KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

PROGRAMY A PROJEKY V JAZYKU SCHEME III

DAVID SKOUPIL

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

Olomouc 2007

Abstrakt

Tento učební text seznamuje čtenáře se základy programování v prostředí jazyka Scheme. Obsahuje řadu příkladů a projektů, při jejichž řešení musí mít čtenář k dispozici počítač a interpretr programovacího jazyka. Předpokládá se přitom využití interpreteru DrScheme, většina textu je však využitelná i na jiných platformách. Text je koncipován zejména jako cvičebnice jazyka, neklade si za cíl nahradit učebnice algoritmizace ani referenční manuál jazyka. V této oblasti existují další učební texty a zahraničí publikace (viz [Abelson96], [R5RS]). Studující použije tento text jako úvodní pomůcku pro zvládnutí jazyka Scheme a dále jako doprovodný materiál při studiu pokročilejších publikací.

Třetí díl této řady se věnuje problematice datových struktur, zejména pak seznamů.

Cílová skupina

Text je primárně určen pro posluchače prvního ročníku bakalářského studijního programu Aplikovaná informatika na Přírodovědecké fakultě Univerzity Palackého v Olomouci. Může však sloužit komukoli se zájmem o počítače a programování v jazyku Scheme. Text předpokládá zvládnutí publikací [Skoupil04A] a [Skoupil04B].

Obsah

1 Jednoduché datové struktury

1.1 Symboly a tečkové páry

Studijní cíle: Po prostudování kapitoly bude studující schopen pracovat se základními jednoduchými datovými strukturami jazyka Scheme: symboly a tečkovými páry.

Klíčová slova: Kvotování, symboly, tečkové páry.

Potřebný čas: 2 hodiny.

Veškeré programy, které jsme doposud napsali, pracovaly výhradně s čísly. Je zřejmé, že to v praxi nedostačuje. V této a následujících kapitolách ukážeme, jak lze jako data programu používat i jiné výrazy a jak lze různé výrazy spojovat a vytvářet tak datové struktury.

1.1.1 Symboly a kvotování

Kvotování potlačí vyhodnocení výrazu.

V úvodu do jazyka Scheme jsme se seznámili se symboly, které jsme doposud používali v jazyce Scheme jako proměnné. Jediným smyslem existence symbolu byla jeho možnost vyhodnocení na aktuální vazbu. Aktuální vazba symbolu přitom vznikala pomocí speciální formy `define` nebo při volání procedury.

Symboly je však možno používat i jako plnohodnotný datový typ. Symbol v tu chvíli nepředstavuje název proměnné, ale sama sebe. Je například možné, aby procedura akceptovala symbol jako svůj parametr a vracela symbol jako svůj výsledek.

Problémem však je, že standardní model vyhodnocování jazyka Scheme předpokládá, že symboly jsou vyhodnocovány na svoji aktuální vazbu. Chceme-li namísto aktuálních vazeb symbolů použít symboly jako takové, musíme je *kvotovat* [1].

Tento princip je dobře známý z přirozených jazyků. Přečte-li si například čtenář příkaz: „Řekni nahlas své jméno“, očekává se od něj, že vysloví nahlas své křestní jméno a své příjmení. Přečte-li si pokyn „Řekni nahlas 'své jméno' “, očekává se od něj zřejmě, že nahlas vysloví slovo „své“ a slovo „jméno“. Rozdíl je způsoben tím, že v druhém pokynu byl řetězec (v terminologii jazyka Scheme symbol) uzavřený v apostrofech, a tudíž neoznačoval svoji aktuální vazbu (tj. jméno a příjmení daného člověka), ale představoval sám sebe.

Kvotovat můžeme symboly stejně jako libovolné jiné výrazy jazyka.

V jazyce Scheme nazýváme tento jev kvotování a k tomuto účelu slouží speciální forma `quote`. Tato forma akceptuje libovolný výraz a tentýž výraz vrátí jako výsledek, aniž by jej vyhodnocovala. Kvotovat tudíž můžeme nejen symboly, ale i jakékoliv jiné výrazy jazyka Scheme.

Speciální forma `quote` je natolik užitečná, že pro ni bylo v jazyce Scheme zavedeno synonymum `'` (apostrof). Na rozdíl od přirozených jazyků není nutné umísťovat apostrof před i za kvotovaný výraz, ale vzhledem k jednoznačnosti prefixového zápisu stačí umístit jeden apostrof před výraz. V našem textu budeme preferovat tuto jednodušší syntaxi. Následující záznam interakce s překladačem jazyka Scheme demonstruje využití speciální formy `quote`.

```
pokus
reference to undefined identifier: pokus
> (quote pokus)
pokus
```

[1] Název kvotování vznikl z anglického slova quoting, čtenář se může setkat v literatuře také s doslovným překladem – uvozování.

```
> 'pokus
pokus
> '(+ 1 2)
(+ 1 2)
> '(1 2 3)
(1 2 3)
> '(josef petr andrea)
(josef petr andrea)
```

Při této interakci s překladačem jsme si nejprve ověřili, že `pokus` je nedefinovaný symbol. Dále jsme pomocí speciální formy `quote` potlačili vyhodnocení tohoto symbolu a překladač nám vrátil samotný symbol. Obdobně jsme potlačili vyhodnocení seznamu `(+ 1 2)` a seznamu `(1 2 3)`. Druhý ze jmenovaných seznamů ostatně ani není možné vyhodnotit, neboť `1` není procedura.

Kvotování umožňuje využívat výrazy jazyka jako datové položky.

Za pomocí speciální formy `quote` můžeme nadefinovat procedury pracující se symboly jako s datovými položkami. Takto by například vypadala procedura, která překládá číslovky z anglického do českého jazyka:[1]

```
(define (preklad slovo)
  (cond ((equal? slovo 'zero) 'nula)
        ((equal? slovo 'one) 'jedna)
        ((equal? slovo 'two) 'dvě)
        ((equal? slovo 'three) 'tři)
        ((equal? slovo 'four) 'čtyři)
        ((equal? slovo 'five) 'pět)
        ((equal? slovo 'six) 'šest)
        ((equal? slovo 'seven) 'sedm)
        ((equal? slovo 'eight) 'osm)
        ((equal? slovo 'nine) 'devět)
        (else 'špatné_slovo)))
```

Tato procedura používá symboly jako datové položky. Akceptuje symboly jako jsou `zero`, `one`, `two` atd. a vyhledává k nim příslušné ekvivalenty. Například:

```
> (preklad 'six)
šest
> (preklad 'ten)
špatné_slovo
```

Apostrof před symboly `six` a `ten` jsou přitom naprosto podstatné. Pokud bychom je neuvedli, překladač by takovýto symbol bral jako proměnnou a pokusil by se jej vyhodnotit.

Průvodce studiem

Pro podobné účely by bylo v jazyce Scheme vhodnější použít datové struktury zvané řetězce. Na rozdíl od symbolů, řetězce jsou označeny dvojitými uvozovkami na začátku i na konci. Vyhodnocují se samy na sebe. Řetězce bychom potřebovali zejména v případě, že bychom na počítači zpracovávali texty.

1.1.2 Tečkové páry

V úvodu do jazyka Scheme jsme explicitně hovořili o třech datových typech: o číslech, logických konstantách a symbolech.[1] K reprezentaci složitějších dat však tyto typy často nestačí. Pro

[1] Procedura `equal?`, kterou v tomto kódu používáme, testuje shodnost dvou výrazů. Pracuje tedy podobně jako procedura `=` s tím rozdílem, že `=` pracuje pouze s čísly zatímco procedura `equal?` umí pracovat s jakýmikoli datovými typy. Lze ji tedy použít i na testování shodnosti dvou symbolů.

budování složitějších datových typů nám programovací jazyk Scheme poskytuje jednoduchý způsob jak sdružit dvě hodnoty libovolného typu: *tečkový pár*.

Tečkový pár je formálně zapsán jako `(<výraz1> . <výraz2>)`, kde `<výraz1>` a `<výraz2>` jsou libovolné výrazy jazyka Scheme. Tečka je od výrazů oddělena zleva i zprava mezerou. Syntaxe tečkových párů připomíná syntaxi seznamu. Mezi seznamy a tečkovými páry je však dost podstatný rozdíl, který objasníme v kapitole 1.2.

Tečkové páry umožňují spojit libovolné dva výrazy do jedné datové struktury.

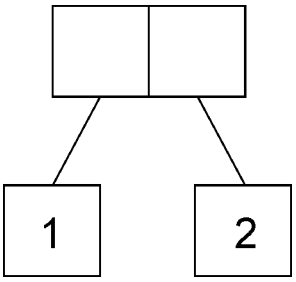
Pomocí tečkových párů můžeme navzájem spojovat čísla, logické konstanty i symboly, například:

- `(-1 . 2)` může být reprezentace polohy bodu v rovině. Vzhledem k počátku má bod x-ovou souřadnici -1 a y-ovou souřadnici 2.
- `(petr . 585254644)` může být část telefonního seznamu. Petr má telefonní číslo 585 254 644.
- `(work . práce)` může být část slovníku, obsahující informaci, že anglické slovo `work` se do češtiny překládá jako `práce`.
- `(program . 205)` může být část statistiky prováděné nad tímto textem: řetězec `program` se v něm vyskytuje 205 krát.

Průvodce studiem

Tečkové páry představují jakési univerzální datové lepidlo. Pomocí tečkového páru můžeme slepit libovolné dva výrazy a

Implementace tečkového páru `(1 . 2)` je pomocí tzv. *krabičkového diagramu* znázorněna na obrázku Obr. 1.



Obr. 1 Implementace tečkového páru

Problém nastane v okamžiku, kdy se pokusíme tečkový pár vyhodnotit:

Tečkové páry obecně nelze vyhodnotit.

```
> (1 . 2)
application: bad syntax (illegal use of '.') in: (1 . 2)
```

Chybové hlášení počítače se v tuto chvíli nebudeme pokoušet interpretovat. Spokojíme se s tvrzením, že tečkový pár není v jazyce Scheme použit jako řídící struktura, ale jako

[1] Uváděli jsme si i datový typ seznam, ale používali jsme jej jen jako prostředek k aktivaci procedury. Nezmiňovali jsme se ani o možnostech použití datového typu procedura.

prostředek datové abstrakce. Pokus o vyhodnocení tečkového páru proto skončí chybou[1]. Chceme-li tečkový pár používat k reprezentaci dat, potřebujeme jej kvotovat, stejně jako jsme v minulé kapitole kvotovali ostatní výrazy.

```
> (quote (1 . 2))
 (1 . 2)
> '(1 . 2)
(1 . 2)
```

1.1.3 Procedury pro práci s tečkovými páry

K datové struktuře musíme mít konstruktory a selektory.

Tečkové páry představují jednoduchou *datovou strukturu*. Na rozdíl od tzv. skalárních typů, datové struktury mají vždy několik částí. Ke každé datové struktuře musíme v jazyku nadefinovat dva typy procedur: *konstruktor*, který akceptuje jednotlivé části a vytvoří z nich datovou strukturu, a *selektory*, které akceptují datovou strukturu a vrátí nám jednotlivé její části. Selektory nám tak umožňují nahlédnout do datové struktury.

Konstruktorem tečkových párů je v jazyce Scheme procedura `cons`, selektory jsou procedury `car` a `cdr`. Jejich použití demonstrujme na příkladu:

```
> (cons 1 2)
(1 . 2)
> (define p (cons 'a 'b))
> p
(a . b)
(car '(1 . 2))
1
(cdr '(1 . 2))
2
(car p)
a
(cdr p)
b
```

Pro práci s tečkovým párem máme procedury `cons`, `car`, `cdr` a `pair?`.

Procedura `cons` akceptuje dva parametry a vytvoří tečkový pár. První parametr procedury bude umístěn před tečkou a druhý parametr bude umístěn za tečkou. Procedura `car` akceptuje tečkový pár a vrací jeho první položku (výraz před tečkou). Procedura `cdr` akceptuje tečkovaný pár a vrací jeho druhou položku (výraz za tečkou).

Navíc máme v jazyce k dispozici i predikát `pair?`, který testuje, jestli jeho parametr je nebo není tečkový pár.

Průvodce studiem

Názvy procedur `car` a `cdr` zůstaly v jazyce Scheme jako relikty původních implementací jazyka Lisp. Zkratka CAR původně znamenala „Content of Adress Register" zatímco zkratka „CDR" znamenala „Content of Decrement Register". Dnes již tečkové páry nemají s registry nic společného. Čtenář by proto v těchto názvech neměl hledat mnemotechnické významy.

1.1.4 Příklad: implementace komplexních čísel

Komplexní čísla.

Ukažme si použití těchto procedur na jednoduchém příkladu. Budeme chtít implementovat vlastní malý systém pro počítání s komplexními čísly. Jako reprezentaci komplexního čísla zvolíme tečkový pár, jehož první složka bude reprezentovat reálnou část komplexního čísla a

[1] … což ovšem obecně neplatí, jak ukážeme v další kapitole.

druhá složka imaginární část komplexního čísla. Výraz `(1 . 2)` bude tedy označovat komplexní číslo 1 + 2i. Napíšeme procedury `c+` a `c*` pro sčítání a násobení dvou komplexních čísel. Tyto procedury akceptují dvě komplexní čísla `c1` a `c2` a vrací výsledek – komplexní číslo.

```
(define (c+ c1 c2)
  (cons (+ (car c1) (car c2))
        (+ (cdr c1) (cdr c2))))

(define (c* c1 c2)
  (cons (- (* (car c1) (car c2)) (* (cdr c1) (cdr c2)))
        (+ (* (car c1) (cdr c2)) (* (cdr c1) (car c2)))))
```

Příklad použití těchto procedur vidíme na následující interakci s překladačem jazyka Scheme:

```
> (c+ '(0 . 1) (c* '(1 . 1) '(3 . 1)))
(2 . 5)
```

Průvodce studiem

Zavedeme-li datové struktury, vynikne nám v programování problém typové kontroly. Selektory `car` a `cdr` očekávají jako své parametry vždy tečkový pár. Pokud chybou programu dostanou jako parametr jiný datový typ, například číslo, dojde k zastavení výpočtu. Tato chybová hlášení typicky signalizují chybu v logice programu.

1.1.5 Vrstvová architektura programu

Program by měl být psán v logických vrstvách.

I když na první pohled vypadá předchozí program na sčítání a násobení reálných čísel zcela korektně, neměli bychom s ním být spokojeni. Kód obou procedur je obtížně čitelný a je zcela nejasné, co vlastně procedury počítají. Srovnejme tento kód s následujícím programem.

```
(define (komplex r i)
  (cons r i))

(define (real c)
  (car c))

(define (imag c)
  (cdr c))

(define (c+ c1 c2)
  (komplex (+ (real c1) (imag c2))
           (+ (real c1) (imag c2))))

(define (c* c1 c2)
  (komplex (- (* (real c1) (real c2)) (* (imag c1) (imag c2)))
           (+ (* (real c1) (imag c2)) (* (imag c1) (real c2)))))
```

Pozorný čtenář zjistí, že tento kód je v zásadě totožný s předchozím. Pouze namísto procedury `cons` využíváme proceduru `komplex`, namísto `car real` a namísto `cdr imag`. Rozdíl je tedy pouze ve jménech procedur, nikoli v algoritmu. Navíc lze kriticky podotknout, že druhý zápis je podstatně delší a program bude navíc pracovat pomaleji.

Přesto budeme tento zápis preferovat z důvodu, který všechny ostatní zastíní: procedury pro práci s komplexními čísly jsou snadno čitelné a jsou odděleny od fyzické implementace tečkových párů v jazyce. Procedury `komplex`, `real` a `imag` vkládají mezi tečkové páry a komplexní aritmetiku *abstraktní bariéru*. Program je tak členěn do *vrstev*, každá vrstva je v zásadě samostatná a navazuje na další vrstvy pouze jasně definovaným rozhraním. Interní změna jedné vrstvy tak neovlivní vrstvy jiné.

Průvodce studiem

Pokud čtenář tohoto textu za sebou nemá rozsáhlejší programový projekt, jen těžko vrstvovou architekturu ocení. Faktem je, že programování v logických vrstvách je dnes již absolutním průmyslovým standardem. U rozsáhlejších aplikací se například setkáváme s tzv. trojvrstvým modelem, kdy je zvlášť řešena práce s databází, logika aplikace a prezentace aplikace. Chceme-li pak přenést aplikaci na jiný typ databáze, je ovlivněna jen jediná vrstva programu. Běžně se s vrstvovou architekturou setkáváme také v knihovnách pro práci s počítačovou sítí. Počet a rozhraní jednotlivých vrstev je zde dokonce určeno normou. Program, který chce komunikovat po síti, pak může pracovat s nejvyšší vrstvou a nemusí se starat o to, kde je nejbližší směrovač, nebo na jaký typ sítě je počítač připojen.

Shrnutí

Speciální forma `quote` potlačí vyhodnocení libovolného výrazu a vrátí samotný výraz. Namísto slova `quote` používáme apostrof před výrazem. Kvotované symboly můžeme používat jako datové prvky. Tečkové páry představují nejjednodušší datovou strukturu. Vytváříme je konstruktorem `cons`, k prvkům přistupujeme selektory `car` a `cdr`. U větších programů používáme vrstvovou architekturu pro oddělení logiky aplikace od implementace datové struktury.

Pojmy k zapamatování

- Kvotování, speciální forma quote,
- tečkový pár,
- skalární proměnná,
- datová struktura,
- konstruktory, selektory
- procedury `cons`, `car` a `cdr`,
- vrstvová architektura programu.

Kontrolní otázky

1. *Co vrací speciální forma quote?*
2. *Co to je skalární proměnná?*
3. *Jak se obecně vyhodnocují tečkové páry?*
4. *Jaké procedury máme k dispozici pro práci s tečkovými páry?*
5. *Jaké jsou výhody vrstvové architektury programu?*

Cvičení

1. Napište proceduru `otoc`, která akceptuje tečkový pár a vrací tečkový pár s přehozenými prvky před a za tečkou. Například:

```
> (otoc '(prvni . druhy))
(druhy . prvni)
```

2. Bod v rovině lze reprezentovat jako tečkový pár jeho souřadnic v kartézském systému. Implementujte konstruktor `bod` a selektory `bod-x` a `bod-y` pro práci s bodem. Dodržujte přitom vrstvovou architekturu programu. Dále napište proceduru `vzdalenost`, která

spočte vzdálenost dvou bodů v rovině.

Úkoly k textu

1. Bez použití počítače určete nebo odhadněte, co vypíše Scheme jako reakci na následující výrazy (předpokládejte, že výrazy jsou postupně zadávány překladači Scheme). Přesvědčte se na počítači o správnosti svých úvah. Pokud se výsledky liší od vašich úvah, objasněte proč.

```
> '123
> 'pokus
> ''pokus
> '''pokus
> '(1 2 3)
> '('1 2 3)
> (define pokus 'pokus)
> pokus
> (define test pokus)
> test
```

2. Napište proceduru `signum2`, akceptující číslo a vracející symbol `kladne`, `zaporne` nebo `nula` na základě znaménka daného čísla. Například:

```
> (signum2 123)
kladne
> (signum2 -123)
zaporne
```

3. Bez použití počítače určete nebo odhadněte, co vypíše Scheme jako reakci na následující výrazy. Přesvědčte se na počítači o správnosti svých úvah. Pokud se výsledky liší od vašich úvah, objasněte proč.

```
> (2 . 1)
> (2.1)
> (quote (2 . 1))
> (quote (2.1))
> '(2 . 1)
> ''(2 . 1)
> '('(+ 1 1) . 2)
> '((+ 1 1) . 2)
```

4. Bez použití počítače určete nebo odhadněte, co vypíše Scheme jako reakci na následující výrazy (předpokládejte, že výrazy jsou postupně zadávány překladači Scheme). Přesvědčte se na počítači o správnosti svých úvah. Pokud se výsledky liší od vašich úvah, objasněte proč.

```
> (cons (+ 1 2) (- 3 4))
> (car (cons 'dobry 'den))
> (cdr (cons 'dobry 'den))
> (define pozdrav (cons dobry den))
> (define odpoved '(dobry . den))
> (cons (cdr pozdrav) (car odpoved))
```

5. Úsečka v rovině je reprezentována dvěma body. Napište konstruktor `usecka` a selektory `usecka-a` a `usecka-b` pro práci s úsečkou za využití existující implementace bodu. Dodržujte přitom vrstvovou architekturu programu. Dále naprogramujte proceduru `stred`, která vypočítá a vrátí bod ve středu úsečky.

Řešení

```
1. (define (otoc p)
     (cons (cdr p) (car p)))

2. (define (bod x y)
     (cons x y))

   (define (bod-x b)
     (car b))

   (define (bod-y b)
     (cdr b))

   (define (vzdalenost b1 b2)
     (sqrt (+ (sqr (- (bod-x b1) (bod-x b2)))
              (sqr (- (bod-y b1) (bod-y b2))))))
```

1.2 Projekt: časová aritmetika

Studijní cíle: Při řešení projektu si studující vedle algoritmizačních dovedností procvičí použití základních operací s tečkovými páry. Pochopí důležitost vrstvové architektury programu.

Klíčová slova: Časová aritmetika, tečkové páry, vrstvová architektura.

Potřebný čas: 3 hodiny.

Cílem projektu je vytvořit počítačovou reprezentaci času a implementovat základní operace pro práci s časem.

1.2.1 Fyzická implementace času

Čas lze v počítači reprezentovat několika způsoby.

Při řešení tohoto projektu nejprve navrhneme nejnižší (fyzickou) implementační vrstvu. Čas zde bude reprezentován pomocí nějak zvolené datové struktury programovacího jazyka. Tato implementační vrstva se bude skládat jednak z popisu syntaxe a sémantiky zvolené datové reprezentace, jednak ze sady konstruktorů a selektorů pro práci s touto reprezentací. Tyto procedury budeme v dalších vrstvách projektu vždy důsledně používat pro přístup k datové struktuře.

Čas dne budeme chápat jako aktuální hodinu a minutu. Čas tedy bude vyjádřen jako dvojice čísel, například 7:25 nebo 21:30. Pro tuto implementaci se nám v jazyku Scheme nabízí použití tečkového páru. Rozhodněme se proto, že čas bude vyjádřen jako tečkový pár hodiny (v rozmezí 0-23) a minuty (v rozmezí 0-59). Tečkový pár `(7 . 25)` tak bude představovat 7:25 ráno, `(21 . 30)` bude představovat 21:30 večer. Tečkové páry `(25 . 60)` nebo `(50 . 7)` nebudou sémanticky správné reprezentace času.

Průvodce studiem

Můžeme si představit, že náš balík pro práci s časem bude využíván například v rámci informačního systému na letišti. Má-li letadlo přiletět v 15:40 a má ohlášeno 90 minut zpoždění,systém musí být schopen určit, že letadlo dorazí v 17:10 . Jestliže má být sytém použitelný v různých zemích, musí být také schopen konvertovat tento údaj na 5:10 pm.

Pro takto navrženou reprezentaci navrhneme konstruktor a selektory. Tyto procedury přímo napojíme na systémové procedury `cons`, `car` a `cdr` bez jakékoli přidané logiky. V tomto projektu se přidržíme anglického označení procedur.

Ke zvolené reprezentaci vytvoříme přístupové procedury.

```
(define (make-time h m)
  (cons h m))

(define (hour t)
  (car t))

(define (minute t)
  (cdr t))
```

Náš balík lze zatím použít například následujícím způsobem:

```
> (define t (make-time 16 40))
> (hour t)
16
> (minute t)
40
```

Úkoly k textu

1. Zapište tento kód do prostředí jazyka Scheme a otestujte.

2. Napište proceduru `print-time`, která akceptuje čas `t` v naší reprezentaci a vytiskne (s použitím procedury `display`) časovou informaci. Procedura pracuje pomocí vedlejšího efektu a nezajímá nás, co vrací. Například:

   ```
   > (define t (make-time 16 40))
   > (print-time t)
   16:40
   ```

1.2.2 Logická implementace času

Další přístupové procedury vytvoříme pro odlišný přístup k datové struktuře.

Nad takto navrženou vrstvou můžeme vybudovat sadu logických konstruktorů a selektorů. v mnoha zemích světa se například využívá symbol AM pro dopoledne a PM pro odpoledne. Hodinový údaj je pak vždy v rozmezí 1-12. Cílem nyní bude vytvořit konstruktor `make-time12` a selektory `hour12` a `ampm`. Procedury musí pracovat následujícím způsobem:

```
> (define t (make-time12 4 40 'pm))
> (print-time t)
16:40
> (hour12 t)
4
> (ampm t)
pm
```

Dříve, než tyto procedury implementujeme, pokusme se pochopit problém a vyplnit následující tabulku. Tato tabulka nám později bude sloužit i jako zdroj testovacích dat. Vlevo máme náhodně vybraný časový údaj ve 24-hodinové notaci, vpravo dopíšeme zápis tohoto časového údaje ve 12-ti hodinové notaci včetně symbolů AM nebo PM.

24-hodinový čas	12-hodinový čas
0:20	
4:20	

12:20	
16:20	

Úkoly k textu

3. Napište konstruktor `make-time12`, který bude konstruovat naši reprezentaci času na základě třech parametrů: hodiny (1-12), minuty (0-59) a symbolu `am` nebo `pm`. POZOR: tato procedura nesmí využívat zabudovanou proceduru `cons`. Namísto toho použijte procedury z kapitoly 1.2.1.

4. Napište selektory `hour12` a `ampm`. Ani zde nepoužívejte systémové procedury `car` nebo `cdr`, ale použijte procedury z kapitoly 1.2.1.

5. Napište proceduru `print-time12`, která bude pracovat obdobně jako procedura `print-time`, vytiskne však časové informace v dvanáctihodinovém formátu.

6. Otestujte vaše procedury na časových údajích z předchozí tabulky.

Průvodce studiem

Využívání procedury `make-time` namísto procedury `cons` může vypadat naprosto zbytečné. Při realizaci většího projektu se nám to však vždy vyplatí. Náš program bude jednak přehlednější, ale hlavně si tak necháváme otevřená zadní dvířka pro případ, že by se fyzická implementace měla v budoucnu změnit.

1.2.3 Časová aritmetika

Časová aritmetika musí řešit problém přetečení a podtečení minut a hodin.

Dalším úkolem je implementovat jednoduchou časovou aritmetiku. K danému časovému intervalu potřebujeme přičíst nebo odečíst daný počet minut. Pro tento účel nadefinujeme procedury `add-minute` a `sub-minute`. Při přičítání musíme dát pozor na přelom hodiny a přelom dne. Přičteme-li například 130 minut k času 16:40, nedostaneme 16:170 ale 18:50. Přičteme-li 130 minut k času 22:40 nedostaneme 24:50 ale 0:50. U odčítání narazíme navíc na problém záporných čísel. Odečteme-li 130 minut od času 1:40, nedostaneme -1:30 ale 23:30.

Následující kód demonstruje použití těchto procedur.

```
> (print-time (add-minute (make-time 16 40) 130))
18:50
> (print-time (add-minute (make-time (22 50) 130))
1:0)
> (print-time (add-minute (make-time 22 40) 130))
0:50
> (print-time (sub-minute (make-time 1 40) 130))
23:30
```

Průvodce studiem

Pro řešení tohoto úkolu můžete zvážit využití Scheme procedur `modulo` *a* `quotient`*. Procedura* `modulo` *pracuje podobně jako nám známá procedura* `remainder`*, vrací však kladný výsledek v situaci, kdy je první parametr záporný.*

Úkoly k textu

7. Implementujte procedury `add-minute` a `sub-minute` a otestujte je na předchozí ukázce kódu.

1.2.4 Časové predikáty

Časové predikáty jsou definované intuitivně.

Posledním úkolem je implementovat časové predikáty. Rádi bychom zjistili, jestli nějaký okamžik v čase nastává *před* nebo *po* jiném časovém okamžiku. Tento problém je složitý v tom, že pojmy *před* a *po* nejsou v 24 hodinovém cyklu jasně definovány. Musíme zde tedy použít zdravý selský rozum a pokusit se navrhnout řešení, které bude většině normálních lidí připadat jako přirozené. Nejprve si proto sami odpovězme na tyto otázky: je 12:10 před 12:20? Je 12:20 před 15:40? Je 12:20 před 10:50? Je 0:10 před 23:50? Je 23:50 po 0:10?

Průvodce studiem

Doufáme, že většina čtenářů došla k názoru, že jeden čas následuje po druhém, pokud rozdíl pozdějšího a dřívějšího času je menší než než 12 hodin.

Úkoly k textu

8. Implementujte predikáty `time-before?`, `time-after?` a `time-within?`, které pracují tak, jak je ukázáno na následujícím příkladě. Procedury důkladně otestujte i na dalších příkladech!

```
> (time-before? (make-time 12 20) (make-time 15 40))
#t
> (time-after? (make-time 0 10) (make-time 23 40))
#t
> (time-within? (make-time 0 10) (make-time 23 15) (make-time 1 05))
#t
```

1.2.5 Změna fyzické implementace

Členění do vrstev umožňuje změnu fyzické implementace..

V kapitole 1.2.3 jste možná došli k názoru, že nejjednodušší pro přičítání a odečítání minut bylo konvertovat časový údaj na počet minut uplynulých od půlnoci (počet hodin krát 60 plus počet minut), k tomuto číslu přičíst nebo odečíst požadovaný počet minut, výsledné číslo upravit modulo 1440 a konvertovat zpět na hodiny a minuty pomocí procedur pro celočíselné dělení a výpočtu zbytku po dělení.

Možná jste také v danou chvíli zjistili, že celá naše fyzická implementace času je zbytečně složitá a čas by bylo možno reprezentovat namísto tečkového páru jednoduše jako jediné číslo.

Průvodce studiem

To je také skutečná fyzická implementace data a času v některých počítačových systémech. Datum a čas bývají reprezentovány jako počet sekund, uplynulých od půlnoci 1.1.1970, měřeno v anglickém Greenwichi. Tento údaj je znám jako tzv. UNIX timestamp. V jazyce Scheme nám tento údaj vrací volání procedury `(current-seconds)`*. Tato implementace je nesmírně výhodná pro účely časové aritmetiky a porovnávání. Zjištění, jestli je soubor starší než jeden den, lze jednoduše provést porovnáním rozdílu časových značek a čísla 86400. Nemusíme se přitom ohlížet na časová pásma, dny v měsíci, měsíce v roce, přestupné roky atd.*

Upravme proto konstruktory a selektory fyzické vrstvy takto:

```
(define (make-time h m)
  (+ (* h 60) m))

(define (hour t)
  (quotient t 60))

(define (minute t)
  (remainder t 60))
```

Úkoly k textu

9. Zopakujte veškeré testy z kapitol 1.2.2, 1.2.3 a 1.2.4. Pokud váš kód správně využíval vrstvovou architekturu programu, programy by měly fungovat stejně jako doposud.

Shrnutí

Při řešení tohoto projektu si studující procvičili práci s tečkovými páry a datovými strukturami. Projekt ukázal postupné budování balíku procedur po jednotlivých vrstvách, přičemž velká pozornost byla věnována datové abstrakci. Počítačová reprezentace času byla chápána jako abstraktní veličina, ke které přistupujeme a kterou manipulujeme pomocí navržených procedur. Tato myšlenka byla zdůrazněna změnou fyzické implementace, která nezasáhla ostatní vrstvy programu.

1.3 Projekt: intervalová aritmetika

Studijní cíle: Při řešení projektu si studující procvičí použití vrstvové architektury programu.

Klíčová slova: Intervalová aritmetika, tečkové páry, vrstvová architektura programu.

Potřebný čas: 2 hodiny.

Intervalová aritmetika se používá v elektrotechnice.

V elektrotechnické praxi je často třeba pracovat s hodnotami, které známe jen s jistou přesností. Elektrický odpor rezistoru může být například uváděn jako 3.5 Ω ± 15%, což znamená, že skutečný elektrický odpor součástky může být kdekoliv v intervalu 2.975 Ω a 4.025 Ω. V tomto cvičení navrhneme a implementujeme intervalovou aritmetiku, která bude umožňo-

umožňovat výpočty s právě takovými hodnotami.

1.3.1 Fyzická implementace intervalu

Pro reprezentaci intervalu v jazyku Scheme použijeme zřejmě tečkové páry. Není však zřejmé, jaké hodnoty se mají nacházet před tečkou a jaké za tečkou. Existují v zásadě dvě zcela ekvivalentní možnosti:

- první číslo páru může obsahovat začátek intervalu a druhé číslo konec intervalu,
- první číslo páru může představovat střed intervalu a druhé číslo toleranci v procentech.

Stejný interval lze tak zobrazit podle zvolené reprezentace jako `(2.975 . 4.025)` nebo `(3.5 . 15)`. Je v podstatě jedno, kterou reprezentaci si vybereme. Náhodně proto vyberme například první z uvedených možností. Implementujeme konstruktor pro vytvoření této datové struktury (`make-interval-bounds`) a dva selektory (`lower-bound` a `upper-bound`).

```
(define (make-interval-bounds lb up)
  (cons lb up))

 (define (lower-bound i)
  (car i))

(define (upper-bound i)
  (cdr i))
```

1.3.2 Logické přístupové procedury

Další sada procedur bude pracovat nad stejnou datovou strukturou, bude k ní však přistupovat pomocí středu intervalu a tolerance. Načrtněme implementaci konstruktoru `make-interval-tolerance` a selektorů `center` a `tolerance`.

Interval lze reprezentovat pomocí mezí nebo tolerance.

```
(define (make-interval-tolerance c t)
  ..........)

(define (center i)
  (/ (+ (lower-bound i) (upper-bound i)) 2))

(define (tolerance i)
  ..........)
```

Úkoly k textu

1. Napište do prostředí jazyka Scheme procedury z kapitol 1.3.1 a 1.3.2. Doplňte chybějící části procedur `make-interval-tolerance` a `tolerance`. Otestujte váš program na následující interakci:

```
> (define i (make-interval-bounds 2.975 4.025))
> (define j (make-interval-tolerance 3.5 15))
> (lower-bound i)
2.975
> (lower-bound j)
2.975
> (upper-bound i)
4.025
> (upper-bound j)
4.025
> (center i)
3.5
> (center j)
3.5
> (tolerance i)
15
```

```
> (tolerance j)
15
```

1.3.3 Intervalová aritmetika

Pravidla intervalové aritmetiky.

V elektrotechnické praxi je třeba s takto nepřesně zadanými intervalovými hodnotami počítat. Zapojíme-li například do série dva odpory, jejichž rezistence bude 3.5 Ω ± 15% a 7.0 Ω ± 5%, bude odpor celého obvodu 10.5 Ω ± 8.3%. Pokud by totiž oba rezistory byly shodou okolností na své spodní mezi tolerance, bude odpor tvořen součtem 2.975 Ω + 6.65 Ω = 9.625 Ω. Pokud by shodou okolností byly na své horní mezi tolerance, bude to 4.025 Ω + 7.35 Ω = 11.375 Ω.

Pravidla intervalové aritmetiky lze (v souhlasu s naší fyzickou reprezentací) formulovat takto:

- $(a,b)+(c,d)=(a+c,b+d)$
- $(a,b)-(c,d)=(a-c,b-d)$
- $(a,b)\times(c,d)=(\min\{ac,ad,bc,bd\},\max\{ac,ad,bc,bd\})$
- $(a,b)\div(c,d)=(a,b)\times(1/d,1/c)$

Úkoly k textu

2. Implementujte procedury `interval-add`, `interval-sub`, `interval-mul` a `interval-div` pro sčítání, odčítání, násobení a dělení intervalů.

1.3.4 Problém na zamyšlení

Celkový odpor obvodu se dvěma paralelně zapojenými rezistory lze vyjádřit dvěma algebraicky identickými vzorci:

$$R=\frac{R_1R_2}{R_1+R_2} \text{ nebo } R=\frac{1}{1/R_1+1/R_2}.$$

Úkoly k textu

3. Implementujte procedury `resist-paralelA` a `resist-paralelB`, které počítají odpor takovéhoto obvodu pomocí těchto dvou vzorců. Dávají obě procedury stejné výsledky. Pokud ne, která z nich je přesnější a proč?

Shrnutí

Intervaly lze reprezentovat jako tečkový pár horní a dolní meze nebo jako tečkový pár středu intervalu a tolerance. Nad zvolenou datovou strukturou lze vybudovat vrstvu fyzických přístupových procedur, logických přístupových procedur a vrstvu intervalové aritmetiky.

2 Seznamy

2.1 Seznamy jako dobře uspořádané tečkové páry

Studijní cíle: Po absolvování kapitoly získá studující schopnost pracovat se seznamy.

Klíčová slova: Seznam, tečkový pár.

Potřebný čas: 1 hodina.

2.1.1 Reprezentace sekvencí

Sekvenci prvků lze pomocí tečkových párů vyjádřit mnoha způsoby.

Jak jsme viděli ve úkolech z minulé kapitoly, tečkové páry mohou jako své prvky obsahovat jiné tečkové páry. Situace, kdy potřebujeme spojit více než dvě hodnoty nastává v praxi velmi často. Chceme-li například vytvořit databázi osobních údajů, potřebujeme spojit příjmení, jméno, datum narození a bydliště. I v tomto případě však plně vystačíme s tečkovým párem. Údaje o osobě lze zapsat pomocí tečkových párů hned několika způsoby, například:

```
((Cimrman . Jara) . (1840 . Liptakov))
(((Cimrman . Jara) . 1840) . Liptakov)
(Cimrman . (Jara . (1840 . Liptakov)))
```

Pro každou z těchto implementací můžeme napsat konstruktor a sadu selektorů, které nám vrátí příjmení, jméno, rok narození a bydliště. Pro první reprezentaci by tyto procedury mohly vypadat například takto:

```
(define (osoba prijmeni jmeno rok bydliste)
  (cons (cons prijmeni jmeno) (cons rok bydliste)))

(define (prijmeni o)
  (car (car o)))

(define (jmeno o)
  (cdr (car o)))

(define (rok o)
  (car (cdr osoba)))

(define (bydliste o)
  (cdr (cdr osoba)))
```

2.1.2 Definice seznamu

Seznam je sekvence vnořených tečkových párů.

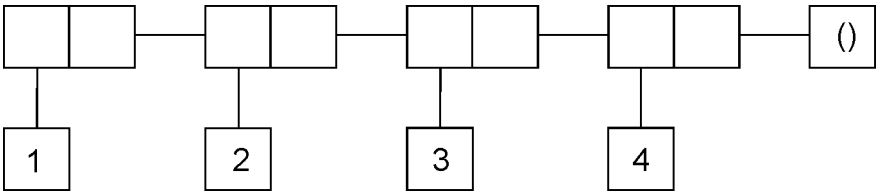
Zápis pomocí několika do sebe vnořených tečkových párů je však poněkud nepřehledný a snadno svádí k programátorských chybám. Tvůrci jazyka Lisp proto pro tyto situace vybrali jednu z implementací a zabudovali do jazyka možnost jejího přehlednějšího zápisu. v našem případě by zvolenou implementační možností bylo:

```
(Cimrman . (Jara . (1850 . (Liptakov . ()))))
```

Zápis poněkud připomíná náš třetí návrh, jako poslední prvek je však do sekvence vložen prázdný seznam. Takto konstruovaný výraz se nazývá *seznam* a dá se zcela ekvivalentním způsobem zapsat jako:

```
(Cimrman Jara 1850 Liptakov)
```

Implementace této datové struktury v paměti, demonstrovaná na krabičkovém diagramu, odpovídá klasické implementaci lineárního spojového seznamu (viz obrázek Obr. 2).



Obr. 2 Spojový seznam

Průvodce studiem

Prázdný seznam je zabudovaným rysem jazyka. Plní zde obdobnou úlohu jako tzv. prázdný ukazatel (null pointer) v jazyku C nebo klíčové slovo `nothing` v jazyku Basic. V dřívějších implementacích jazyka byl prázdný seznam totožný se symbolem `#f`, to však vedlo k nejednoznačnostem a moderní implementace by proto měly mezi `#f` a `()` rozlišovat. Z čistě technického hlediska je jedno, čím budeme seznam ukončovat, je však praktické, aby to byl výraz, který se nepoužívá v žádné jiné souvislosti.

Čtenář se v tuto chvíli jistě právem ptá, jaký je rozdíl mezi seznamy, které představují základní řídící prvek celého jazyka, a mezi seznamy, které jsme si právě zavedli pomocí tečkových párů. Odpověď je, že tyto struktury jsou zcela totožné. Vyhodnocení výrazu:

```
> (+ . (2 . (3 . ())))
```

tedy probíhá zcela stejně jako vyhodnocení výrazu:

```
> (+ 2 3)
```

protože oba výrazy ve skutečnosti **přestavují totéž**. Druhý zápis je jen poněkud elegantnější forma **zobrazení** stejné struktury. Programovací jazyk Scheme tak nedělá rozdíl mezi řídícími a datovými strukturami.

2.1.3 Procedury pro práci se seznamy

Protože seznamy jsou v podstatě jen jistým jasně definovaným způsobem vytvořené tečkové páry, dají se na ně použít i procedury `cons`, `car` a `cdr`. Tyto procedury jen získávají poněkud jiný význam. Demonstrujme si nejprve použití těchto procedur na příkladech a pak slovy formulujme, jaký je jejich efekt.

```
> (cons 1 '(2 3 4))
(1 2 3 4)
> (car '(1 2 3 4))
1
> (cdr '(1 2 3 4))
(2 3 4)
```

Efekt procedur lze slovně zhodnotit takto:

Procedury pro práci s tečkovými páry používáme i pro seznamy.

- Procedura `cons` akceptuje prvek a seznam a přidává daný prvek na začátek seznamu,
- procedura `car` akceptuje seznam a vrací první prvek seznamu,
- procedura `cdr` akceptuje seznam a vrací seznam bez prvního prvku.

Bylo by chybou použít procedury `car` nebo `cdr` na prázdný seznam.

Průvodce studiem

Pokud vám činnost procedur `cons`, `car` nebo `cdr` ve spojitosti se seznamy není jasná, rozepište si seznamy v předchozí interakci s jazykem Scheme pomocí tečkových párů.

Často potřebujeme kombinovat volání několika procedur `car` a `cdr` (to jsme ostatně viděli u prvního příkladu v kapitole 2.1.1.) Protože manuální skládání těchto procedur zhoršuje čitelnost programu, jsou v programovacím jazyce Scheme zavedena synonyma pro složeniny těchto procedur. Některé z nich ukazuje tabulka. V levém sloupci je kombinace volání procedur `car` a `cdr`, v pravém sloupci pak ekvivalentní zabudovaná procedura.

`(car (car x))`	`(caar x)`
`(car (cdr x))`	`(cadr x)`
`(cdr (cdr x))`	`(cddr x)`
`(car (car (cdr x)))`	`(caadr x)`
`(car (cdr (car x)))`	`(cadar x)`
`(car (cdr (cdr x)))`	`(caddr x)`
…	…

Většina datových struktur by měla poskytovat i predikát, který dokáže zjistit, jestli libovolný výraz je nebo není danou strukturou. v tomto případě můžeme využít predikát `list?` Pokud by tento predikát nebyl zabudován v jazyku, můžeme si jej snadno vytvořit. Budeme pro to potřebovat predikáty `pair?` (test na tečkový pár) a predikát `null?` (test na prázdný seznam). Rekurzivní definice seznamu nám pak říká že:

- prázdný seznam je seznam,
- tečkový pár je seznamem pokud prvek za tečkou se seznam.

Tuto definici můžeme snadno převést na proceduru `list?`:

```
(define (list? s)
  (cond ((null? s) #t)
        ((pair? s) (list? (cdr s)))
        (else #f)))
```

Cvičení

1. Rozepište seznamy v argumentech následující sady příkladů na tečkové páry. Napište výsledky vyhodnocení těchto výrazů ve tvaru seznamů a tečkových párů.

   ```
   > (cons 1 '(2 3 4))
   > (car '(1 2 3 4))
   > (cdr '(1 2 3 4))
   ```

2. Vyhodnocením výrazu `(cons 1 (cons 2 ()))` je seznam `(1 2)`. Za pomocí procedury `cons` a prázdného seznamu napište výrazy, které budou vyhodnoceny na seznamy:

   ```
   (1 2 3 4)
   ((1 2) (3 4))
   (1 (2 3) 4)
   (((1)))
   ```

Úkoly k textu

1. Podobně jako v kapitole 2.1.1 implementujte konstruktor a selektory i pro všechny zbývající možné implementace záznamu osoby. Nezapomeňte všechny procedury řádně otestovat.

2. Napište sekvenci příkazů `car` a `cdr`, které z následujících seznamů vrátí číslo 7.

```
(1 (2 3 (5 7) 9))
((7))
(1 (2 (3 (4 (5 (6 7))))))
```

3. Bez použití počítače určete nebo odhadněte, co vypíše Scheme jako reakci na následující výrazy. Přesvědčte se na počítači o správnosti svých úvah. Pokud se výsledky liší od vašich úvah, objasněte proč.

```
> (* . (2 . ((+ . (1 . (2 . ()))) . ())))
> '(1 . (2 . (3 . ())))
> '(1 . (2 3))
> '(1 2 . (3 . ()))
> '('1 '2)
> '('1 . '2)
```

4. Co představuje následující kód v jazyce Scheme?

```
(define . ((sqr . (x . ())) . ((* . (x . (x . ()))) . ())))
```

Řešení

1.
```
> (cons 1 '(2 3 4))
> (cons 1 '(2 . (3 . (4 . ()))))
(1 2 3 4)
(1 . (2 . (3 . (4 . ()))))

> (car '(1 2 3 4))
> (car '(1 . (2 . (3 . (4 . ()))))
1

> (cdr '(1 2 3 4))
> (cdr '(1 . (2 . (3 . (4 . ()))))
(2 3 4)
(2 . (3 . (4 . ())))
```

2.
```
> (cons 1 (cons 2 (cons 3 (cons 4 '()))))
(1 2 3 4)
> (cons (cons 1 (cons 2 '())) (cons (cons 3 (cons 4 '())) '()))
((1 2) (3 4))
> (cons 1 (cons (cons 2 (cons 3 '())) (cons 4 '())))
(1 (2 3) 4)
> (cons (cons (cons 1 '()) '()) '())
(((1)))
```

2.2 Operace se seznamy: průchod seznamem

Studijní cíle: Po zvládnutí kapitoly bude studující schopen pracovat s procedurami, které akceptují seznam a vrací skalární data.

Klíčová slova: Seznamy, prvek seznamu, délka seznamu.

Potřebný čas: 3 hodiny.

Algoritmy pracující se seznamy jsou tradičně rekurzivní a u většiny z nich lze rozpoznat typický „modus operandi“. Uveďme a popišme si alespoň několik základních příkladů algoritmů pracujících se seznamy. V této kapitole se omezíme na procedury, které procházejí seznam a vrací nám o něm skalární informaci.

Procedury na průchod seznamem mají typickou strukturu.

2.2.1 Příklad: prvek seznamu

Prvním příkladem bude predikát `prvek?`, který testuje, zda daný prvek je nebo není členem seznamu.

```
> (prvek? 'c '(d g f c b a))
#t
> (prvek? 'x '(d g f c b a))
#f
```

Označme `x` prvek, jehož přítomnost v seznamu zkoumáme. Stejně jako u libovolné jiné rekurzivní procedury musíme stanovit mezní podmínku rekurze a rekurzivní předpis.

Vždy nejprve stanovíme mezní podmínku rekurze a rekurzivní předpis.

- Mezní podmínka rekurze 1: je-li seznam prázdný, pak neobsahuje žádný prvek. Proto jistě nebude obsahovat ani daný prvek `x`. V tomto případě může procedura vrátit hodnotu `#f`.
- Mezní podmínka 2: pokud seznam není prázdný, pak jistě obsahuje alespoň jeden prvek. Jeho první prvek můžeme zjistit pomocí volání procedury `car`. Je-li hledaný prvek `x` roven prvnímu prvku seznamu, potom seznam prvek obsahuje a my můžeme vrátit jako výsledek `#t`.
- Rekurzivní předpis: pokud seznam není prázdný a první prvek není roven hledanému prvku `x`, můžeme celý problém redukovat na rekurzivní volání procedury `prvek?` se seznamem bez prvního prvku. Seznam bez tohoto prvku lze získat pomocí procedury `cdr`.

V jazyce Scheme bude zápis této procedury vypadat takto:

```
(define (prvek? x s)
  (cond ((null? s) #f)
        ((equal? x (car s)) #t)
        (else (prvek? x (cdr s)))))
```

Procedura `prvek?` je koncově rekurzivní procedura, která postupně probírá jednotlivé prvky seznamu. Narazí-li na prvek `x`, vrátí jako svoji hodnotu symbol `#t`. Dojde-li až na konec seznamu aniž by narazila na prvek `x`, vrátí symbol `#f`.

Celou situaci můžeme demonstrovat na trasování procedury prvek?

```
> (prvek? 'c '(d g f c b a))
|(prvek? c (d g f c b a))
| (prvek? c (g f c b a))
| |(prvek? c (f c b a))
| | (prvek? c (c b a))
| | #t
| |#t
| #t
|#t
#t
> (prvek? 'x '(d g f c b a))
|(prvek? x (d g f c b a))
| (prvek? x (g f c b a))
| |(prvek? x (f c b a))
| | (prvek? x (c b a))
| | |(prvek? x (b a))
| | | (prvek? x (a))
```

```
| | | |(prvek? x ())
| | | |#f
| | | #f
| | |#f
| | #f
| |#f
| #f
|#f
#f
```

Všimněme si, že při každém volání procedury se uplatnila jiná limitní podmínka rekurze. V prvním případě byl prvek nalezen a překladač vrátil hodnotu `#t`. Zbývající prvky seznamu již nemusely být zkoumány. V druhém případě prvek nalezen nebyl a počítač musel prověřit všechny prvky seznamu až narazil na prázdný seznam.

Průvodce studiem

Počítač vždy „vidí" ze seznamu pouze první prvek. Představme si situaci, kdy lidé čekají spořádaně seřazeni za sebou v úzké uličce a malý chlapec mezi nimi hledá své rodiče. Chlapec, protože je malý, vidí vždy pouze první osobu ve frontě. Pokud to není jeho rodič, musí požádat osobu aby ustoupila a on tak uviděl dalšího člověka ve frontě. Takto postupuje dokud buď svého rodiče nenajde nebo dokud není ulička prázdná.První osobu ve frontě nám vrací procedura `car`*, frontu bez první osoby nám vrací procedura* `cdr`*.*

2.2.2 Příklad: délka seznamu

Druhým modelovým příkladem bude výpočet délky seznamu. Délkou seznamu přitom rozumíme počet jeho prvků.

```
> (delka '(d g f c b a))
6
> (delka '(d g a))
3
```

Opět si stanovíme mezní podmínku rekurze a rekurzivní předpis.

- Mezní podmínka rekurze: prázdný seznam neobsahuje žádný prvek, jeho délka je tedy rovna nule.
- Rekurzivní předpis: pokud seznam není prázdný, obsahuje první prvek. Jeho délku můžeme určit tak, že spočteme délku seznamu bez tohoto prvního prvku a k výsledku přičteme jedničku.

Kód v jazyce Scheme vypadá takto:

```
(define (delka s)
  (if (null? s) 0
      (+ 1 (delka (cdr s)))))
```

Výsledek trasování procedury `delka` znázorňuje následující výpis.

```
> (delka '(d g a))
|(delka (d g a))
| (delka (g a))
| |(delka (a))
| | (delka ())
| | 0
| |1
| 2
|3
3
```

Jde tedy o lineárně rekurzivní proceduru, která ve fázi navíjení „odloupne“ všechny prvky seznamu až narazí na prázdný seznam. Ve fázi odvíjení pak spočte kolik prvků v seznamu vlastně je.

Průvodce studiem

V anglickém programátorském slangu se setkáme přímo s výrazem „cdr-ing down the list“ (vyslov [kədəriŋ daun]). Výraz se používá i v přeneseném smyslu. Chceme-li na schůzce přejít k dalšímu bodu programu, můžeme světácky prohodit „Let's cdr down the agenda! “

Tuto proceduru můžeme přepsat i do koncově rekurzivního tvaru. Jak jsme naznačili v předchozím textu, musíme u koncově rekurzivní procedury přidat jeden parametr, který bude sloužit pro kumulaci výsledku. Tento parametr pak budeme vracet v mezní podmínce rekurze.

Některé procedury lze přepsat do koncově rekurzivního tvaru.

```
(define (delka-iter c s)
  (if (null? s) c
      (delka (+ c 1) (cdr s))))

(define (delka s)
  (delka-iter 0 s))
```

Shrnutí

Procedury, které pracují se seznamy, musí rekurzivně projít všechny prvky seznamu dokud nenarazí na konec – prázdný seznam. Mezní podmínka rekurze proto typicky obsahuje volání procedury `null?`, rekurzivní předpis typicky volá procedury se seznamem zkráceným o jeden prvek pomocí procedury `cdr`.

Cvičení

1. Napište proceduru, vracející pozici daného prvku v seznamu. Pokud prvek nebude v seznamu, vraťte číslo 0. Například:

   ```
   > (pozice 'petr '(jan aleš markéta petr josef))
   4
   ```

2. Napište proceduru, vracející n-tý prvek seznamu. Předpokládejte přitom, že seznam má alespoň n prvků. Například:

   ```
   > (nprvek 4 '(jan aleš markéta petr josef))
   petr
   ```

Úkoly k textu

1. Napište proceduru, která zjistí počet výskytů prvku v seznamu. Například:

   ```
   > (pocet-vyskytu 'petr '(marek petr andrea jan jindra petr))
   2
   > (pocet-vyskytu 'petr '(marek ales andrea jan jindra martin))
   0
   ```

2. Napište predikát `stejne?`, který ověří jestli jsou dva seznamy jsou totožné. Například:

```
> (stejne? '(petr jan michal katerina) '(petr jan michal kateřina))
#t
> (stejne? '(petr jan marek katerina) '(petr jan michal kateřina))
#f
```

3. Napište predikát `obsahuje?`, který ověří jestli všechny prvky jednoho seznamu jsou obsaženy v druhém. Například:

```
> (obsahuje? '(petr jan) '(michal petr kateřina jan))
#t
> (obsahuje? '(petr jan) '(michal zbyněk kateřina adolf))
#f
```

4. Napište predikát `podobne?`, který zjistí, jestli dva seznamy obsahují stejné prvky. Využijte přitom predikát `obsahuje?`. Například:

```
> (podobne? '(petr jan michal jan kateřina) '(michal petr kateřina jan))
#t
> (podobne? '(petr jan michal jan kateřina)'(michal zbyněk kateřina jan))
#f
```

Řešení

1. Rekurzivně lze problém řešit následujícím způsobem. Bylo by možné vytvořit i iterativní verzi.

```
(define (pozice x s)
  (cond ((null? s) 0)
        ((equal? x (car s)) 1)
        (else (+ 1 (pozice x (cdr s))))))
```

```
2. (define (nprvek n s)
     (if (= n 1) (car s)
         (nprvek (- n 1) (cdr s))))
```

2.3 Operace se seznamy: konstrukce seznamů

Studijní cíle: Po zvládnutí kapitoly bude studující schopen pracovat s procedurami, které akceptují i vrací seznamy.

Klíčová slova: Seznamy, spojování seznamů, filtrace seznamů, reverze seznamů, třídění seznamů.

Potřebný čas: 3 hodiny.

Dovednosti, které jsme získali v minulé kapitole, nyní doplníme o vytváření procedur, které vrací jako svůj výsledek seznamy. V mnoha případech se jedná o procedury, které akceptují seznam a na výstupu vrací modifikovanou verzi tohoto seznamu. Také tyto procedury mají velmi typickou strukturu. Zatímco na jedné straně ukrajují pomocí `cdr` prvky ze vstupního seznamu, na druhé straně konstruují výsledek pomocí procedury `cons`. Procedury tohoto typu si představíme na několika modelových příkladech.

Procedury zkracují vstupní seznam a současně konstruují výsledný seznam.

2.3.1 Příklad: spojování seznamů

Prvním příkladem bude procedura, která akceptuje dva seznamy a vrátí seznam vzniklý zřetězením těchto seznamů. Například:

```
> (spoj '(1 2 3) '(a b c))
(1 2 3 a b c)
```

Problém budeme řešit postupným průchodem přes první seznam. Pokusme se formulovat mezní podmínku rekurze a rekurzivní předpis.

- Mezní podmínka rekurze: je-li první ze seznamů prázdný, je výsledkem druhý seznam. V tomto případě není co spojovat.
- Rekurzivní předpis: pokud není první seznam prázdný, pak jistě obsahuje první prvek. Tento prvek lze získat voláním procedury `car`. První seznam bez prvního prvku lze získat voláním procedury `cdr`. Náš problém jistě redukujeme na jednodušší, pokud vyvoláme proceduru `spoj` s prvním seznamem bez prvního prvku. Na začátek výsledku tohoto vyhodnocení však ještě musíme přidat první prvek prvního seznamu.

Tento rekurzivní předpis je poněkud komplikovanější, neboť obsahuje jak zkracování seznamu, tak konstrukci nového seznamu. Podívejme se na kód v jazyce Scheme a na výsledek trasování výpočtu.

```
(define (spoj s1 s2)
  (if (null? s1) s2
      (cons (car s1) (spoj (cdr s1) s2))))
```

```
> (spoj '(1 2 3) '(a b c))
|(spoj (1 2 3) (a b c))
| (spoj (2 3) (a b c))
| |(spoj (3) (a b c))
| | (spoj () (a b c))
| | (a b c)
| |(3 a b c)
| (2 3 a b c)
|(1 2 3 a b c)
(1 2 3 a b c)
```

Jak je vidět z trasování výpočtu, procedura generuje lineárně rekurzivní výpočetní proces. Ve fázi navíjení se v každém kroku výpočtu odloží první prvek seznamu `s1` na zásobník a seznam se zkrátí. Jakmile dosáhneme mezní podmínky rekurze, vrátíme druhý seznam jako výsledek. Ve fázi odvíjení se pak na tento výsledek postupně konstruují odložené prvky prvního seznamu.

Průvodce studiem

Pro spojování seznamů existuje v jazyce Scheme zabudovaná procedura append. *My si tento příklad uvádíme jen proto, že názorně demonstruje typickou práci procedury při průchodu seznamem a při konstrukci seznamu.*

2.3.2 Příklad: filtrování seznamu

V druhém modelovém příkladě si ukážeme možnosti odstranění prvku ze seznamu. Procedura `odstran-prvni` odstraní ze seznamu první výskyt zadaného prvku. Například:

```
> (odstran-prvni 'jan '(petr marcela jan marek jan david))
(petr marcela marek jan david)
```

Formulujme si mezní podmínku a rekurzivní předpis.

- Mezní podmínka rekurze 1: pokud je seznam prázdný, neobsahuje žádný prvek. Neobsahuje tedy ani prvek `x` a není co odstraňovat. Výsledkem je prázdný seznam.
- Mezní podmínka rekurze 2: pokud seznam není prázdný, obsahuje první prvek. Je-li tento první prvek náhodou roven prvku `x`, našli jsme první výskyt a výsledkem je seznam bez prvního prvku.
- Rekurzivní předpis: pokud seznam není prázdný a první prvek je různý od prvku `x`, potřebujeme zachovat první prvek seznamu a pokusit se odstranit prvek `x` ze zbytku seznamu. Použijeme tedy stejnou techniku jako v příkladě 2.3.1 a konstruujeme první prvek na rekurzivní volání.

Procedura může mít více mezních podmínek rekurze.

Proceduru a výsledek jejího trasování ukazuje následující text.

```
(define (odstran-prvni x s)
  (cond ((null? s) '())
        ((equal? (car s) x) (cdr s))
        (else (cons (car s) (odstran-prvni x (cdr s))))))
```

```
> (odstran-prvni 'jan '(petr marcela jan marek jan david))
|(odstran-prvni jan (petr marcela jan marek jan david))
| (odstran-prvni jan (marcela jan marek jan david))
| |(odstran-prvni jan (jan marek jan david))
| |(marek jan david)
| (marcela marek jan david)
|(petr marcela marek jan david)
(petr marcela marek jan david)
```

Jako modifikaci této procedury si uvedeme proceduru `odstran-vsechny`, která odstraní nejen první, ale všechny výskyty prvku v seznamu. V návrhu procedury se nám změní druhá mezní podmínka rekurze v další rekurzivní předpis. Doposud jsme po nalezení prvního výskytu prvku v seznamu dále nepostupovali a přímo vrátili zbytek seznamu. V tomto zbytku se ale mohl vyskytovat hledaný prvek. Musíme proto zajistit odstranění prvku i ze zbytku seznamu. Jednotlivé rekurzivní předpisy se od sebe nyní liší jen tím, že v prvním z nich prvek `x` zahazujeme zatímco v druhém jej zachováváme – konstruujeme je na výstup.

```
(define (odstran-vsechny x s)
  (cond ((null? s) '())
        ((equal? (car s) x) (odstran-vsechny x (cdr s)))
        (else (cons (car s) (odstran-vsechny x (cdr s))))))
```

```
> (odstran-vsechny 'jan '(petr marcela jan marek jan david))
|(odstran-vsechny jan (petr marcela jan marek jan david))
| (odstran-vsechny jan (marcela jan marek jan david))
| |(odstran-vsechny jan (jan marek jan david))
| | (odstran-vsechny jan (marek jan david))
| | |(odstran-vsechny jan (jan david))
| | | (odstran-vsechny jan (david))
| | | |(odstran-vsechny jan ())
| | | |()
| | | (david)
| | |(david)
| | (marek david)
| |(marek david)
| (marcela marek david)
|(petr marcela marek david)
(petr marcela marek david)
```

Průvodce studiem

Hovořit o odstranění prvku ze seznamu není zcela přesné. Původní seznam, dodaný jako parametr, se totiž nijak nezmění a prvek v něm zůstane. Co se doopravdy děje, je konstrukce zcela nového seznamu, ve kterém budou prvky z původního seznamu až na fil-

trovaný prvek. Jak původní, tak výsledný seznam pak existují v paměti počítače. To může být v některých případech výhodou, jindy je to nežádoucí. Problém by nastal zejména tehdy, když by se jednalo o rozsáhlé datové struktury a vytváření jejich (i když modifikovaných) kopií by bylo náročné na čas i na paměť počítače. Efektivnější řešení tohoto problému vyžaduje hlubší porozumění mechanismu ukládání dat do paměti a zavedení přiřazovacího příkazu.

2.3.3 Příklad: reverzní spojování seznamů a reverze seznamu

Třetím vyřešeným příkladem bude modifikace procedury `spoj` z kapitoly 2.3.1 nazvaná `rev-spoj`, která spojí dva seznamy s tím rozdílem, že prvky prvního seznamu budou připojeny na druhý v obráceném pořadí.

```
> (rev-spoj '(1 2 3) '(a b c))
(3 2 1 a b c)
```

Průvodce studiem

Možná čtenáře překvapilo, že jsme se v předchozích příkladech nepokusili vytvořit iterativní verze algoritmů. Iterativní verze (verze „bez paměti“) však přináší poněkud odlišné výsledky.

Analýza problému bude velmi podobná té z příkladu 2.3.1, využijeme však koncovou rekurzi.

Některé procedury nelze jednoduše přepsat na iterativní výpočetním proces.

- Mezní podmínka rekurze: je-li první ze seznamů prázdný, je výsledkem druhý seznam. V tomto případě není co spojovat.
- Rekurzivní předpis: jestliže není první seznam prázdný, pak jistě obsahuje první prvek. Tento prvek lze získat voláním procedury `car`. První seznam bez prvního prvku lze získat voláním procedury `cdr`. Připojíme-li pomocí procedury `cons` první prvek prvního seznamu na začátek druhého seznamu, získáme část požadovaného výsledku. Pak už jen stačí na tento seznam rekurzivně zavolat proceduru `rev-spoj`.

Demonstrujme opět tento algoritmus na kódu v jazyce Scheme a na ukázkovém příkladě.

```
(define (rev-spoj s1 s2)
  (if (null? s1) s2
      (rev-spoj (cdr s1) (cons (car s1) s2))))
```

```
(rev-spoj '(1 2 3) '(a b c))
|(rev-spoj (1 2 3) (a b c))
| (rev-spoj (2 3) (1 a b c))
| |(rev-spoj (3) (2 1 a b c))
| | (rev-spoj () (3 2 1 a b c))
| | (3 2 1 a b c)
| |(3 2 1 a b c)
| (3 2 1 a b c)
|(3 2 1 a b c)
(3 2 1 a b c)
```

Průvodce studiem

Napište si všechny zde prezentované kódy do počítače. Vyzkoušejte si, že tyto procedury opravdu fungují. Zapněte si trasování všech procedur a prozkoumejte, jak probíhá vý-

počet. Pro další programování v jazyce Scheme je nezbytné důkladně pochopit princip a implementaci těchto příkladů!

Pomocí procedury `rev-spoj` již snadno implementujeme i proceduru na reverzi seznamu. Jediné, co je potřeba, je připojit daný seznam na prázdný seznam!

```
(define (reverze s)
  (rev-spoj s '()))
```

```
> (reverze '(a b c d e f))
(f e d c b a)
```

Shrnutí

V této kapitole jsme se zabývali procedurami, které akceptují seznam a vrací modifikovaný seznam. Procedury ve svém těle používají proceduru `cons` na konstrukci výsledku. Typicky konstruujeme první prvek vstupního seznamu na výsledek rekurzivního volání (v případě rekurzivního výpočetního procesu) nebo na parametr rekurzivního volání (v případě iterativního výpočetního procesu).

Cvičení

1. Napište proceduru `rev2-spoj`, která pracuje podobně jako procedura `rev-spoj` s tím rozdílem, že obrací druhý seznam namísto prvního, tedy:

   ```
   > (rev2-spoj '(1 2 3) '(a b c))
   (1 2 3 c b a)
   ```

   K psaní této procedury však použijte pouze proceduru `rev-spoj` a žádnou jinou proceduru!

2. Napište proceduru `odstran-vetsi`, která ze seznamu čísel odstraní všechny prvky větší než daný prvek. Například:

   ```
   > (odstran-vetsi 4 '(5 8 2 5 3 7 9 0 4))
   (2 3 0 4)
   ```

Úkoly k textu

1. Napište proceduru `odstran-sude`, která ze seznamu čísel odstraní všechna sudá čísla. Například:

   ```
   > (odstran-sude '(5 8 2 5 3 7 9 0 4))
   (5 5 3 7 9)
   ```

2. Napište proceduru `vloz`, která vloží číslo do seznamu těsně před první větší číslo. Například:

   ```
   > (vloz 5 '(4 2 3 6 7 1))
   (4 2 3 5 6 7 1)
   ```

3. Pomocí procedury `vloz` napište proceduru `setrid`, která setřídí vzestupně seznam čísel. Postupujte tak, aby procedura postupně vkládala prvky do původně prázdného seznamu (tento algoritmus se nazývá třídění přímým vkládáním). Například:

```
> (setrid '(4 2 3 6 7 1))
(1 2 3 4 6 7)
```

Řešení

```
1. (define (rev-spoj2 s1 s2)
     (rev-spoj (rev-spoj s1 '()) (rev-spoj s2 '())))

2. (define (odstran-vetsi x s)
     (cond ((null? s) '())
           ((> (car s) x) (odstran-vetsi x (cdr s)))
           (else (cons (car s) (odstran-vetsi x (cdr s))))))
```

2.4 Projekt: simulátor výtahu

Studijní cíle: Při řešení projektu si studující procvičí použití procedur, představených v minulých kapitolách, na praktickém příkladu. Projekt zároveň slouží jako cvičení analýzy reálného problému.

Klíčová slova: Simulátor výtahu, analýza problému, seznamy.

Potřebný čas: 2 hodiny.

V tomto projektu vytvoříme počítačovou simulaci výtahu v osmipatrové budově. Pro řešení potřebujeme teachpack `elevator.scm`.

2.4.1 Popis problému

Stav výtahu v budově je plně popsán následujícími hodnotami. Tyto hodnoty vstupují jako parametry do procedury, která řídí pohyb výtahu po budově.

Stav systému je popsán skupinou hodnot.

- `d` – směr, kterým se výtah pohybuje (případně kterým se naposledy pohyboval). Může nabýt dvou hodnot: symbolu `up` a symbolu `down`.
- `f` – aktuální patro, ve kterém se výtah nachází.
- `all-calls` – seznam všech pater budovy, které vyžadují zastavení výtahu.
- `up-calls` – seznam všech pater budovy, kde bylo stisknuto tlačítko „nahoru“.
- `down-calls` – seznam všech pater budovy, kde bylo stisknuto tlačítko „dolů“.
- `demands` – seznam všech tlačítek stisknutých uvnitř výtahu.

Všechny seznamy pater jsou vždy setříděny vzestupně od nejnižšího patra po nejvyšší. Seznam `all-calls` je pouze sjednocením seznamů `up-calls`, `down-calls` a `demands`.

Stav výtahu se mění pokaždé, když se změní kterýkoli z parametrů, tedy když výtah změní směr, přesune se z jednoho patra do jiného nebo když je stisknuto libovolné tlačítko. Když výtah zastaví v některém z pater, číslo patra je automaticky odstraněno ze všech seznamů.

Kdykoli se stav výtahu změní, systém zavolá řídící proceduru `controller`. Řídící procedura akceptuje jako parametry všechny stavové hodnoty výtahu a vrací jako svůj výsledek **číslo patra**, do kterého se má výtah přesunout a zastavit.

2.4.2 Jednoduché řešení

Následující kód představuje jednoduché řešení řídící procedury.

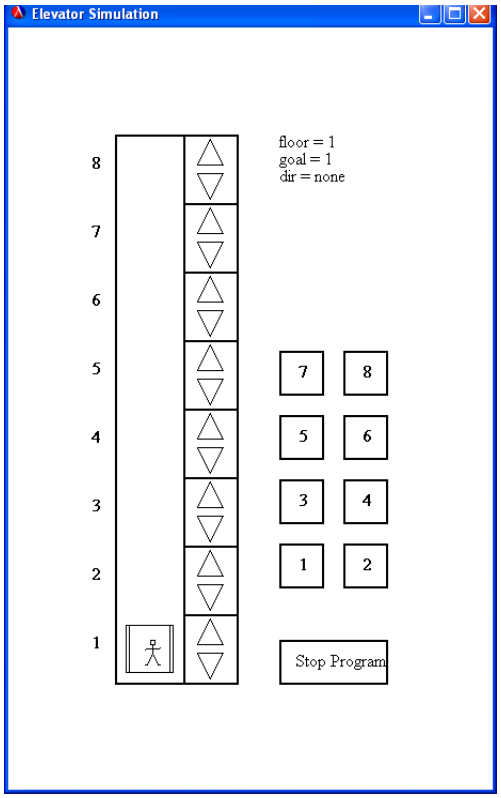
```
(define (controller d f all-calls up-calls down-calls demands)
  (if (null? all-calls)
      f
      (car all-calls)))

(run controller)
```

Původní verze řídící procedury je značně nedokonalá.

Tato verze řídící procedury využívá ze stavových informací pouze parametry `f` a `all-calls`. Ve zkratce lze říci, že pokud není nikde v budově žádný požadavek na výtah (žádné tlačítko není stisknuto a seznam `all-calls` je prázdný), návratová hodnota procedury je `f` a výtah proto zůstává v aktuálním patře. Pokud ale v budově existuje alespoň jeden požadavek na výtah (seznam `all-calls` není prázdný), výtah se přesune do nejnižšího patra, pro které byl vznesen požadavek (první prvek seznamu `all-calls`).

Po spuštění systému uvidíte grafické okno jako na obrázku Obr. 3. Tlačítka ve tvaru šipek odpovídají tlačítkům výtahu v jednotlivých patrech (seznamy `up-calls` a `down-calls`). Čtvercová tlačítka vpravo představují tlačítka uvnitř výtahu (`demands`). Zároveň si všimněte okna interakcí jazyka Scheme, kde jsou vypisovány stavové informace výtahu pokaždé, když je procedura `controller` aktivována.



Obr. 3 Grafické okno teachpacku „elevator“.

Úkoly k textu

1. Přidejte do jazyka teachpack, napište výše uvedený kód a experimentujte se systémem. Pokuste se pochopit princip práce výtahu.

2. Co se stane, když na třetím řádku procedury `controller` nahradíme symbol `f` číslem `1`?

2.4.3 První revize řešení

Řídící proceduru upravujeme na základě uživatelských připomínek.

Jak jsme si ověřili v předchozí kapitole, implementovaná logika výtahu „nějak“ funguje a proto můžeme náš fiktivní výtah předat zákazníkovi k testování. Po jisté době dostaneme od zákazníka následující dvě připomínky.

- „Ve většině případů nám výtah funguje, ale v některých případech nastává následující problém: pokud je výtah na cestě z prvního do posledního patra a někdo v přízemí se pokusí přivolat výtah, kabina z ničeho nic změní směr a vrátí se zpět do přízemí. Pro zaměstnance, jejichž kanceláře jsou v posledním patře, je tak skutečně obtížné dostat se do práce. “
- „S tímto problémem možná souvisí další potíž: pokud je výtah na cestě například z posledního patra do přízemí a někdo chce z výtahu vystoupit řekněme ve čtvrtém patře, výtah zde nezastaví. Sjede až úplně dolů do přízemí a pak se vrátí zpět do čtvrtého patra. “

Úkoly k textu

1. Pochopte problém, který zákazník popisuje, a ověřte, že skutečně nastává.
2. Upravte logiku procedury `controller` tak, aby tento problém nenastával. Prozatím pracujte pouze s parametry `d`, `f` a `all-calls`!
3. Důkladně otestujte upravený systém. Jede-li kabina nahoru, musí zastavit postupně ve všech patrech, kde byl přivolán výtah (ať už byla stisknuta šipka nahoru, dolů, nebo tlačítko ve výtahu). Jede-li výtah dolů, musí se systém chovat obdobně. Výtah se v žádném případě nesmí otočit a změnit směr jízdy nebo přeskočit patro, kde je stisknuté tlačítko výtahu.

Průvodce studiem

Pochopení problému a otestování, že skutečně nastává, je bezpodmínečně nutným předpokladem úspěchu řešení. Pokud problém nebo chybu programu není možné deterministicky vyvolat nelze ji ani řešit. Uživatele svých programů musíme proto „vycvičit“ tak, aby vždy úplně popsali seznam kroků, spolehlivě vedoucích k popisované chybě.

2.4.4 Druhá revize řešení

Pokud se nám podařilo odstranit chyby popisované zákazníkem v předchozí kapitole, dostaneme se k dalším potížím. Uživatel si všimne, že přivolávací tlačítka ve tvaru šipek se chovají zcela stejně – nezáleží na tom, které z nich stiskneme. Při řešení budeme muset začít v kódu používat proměnné `up-calls`, `down-calls` a `demands`.

- „Výtah zcela ignoruje, jestli stiskneme přivolávací tlačítko ve směru nahoru nebo dolů. Konkrétně nastávají tyto dva problémy: pokud je výtah na cestě nahoru a projíždí patrem, kde je stisknuta šipka dolů, výtah přesto zastaví. Obráceně, pokud je výtah na cestě dolů a projíždí patrem kde je stisknuta šipka nahoru, výtah taktéž zastaví. “

Náš výtah by měl fungovat sběrným systémem – při cestě nahoru by měl brát v potaz pouze stisknuté šipky nahoru a při cestě dolů by měl akceptovat pouze stisknutá tlačítka ve tvaru šipky dolů. Nezávisle na směru jízdy by však měl vždy zastavit, pokud někdo z výtahu chce v aktuálním patře vystoupit.

Testování systému je velmi důležité.

Úkoly k textu

4. Upravte logiku procedury `controller` tak, aby vyřešila popisovaný problém. Důkladně otestujte výsledný systém.

5. Proveďte na vašem systému následující testy:

   A. Je-li výtah v přízemí, stiskněte rychle po sobě tlačítko 8, šipku nahoru v šestém patře a šipku dolů v sedmém patře. Výtah musí zastavit v šestém patře, ale nesmí zastavit v sedmém patře.

   B. Je-li výtah v posledním patře, stiskněte rychle po sobě tlačítko 1, šipku dolů ve třetím patře a šipku nahoru ve druhém patře. Výtah musí zastavit ve třetím patře ale nesmí zastavit ve druhém patře.

   C. Je-li výtah v přízemí, stiskněte rychle po sobě šipku nahoru v pátém a sedmém patře a tlačítko 6. Výtah musí zastavit postupně ve všech třech patrech.

   D. Je-li výtah v posledním patře, stiskněte rychle po sobě šipku dolů v čtvrtém a druhém patře a tlačítko 3. Výtah musí zastavit postupně ve všech třech patrech.

   E. Je-li výtah v přízemí, stiskněte rychle po sobě tlačítka 6 a 8 a šipku nahoru v sedmém patře. Výtah musí zastavit postupně ve všech třech patrech.

   F. Je-li výtah v posledním patře, stiskněte rychle po sobě tlačítka 1 a 3 a šipku dolů ve druhém patře. Výtah musí zastavit postupně ve všech třech patrech.

   Pokud jste při testování zjistili nějaké problémy, proceduru upravte.

Průvodce studiem

Většina studujících je překvapena, jak komplikovaná logika je skryta v zařízení tak běžném a zdánlivě jednoduchém, jakým je výtah pracující sběrným systémem. Zároveň se při testování systému přesvědčíme, kolik věcí může být v programu „špatně“, ačkoliv se program chová zdánlivě korektně. Chyby se projevují jen v jistých přesně definovaných situacích.

Shrnutí

V projektu jsme vytvářeli logiku výtahu pracujícího sběrným systémem v osmipatrové budově. Procvičili jsme použití komplexních podmíněných výrazů a řady pomocných procedur z předchozích kapitol. Dále jsme demonstrovali analýzu a řešení problému na základě uživatelských připomínek. Zdůraznili jsme také potřebu komplexního testování programu.

2.5 Projekt: Josephův problém

Studijní cíle: Při řešení tohoto projektu si studující procvičí analýzu problému a základní operace se seznamy.

Klíčová slova: Josephův problém, seznamy.

Potřebný čas: 1 hodina.

Josephův problém řešíme pomocí počítačové simulace.

Tento projekt je založen na poněkud amorálním a morbidním příběhu, který je připisován židovskému vojákovi Flaviu Josephovi. Josephus využil svůj matematický talent k tomu, aby poněkud zbaběle a podvodně unikl z římského zajetí.

V době plného rozkvětu římského impéria byla skupina 41 židovských vzbouřenců obklíčena Římany. Bylo nemožné, aby celá skupina unikla a všech 41 vojáků tak čelilo jisté smrti. Josephovi soudruzi se rozhodli, že než by měli zahynout rukou nepřítele, raději ukončí svůj život sami. Vymysleli následující likvidační schéma. Všech 41 vojáků vytvoří řadu a každý třetí voják v řadě bude ostatními zabit. Tak budou pokračovat až zbude naživu poslední voják, který spáchá sebevraždu.

Josephovi a jeho příteli se tato myšlenka příliš nepozdávala. Spočítali proto, kde se musí v řadě postavit, aby zůstali jako poslední dva vojáci naživu. V tomto případě se Josephus postavil na 16. místo v řadě a jeho přítel na 31. místo v řadě. Poté, co byli ostatní vojáci zabiti, Josephus a jeho přítel pod rouškou noci uprchli.

2.5.1 Vytvoření řady

Problém budeme řešit pomocí simulace. Řada vojáků bude reprezentována seznamem čísel. Kdyby bylo v židovské skupině například pouze 15 vojáků, byla by řada reprezentována seznamem čísel od 1 do 15:

```
(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15)
```

Po té, co byl třetí voják zabit, vznikla by řada:

```
(4 5 6 7 8 9 10 11 12 13 14 15 1 2)
```

V dalším „kroku“ pak dostaneme řadu

```
(7 8 9 10 11 12 13 14 15 1 2 4 5).
```

Nejprve napíšeme pomocnou proceduru, která vytvoří naši řadu.

Úkoly k textu

1. Napište proceduru `vytvor-radu`, která akceptuje číslo `n` a vrátí seznam čísel od `1` do `n`. Například:

   ```
   > (vytvor-radu 10)
   (1 2 3 4 5 6 7 8 9 10)
   ```

2.5.2 Eliminace řady

Dalším krokem je vytvoření samotné procedury `josephus`, která vyřeší Josephův problém. Procedura bude akceptovat počet vojáků ve skupině a modulo `m` – číslo, určující kolikátý voják v pořadí má být odstraněn. Vrátí pak seznam zbývajících `m-1` vojáků. Při řešení použijte následující postup:

- vytvořte pomocnou proceduru, která akceptuje řadu (seznam čísel vytvořený pomocí procedury `vytvor-radu`) a počítadlo; použijte počítadlo k tomu, abyste určili, který voják má být zabit,
- ukončete rekurzivní běh procedury v okamžiku, kdy délka řady bude menší než číslo m,
- posílejte vojáky (čísla v řadě), kteří nemají být odstraněni na konec řady.

Úkoly k textu

2. Napište proceduru `josephus`, která akceptuje délku řady a modulo a řeší Josephův problém. Například:

```
> (josephus 41 3)
(16 31)
```

Shrnutí

Josephův problém je příkladem využití seznamů pro simulaci problému. Při řešení projektu jsme si procvičili práci se seznamy.

2.6 Projekt: polynomiální aritmetika

Studijní cíle: Při řešení projektu studující kromě praxe v práci s datovými strukturami získá představu o využití seznamů pro symbolickou reprezentaci výrazu a vytvoření programů pro manipulaci symbolických výrazů.

Klíčová slova: Seznamy, polynomy, symbolická manipulace.

Potřebný čas: 4 hodiny.

V tomto projektu navrhneme datovou strukturu polynomu jedné neznámé a implementujeme procedury pro práci s touto datovou strukturou. Polynomem přitom rozumíme výraz

$a_n x^n + a_{n-1} x^{n-1} + \ldots + a_1 x + a_0$,

Kde x je proměnná a a_n až a_0 jsou koeficienty u jednotlivých mocnin.

2.6.1 Fyzická implementace polynomu

Polynom reprezentujeme jako sezname jeho koeficientů.

Jako reprezentaci polynomu zde zvolíme seznam koeficientů polynomu v pořadí vzrůstajících mocnin neznámé. Například polynom $3x^3 + 2x - 4$ bude reprezentován seznamem `(-4 2 0 3)`. Nula na třetí pozici seznamu zastupuje nulový koeficient u členu x^2. Seznam `(0 0 0 0 0 1 0 0 1)` bude naší reprezentací polynomu $x^8 + x^5$. Existují jistě i jiné implementace polynomů, pro většinu algoritmů bude však tato implementace nejvhodnější.

V zájmu zjednodušení stanovíme pro naši implementaci následující podmínky:

- Stupeň polynomu bude vždy jednoznačně určen délkou seznamu. Nebudeme proto připouštět seznamy mající nulu na místě nejvyšší mocniny (jako například seznam `(-4 2 0 3 0)`).

- Budeme formálně rozlišovat mezi číslem a polynomem stupně nula (například číslem `5` a polynomem `(5)`).
- Seznam obsahující nulu (`(0)`) a prázdný seznam (`()`) budou v naší implementaci představovat sobě rovné implementace nulového polynomu.

Průvodce studiem

Nejasná implementace nulového polynomu představuje v našem systému jistě nekonzistenci. Tato nekonzistence nám však výrazně zjednoduší práci při implementaci velké části procedur.

Sada procedur oddělí fyzickou implementaci od zbytku systému.

Nad touto reprezentací vybudujeme logickou vrstvu procedur, která nás oddělí od fyzické reprezentace a umožní nám v budoucnu fyzickou implementaci změnit. Vzhledem ke složitosti operací s polynomy bude vytvoření vhodných a potřebných konstruktorů a selektorů o něco obtížnější než v jiných případech.

Základní operace se seznamy provádějí procedury `cons`, `car` a `cdr`. Tyto procedury musíme „zabalit" do vhodných názvů a dát k dispozici programátorům, pracujícím s polynomy. Procedura `car` v naší reprezentaci představuje *nulový koeficient polynomu* (koeficient u nulové mocniny x). Procedura `cdr` provádí *snížení stupně polynomu*, po její aplikaci dojde k vydělení polynom proměnnou x a odstranění nulového koeficientu. Provedeme-li redukci polynomu stupně nula, dostaneme jako výsledek nulový polynom. Procedura `cons` provádí naopak *zvýšení stupně polynomu*, při její aplikaci dojde k vynásobí polynomu proměnnou x a přidání nového nulového koeficientu.

S ohledem na budoucí práci s polynomy zvolíme v tomto projektu následující sadu procedur, pracujících na fyzické vrstvě. Procedury `poly-dec` a `poly-inc` provádějí snížení respektive zvýšení stupně polynomu, procedura `poly-coef` vrací koeficient polynomu u zadané mocniny a predikát `poly-zerodeg?` představuje test na polynom nulového stupně.

```
 (define (poly-dec p)
  (cdr p))

(define (poly-inc coef p)
  (cons coef p))

(define (poly-coef n p)
  (if (= n 0) (car p)
      (poly-coef (- n 1) (cdr p))))

(define (poly-zerodeg? p)
  (null? (cdr p)))
```

K dalším potřebným procedurám na fyzické úrovni patří a test na nulový polynom a vytvoření polynomu ze seznamu jeho koeficientů. Součástí vytvoření polynomu by mělo být formátování koeficientů tak, aby výsledná datová struktura odpovídala našim požadavkům na implementaci této datové struktury.

```
(define (poly-zero? p)
  (or (null? p)
      (and (poly-zerodeg? p) (= (poly-coef 0 p) 0))))

(define (make-poly coef0 . coefs)
  (poly-format (cons coef0 coefs)))[1]
```

[1] Tečka v seznamu formálních parametrů procedury označuje možnost proměnlivého počtu parametrů. v tomto případě musí být procedura `vytvor-poly` volána s alespoň jedním parametrem (nulovým koefi-

Úkoly k textu

1. Napište kód do prostředí jazyka Scheme. Doplňte chybějící pomocnou proceduru `poly-format` (označena tučně), která odstraní nuly na konci seznamu. Například:

```
> (poly-format '(4 3 2 1 0 0 0))
(4 3 2 1)
> (poly-format '(4 3 2 1))
(4 3 2 1)
```

Průvodce studiem

V dalších kapitolách tohoto projektu budeme důsledně používat navržené procedury pro přístup k datové struktuře polynomu. Jakékoli přímé použití procedur `cons`, `car`, `cdr` nebo `null?` bude považováno za chybu!

2.6.2 Stupeň polynomu

Pro zjištění stupně polynomu lze na základě navržených přístupových procedur snadno formulovat rekurzivní algoritmus. Máme totiž k dispozici proceduru, která detekuje polynom stupně nula (`poly-zerodeg?`) a proceduru, která sníží stupeň polynomu o jedničku (`poly-dec`).

Stupeň polynomu odpovídá délce seznamu.

Úkoly k textu

2. Implementujte proceduru `poly-deg`, která vrátí stupeň polynomu. Například:

```
> (poly-deg '(1 3 2 1))
3
```

Průvodce studiem

V této i dalších podkapitolách bychom se ani při testování neměli spoléhat na známost fyzické implementace a měli bychom důsledně využívat konstruktor `make-poly`. Při testování předchozí procedury bychom proto měli správně psát:

```
> (poly-deg (make-poly 1 3 2 1))
```

V zájmu zachování jednoduchosti a přehlednosti zde však budeme tento malý prohřešek tolerovat.

---

cientem). První parametr se naváže na symbol `nulovy-koef` a seznam všech ostatních parametrů se naváže na symbol `dalsi-koef`.

2.6.3 Hodnota polynomu v bodě

Hodnotu polynomu v bodě zjistíme bez nutnosti výpočtu mocnin.

Pro výpočet hodnoty polynomu v daném bodě x použijeme algoritmus známý jako *Hornerovo schéma*. Tento algoritmus vychází z toho, že např. polynom $1+3x+2x^2+x^3$ se dá zapsat jako $1+x(3+2x+x^2)$. Použijeme-li toto řešení rekurzivně, nemusíme počítat jednotlivé mocniny čísla x. Abychom zjistili hodnotu polynomu **třetího** stupně v bodě x, stačí určit hodnotu polynomu **druhého** stupně v bodě x, vynásobit ji číslem x a přičíst konstantu. Časem se tak dopracujeme k nulového polynomu, jehož hodnota je jistě 0 nezávisle na velikosti x.

Úkoly k textu

3. Implementujte proceduru `poly-val`, která spočte hodnotu polynomu v daném bodě. Využijte přitom algoritmus Hornerova schématu. Například:

```
> (poly-val '(1 3 2 1) 3)
55
```

2.6.4 Pomocné procedury

Pro další práci si implementujeme několik pomocných procedur. Procedura `poly-add-c` přičte k danému polynomu konstantu, procedura `poly-mul-c` vynásobí polynom konstantou, procedura `poly-mul-x` vynásobí polynom proměnnou `x` a procedura `poly-mul-xn` vynásobí polynom `n`-tou mocninou proměnné `x`. Procedury zde uvádíme.

```
(define (poly-add-c p c)
  (poly-inc (+ (poly-coef 0 p) c) (poly-dec p)))

(define (poly-mul-c p c)
  (if (poly-zero? p) p
      (poly-inc (* (poly-coef 0 p) c) (poly-mul-c (poly-dec p) c))))

(define (poly-mul-x p)
  (poly-inc 0 p))

(define (poly-mul-xn p n)
  (if (<= n 0) p
      (poly-mul-x (poly-mul-xn p (- n 1)))))
```

Průvodce studiem

Nezapomeňte tyto procedury nejen mechanicky zapsat do vašeho projektového souboru, ale také je pochopit a důkladně je otestovat. Způsob, jakým tyto procedury využívají rozhraní vrstvy fyzické implementace vám může posloužit jako inspirace při řešení dalších úkolů. Důkladné testování vede kromě odhalení možných překlepů také k tomu, že pochopíte, jak tyto procedury používat.

2.6.5 Sčítání polynomů

S polynomy provádíme symbolické úpravy.

Vzhledem k tomu, jakou jsme vybrali fyzickou implementaci polynomů a jaké jsme nad touto implementací navrhli rozhraní, je sčítání polynomů efektivní i snadné. Stačí postupně sčítat jednotlivé sobě si odpovídající koeficienty členy obou polynomů a vytvářet tak polynom no-

vý. Je-li jeden polynom vyššího stupně než druhý, ukončíme sčítání v okamžiku, kdy narazíme na nulový polynom. Výsledkem je pak druhý polynom.

Problém, na který je třeba upozornit, však může vzniknout tehdy, když se koeficienty u nejvyšších mocnin obou polynomů navzájem vyruší. Sečtením dvou polynomů tak může vzniknout polynom menšího stupně než je stupeň libovolného z nich. Vzhledem k tomu, že naše reprezentace nepovoluje nuly na místě koeficientů u nejvyšších mocnin, je třeba nuly z konce seznamu odstranit pomocí naší procedury `poly-format`.

Uvnitř samotné procedury `poly-add` si proto pomocí lokální definice vytvoříme proceduru `poly-add-temp`, která bude provádět sčítání bez kontroly na nuly na místech nejvyšších mocnin, a výsledek této procedury proženeme přes formátovací funkci. Kostra procedury `poly-add` může vypadat takto:

```
(define (poly-add p1 p2)

  (define (poly-add-temp p1 p2)
    (cond
      ((poly-zero? p1) p2)
      ((poly-zero? p2) p1)
      (else ..........)))

  (poly-format (poly-add-temp p1 p2)))
```

Úkoly k textu

4. Doplňte vynechaná místa v proceduře `poly-add` a otestujte ji na následujících příkladech:

```
> (poly-add '(1 2 1) '(1 3 2 1))
 (2 5 3 1)

> (poly-add '(1 3 2 1) '(1 -3 -2 -1))
 (2)
```

2.6.6 Odčítání polynomů

Tuto proceduru lze snadno získat kombinací procedur `poly-add` a `poly-mul-c`.

Úkoly k textu

5. Implementujte proceduru `poly-sub`, realizující odčítání polynomů. Například:

```
> (poly-sub '(1 2 1) '(1 3 2 1))
(0 -1 -1 -1)
```

2.6.7 Násobení polynomů

Násobení polynomů je snadné, použijeme-li důsledně distributivní a asociativní zákon. Uvažujeme-li například násobení polynomu $4+2x+3x^2$ a libovolného polynomu P, lze výraz $(4+2x+3x^2)\times P$ vyjádřit jako $4P+x((2+3x)\times P)$. Násobení polynomu stupně n a libovolného polynomu P lze tedy rozepsat pomocí operací součtu polynomů, násobení polynomu konstantou, násobení polynomu proměnnou x a násobení polynomu stupně n-1 a polynomu P. Tím tedy celý problém redukujeme na známé operace a na problém jednodušší – víme totiž, že

výsledkem násobení nulovým polynomem je vždy nulový polynom, a máme tak k dispozici mezní podmínku rekurze.

Úkoly k textu

6. Implementujte proceduru `poly-mul` a otestujte ji na následujícím příkladu:

```
> (poly-mul '(1 2 1) '(1 3 2 1))
(1 5 9 8 4 1)
```

2.6.8 Dělení polynomů

Dělení polynomů je náročnější než jiné symbolické úpravy.

Dělení polynomů je asi nejsložitější problém této podkapitoly. Situace však nemusí být tak složitá, pokud si dobře formalizujeme algoritmus pro dělení polynomů, známý ze střední školy. Předpokládejme, že chceme spočítat podíl polynomů P_1 a P_2. Předpokládejme dále, že polynom P_1 lze zapsat jako

$$a_m x^m + a_{m-1} x^{m-1} + ... + a_1 x + a_0$$

a polynom P_2 jako

$$b_n x^n + b_{n-1} x^{n-1} + ... + b_1 x + b_0 .$$

Je-li stupeň polynomu P_1 menší než stupeň polynomu P_2, není třeba nic počítat: výsledkem je nulový polynom a zbytkem po dělení je polynom P_1. V opačném případě lze využít toho, že platí rovnost[1]:

$P_1 \div P_2 = Q + ((P_1 - QP_2) \div P_2)$, kde polynom Q je volen jako: $Q = \frac{a_n}{b_m} x^{n-m}$.

Podstatné je, že hodnota polynomu Q byla záměrně volena tak, aby výraz QP_2 eliminoval nejvyšší mocninu polynomu P_1. Polynom $P_1 - QP_2$ je tedy polynomem nutně nižšího stupně, než polynom P_1. Tím problém dělení polynomů převádíme na známé operace a na problém jednodušší. Povšimněme si ještě, že tento algoritmus je korektní, neboť koeficient b_m, vyskytující se ve jmenovateli výše uvedeného výrazu, je koeficientem u nejvyšší mocniny polynomu, a v naší implementaci je tudíž vždy nenulový.

Výsledkem dělení polynomů je podíl a zbytek.

Vypořádat se musíme i s problémem, že dělení polynomů musí dávat dva výsledky: podíl a zbytek po dělení. Například podílem polynomů $x^3 + 2x^2 + 3x + 1$ a $x^2 + 2x + 1$ je polynom x a zbytkem po dělení je polynom $x + 2$. Vyřešíme to tak, že namísto jedné procedury implementujeme procedury dvě – jedna bude vracet podíl a druhá zbytek po dělení. Nazveme je `poly-quotient` a `poly-remainder`.

```
(define (poly-quotient p1 p2)
  (define Q ..........)
  (if (< (poly-deg p1) (poly-deg p2)) (make-poly 0)
      ..........))
```

[1] Důkaz tohoto tvrzení je mechanický a přenecháme ho čtenářům.

Úkoly k textu

7. Doplňte chybějící kód v proceduře `poly-quotient` a otestujte ji na následujícím příkladě:

```
> (poly-quotient '(1 3 2 1) '(1 2 1))
(0 1)
```

8. Obdobně implementujte proceduru `poly-remainder` a otestujte ji na následujícím příkladě:

```
> (poly-remainder '(1 3 2 1) '(1 2 1))
 (1 2))
```

2.6.9 Největší společný dělitel polynomů

Průvodce studiem

Při zmínění výpočtu největšího společného dělitele dvou polynomů jistě většině čtenářů naskakuje husí kůže. Problém je však mimořádně jednoduchý. Uvědomme si, že máme k dispozici funkční procedury na výpočet podílu a zbytku po dělení dvou polynomů. S polynomy tedy můžeme pracovat v zásadě stejně snadno jako s čísly a použít na ně Euklidův algoritmus. Program na výpočet největšího společného dělitele dvou polynomů má tak v jazyce Scheme cca tři řádky a je prakticky totožný s programem na výpočet největšího společného dělitele dvou čísel.

Pro výpočet největšího společného dělitele použijeme Euklidův algoritmus.

Postup výpočtu největšího společného dělitele pomocí Euklidova algoritmu byl popsán v [Skoupil04B]. V této chvíli je nám již jedno, počítáme-li největšího společného dělitele dvou čísel nebo dvou polynomů, protože máme z předchozích podkapitol k dispozici veškeré potřebné operace pro symbolickou manipulaci s polynomy.

Připomínáme, že algoritmus výpočtu největšího společného dělitele vychází z pozorování, že jestliže hodnota r je zbytek po dělení hodnoty x hodnotou y, pak největší společný dělitel x a y je stejný, jako největší společný dělitel hodnot y a r. Tím ovšem problém redukujeme na jednodušší, neboť zbytek po dělení musí být vždy menší než dělitel. Pokud budeme algoritmus opakovat, musíme dříve či později dojít k situaci, kdy zbytek po dělení bude nulový polynom.

Je třeba upozornit, že největších společných dělitelů dvou polynomů je nekonečně mnoho, liší se však jen o konstantní násobek. Největším společným dělitelem polynomů $x^3 + 5x^2 + 7x + 3$ a $x^3 + 2x^2 + x$ je polynom $x^2 + 2x + 1$ stejně jako $3x^2 + 6x + 3$ atd.

Úkoly k textu

9. Implementujte proceduru `poly-gcd`, která počítá největšího společného dělitele dvou polynomů pomocí Euklidova algoritmu. Otestuje proceduru na následujícím příkladě:

```
> (poly-gcd '(3 7 5 1) '(0 1 2 1))
(3 6 3)
```

2.6.10 Symbolická derivace polynomu

Symbolická derivace je v naší implementaci polynomu snadná. Stačí si uvědomit, že například derivováním polynomu $x^3 + x^2 + x + 1$ získáme polynom $3x^2 + 2x + 1$, což, zapsáno pomocí seznamů, znamená že derivací polynomu `(1 1 1 1)` je polynom `(1 2 3)`.

Symbolická derivace a integrace polynomu jsou si v mnohém podobné.

Výsledný polynom tedy formálně vznikne tak, že první prvek seznamu (koeficient u nulové mocniny x) vynecháme a zbytek seznamu upravíme tak, že jeho první prvek vynásobíme číslem 1, druhý prvek číslem 2 atd. Postup konstrukce výsledného polynomu bude v jistém smyslu analogický postupu, který jsme použili v proceduře `poly-mul-c`.

Úkoly k textu

10. Implementujte proceduru `poly-deriv`, která provádí symbolickou derivaci polynomu. Otestujte ji na následujícím příkladě:

```
> (poly-deriv '(3 7 5 1))
(7 10 3)
```

2.6.11 Symbolická integrace polynomu

Symbolická integrace je v mnohém obdobná symbolické derivaci. Také kód obou procedur bude až na formální změny téměř stejný. Zatímco derivace snižuje řád polynomu, integrace jej bude zvyšovat a zatímco u derivace jsme jednotlivé koeficienty polynomu násobili, u integrace je budeme dělit.

Odlišností oproti derivaci je existence integrační konstanty. Procedura bude proto akceptovat druhý parametr, který poslouží jako hodnota integrační konstanty. Integrační konstanta bude ve výsledku tvořit koeficient u nulové mocniny x.

Úkoly k textu

11. Implementujte proceduru `poly-integ`, která provádí symbolickou integraci polynomu. Otestujte ji na následujícím příkladě:

```
> (poly-integ '(3 7 5 1) 9)
(9 3 7/2 5/3 1/4)
```

Shrnutí

Polynomy je možno v jazyce Scheme jednoduše symbolicky reprezentovat pomocí seznamů. Pro konstrukci a přístup k fyzické implementaci polynomů si vybudujeme vrstvu procedur. Pomocí těchto procedur pak realizujeme algoritmy na polynomiální aritmetiku. Snadno realizovatelné jsou i algoritmy na výpočet největšího společného dělitele dvou polynomu a na symbolickou derivaci a integraci polynomu.

2.7 Projekt: reprezentace množin

Studijní cíle: Při řešení tohoto projektu se studující seznámí s možností využití seznamů pro reprezentaci množin. Procvičí se v návrhu vrstvové architektury programu a v implementaci rekurzivních procedur na práci se seznamy.

Klíčová slova: Množiny, seznamy.

Potřebný čas: 2 hodiny.

Množinu můžeme neformálně zavést jako skupinu navzájem odlišitelných objektů[1]. Každý objekt je v množině pouze jednou a pořadí objektů v množině není určeno. v informatice můžeme navíc množinu chápat jako datovou strukturu a použít techniky datové abstrakce. Každá datová struktura (nezávisle na své fyzické implementaci) je popsána operacemi, které nad touto strukturou smíme provádět. V případě množin chceme umět zjistit, jestli je objekt prvkem množiny a chceme mít možnost přidat prvek do množiny. Dále nás budou zajímat operace množinové aritmetiky.

2.7.1 Volba reprezentace

Pro reprezentaci množin v jazyku Scheme zvolíme celkem přirozeně seznam. Stanovíme si přitom následující dvě podmínky.

Množinu implementujeme jako seznam prvků bez duplicit.

- Libovolný prvek se může v množině vyskytovat nejvýše jednou, duplicity nejsou povoleny. Seznam `(1 7 3 5)` je platnou reprezentací množiny {1, 3, 5, 7}, zatímco seznam `(1 7 3 7 5)` není reprezentací žádné množiny.
- Na pořadí prvků v množině nezávisí. Seznamy `(1 7 3 5)` a `(7 3 1 5)` jsou navzájem rovnocenné reprezentace množiny {1, 3, 5, 7}.

Narozdíl od předchozích příkladů zde nebudeme budovat oddělenou vrstvu pro fyzický a logický přístup k datové struktuře. Všechny zde implementované procedury budou tedy závislé na volbě reprezentace množiny. Začneme implementací dvou základních procedur:

- `prvek?` – predikát, umožňující rozhodnout, jestli libovolný objekt je nebo není prvkem množiny,
- `pridej` – procedura, která korektně přidá prvek do množiny (zabraňuje vzniku duplicit).

```
(define (prvek? x set)
  (cond ((null? set) #f)
        ((equal? x (car set)) #t)
        (else (prvek? x (cdr set)))))

(define (pridej x set)
  (if (prvek? x set)
      set
      (cons x set)))
```

Úkoly k textu

1. Napište tento kód do prostředí jazyka Scheme a otestujte jej.

[1] Korektní matematická definice množiny je netriviální téma. Množiny se v matematice zavádějí axiomaticky: vyjmenujeme chování, které od množin očekáváme (axiomy) a oproti těmto axiomům pak dokazujme platnost všech ostatních tvrzení.

2.7.2 Sjednocení, průnik a rozdíl množin

Sjednocení a průnik jsou základní množinové operace.

Sjednocením dvou množin vznikne nová množina, která obsahuje prvky obsažené v kterékoli z obou množin. Sjednocení množin je tedy v při naší reprezentaci množiny velmi podobné spojení seznamů (také kód obou procedur bude téměř identický), musíme však dbát na možný vznik duplicit.

Do průniku dvou množin pak patří prvky, které se vyskytují v obou množinách.

Rozdílem dvou množin rozumíme všechny ty prvky první množiny, které se nevyskytují v druhé množině.

Úkoly k textu

2. Implementujte proceduru `sjednoceni`, která vrátí výsledek sjednocení dvou množin. Aby při spojování množin nevznikaly duplicity, musí vaše procedura využívat proceduru `pridej`. Například:

```
> (sjednoceni '(7 2 9 4) '(2 3 4 5))
(7 2 9 4 3 5)
```

3. Implementujte proceduru `prunik`, která vrátí výsledek průniku dvou množin. Například:

```
> (prunik '(7 2 9 4) '(2 3 4 5))
(2 4)
```

4. Implementujte proceduru `rozdil`, která vrátí výsledek rozdílu dvou množin. Například:

```
> (rozdil '(1 2 3 4 5 6 7 8) '(3 8 5))
(1 2 4 6 7)
```

2.7.3 Podmnožiny a rovnost množin

Test podmnožiny je základní množinový predikát.

Po zvládnutí množinové aritmetiky přistoupíme k implementaci základních predikátů. Pro dvě množiny je potřeba zjistit jestli jsou identické a jestli je jedna podmnožinou druhé. Opět zde nemůžeme použít prosté porovnání seznamů, protože u množin není určeno pořadí prvků.

Úkoly k textu

5. Implementujte predikát `podmnozina?`, který otestuje jestli je jedna množina podmnožinou druhé. Například:

```
> (podmnozina? '(7 2 9 4) '(1 2 3 4 5 6 7 8))
#f
> (podmnozina? '(7 2 8 4) '(1 2 3 4 5 6 7 8))
#t
```

6. Implementujte predikát `rovno?`, který otestuje rovnost dvou množin. Tento predikát by měl využívat predikát `podmnozina?`.

Průvodce studiem

Pocit, který jste mohli zaznamenat při řešení posledních dvou úkolů, nespadá do kategorie označované v odborné literatuře jako Déjà Vu. Tyto procedury jsme již skutečně v rámci tohoto textu implementovali.

2.7.4 Potenční množina

Výpočet potenční množiny využívá rekurzivní algoritmus.

Implementace procedury, která vrací potenční množinu, je kvalitativně složitější než řešení předchozích úkolů. Potenční množinou rozumíme množinu všech podmnožin dané množiny. Má-li množina n prvků, bude mít její potenční množina 2^n prvků. Například:

```
> (poten '())
(())

> (poten '(1))
(() (1))

> (poten '(1 2))
(() (2) (1) (1 2))

> (poten '(1 2 3))
(() (3) (2) (2 3) (1) (1 3) (1 2) (1 2 3))
```

Algoritmus pro určení potenční množiny bude zřejmě rekurzivní. Nabízí se mezní podmínka rekurze: potenční množina prázdné množiny je jednoprvková množina, obsahující prázdnou množinu. Protože na pořadí prvků v množinách nezáleží, přepišme si předchozí výsledky do následující tabulky, která jasněji odhaluje povahu algoritmu.

```
(poten      '())                 (())
(poten     '(1))                 (() (1))
(poten   '(2 1))                 (() (1) (2) (2 1))
(poten '(3 2 1))                 (() (1) (2) (2 1) (3) (3 1) (3 2) (3 2 1))
```

Potenční množina vznikne spojením výsledků dvou rekurzivních volání.

Porovnáme-li poslední dva řádky, vidíme, že potenční množina množiny `(3 2 1)` vznikne spojením (nabízí se použití systémové procedury `append` nebo procedury `spoj` z kapitoly 2.3.1) dvou seznamů:

- potenční množiny množiny `(2 1)`, tedy seznamu `(() (1) (2) (2 1))`
- a ještě jednou potenční množiny množiny `(2 1)` s tím, že do každé podmnožiny této množiny byl na začátek přidán prvek 3, tedy seznamu `((3) (3 1) (3 2) (3 2 1))`.

Tím jsme v podstatě prozradili celý algoritmus.

Průvodce studiem

Tento algoritmus je silně založen na použití rekurze, podobně jako tomu bylo například u algoritmu na řešení problému Hanojských věží. Uděláme-li v takovémto rekurzivním programu jakoukoli chybu, nedostaneme pravděpodobně žádný, nebo velmi podivně vyhlížející výsledek. Znovu proto opakujeme, že potenční množina prázdné množiny ***není*** *prázdná množina a že potenční množina* ***není*** *konstrukcí (cons), ale spojením (append) dvou seznamů.*

Úkoly k textu

7. Napište pomocnou proceduru `pridej-do-podmnozin`, která přidá prvek `x` do každé z podmnožin dané množiny, například:

```
> (pridej-do-podmnozin 5 '((1) (2) (3) (4))
((5 1) (5 2) (5 3) (5 4))
```

8. Implementujte proceduru `poten`, která vrací potenční množinu dané množiny.

Shrnutí

Množinu lze v jazyku Scheme zavést jako neuspořádaný seznam prvků bez duplicit. Nad touto reprezentací můžeme implementovat základní množinové operace.

3 Závěr

Tato publikace navazovala na publikaci [Skoupil04B]. Využili jsme znalosti základů programovacího jazyka Scheme a představili jsme studujícím jednoduché i složitější datové typy. Základní pozornost byla věnována tečkovým párům a seznamům. Nad seznamy jsme pak prezentovali řadu rekurzivních procedur. Studující měl možnost si osvojit a procvičit diskutovanou problematiku prostřednictvím řady menších i rozsáhlejších projektů.

4 Seznam literatury

[Abelson96] ABELSON, H., SUSSMAN, G. *Structure and Interpretation of Computer Programs*. 2. vyd. Cambridge, Massachusetts: MIT Press, 1996. ISBN 0-262-01153-0. Dostupné z WWW:

[Dybvig03] DYBVIG, R. K. *The Scheme Programming Language*. 3. vyd. Cambridge, Massachusetts: MIT Press, 2003. ISBN 0-262-541483-0.

[Friedman94] FRIEDMAN, D. P., WAND, M., HAYNES, C. T. *Essentials of Programming Languages.* 1. vyd. Cambridge, Massachusetts: MIT Press, 1994. ISBN 0-262-06145-7.

[Friedman96a] FRIEDMAN, D. P., FELLEISEN, M. *The Little Schemer*. 4. vyd. Cambridge, Massachusetts: MIT Press, 1996. ISBN 0-262-56099-2.

[Friedman96b] FRIEDMAN, D. P., FELLEISEN, M *The Seasoned Schemer.* 4. vyd. Cambridge, Massachusetts: MIT Press, 1996. ISBN 0-262-56100-X.

[R5RS] KELSEY, R, CLINGER, W, REEDS, J. (eds.), Revised5 Report on the Algorithmic Language Scheme, *Higher-Order and Symbolic Computation*, 1998, roč. 11, č. 1.

[Skoupil97] SKOUPIL, D., KOPKA, M. *Průvodce jazykem Scheme*. 1. vyd. Olomouc: Vydavatelství UP, 1997. ISBN 80-7067-693-0.

[Skoupil04A] SKOUPIL, D. *Programy a projekty v jazyku Scheme I.*

[Skoupil04B] SKOUPIL, D. *Programy a projekty v jazyku Scheme II.*

[Steele76a] STEELE, G. L., SUSSMAN, G. J. *LAMBDA: The Ultimate Imperative.*MIT AI Lab Memo AIM-353, 1976. Dostupné z FTP: ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-353.pdf

[Steele76b] STEELE, G. L. LAMBDA: *The Ultimate Declarative*. MIT AI Lab. AI Lab Memo AIM-379, 1976. Dostupné z FTP: ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-379.pdf

[Steele78] Steele, G. L. *Revised Report on Scheme –Dialect of Lisp.* MIT AI Lab. AI Lab Memo AIM-452, 1978. Dostupné z FTP: ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-452.pdf

[Steele93] STEELE, G. L., GABRIEL, R. *The Evolution of Lisp*. SIGPLAN Notices 1993, roč 28, č 3, s. 231-270. ISSN 0362-1340. Dostupné z FTP: ftp://ftp.cs.indiana.edu/pub/scheme-repository/doc/pubs/Evolution-of-Lisp.ps

5 Seznam obrázků

6 Rejstřík