eval-exp`, pouze s tím rozdílem, že namísto `extend-env` bude použito `extend-rec-env`, čímž bude zajištěno vytvoření nového prostředí, ve kerém bude vyhodnoceno tělo procedury.

Příklad 1.12 - Část interpretu - větev `eval-exp` pro vyhodnocení `letrecproc`

```
(letrecproc (procdecls body)
  (eval-exp body (extend-rec-env procdecls env)))
```

1.7 Dynamický rozsah platnosti proměnných a dynamické přiřazení

V této kapitole si ukážeme dvě možnosti pro zajištění dynamického přeuspořádávání informací v prostředí. První přístup, dynamický rozsah platnosti, je druh vázacího mechanizmu, který zvyšuje vyjadřovací sílu za cenu obtížnější pochopitelnosti programu. Druhý přístup, dynamické přiřazení, poskytuje skoro stejnou vyjadřovací schopnost jako dynamický rozsah platnosti a navíc nezpůsobuje ztrátu přehlednosti programu.

1.7.1 Dynamický rozsah platnosti proměnných

Zatím jsme stále předpokládali, že tělo procedury je vždy vyhodnoceno v prostředí, v jakém byla tato procedura vytvořena. To je nutné, pokud mají být proměnné lexikálně vázané, ovšem, jak jsme mohli vidět, je na druhou stranu nutné vytvořit vždy nový uzávěr při každém volání procedury. Podstatně jednodušší by bylo, vyhodnocovat tělo každé procedury vždy v prostředí, ze kterého je tato procedura právě volána - použít *dynamický rozsah platnosti proměnných* nebo též *dynamické vázání*. Situaci, jež v tom případě nastane, ilustruje následující příklad.

Příklad 1.13 - Rozdíl mezi technikami dynamický rozsah platnosti a lexikální vázání proměnných

```
let a = 3
in let p = proc (x) +(x, a);
       a = 5
   in *(a, p(2))
```

Pokud použijeme lexikální vázání proměnných, bude se tělo procedury `p` vyhodnocovat v prostředí, v jakém bylo vytvořeno, tedy proměnná `a` bude mít v průběhu vyhodnocování hodnotu 3 a výsledek celého výrazu bude 25. Pokud ale použijeme dynamický rozsah platnosti, bude se tělo procedury `p` vyhodnocovat v prostředí, v jakém bylo vyvoláno, tedy proměnná `a` bude mít v průběhu vyhodnocování hodnotu 5 a výsledek celého výrazu bude 35.

Dynamické vázání se řídí následujícím pravidlem:

Dynamická vazba je *platná* po dobu vyhodnocování těla asociovaného s vázací formou. Reference na dynamicky vázanou proměnnou odkazují vždy na *nejposlednější* platné vázání této proměnné.

Ačkoliv je dynamické vázání proměnných obtížnější než lexikální vázání co se týče pochopitelnosti zdrojového programu, je podstatně jednodušší při implementaci překladače. Jelikož procedury již nejsou uzavřeny vzhledem k prostředí, nemusíme vytvářet jejich uzávěry obsahující prostředí, v němž byly vytvořeny, ale budeme používat pouze seznam formálních parametrů.

Při vyhodnocování procedury tedy musí `apply-proc` pouze doplnit aktuální prostředí tak, aby získala nové prostředí, v němž se bude vyhodnocovat tělo procedury a toto prostředí předat do `eval-exp`.

Pokud je do prostředí přidána nějaká vazba, pak zůstává platná tak dlouho, dokud nejsou odstraněny všechny následně přidané vazby. Jinak řečeno, prostředí se chová jako struktura typu LIFO (last-in-first-out). Tohoto chování můžeme výhodně využít a implementovat ADT prostředí tak, aby používal globální zásobník prostředí, čímž odpadne nutnost složitého předávání prostředí jako pomocného parametru procedur. Předání nového prostředí z `apply-proc` do `Eval-exp` (jak bylo naznačeno v předchozím odstavci), tedy bude provedeno jednoduše jeho uložením na globální zásobník prostředí v `apply-proc`. Po vyhodnocení těla procedury je samozřejmě nutné toto nové prostředí ze zásobníku odstranit, o což se též stará `apply-proc`.

Příklad 1.14 - Interpret podporující dynamický rozsah proměnných

```
(define eval-exp
  (lambda (exp)
    (variant-case exp
      (lit (datum) datum)
      (varref (var) (cell-ref (lookup-in-env var)))
      (app (rator rands)
        (let ((proc (eval-exp rator)) (args (eval-rands rands)))
          (apply-proc proc args)))
      (if (test-exp then-exp else-exp)
        (if (true-value? (eval-exp test-exp))
            (eval-exp then-exp)
            (eval-exp else-exp)))
      (proc (formals body) exp)
      (var-assign (var exp)
        (cell-set! (lookup-in-env var) (eval-exp exp)))
     (else (error "Invalid abstract syntax: " exp)))))

(define eval-rands
  (lambda (rands)
    (map (lambda (exp) (eval-exp exp)) rands)))

(define apply-proc
  (lambda (proc args)
    (variant-case proc
      (prim-proc (prim-op) (apply-prim-op prim-op args))
      (proc (formals body)
        (push-env! formals (map make-cell args))
        (let ((value (eval-exp body)))
          (pop-env!) value))
      (else (error "Invalid procedure: " proc)))))
```

Implementace s globálním zásobníkem prostředí může být nevýhodná v případě, že si vyhodnocení programu vyžádá mnoho proměnných a jejich hluboké zanoření v zásobníku prostředí. Jelikož tento musí být při vyhledávání nejposlednější platné verze dané proměnné procházen sekvenčně, může být celé vyhodnocování programu značně neefektivní. Interpret však lze modifikovat i tak, že se pro každou proměnnou použije separátní zásobník a v tom případě vyhledávání zcela odpadne, neboť nejposlednější platná verze je vždy na vrcholu příslušného zásobníku. Ovšem, v tomto případě může být poněkud časově složitější vyvolávání procedur, neboť tato akce pak obecně představuje ukládání údajů do několika zásobníků a po vyhodnocení procedury jejich opětné vyjímání.

Poměrně podstatnou nevýhodou dynamického rozsahu platnosti proměnných však je to, že při jeho použití nelze na program aplikovat různé transformační techniky jako jsou například α-, β-, η-konverze, které je možné aplikovat při použití lexikálního vázání proměnných.

1.7.2 Dynamické přiřazení

Interprety, které jsme vytvořili před zavedením dynamického rozsahu platnosti proměnných, používaly lexikální vázání proměnných. V tomto případě byla platnost všech nadefinovaných vazeb neomezená. Nabízí se tedy možnost explicitně omezit platnost některých vazeb jen na rozsah určité části programu, například na tělo jedné procedury. V definovaném jazyce za tímto účelem použijeme následující konkrétní a abstraktní syntaxi:

```
<exp> ::= <var> := <exp> during <exp>              dynassign (var exp body)
```

Efekt příkazu *var* `:=` *exp* `during` *body* je takový, že v průběhu vyhodnocování *body* dočasně přiřadí proměnné *var* hodnotu *exp* a po vyhodnocení *body* vrátí proměnné *var* její původní hodnotu.

Příklad 1.15 - Interpret s dynamickým přiřazením

```
(define eval-exp
  (lambda (exp env)
    (variant-case exp
      (lit (datum) datum)
      (varref (var) (cell-ref (apply-env env var)))
      (app (rator rands)
        (let ((proc (eval-exp rator env)) (args (eval-rands rands env)))
          (apply-proc proc args)))
      (if (test-exp then-exp else-exp)
        (if (true-value? (eval-exp test-exp env))
            (eval-exp then-exp env)
            (eval-exp else-exp env)))
      (let (decls body)
        (let ((vars (map decl->var decls)) (exps (map decl->exp decls)))
          (let ((new-env (extend-env vars (eval-rands exps env) env)))
            (eval-exp body new-env))))
      (proc (formals body)
        (make-closure formals body env))
      (var-assign (var exp)
        (cell-set! (apply-env env var) (eval-exp exp env)))
```

```
      (dynassign (var exp body)
        (let ((a-cell (apply-env env var))
              (b-cell (make-cell (eval-exp exp env))))
          (cell-swap! a-cell b-cell)
          (let ((value (eval-exp body env)))
            (cell-swap! a-cell b-cell)
            value)))
      (else (error "Invalid abstract syntax: " exp)))))

(define apply-proc
  (lambda (proc args)
    (variant-case proc
      (prim-proc (prim-op) (apply-prim-op prim-op args))
      (closure (formals body env)
        (eval-exp body (extend-env formals (map make-cell args) env)))
      (else (error "Invalid procedure: " proc)))))
```

1.7.3 Vlastnosti a porovnání obou metod

Asi nejběžnějším využitím obou metod je přesměrování vstupně-výstupních operací, případně zpracování výjimek ve většině programovacích jazyků. Dynamické přiřazení přitom odstraňuje většinu obtíží, které sebou přináší použití dynamického rozsahu platnosti proměnných. Například lze provádět α-konverze. Výhodou dynamického přiřazení je i to, že je lze využít i v lexikálně zaměřených jazycích, ve kterých rozhodně nelze využít dynamického rozsahu platnosti proměnných. Ovšem, na druhou stranu, dynamické přiřazení sdílí problémy ostatních forem přiřazení. Kupříkladu procedury se nechovají funkcionálně: Výsledky, které vracejí, nemusejí záviset pouze na hodnotách jejich parametrů.

2 Předávání parametrů

2.1 Podpora polí

Mnoho programovacích jazyků podporuje datové struktury skládající se z mnoha prvků, jako jsou například pole nebo záznamy. Takovéto struktury nazýváme *agregáty*. Tyto typy bývají v paměti obvykle uloženy jako souvislý blok, obsahující všechny elementy této struktury. Pokud je agregát předáván do procedury, je obvykle předáván pouze ukazatel na první buňku příslušného bloku paměti. Tomuto přístupu říkáme *nepřímá* reprezentace, protože hodnoty agregátu následně nejsou získávány přímo, ale přes ukazatel. Volaná procedura a část programu, ze které je tato procedura volána, používají stejný úsek paměti. Tedy, pokud procedura provede nějaké změny v hodnotách agregátu, projeví se tyto změny následně i v kódu, který tuto proceduru vyvolal.

I do našeho jazyka teď přidáme podporu polí s následující konkrétní a abstraktní syntaxí:

```
<form>    ::= definearray <var> [<exp>]     definearray (var dim-exp)
<exp>     ::= letarray <arrdcls> in <exp>   letarray (arraydecls body)
            | <arrexp> [<exp>]              arrayref (array index)
            | <arrexp> [<exp>] := <exp>     arrayassign (array index exp)
<arrexp>  ::= <exp>
<arrdcls> ::= <arrdcl> {; <arrdcl>}*
<arrdcl>  ::= <var> [<exp>]                 decl (var exp)
```

Pole je sekvence buněk obsahujících vyjadřované hodnoty, což lze vyjádřit takto:

Pole = buňka * (vyjadřovaná hodnota)

Jelikož náš definující jazyk nepodporuje pole, budeme je implementovat pomocí vektorů, které také poskytují sekvence proměnlivých elementů. Indexování polí bude řešeno tak, že první prvek má vždy index nula. Vyjadřované hodnoty v definovaném jazyce zůstanou i po přidání polí stále reprezentovány pomocí buněk obsahujících vyjadřované hodnoty:

Vyjadřovaná hodnota = číslo + procedura + pole
Označovaná hodnota = buňka (vyjadřovaná hodnota)

Příklad 2.1 - Abstraktní datový typ pole

```
(define-record aggregate (vector))

(define make-array
  (lambda (dimension)
    (make-aggregate (make-vector dimension))))

(define array? aggregate?)

(define array-ref
  (lambda (aray index)
    (vector-ref (aggregate->vector array) index)))
```

```
(define array-set!
  (lambda (array index value)
    (vector-set! (aggrefate->vector array) index value)))

(define array-copy
  (lambda (array)
    (make-aggregate (list->vector (vector->list
                                     (aggregate->vector array))))))
```

Příklad 2.2 - Interpret s podporou polí

```
(define eval-exp
  (lambda (exp env)
    (variant-case exp
      (lit (datum) datum)
      (varref (var) (denoted->expressed (apply-env env var)))
      (app (rator rands)
        (apply-proc(eval-rator rator env) (eval-rands rands env)))
      (if (test-exp then-exp else-exp)
        (if (true-value? (eval-exp test-exp env))
            (eval-exp then-exp env)
            (eval-exp else-exp env)))
      (proc (formals body)
        (make-closure formals body env))
      (varassign (var exp)
        (denoted-value-assign! (apply-env env var) (eval-exp exp env)))
      (letarray (arraydecls body)
        (eval-exp body
          (extend-env (map decl->var array decls)
            (map (lambda (decl)
                   (do-letarray (eval-exp (decl->exp decl) env)))
                 arraydecls)
            env)))
      (arrayref (array index)
        (array-ref (eval-array-exp array env)
          (eval-exp index env)))
      (arrayassign (array index exp)
        (array-set! (eval-array-exp array env)
          (eval-exp index env)
          (eval-exp exp env)))
    (else (error "Invalid abstract syntax: " exp)))))

(define apply-proc
  (lambda (proc args)
    (variant-case proc
      (prim-proc (prim-op) (apply-prim-op prim-op args))
      (closure (formals body env)
        (eval-exp body (extend-env formals (map make-cell args) env)))
      (else (error "Invalid procedure: " proc)))))

(define eval-rator
  (lambda (rands env)
    (eval-exp rator env)))

(define eval-rands
  (lambda (rands env)
    (map (lambda (rand) (eval-rand rand env)) rands)))

(define eval-rand
```

```
  (lambda (exp env)
    (expressed->denoted (eval-exp exp env))))

(define apply-proc
  (lambda (proc args)
    (variant-case proc
      (prim-proc (prim-op)
        (apply-prim-op prim-op (map denoted->expressed args)))
      (closure (formals body env)
        (eval-exp body (extend-env formals args env)))
      (else (error "Invalid procedure: " proc)))))
```

Interpret uvedený v předchozím příkladu neobsahuje definice některých procedur. V závislosti na realizaci těchto definicí je možné specifikovat chování interpretu při předávání polí. Definice těchto procedur pro nepřímé předávání polí jsou uvedeny v následujícím příkladu.

Příklad 2.3 - Definice pomocných procedur pro pro nepřímé předávání polí

```
(define denoted->expressed cell-ref)
(define denoted-value-assign! cell-set!)
(define do-letarray (compose make-cell make-array))
(define eval-array-exp eval exp)
(define expressed->denoted make-cell)
```

Pokud potřebujeme, aby interpret prováděl přímé předávání polí (předávání hodnotou), můžeme tyto procedury definovat, jak ukazuje následující příklad.

Příklad 2.4 - Definice některých pomocných procedur pro přímé předávání polí

```
(define denoted->expressed
  (lambda (den-val)
    (let ((exp-val (cell-ref den-val)))
      (if (arrray? exp-val) (array-copy exp-val) exp-val))))

(define eval-array-exp
  (lambda (exp env)
    (variant-case exp
      (variant-case exp
        (varref (var) (let ((exp-val (cell-ref (apply-env env var))))
                        (if (array? exp-val)
                            exp-val
                            (error "expecting an array: " exp-val))))
        (else (eval-exp exp env))))))
```

V uvedeném modelu je realizováno předávání parametrů hodnotou. Každý formální parametr je navázán na buňku obsahující kopii sekvence hodnot původního pole. Jak je patrné z příkladu, je kopírování obsahu prováděno pouze při předávání polí. Předpokládá se totiž, že definující jazyk implicitně používá přímou reprezentaci předávání všech proměnných kromě polí (ta samozřejmě nejsou implicitně podporována vůbec).

2.2 Předávání parametrů odkazem

Při programování často nastává situace, kdy potřebujeme, aby se změny, které provede volaná procedura ve svých parametrech, projevily i v části programu, ze které byla tato procedura volána. Jinými slovy, aby procedura mohla ve svých prametrech předávat návratové hodnoty. Toho může být dosaženo použitím jiné techniky předávání prametrů, která spočívá v předávání referencí na proměnné, nikoliv jejich vyjadřovaných hodnot. Tento pirincip předávání parametrů je známý jako *předávání odkazem*.

Obecně platí, že tímto způsobem lze jako parametry předávat pouze proměnné, popřípadě jiné datové struktury, na které lze získat referenci do paměti. V některých jazycích je však možné použít předávání odkazem i pro zcela jiné datové struktury (například aplikace). Mechanizmus předávání je pak řešen tak, že hodnota daného operandu je uložena do nové paměťové buňky a její adresa je předána volané proceduře. Přiřazení do takového prametru uvnitř procedury pak ale samozřejmě nemá žádný efekt v rámci volajícího kódu.

Zajímavý je případ, kdy dva parametry předávané odkazem odkazují na shodnou paměťovou buňku, jak demonstruje následující příklad.

Příklad 2.5 - Dva parametry odkazující na stejnou buňku paměti

```
let b = 3:
    p = proc (x, y)
          begin
            x := 4;
            y
          end
in p(b, b);
```

Jelikož oba parametry procedury `p` odkazují na stejnou buňku paměti (proměnná `b`), způsobí přiřazení `x := 4` zároveň přiřazení do `y`, čímž dojde k přepsání jeho původní hodnoty 3. Výsledek volání procedury `p(b, b)` je tedy 8 a nikoliv 7, jak by se mohlo na první pohled zdát.

Fenomén demonstrovaný v předchozím příkladu je známý pod pojmem *aliasing* a je velmi nebezpečný, neboť jej nelze jednoduše automaticky detekovat a pravděpodobnost jeho přehlédnutí v programu je vysoká. Jedinou možností detekce jsou náročné run-time testy.

Jak budeme modelovat předávání odkazem? Vodítkem nám bude to, že označované hodnoty volající části programu a volané procedury jsou shodné. Tedy, pokud operand bude proměnná, můžeme předat její vazbu přímo do procedury namísto kopírování její hodnoty do nové buňky. Změníme tedy pravidlo gramatiky našeho jazyka

```
<operand> ::= <exp>
```

na

```
<operand> ::= <varref>
```

a zároveň změníme proceduru `eval-rand` takto:

Příklad 2.6 - Procedura `eval-rand` pro předávání parametrů odkazem

```
(define eval-rand
  (lambda (rand env)
    (variant-case rand
      (varref (var) (apply-env env var))
      (else error "Invalid operand: " rand)))))
```

Situace se poněkud zkomplikuje v okamžiku, kdy přidáme požadavek na předávání jednotlivých prvků polí odkazem. Až do této chvíle jsme při předávání polí odkazem předpokládali předání ukazatele na první prvek pole. V typické implementaci se jedná pouze o předání ukazatele na příslušný prvek pole. Náš definující jazyk však nedokáže získat ukazatel na libovolný prvek vektorového elementu. Proto budeme jednotlivé prvky pole reprezentovat jako záznamy, jež budou obsahovat pole a index tohoto prvku uvnitř pole.

```
(define-record ae (array index))
```

Obecně je tedy jakákoliv L-hodnota v tomto programovacím jazyce buď buňka paměti nebo prvek pole. V každém případě L-hodnota představuje vyjadřovanou hodnotu, což můžeme zapsat takto:

L-hodnota = buňka (vyjadřovaná hodnota) + prvek pole (vyjadřovaná hodnota)
Označovaná hodnota = L-hodnota

Nyní přidáme do našeho jazyka možnost předávání odkazem pro libovolné výrazy, přičemž jejich hodnoty budou předávány hodnotou v nových paměťových buňkách s následující konkrétní a abstraktní syntaxí:

```
<operand> ::= <varref>
            | <array-exp> [<exp>]                  arrayref (array index)
            | <exp>
```

Abychom získali interpret provádějící předávání parametrů odkazem, provedeme modifikaci interpretu pro předávání hodnotou a modifikujeme pomocné procedury. V souladu s uvedenou gramatikou pro operandy bude `eval-rand` obsahovat tři větve. Pokud operand bude proměnná, případně reference na pole, bude do volané procedury předána korespondující L-hodnota. V opačném případě bude výraz vyhodnocen, zkopírován do nové paměťové buňky a předán hodnotou. Jelikož jsme změnili množinu označovaných hodnot, potřebujeme změnit procedury, které s nimi pracují (`denoted->expressed` a `denoted-value-assign!`).

Příklad 2.7 - Pomocné procedury pro předávání parametrů odkazem s nepřímým předáváním polí

```
(define eval-rand
  (lambda (rand env)
    (variant-case rand
      (varref (var) (apply-env env var))
      (arrayref (array index)
        (make-ae (eval-array-exp array env) (eval-exp index env)))
      (else (make-cell (eval-exp rand env))))))

(define denoted->expressed
  (lambda (den-val)
```

```
    (cond
      ((cell? den-val) (cell-ref den-val))
      ((ae? den-val) (array-ref (ae->array den-val) (ae->index den-val)))
      (else (error "Can't dereference denoted value: " den-val)))))

(define denoted-value-assign!
  (lambda (den-val val)
    (cond
      ((cell? den-val) (cell-set! den-val val))
      ((ae? den-val) (array-set! (ae->array den-val)
                                 (ae->index den-val) val))
      (else (error "Can't assign to denoted value: " den-val)))))
```

S kombinací přímého a nepřímého předávání parametrů se v programovacích jazycích setkáváme velice často - například při předávání pole jako var parametru v Pascalu.

2.3 Předávání hodnotou-odkazem a předávání výsledkem

Technika předávání *hodnotou-odkazem* v sobě kombinuje efektivnost metody předávání hodnotou s možností navrácení výsledku jako u metody předávání odkazem. Trik pro splnění těchto vlastností je poměrně jednoduchý. Při volání se parametr nejprve zkopíruje do nové paměťové buňky, v těle procedury se používá jako by byl předávaný hodnotou a v okamžiku návratu je jeho konečná hodnota zkopírována na místo původního parametru (původní hodnota je přepsána). Přístup k parametrům v těle procedury je velmi rychlý (stejně jako při použití předávání hodnotou), ovšem režie při vyvolání a ukončení procedury je vysoká, (také stejně jako při předávání hodnotou), neboť probíhá kopírování paměťových buněk. Díky tomu se tento přístup jeví jako výhodný v případě, že potřebujeme mít možnost navracet v parametrech výsledky a zároveň se jednotlivé parametry v těle procedury často používají (například v těle cyklu). V případě malého počtu použití se jeví jako výhodnější použití metody předávání odkazem.

Existuje však ještě jeden důvod, proč preferovat předávání hodnotou-odkazem před předáváním odkazem. Předávání hodnotou-odkazem je odolné vůči vzniku aliasingu (každý parametr je uložen v jiné paměťové buňce), což zvyšuje odolnost programu vůči skrytým chybám.

Technika *předávání parametrů výsledkem* je použitelná pouze pokud daným parametrem nepotřebujeme předávat data do procedury, ale potřebujeme je v něm vrátit. Jedná se o jednoduchou variantu předávání hodnotou-odkazem, ve které je lokální L-hodnota parametru neinicializovaná.

Předávání hodnotou, předávání hodnotou-odkazem a předávní výsledkem bývají někdy souhrnně označovány jako *předávání kopií*, neboť při vyvolání procedury se vždy kopíruje hodnota proměnné předávané jako parametr.

2.4 Vyjadřované nebo označované hodnoty?

Jazyky, které jsme zatím vytvořili, stejně jako náš definující jazyk měly bohatou množinu vyjadřovaných hodnot, ale pouze jediný druh ozančované hodnoty - buňky obsahující vyjadřované

hodnoty. Mít v jazyce bohatou množinu vyjadřovaných hodnot je velice důležité, neboť hlavní cestou, kterou se vrací vypočtené informace, je návratová hodnota procedury - označovaná hodnota.

V tradičních imperativních jazycích se však výrazy vyskytují především v přiřazovacích příkazech, takže množina vyjadřovaných hodnot zhruba koresponduje s množinou hodnot, které mohou být uloženy v paměťových lokacích. Obvykle je tedy velmi malá, zatímco množina označovaných hodnot bývá obvkle velmi bohatá. Například v Pascalu jsou vyjadřovanými hodnotami skaláry jako jou integery nebo znaky, zatímco označované hodnoty jsou procedury a pole. Nyní zkusíme tento princip implementovat do našeho interpretu.

Nový jazyk bude mít nejjednodušší možnou množinu vyjadřovaných hodnot:

Vyjadřovaná hodnota = číslo

zato množinu označovaných hodnot vytvoříme poněkud bohatší:

Označovaná hodnota = L-hodnota + pole + procedura

Pole a procedury tedy nyní budou označované hodnoty, nikoliv vyjadřované. Tato změna nezpůsobí vážnější problémy u polí, neboť jsou vytvářena pomocí `letarray`, ale u procedur je situace již poněkud složitější. Fakt, že procedury jsou vyjadřované hodnoty, je začleněn přímo v syntaxi jazyka a tím i v interpretu. Procedury jsou totiž vytvářeny jako vyjadřované hodnoty pomocí `proc` a zároveň navráceny jako vyjadřované hodnoty v okamžiku, kdy `eval-exp` vyhodnotí operátorovou část aplikace.

Nejdříve tedy musíme změnit syntaxi našeho jazyka. Procedura nebo pole budou nyní moci existovat pouze jako vazba na proměnnou a zároveň potřebujeme zavést nějakou formu jako `let`, která umožní deklaraci proměnných bez použití aplikace procedury. Z původní gramatiky tedy odstraníme pravidla

```
<exp>       ::= proc <varlist> <exp>                    proc (formals body)
<operator>  ::= <exp>
<array-exp> ::= <exp>
```

a nahradíme je následujícími pravidly

```
<exp>       ::= letrecproc <procdecls> in          letrecproc (procdecls
                  <exp>                              body)
              | local <decls> in <exp>               local (decls body)
operator    ::= <varref>
<array-exp> ::= <varref>
```

Pro předávání parametrů do procedur použijeme techniku předávání odkazem. K tomu využijeme interpret z kapitol 2.1 až 2.2 a provedeme potřebné modifikace.

Příklad 2.8 - Interpret s procedurami a poli jako označovanými hodnotami

```
(define eval-exp
  (lambda (exp env)
    (variant-case exp
      (lit (datum) datum)
      (varref (var) (denoted->expressed (apply-env env var)))
```

```
      (app (rator rands)
        (apply-proc(eval-rator rator env) (eval-rands rands env)))
      (if (test-exp then-exp else-exp)
        (if (true-value? (eval-exp test-exp env))
            (eval-exp then-exp env)
            (eval-exp else-exp env)))
      (letproc (procdecls body)
        (let ((vars (map procdecl->var procdecls))
              (closures (map (lambda (decl)
                               (make-closure
                                 (procdecl->formals decl)
                                 (procdecl->body decl)
                                 env))
                             procdecls)))
          (let ((new-env (extend-env vars closures env)))
            (eval-exp body new-enw))))
      (local (decls body)
        (let ((vars (map decl->var decls))
              (exps (map decl->exp decls)))
          (let ((new-env (extend-env vars
                           (map (lambda (exp)
                                  (make-cell (eval-exp exp env)))
                                exps)
                           env)))
            (eval-exp body new-env))))
      (var-assign (var exp)
        (denoted-value-assign! (apply-env env var) (eval-exp exp env)))
      (letarray (arraydecls body)
        (eval-exp body
          (extend-env (map decl->var arraydecls)
            (map (lambda (decl)
                   (do-letarray (eval-exp (decl->exp decl) env)))
                 arraydecls)
            env)))
      (arrayref (array index)
        (array-ref (eval-array-exp array env)
          (eval-exp index env)))
      (arrayassign (array index exp)
        (array-set! (eval-array-exp array env)
          (eval-exp index env)
          (eval-exp exp env)))
    (else (error "Invalid abstract syntax: " exp)))))

(define eval-rator
  (lambda (rator env)
    (let ((den-val (apply-env env (varref->var rator))))
      (if (closure? den-val)
          den-val
          (denoted->expressed den-val)))))
```

V interpretu byly provedeny dvě změny. První se týká procedury `eval-rator`, která původně používala `eval-exp`. Nyní používá `apply-env`, neboť `rator` je reference na proměnnou. Druhou změnou je nová konstrukce `letproc`. Kód `letproc` vytvoří seznam uzávěrů a potom vyhodnotí tělo procedury v novém prostředí, ve kterém je každé ze jmen vázáno k odpovídajícímu uzávěru.

Jelikož se změnily množiny označovaných a vyjadřovaných hodnot, musíme změnit také pomocné procedury uvedené v kapitole 2.2, které pracují s označovanými hodnotami. Procedury `denoted->expressed` a `denoted-value-assign!` mohou zůstat bez změny, neboť již obsahují testy zajišťující kontrolu, zda jsou dereferovány či měněny pouze platné L-hodnoty. Podobně mohou zůstat bez změn i `expressed->denoted` a `eval-rand`.

Příklad 2.9 - Pomocné procedury pro předávání parametrů odkazem s poli jako označovanými hodnotami

```
(define denoted->expressed
  (lambda (den-val)
    (cond
      ((cell? den-val) (cell-ref den-val))
      ((ae? den-val) (array-ref (ae->array den-val) (ae->index den-val)))
      (else (error "Can't dereference denoted value: " den-val)))))

(define denoted-value-assign!
  (lambda (den-val val)
    (cond
      ((cell? den-val) (cell-set! den-val val))
      ((ae? den-val) (array-set! (ae->array den-val)
                                 (ae->index den-val) val))
      (else (error "Can't assign to denoted value: " den-val)))))

(define do-letarray make-array)

(define eval-array-exp
  (lambda (array-exp env)
    (apply-env env (varref->var array-exp))))

(define eval-rand
  (lambda (rand env)
    (variant-case rand
      (varref (var) (apply-env env var))
      (arrayref (array index)
        (make-ae (eval-array-exp array env) (eval-exp index env)))
      (else (make-cell (eval-exp rand env))))))

(define expressed->denoted make-cell)
```

V této kapitole jsme se pouze lehce dotkli různých struktur hodnot v programovacích jazycích. Studium označovaných a vyjadřovaných hodnot jazyka nám umožní lépe proniknout do jeho vnitřní struktury. Struktura hodnot v jazyce určuje velkou měrou jeho vyjadřovací sílu a efektivnost a ovlivňuje implementační strategie. Například, pokud procedury nejsou vyjadřovanými hodnotami, není možné použít většinu funkcionálních technik. Na druhou stranu, pokud jsou procedury vyjadřovanými hodnotami, musí být alokovány na hromadě, což je obecně méně efektivní, než alokace na zásobníku.

2.5 Předávání jménem a předávání s vyhodnocením při potřebnosti

Výrazy lambda kalkulu mohou být vyhodnocovány použitím β-redukcí. V tomto modelu jsou procedury volány vykonáním jejich těla poté, co jsou operandy substituovány pro každou referenci za odpovídající proměnnou. Tato substituce může vyžadovat α-konverzi, aby se zabránilo případnému navázání proměnných, které jsou v operandech volné. Takovéto přepisovací techniky, které vyžadují manipulaci s textem programu v průběhu jeho vyhodnocování, bývají intenzivně využívány při teoretických studiích programovacích jazyků. Pro praktické využití však většinou bývají velmi neefektivní.

β-redukce má jednu zajímavou vlastnost, která má občas i praktický význam a není možná při použití technik předávání parametrů uvedených dříve - umožňuje odložit vyhodnocení operandů (použitím normálního pořadí vyhodnocení) až do okamžiku, kdy budou skutečně potřeba. To může zabránit zbytečnému, případně nekonečnému, vyhodnocování, ovšem za cenu opakovaného vyhodnocování každé reference v případě, že tělo funkce obsahuje více referencí na jeden parametr.

Aby bylo možné dosáhnout odložení vyhodnocení operandu bez nutnosti přepsat program, je možné předávat jako parametry reference na abstraktní syntaktické stromy (případně na kompilovaný kód) jednotlivých operandů. Pak je ale nutné zajistit, aby nemohlo dojít k nechtěnému navázání původně volných proměnných ve volané proceduře. Tento problém lze vyřešit poměrně jednoduše - uzavřít operandy v prostředí, ze kterého je procedura volána. Toho dosáhneme zformováním datové struktury, která bude obsahovat textovou reprezentaci operandu a také vazby všech proměnných, které jsou v tomto operandu volné. Takovéto uzávěry budeme nazývat *odložené výpočty* (*thunks*). Technika předávání parametrů využívající zpožděné vyhodnocování všech argumentů s využitím odložených výpočtů a provádějící vyhodnocení každého operandu (tedy i opakované, pokud je daný parametr použit vícekrát) v okamžiku, kdy je potřeba, se nazývá *předávání jménem*.

Nyní vytvoříme interpret používající volání jménem a nepřímé předávání parametrů. Nejprve určíme vyjadřované a označované hodnoty. Vyjdeme z interpretu uvedeného v kapitole 2.1. Množinu vyjadřovaných hodnot ponecháme beze změny:

$$\text{Vyjadřovaná hodnota} = \text{číslo} + \text{procedura} + \text{pole}$$

Množina popisovaných hodnot ale nyní musí obsahovat i odložené výpočty, které zapouzdřují každý operand i s jeho prostředím. Jelikož se formální parametr může vyskytovat na levé straně přiřazení, vyvolání odloženého výpočtu musí vracet L-hodnotu, což zapíšeme jako

$$\text{odložený výpočet} = () \rightarrow \text{L-hodnota}$$

ve významu, že odložený výpočet je procedura bez parametrů, která vrací L-hodnotu, pokud je zavolána. Odložený výpočet obsahující operand a prostředí budeme reprezentovat následujícím záznamem:

```
(define-record thunk (exp env))
```

Stejně jako v případě předávání odkazem, bude množian L-hodnot obsahovat i pole:

L-hodnota = buňka (vyjadřovaná hodnota) + prvek pole (vyjadřovaná hodnota)

Jelikož množina L-hodnot zůstala stejná, můžeme i většinu pomocných procedur ponechat beze změny.

Protože použití mechanizmu odložených výpočtů i pro lokální proměnné procedur by bylo značně neefektivní, budeme je reprezentovat obvyklým způsobem. Označované hodnoty tedy budou buď L-hodnoty nebo odložené výpočty:

Označovaná hodnota = L-hodnota + odložený výpočet

Interpret, který nyní vytvoříme bude obsahovat jednu podstatnou změnu oproti interpretu z kapitoly 2.1. Procedura `eval-rands` nebude vyhodnocovat operandy, ale bude je zapouzdřovat do odložených výpočtů (záznamů thunks). Jednotlivé odložené výpočty budou vyhodnocovány až v proceduře.

Příklad 2.10 - Interpret s předáváním prametrů jménem

```
(define eval-exp
  (lambda (exp env)
    (variant-case exp
      (local (decls body)
        (let ((vars (map decl->var decls))
              (exps (map decl->wxp decls)))
          (let ((new-env (extend-env vars
                           (map (lambda (exp)
                                  (make-cell (eval-exp exp env)))
                                exps)
                           env)))
            (eval-exp body new-env))))
      (lit (datum) datum)
      (varref (var) (denoted->expressed (apply-env env var)))
      (app (rator rands)
        (apply-proc(eval-rator rator env) (eval-rands rands env)))
      (if (test-exp then-exp else-exp)
        (if (true-value? (eval-exp test-exp env))
            (eval-exp then-exp env)
            (eval-exp else-exp env)))
      (proc (formals body)
        (make-closure formals body env))
      (varassign (var exp)
        (denoted-value-assign! (apply-env env var) (eval-exp exp env)))
      (letarray (arraydecls body)
        (eval-exp body
          (extend-env (map decl->var arraydecls)
            (map (lambda (decl)
                   (do-letarray (eval-exp (decl->exp decl) env)))
                 arraydecls)
            env)))
      (arrayref (array index)
        (array-ref (eval-array-exp array env)
          (eval-exp index env)))
      (arrayassign (array index exp)
        (array-set!
```

```
          (eval-array-exp array env)
          (eval-exp index env)
          (eval-exp exp env)))
    (else (error "Invalid abstract syntax: " exp)))))

(define eval-rands
  (lambda (rands env)
    (map (lambda (rand) (make-thunk rand env)) rands)))
```

Jelikož vyvolání odloženého výpočtu vrací L-hodnotu, nemůžeme v `eval-rand` použít `eval-exp`. Namísto toho použijeme gramatiku pro operandy, která rozpoznává speciální případ, kdy operandem je proměnná nebo reference na pole. V tomto případě se bude vracet přímo L-hodnota. Zavedeme gramatiku pro operandy

```
<operand> ::= <varref>
            | <array-exp> [<exp>]                  arrayref (array index)
            | <exp>
```

a vyvoláme odložené výpočty pomocí lehce modifikované verze `eval-rand` z interpretu pro předávání parametrů odkazem.

Příklad 2.11 - pomocné procedury pro interpret s předáváním prametrů jménem

```
(define eval-rand
  (lambda (rand env)
    (variant-case rand
      (varref (var) (denoted->L-value (apply-env env var)))
      (arrayref (array index)
        (make-ae (eval-array-exp array env) (eval-exp index env)))
      (else (make-cell (eval-exp rand env))))))

(define denoted->L-value
  (lambda (den-val)
    (if thunk? den-val)
    (eval-rand (thunk->exp den-val (thunk->env den-val)) den-val)))

(define denoted->expressed
  (lambda (den-val)
    (let ((l-val (denoted->L-value den-val)))
      (cond
        ((cell? l-val) (cell-ref l-val))
        ((ae? den-val)
          (array-ref (ae->array l-val) (ae->index l-val)))
        (else (error "Can't dereference denoted value: " l-val))))))

(define denoted-value-assign!
  (lambda (den-val exp-val)
    (let ((l-val (denoted->L-value den-val)))
      (cond
        ((cell? l-val) (cell-set! l-val exp-val))
        ((ae? l-val) (array-set! (ae->array l-val)
                                 (ae->index l-val) exp-val))
        (else (error "Can't assign to denoted value: " den-val))))))

(define expressed->denoted make-cell)

(define do-letarray (compose make-cell make-array))
```

```
(define eval-array-exp eval-exp)

(define eval-rator eval-exp)
```

Pokud odložené výpočty obsahují proměnné nebo reference na pole, budou vráceny odpovídající L-hodnoty. Vyvolání `(apply-env env var)` může způsobit navrácení jiného odloženého výpočtu namísto L-hodnoty, proto je volána ještě procedura `denoted->L-value`, která tento odložený výpočet případně vyvolá.

Procedura `denoted->expressed` nejprve převede pomocí `denoted->L-value` zadanou označovanou hodnotu na L-hodnotu a potom provede její dereferenci, jako tomu bylo při předávání odkazem. Procedura `denoted-value-assign!` pracuje obdobně.

Pokud není situace zkomplikována nějakými vedlejšími efekty, vrací zpožděný argument vždy stejné hodnoty, kdykoliv je vyvolán. Opakované vyhodnocování stejného argumentu je samozřejmě neefektivní. Uložením hodnoty získané při prvním vyhodnocení do paměti a jejím opětovným použitím lze vícenásobnému vyvolávání jednoho argumentu zabránit. Tato technika bývá označována jako *předávání s vyhodnocením při potřebnosti*. Úprava interpretu pro tuto techniku je poměrně jednoduchá.

Musíme lehce změnit množinu označovaných hodnot:

Popisovaná hodnota = L-hodnota + memo
Memo = buňka (odložený výpočet + L-hodnota)

Popisovaná hodnota tedy nyní bude buď L-hodnota nebo memo, což je buňka obsahující buď odložený výpočet nebo jeho výslednou L-hodnotu. Memo budeme reprezentovat jako záznam:

```
(define-record memo (cell))
```

V interpretu je nutné změnit pouze procedury `eval-rands` a `denoted->l-value` tak, aby pracovaly s položkami memo.

Příklad 2.12 - pomocné procedury pro interpret s předáváním prametrů při potřebnosti

```
(define eval-rands
  (lambda (rands env)
    (map (lambda (rand)
           (make-memo (make-cell (make-thunk rand env))))
         rands)))

(define denoted->L-value
  (lambda (den-val)
    (if memo? den-val)
      (let ((cell (memo->cell den-val)))
        (let ((contents (cell-ref cell)))
          (if (thunk? contents)
              (let ((l-val (eval-rand
                             (thunk->exp contents)
                             (thunk->env contents))))
                (cell-set! cell l-val)
                l-val)
              contents)))
        den-val)))
```

3 Závěr

V první kapitole byl postupně vytvořen interpret demonstrující různé vlastnosti programovacích jazyků. V druhé kapitole byl tento interpret dále upravován tak, aby poskytoval podporu pro různé druhy předávání parametrů, které se používají, i když některé pouze ve velmi omezené míře. Všechny interprety jsou psány ve funkcionálním jazyce, čímž je do popředí zájmu vynesena sémantika popisovaných jazyků. V současné době v praxi značně převládádají jazyky imperativní nad jazyky funkcionálními. Jednak je jejich výsledný kód většinou několikanásobně rychlejší než kód ekvivalentního programu přeloženého překladačem funkcionálního jazyka a navíc jsou pro zpracování imperativních programů přímo uzpůsobeny prakticky všechny počítače. Pro zpracování funkcionálních jazyků by byly mnohem vhodnější počítače řízené daty, zatímco současné počítače jsou řízeny programem.

Překladače imprerativních jazyků však nedokáží a ani nemohou dokázat zkontrolovat sémantiku překládaných programů. Prakticky všechny modely kompilátorů imperativních jazyků mají společnou vlastnost - jejich základem je syntaktický analyzátor. Ten vytvoří ze vstupního programu syntaktický strom a na základě jeho znalosti kompilátor určuje sémantiku překládaného programu.

Otázkou tedy zůstává, jak jednoznačně určit požadovanou sémantiku programu. U funkcionálních jazyků to samozejmě není problém, ovšem u imperativních je situace zcela jiná. Další otázkou zůstává, do jaké míry jsou současné prostředky pro popis sémantiky, jako je například λ-kalkul, vhodné pro specifikaci a následnou kontrolu sémantiky imperativních programů.

Použitá literatura

Friedman, D. P.; Wand, M.; Haynes, Ch. T. *Essentials of Programming Languages.* McGraw-Hill, 1992. ISBN 00-7022-443-9