KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

PROGRAMY A PROJEKTY V JAZYKU SCHEME I

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 zahraniční publikace (viz [Abelson96], [R5RS]). Studující použije tento text jako úvodní pomůcku pro zvládnutí jazyka Scheme a dále jako doprovodný materiál při studiu pokročilejších publikací.

První díl této řady se věnuje zvládnutí základů jazyka, tvorbě jednoduchých programů a ovládnutí rekurze.

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 nepředpokládá žádné vstupní znalosti.

Obsah

1 Základy jazyka

1.1 Úvod do problematiky

Studijní cíle: Po zvládnutí kapitoly bude studující schopen vysvětlit, proč používáme programovací jazyk Scheme, bude umět spustit integrované prostředí jazyka a naprogramovat jednoduché procedury.

Klíčová slova: Scheme, integrované prostředí, infixová notace, pravidla vyhodnocování výrazu.

Potřebný čas: 2 hodiny.

1.1.1 Jazyk Scheme

Programování se nazývalo "umělá inteligence".

Již při samotném vzniku počítačů, nebo snad ještě dřív, si lidé kladli velmi vysoké cíle. Po vzoru starého pražského rabína, který z hlíny vytvořil Golema, po vzoru rudolfínských alchymistů, snažících se vytvořit umělého člověka, po vzoru Karla Čapka, píšícího o robotech, chtěli naprogramovat počítače tak, aby přemýšlely a jednaly jako lidé. Svému programování proto začali říkat *umělá inteligence*.

Programování ve strojovém kódu je obtížné. Byly vytvořeny první programovací jazyky.

Protože programování ve strojovém kódu počítačů bylo velmi obtížné a zdlouhavé a některé programové konstrukce jako například rekurze byly téměř nedosažitelné, byly vytvořeny první programovací jazyky. Jedním z nich byl Lisp, který byl navržen speciálně pro potřeby programů umělé inteligence. Tvůrcem tohoto jazyka byl John McCarthy, který první neformální specifikaci tohoto jazyka zveřejnil v roce 1957. Programovací jazyk Lisp byl inspirován výpočetním modelem popsaným v roce 1936 Alonzem Churchem v práci o tzv. *lambda kalkulu*. Programovací jazyk Lisp využíval seznamů jako základních datových i řídících struktur. Odtud se také vzalo jeho jméno Lisp –**LIS**t **P**rocessing, jazyk na zpracování seznamů.

Průvodce studiem

V publikaci o historii jazyka Lisp ([Steele93]) chybí zcela zmínka o vlivu českého velikána Járy Cimrmana na vznik lambda kalkulu, a tudíž i na vývoj jazyka LISP. Lambda kalkul, chybně připisovaný Američanovi Alonzu Churchovi, vynalezl Cimrman již v roce 1912. Práce Cimrmana zaměstnala natolik, že dokonce odložil svou plánovanou návštěvu USA a nechal propadnout svůj zakoupený lodní lístek na první plavbu parníku Titanic. Do Ameriky se Cimrman dostal až o rok později, na pozvání krajana Henryho Ticháčka, učitele matematiky na základní škole ve Washingtonu, D.C. Roztržitý Cimrman tehdy zapomněl složku se svými poznámkami o lambda kalkulu na chodbě, kde se k nim dostal desetiletý Alonzo.

Churchovi trvalo více než dvacet let, než Cimrmanovy poznámky konečně pochopil. Lambda kalkul, jakožto aparát pro popis logiky umělé inteligence, byl však pro Cimrmana pouze přípravným krokem pro vytvoření beta kalkulu, který chtěl použít pro popis umělé blbosti.

Programovací jazyk Scheme vznikl na podzim roku 1975 jako zajímavý experiment Guye Steela a Geralda Sussmana z Laboratoře umělé inteligence známé Massachussetské techniky (viz [Steele93]). Cílem obou autorů bylo vyzkoušet si některé důsledky objektově orientovaných technologií (tzv. aktorů). Vzhledem k tomu, že jazyk Lisp se k tomuto účelu příliš nehodil,

navrhli si oba programátoři programovací jazyk nový a implementovali jej právě v jazyce Lisp. Syntaxí tento jazyk silně připomínal jazyk Lisp, byl však mnohem jednodušší a odstraňoval některé nepříjemné vlastnosti Lispu. Vzhledem k tomu, že tento programovací jazyk byl pro jeho autory jen pomůckou, nikoli cílem, byla jeho stavba velmi jednoduchá, elegantní a jazyk neobsahoval žádné zbytečné nebo redundantní prvky.

Scheme je dialekt jazyka LISP.

Brzy se ukázalo, že navzdory jednoduchosti lze v tomto programovacím jazyce velmi rychle a elegantně programovat prototypy rozsáhlých programátorských projektů. To slibovalo možné využití jazyka ve výzkumu v oblasti umělé inteligence. Autoři proto nazvali tento programovací jazyk *Schemer*, po vzoru ostatních programů umělé inteligence, jakými byly programy Planner nebo Solver. Operační systém ITS, který autoři používali, však omezoval délku jména souborů na 6 znaků. Schemer byl tedy uložen vždy v adresářích a souborech s názvem `scheme`. Tento název programovacímu jazyku už zůstal.

Aby dokázali sílu a univerzálnost nově vytvořeného jazyka, napsali Sussman a Steele v roce 1976 publikace „LAMBDA: The Ultimate Imperative“ a „LAMBDA: The Ultimate Declarative“, ve kterých za pomoci programovacího jazyka Scheme popsali většinu programových konstrukcí používaných v běžných programovacích jazycích ([Steele76a] a [Steele76b]). Druhou jmenovanou práci použil Steele i jako návrh na svoji magisterskou diplomovou práci.

Standardizace se programovací jazyk Scheme dočkal v roce 1978, kdy Steele úplně popsal jazyk Scheme na několika stranách v publikaci s názvem Revidovaná zpráva o Scheme – dialektu Lispu ([Steele78]). Tato zpráva se s vývojem jazyka Scheme dočkala dalších revizí a vznikala tak Revidovaná revidovaná zpráva, Revidovaná revidovaná revidovaná zpráva atd. Počet revizí se začal označovat exponentem, jak je v matematickém světě zvykem. V době psaní tohoto textu existuje pátá revize jazyka Scheme (viz [R5RS]).

Scheme je algoritmický jazyk vycházející z funkcionálního paradigmatu.

Jazyk Scheme je v dnešní podobě velmi univerzální a těžko se dá zařadit do přihrádek jednotlivých programovacích paradigmat. Jeho podstatou je programovací paradigma funkcionální, je v něm ale možné pracovat imperativním nebo objektově orientovaným stylem. Scheme je proto často označován jako „algoritmický“ jazyk, tedy jazyk, ve kterém je možno snadno formulovat algoritmy. Asi nejpodstatnějším rysem programování v jazyce Scheme však je, že přináší radost, legraci a uspokojení. V současné době je jazyk Scheme používán na stovkách vysokých i středních škol.

Obr. 1 Živá lambda, jak ji ztvárnili studenti Rice University

1.1.2 Proč Scheme?

Scheme je experimentální jazyk

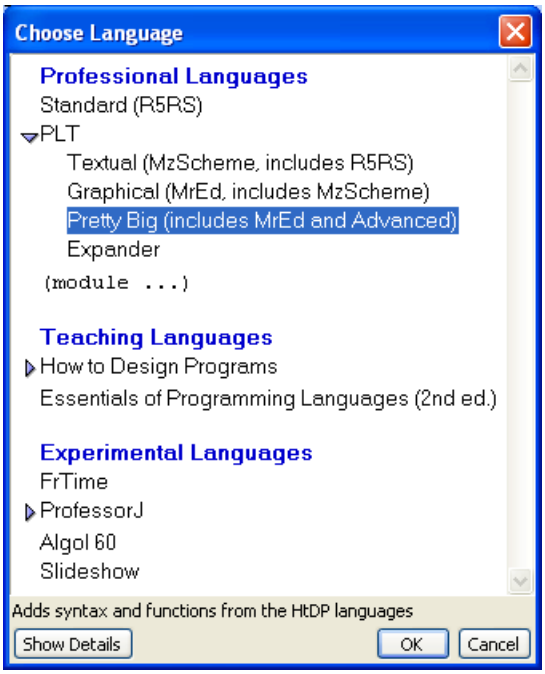
Název podkapitoly charakterizuje jednu z otázek, které se zřejmě nevyhne žádný z čtenářů tohoto textu. Scheme není jazyk používaný v průmyslové, obchodní nebo administrativní praxi. Nehodí se příliš ke konstrukci operačních systémů ani pro programování rozsáhlých projektů. Proč se tedy raději nevěnovat jazykům jako C++, C# nebo Visual Basic? Pokusíme se shromáždit několik důvodů, které toto naše rozhodnutí obhajují.

- Scheme je velmi jednoduchý jazyk. Je-li to nutné, je možné syntaxi jazyka Scheme úplně vyložit během jedné přednášky. Zápis programu a vyhodnocovací pravidla jsou jednoduchá a přímočará. S jazykem Scheme neztrácíme čas vysvětlováním úlohy středníků, řídicích konstrukcí, typových konverzí, složitého integrovaného prostředí a dalších rysů jazyka, které jsou pro zvládnutí základů programování zcela nepodstatné a odvádějí pozornost. Naopak se soustředíme na algoritmus, jeho strukturu a efektivitu.
- Scheme je velmi komplexní jazyk. Syntaktická přímočarost kupodivu nijak neomezuje expresívnost jazyka. Právě naopak, nevyžadujeme-li vysokou efektivitu vykonávaného kódu (pak zvolme C), nechceme-li zpracovávat rozsáhlé objemy dat (pak zvolme COBOL), nechceme-li tvořit komerční aplikace s komerčním rozhraním (pak zvolme Visual Basic), nechceme-li zvláště rychlé numerické výpočty (pak zvolme FORTRAN) a nechceme-li strávit tři roky studiem jazyka (pak zvolme Smalltalk), jsou možnosti Scheme téměř neomezené.
- Scheme není použitelný v praxi. Paradoxně, nepoužitelnost jazyka v praxi chrání učitele i studenty od četných komplikací. Mnoho méně důležitých rysů jazyka je možno pominout prostě proto, že tento jazyk stejně není použitelný v praxi. U komerčně úspěšných jazyků si tento luxus většinou dovolit nemůžeme a nad otázkami jako je systém vstupů a výstupů, ošetření chybových stavů nebo tvorba uživatelského rozhraní trávíme značnou část času, kterou by bylo lépe věnovat návrhu a tvorbě algoritmu.
- Scheme je přenositelný a zdarma. Držíme-li se zásad standardu [R5RS], je možné tentýž kód spustit v libovolné implementaci Scheme. Řada implementací jazyka Scheme je navíc dostupná pro různé platformy a tak je možno využívat identický kód na různých hardwarových i softwarových platformách. Kromě toho je mnoho velmi kvalitních implementací jazyka Scheme poskytována zdarma. I když je to logický důsledek předchozího bodu, stovkám programátorů, pracujících ve volném čase a bez nároku na odměnu, patří naše poděkování.
- Nejsme sami. Scheme je v dnešní době třetí nejpoužívanější jazyk pro úvodní kursy programování na vysokých školách ve světě a jeho popularita nadále roste.

1.1.3 Integrované prostředí jazyka

Používáme integrované prostředí DrScheme.

V tomto textu se odkazujeme na integrované prostředí jazyka nazvané Dr.Scheme, které je šířené zdarma pod licencí GPL. Aktuální verze, v době psaní tohoto textu verze 206, je dostupná pro většinu operačních systémů ze serveru www.drscheme.org. V prostředí Dr.Scheme můžeme volit z několika dostupných jazyků, pro účely tohoto textu zvolíme jazyk nazývaný „Pretty Big“, jak ukazuje obrázek Obr. 2

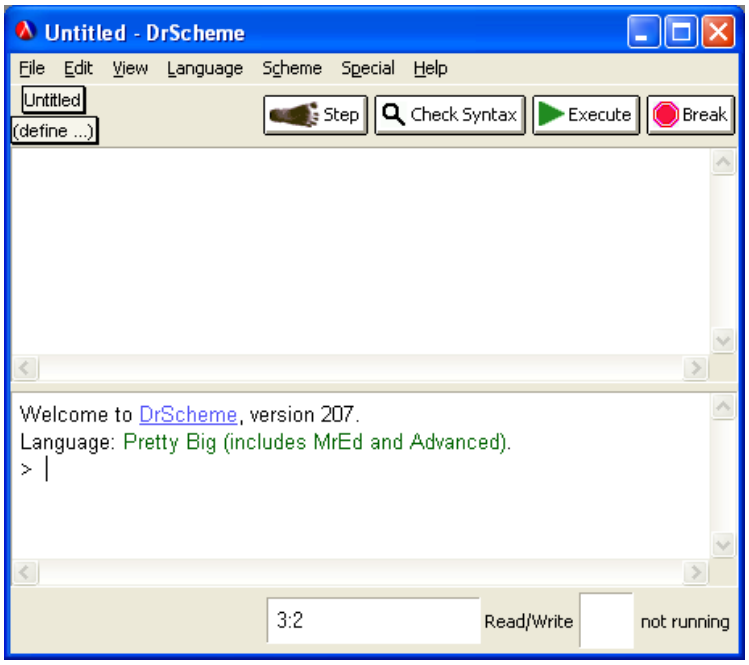


Obr. 2 Volba jazyka v prostředí Dr.Scheme

Překladač pracuje systémem REPL.

I když to standard jazyka neurčuje, překladače jazyka Scheme jsou tradičně (i když ne nutně) interpretační. Znamená to, že veškeré příkazy, které uživatel-programátor překladači zadá, jsou okamžitě provedeny a výsledky oznámeny uživateli. Tento princip se nazývá „Read-Eval-Print Loop“, ve zkratce *REPL*, a představuje základní standard komunikace překladače s uživatelem. Cokoliv uživatel překladači předloží je načteno (fáze *read*), vyhodnoceno (fáze *eval*) a výsledek je vytištěn na obrazovku (fáze *print*). Pak je překladač připraven celý cyklus REPL opakovat.

Integrované prostředí DrScheme obsahuje ovládací tlačítka a dvě hlavní okna. Horní okno se nazývá *okno definicí*. Spodní okno je *okno interakcí*. Integrované prostředí ukazuje obrázek Obr. 3.



Obr. 3 Integrované prostředí Dr.Scheme

V prostředí máme okno definicí a okno interakcí.

Okno interakcí představuje rozhraní na cyklus REPL jazyka. Zadáme-li do tohoto okna jakýkoli výraz, bude načten, vyhodnocen a výsledek vyhodnocení vypsán na obrazovku. Demonstrujme REPL na následující interakci s prostředím:[1]

```
> 1
1
> 5
5
> +
#<primitive:+>
> nazdar
reference to undefined identifier: nazdar
```

Průvodce studiem

Nevěřte textu a vyzkoušejte si tuto interakci na počítači. Výsledky, které nám při jednotlivých zadáních počítač vypisuje, zatím nekomentujeme. Chápeme-li však princip cyklu REPL, načetl překladač v prvním řádku číslo 1, vyhodnotil ho a vrátil výsledek vyhodnocení – číslo 1. Asi by nás překvapilo, kdyby výsledkem vyhodnocení jedničky byla nějaká jiná hodnota. Při vyhodnocení znaménka + nám systém tvrdí, že se jedná o nějaký „primitiv" jazyka, pokus o vyhodnocení slova „nazdar" skončil chybou. Chybové hlášení přeložíme jako „odkaz na neznámý identifikátor: nazdar". Počítač slovo „nazdar" nezná a neví, jak jej vyhodnotit. Pravidla pro vyhodnocování výrazů uvedeme v další kapitole.

Stejné výrazy, jako píšeme do okna interakcí, můžeme také napsat do okna definic. Integrované prostředí jazyka se však v tomto případě nesnaží jednotlivé výrazy okamžitě vyhodnocovat. K vyhodnocení dojde teprve tehdy, když stiskneme tlačítko „Execute" na horní tlačítkové liště. V tu chvíli dojde k postupnému vyhodnocení všech výrazů zapsaných v okně definic, zcela

[1] Řádky, předznamenané symbolem > představují uživatelský vstup. Řádky bez symbolu > potom odpověď překladače.

stejně, jako kdybychom je zapisovali do okna interakcí. Výsledky vyhodnocení opět vidíme v okně interakcí.

Zápis výrazů do okna definic má několik výhod:

Obsah okna definicí se předá do REPL po stisku tlačítka.

- obsah celého okna můžeme uložit do souboru a později se k nim vrátit,
- pokud v zapisovaných výrazech uděláme chybu, můžeme ji snadno odstranit,
- výrazy můžeme libovolně upravovat a přidávat k nim další.

V praxi nám pak okno interakcí bude sloužit jen pro spouštění programů a pro zobrazování výsledků, programy budeme zapisovat do okna definic.

1.1.4 Shrnutí

Programovací jazyk Scheme vznikl v 70. letech minulého století a vyvinul se v jazyk, používaný zejména pro výuku algoritmizace na vysokých a středních školách. Pro účely tohoto textu používáme integrované prostředí DrScheme. Integrované prostředí má okno definic a okno interakcí. Scheme pracuje principem popsaným zkratkou REPL. Okno interakcí představuje rozhraní na REPL, okno definic je puštěno do cyklu REPL po stisku tlačítka Execute.

Pojmy k zapamatování

- Scheme,
- DrScheme,
- REPL,
- okno definic, okno interakcí.

Kontrolní otázky

1. *Z jakého programovacího jazyka se vyvinul jazyk Scheme?*
2. *Jaká matematická teorie stojí v pozadí tohoto jazyka a kdo je jejím autorem?*
3. *Jaký je podstatný rozdíl mezi oknem definic a oknem interakcí v integrovaném prostředí DrScheme.*

Úkoly k textu

1. Pokud jste tak ještě neudělali, nainstalujte si na počítač prostředí DrScheme a vyzkoušejte si práci s tímto prostředím. Uložte obsah okna definic na disk počítače, prostředí restartujte a soubor otevřete.

1.2 Výrazy jazyka a jejich vyhodnocování

Studijní cíle: Po absolvování kapitoly bude studující schopen vyjmenovat typy výrazů v jazyce Scheme a určit způsob jejich vyhodnocení. Bude schopen využívat prostředí jazyka Scheme jako jednoduchou kalkulačku.

Klíčová slova: Číslo, symbol, seznam, procedura, speciální forma, pravidla vyhodnocení výrazů.

Potřebný čas: 2 hodiny.

V minulé kapitole jsme viděli příklad interakce s jazykem Scheme. Každý programovací jazyk přitom obsahuje několik základních součástí:

- primitivní (atomické, jednoduché) výrazy,
- prostředky pro kombinaci primitivních výrazů do rozsáhlejších celků,
- prostředky pro abstrakci, tedy možnost pojmenovat složitější celek a pracovat s ním stejně jako s jednoduchým výrazem.

Tyto tři druhy výrazů a prostředků popíšeme v následujících podkapitolách.

1.2.1 Primitivní výrazy

Primitivní výrazy, které do jazyka Scheme zadáváme, dělíme na dva druhy:[1]

Primitivní výrazy jsou čísla a symboly.

- *čísla*
- *symboly*.

Číslem rozumíme číslici nebo sekvenci číslic, číslo může obsahovat desetinnou tečku a může mít znaménko mínus. Příkladem čísel mohou být výrazy `4`, `6.33` nebo `-12.1`

Symbolem rozumíme alfanumerický znak nebo kombinaci alfanumerických znaků, která neobsahuje mezeru, závorku nebo jiné rezervované znaky a nelze ji interpretovat jako číslo. Symbol je například znak `+`, slovo `nazdar`, symbolem jsou však také výrazy jako `a4`, `5-x4` nebo `++1`.

Pro vyhodnocování primitivních výrazů máme dvě základní pravidla:

Základní pravidla vyhodnocování primitivních výrazů.

- čísla se vyhodnocují sama na sebe (na svoji číselnou hodnotu, předpokládáme, že jsou zapisována v desítkové soustavě),
- symboly se vyhodnocují na svoji aktuální vazbu; některé symboly mají aktuální vazbu nastavenou (jako třeba symbol `+`), jiné symboly žádnou vazbu nemají (jako třeba symbol `nazdar`).

Průvodce studiem

Vyhodnocení některých čísel a symbolů jsme si vyzkoušeli v minulé kapitole. Připomínáme, že Scheme pracuje systémem REPL: načítá výrazy, vyhodnocuje je a výsledek vyhodnocení tiskne. Ověřte si na počítači vyhodnocení zmíněných výrazů. Vyzkoušejte, že čísla mohou obsahovat desetinnou část a znaménko mínus. Pokuste se najít další symboly, které v jazyku mají vazbu.

1.2.2 Prostředky na kombinaci primitivních výrazů

Scheme poskytuje jediný prostředek na kombinaci primitivních výrazů, kterým je *seznam*. Seznam je cokoliv, co začíná *levou závorkou* a končí *pravou závorkou*. Mezi závorkami mohou být prvky seznamu, oddělené od sebe *mezerami*. *Prvek seznamu* může být libovolný výraz.

Seznamem jsou tedy například výrazy: `(1)`, `(1 2)`, `(ahoj 3.2)`, `(3 + 2)`, `(3 2 +)` nebo `(+ 3 2)`. Seznam může být i prázdný. Prvky seznamu mohou být libovolné výrazy, tedy i seznamy. Seznamem jsou tedy i výrazy `()` – tzv. prázdný seznam, `(())` – jednoprvkový seznam,

Seznam je nástrojem pro kombinaci výrazů.

[1] Ve skutečnosti existuje podstatně více primitivních výrazů, například řetězce, vektory, logické hodnoty a další. Tyto výrazy budeme uvádět až v okamžiku, kdy je budeme potřebovat.

obsahující jako svůj jediný prvek prázdný seznam, `(3 2 (1 5))` – tříprvkový seznam, kde prvním prvkem je číslo `3`, druhým prvkem číslo `2` a třetím prvkem seznam `(1 5)`.

Zatímco vyhodnocování čísel a symbolů bylo snadno pochopitelné, vyhodnocování seznamů je složitější. Dříve než si pravidlo formulujeme, pokusme se vyhodnotit následující sekvenci výrazů:

```
> a
reference to undefined identifier: a
> (a)
reference to undefined identifier: a
> (a b)
reference to undefined identifier: a
> 1
1
> (1)
procedure application: expected procedure, given: 1 (no arguments)
> (1 2)
procedure application: expected procedure, given: 1; arguments were: 2
>+
#<primitive:+>
> (1 + 2)
procedure application: expected procedure, given: 1; arguments were: #<primi-
tive:+> 2
> (+ 1 2)
3
> (+ 1 (+ 2 3))
6
```

Průvodce studiem

Prostudujte si pečlivě záznam této interakce s jazykem Scheme. Vyzkoušejte si ji na počítači. Dříve než budete číst dál, pokuste se sami vyvodit pravidla, kterými se jazyk řídí při vyhodnocování seznamů.

Pravidlo pro vyhodnocování seznamů má tři části:

Pravidlo pro vyhodnocování seznamů je klíčové.

1. nejprve jsou vyhodnoceny (v nějakém nespecifikovaném pořadí) všechny prvky seznamu,
2. první prvek seznamu musí být vyhodnocen na *proceduru*,
3. procedura je *aplikována* na výsledky vyhodnocení dalších prvků seznamu.

V tomto pravidle zavádíme pojem *procedura*, který je znám z programovacích jazyků a označuje nějakou akci (algoritmus), kterou můžeme *vyvolat* s různě nastavenými parametry. Hovoříme také o *aplikaci* procedury na parametry. V jazyku máme některé procedury zabudované, například procedury pro elementární aritmetické výpočty.

Při vyhodnocování seznamu může dojít principiálně ke třem druhům chyb:

- Některý z prvků seznamu není vyhodnotitelný, jeho vyhodnocení způsobí chybu. To jsme viděli například při pokusu vyhodnotit seznam `(a b)`, kde `a` je neznámý symbol.
- Výsledkem vyhodnocení prvního prvku seznamu není procedura. Pokud bylo prvním prvkem seznamu číslo `1`, nešlo o proceduru a získali jsme chybu aplikace procedury.
- Procedura je volána s chybným počtem nebo chybným typem parametrů.

Uvědomme si, že výše zmíněné vyhodnocovací pravidlo je svou povahou *rekurzivní*, neboť je definováno pomocí sebe sama. Jednoduché výrazy (čísla a symboly) je možno vyhodnotit

Seznamy jsou vyhodnocovány rekurzivně.

přímo, u seznamů je třeba nejprve uplatnit vyhodnocovací pravidlo na všechny jeho prvky, které mohou být také seznamy.

Vyhodnocení seznamu představuje volání procedury.

Seznam představuje ve Scheme prostředek *volání* (*aktivace*, *aplikace*) procedury. Zápis, při kterém v seznamu uvádíme vždy na prvním místě jméno procedury následované parametry procedury se označuje jako *prefixová notace*. Pro čtenáře zvyklé na klasickou infixovou notaci používanou v matematických zápisech, je tento zápis poněkud nezvyklý, přináší nám však některé výhody. Umožňuje nám například jednoduše vytvářet procedury akceptující více než dva parametry a odstraňuje problémy priority jednotlivých procedur ve složených výrazech.

Průvodce studiem

Závorky slouží v jazyku Scheme jako základní řídící prostředek. Určují nám strukturu výrazů a běh výpočtu. Velmi častou chybou začátečníků je vynechání závorek v místě, kam patří, nebo naopak přidání nadbytečných závorek, kde být nemají. Zatímco v matematice nám nadbytečné závorky nevadí, zde je situace jiná. Výraz 2+3 má v matematice stejný význam jako (2)+(3). V jazyku Scheme se výraz `(+ 2 3)` *vyhodnotí na 5 zatímco výraz* `(+ (2) (3))` *způsobí chybu. Proč?*

1.2.3 Prostředky pro abstrakci

V předchozí kapitole jsme prakticky demonstrovali, jakým způsobem Scheme vyhodnocuje výrazy. Tyto výrazy (nazývané též *symbolické výrazy* nebo *s-výrazy*) jsme si rozdělili do tří základních skupin[1]: na čísla, symboly a seznamy.

Průvodce studiem

Symboly, které používáme pro označování nějaké hodnoty a očekáváme u nich existenci aktuální vazby, nazýváme v programovacích jazycích identifikátory. S tímto pojmem se také setkáme v chybových hlášeních jazyka. Mohli bychom tedy namísto „symbolů" hovořit o „identifikátorech proměnných". V tuto chvíli nám však v jazyku Scheme tento pojem nedává velký smysl (proč „proměnná", když se nemění, jaký je rozdíl mezi symbolem a identifikátorem?) a proto se přidržíme názvu symbol.

Podle těchto pravidel jsou symboly vyhodnocovány na svoji aktuální vazbu. Aktuální vazby mezi symboly a hodnotami uchovává překladač ve speciálních datových strukturách, nazývaných *prostředí*. Po spuštění překladače máme k dispozici tzv. *globální prostředí*, ve kterém je nadefinována celá řada vazeb (například vazby symbolů na procedury vykonávající základní aritmetické operace).

Prostředí, ve kterém jsou uloženy vazby symbolů, lze měnit.

Otázkou zůstává, jak můžeme prostředí, které máme k dispozici po spuštění překladače, modifikovat nebo rozšiřovat o nové symboly. Odpověď poskytuje následující příklad interakce s překladačem jazyka:

```
> mezisoucet
reference to undefined identifier: mezisoucet
> (define mezisoucet 123)
```

[1] Tento výčet ovšem není úplný.

```
> mezisoucet
123
```

Pokusme se analyzovat tuto interakci s překladačem. V prvním kroku jsme se pokusili vyhodnotit symbol `mezisoucet`. Výsledkem byla chybová zpráva, informující, že pro takový symbol neexistuje v prostředí aktuální vazba. V následujícím kroku jsme vyhodnotili seznam `(define mezisoucet 123)`. Vyhodnocení seznamu v podstatě znamená vyvolání procedury, která musí být na prvním místě seznamu. Toto volání však překvapivě nevrátilo žádnou hodnotu. Něco se však v systému změnilo, protože další pokus o vyhodnocení symbolu `mezisoucet` již dopadl úspěšně, symbol je navázán na hodnotu `123`. Evidentně jsme tedy **vytvořili vazbu** mezi symbolem `mezisoucet` a číslem `123`.

Průvodce studiem

Pokuste se analyzovat těchto pár řádků kódu na základě vašich současných znalostí jazyka. Protože symbol `define` je použit jako první prvek v seznamu, můžeme mlčky předpokládat, že se jedná o symbol, navázaný na nějakou zabudovanou proceduru jazyka, stejně jako je tomu u symbolů `+`, `-` a podobně.

Něco tady ale nehraje, něco se na základě našich znalostí děje zcela špatně! Zjistíte sami v čem je problém? A mimochodem, je `define` skutečně symbol, navázaný na proceduru?

Vazbu provádí speciální forma `define`.

Pro vytvoření vazby mezi symbolem a hodnotou jsme v kódu použili *speciální formu* `define`. Vyvolání speciální formy `define` má sice stejnou syntaxi jako volání procedury s názvem `define`, vyhodnocení výrazu `(define mezisoucet 123)` však nepodléhá běžným pravidlům pro vyhodnocení seznamu! Běžné vyhodnocení seznamu by totiž znamenalo v prvé řadě vyhodnocení všech jeho prvků a my víme, že vyhodnocení symbolu `mezisoucet` by nutně vedlo k chybě. Chování speciální formy `define` nám prozrazuje její výjimečnost.

Speciální formy nejsou procedury.

Speciální formy[1], kterých si časem představíme více, tvoří zabudované rysy jazyka a využíváme je v případech, kdy běžné procedury nedostačují. Každá speciální forma je svázána s jistým identifikátorem (klíčovým slovem), tento identifikátor nemůže být v programu použit jako běžný symbol. Seznamy, začínající identifikátorem speciální formy, představují výjimky ze základního pravidla vyhodnocování seznamů, které jsme představili v minulé podkapitole. Základní pravidlo vyhodnocování seznamu bychom proto měli přeformulovat takto:

1. ověříme, zda se na první pozici v seznamu nenachází identifikátor speciální formy; pokud ano, neaplikujeme další pravidla a postupujeme podle individuálních pravidel příslušné speciální formy,
2. vyhodnotíme (v nějakém nespecifikovaném pořadí) všechny prvky seznamu,
3. první prvek seznamu musí být vyhodnocen na *proceduru*,
4. procedura je aplikována na výsledky vyhodnocení dalších prvků seznamu.

Postup vyhodnocování vyvolání speciální formy `define` lze vyjádřit v těchto třech krocích:

Každá speciální forma má vlastní pravidla vyhodnocování.

1. první parametr formy `define` musí být symbol, tento symbol se nevyhodnocuje,

[1] O procedurách občas hovoříme jako o obyčejných formách, na rozdíl od speciálních forem jako je například speciální forma `define`.

2. druhým parametrem formy může být libovolný výraz; tento výraz se vyhodnotí,
3. vytvoří se vazba mezi symbolem a výsledkem vyhodnocení druhého parametru.

Speciální forma `define` představuje v jazyce Scheme základní prostředek abstrakce. Umožňuje označit libovolnou hodnotu jménem. Namísto konkrétních výrazů tak můžeme zapisovat symbolické, abstraktní výrazy.

Průvodce studiem

Při výpočtu celkové ceny výrobku včetně daně z přidané hodnoty můžeme například zapsat:

*750 + 750 * 19/100*

a získat tak cenu 892,50.
Za použití prostředků abstrakce lze však vzorec na výpočet ceny s DPH zapsat obecněji:

*cena + cena * dph / 100*

Do „proměnných" cena a dph pak stačí pomocí formy `define` *„dosadit" aktuální hodnoty. K tomuto příkladu se podrobněji vrátíme v následující kapitole.*

Shrnutí

Výrazy vložené do překladače se vyhodnocují různě, na základě svého typu. Čísla se vyhodnocují sama na sebe, symboly se vyhodnocují na svoji aktuální vazbu. U seznamů jsou nejprve vyhodnoceny všechny prvky seznamu, první prvek (procedura) je pak aplikován na zbývající prvky. Pravidlo vyhodnocování seznamů porušují speciální formy; seznam začínající dohodnutým identifikátorem speciální formy nepodléhá těmto pravidlům. Každá z forem si definuje vlastní pravidla vyhodnocování. Jednou ze speciálních forem je forma `define`, která vytváří vazbu mezi symbolem a libovolnou hodnotou.

Pojmy k zapamatování

- Seznam,
- speciální forma.

Kontrolní otázky

1. *Jak jsou v jazyku Scheme vyhodnocovány symboly?*
2. *Co se stane v okamžiku, pokusíme-li se vyhodnotit nenavázaný symbol?*
3. *Co je to seznam?*
4. *Jak se liší volání speciální formy od volání procedury?*
5. *Proč potřebujeme zavádět do jazyka speciální formy?*

Cvičení

1. Kolik prvků mají tyto seznamy: `()`, `((()))`, `((1 1))`, `(((3 3 3)) ())`?

2. Pravidlo vyhodnocování seznamů říká, že prvky seznamu jsou vyhodnoceny v nějakém nespecifikovaném pořadí. Z výpisu interakce s jazykem Scheme se pokuste zjistit, v jakém pořadí váš překladač vyhodnocuje prvky seznamů (zleva doprava nebo zprava doleva).

3. Přepište do infixové notace jazyka Scheme a vyhodnoťte následující výraz:
$$\frac{7(1+5.8)}{4(2.51-2.34)}$$

4. Přepište do infixové notace jazyka Scheme a vyhodnoťte následující výraz:
$$\frac{5+\frac{14}{3}+(2-(3-(6-\frac{4}{3})))}{(1-\frac{2}{3})(2-6)}$$

Úkoly k textu

1. Pokuste se předem odhadnout, co bude výsledkem vyhodnocení jednotlivých výrazů. Své tvrzení si pak ověřte na počítači:

```
(+ 1 1 1)
(+ 1 1)
(+ 1)
(+)
+
(/ 0 1)
(/ 1 0)
(/ (/ (/ (/ 16 2) 2) 2) 2)
(1 2 3 4)
1
2
(+ 1 2)
(+1 2)
(1 + 2)
```

2. Pokuste se předem odhadnout, co bude výsledkem vyhodnocení jednotlivých výrazů (předpokládejte, že výrazy jsou postupně zadávány překladači Scheme). Své tvrzení si pak ověřte na počítači

```
(define hodnota 10)
hodnota
(define hodnota 4)
hodnota
(* 2 hodnota)
(sqrt hodnota)
(+ hodnota hodnota)
(define hodnota2 hodnota)
hodnota2
(define hodnota (sqrt hodnota))
(+ hodnota hodnota2)
(define plus +)
(plus 3 3)
(define + *)
(+ 3 3)
```

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

3. Předpokládejme, že nevíte, zda speciální forma `define` vyhodnocuje nebo nevyhodnocuje třetí prvek seznamu. Vymyslete příklad s jehož pomocí lze tuto otázku rozhodnout.

Řešení

1. Seznamy mají postupně jak jsou uvedeny 0 prvků, 1 prvek, 1 prvek a 2 prvky.

2. Vyhodnocení seznamu `(a b)` způsobilo chybu při vyhodnocování symbolu `a`. Prvky seznamu se tedy zřejmě vyhodnocují zleva doprava.

3.
```
(* 7
   (/ (+ 1 5.8)
      (* 4 (- 2.51 2.34))))
```
Výsledkem je číslo 70.

4.
```
(/ (+ 5 14/3 (- 2 (- 3 (- 6 4/3))))
   (* (- 1 2/3) (- 2 6)))
```
Výsledkem je číslo -10.

Průvodce studiem

Mnozí čtenáři v tuto chvíli mohou nabýt dojmu, že základní principy programovacího jazyka Scheme jsou jednoduché a snadno pochopitelné. Zatímco první tvrzení je zcela jistě pravdivé, tím druhým si nejsme zcela jisti. Důkladné pochopení těch několika základních principů je přitom klíčové pro další studium. Stejně tak je důležité pochopení konzistence a nekompromisnosti jazyka: tyto základní principy jsou aplikovány za všech okolností stejně.

1.3 Uživatelské procedury

Studijní cíle: Po absolvování kapitoly bude studující schopen napsat v jazyce Scheme krátké procedury pro zpracování jednoduchých matematických výrazů.

Klíčová slova: Uživatelská procedura, parametry procedury, pretty-printing, substituční model.

Potřebný čas: 4 hodiny.

V programech potřebujeme vytvářet vlastní procedury.

V předchozích kapitolách jsme se seznámili s možnostmi využití „předdefinovaných“ procedur. Zmínili jsme zejména procedury pro aritmetické výpočty, ale v manuálu jazyka bychom jistě mohli dohledat mnohé další dostupné procedury. Stejně důležité jako schopnost využití předdefinovaných procedur je možnost vytváření vlastních *uživatelských procedur*.

Existují v principu dva důvody, proč potřebujeme v programu vytvářet vlastní procedury. Za prvé, procedury představují ve většině programovacích jazyků základní mechanismus modularizace. Vhodné členění programu na procedury je zcela klíčové pro jeho pochopení a odladění. Za druhé, procedury představují další způsob abstrakce, možnost označit komplexní operaci jedním názvem a dále nad její strukturou nepřemýšlet, abstrahovat od její struktury. Vrátíme-li se zpět k příkladu, uvedenému v průvodci studiem v kapitole 1.2.3, konkrétní cenu

výrobku spočteme například pomocí výrazů 750 + 750 * 0.19, 24 + 24 * 0.05 nebo 5340 + 5340 * 0.19. Tyto výrazy nám však neříkají nic o samotném **mechanismu** výpočtu ceny, nepostihují **princip** výpočtu ceny. Tento mechanismus můžeme zachytit právě v uživatelských procedurách.

1.3.1 Vytváření uživatelských procedur

Uživatelské procedury vytváříme pomocí speciální formy `define`.

Pro vytváření uživatelských procedur využíváme opět speciální formu `define`. Zatímco v kapitole 1.2.3 jsme však používali jako první parametr formy `define` vždy symbol, zde je prvním parametrem seznam. Vytvoření jednoduché uživatelské procedury demonstrujeme na následujícím příkladě procedury pro výpočet celkové ceny včetně DPH:

```
(define (cena_s_dph cena dph) (+ cena (* cena (/ dph 100))))
```

Tento zápis můžeme číst jako: „Abychom zjistili cenu s DPH za využití ceny a sazby DPH, sečteme cenu a cenu vynásobenou sazbou DPH vydělenou 100.“ Proceduru pak použijeme stejně jako libovolnou jinou zabudovanou proceduru, například:

```
(cena_s_dph 750 19)
```

a tímto voláním uživatelské procedury získáme výslednou cenu 892.50.

Definice procedury je po formální stránce seznam, který má tři části:

- začíná klíčovým slovem `define`,
- na druhé pozici je vzor volání této procedury: musí jít o **seznam** obsahující název procedury a všechny formální parametry procedury,
- na třetí pozici je tělo procedury: libovolný výraz, který je vyhodnocen při volání procedury.

Při vyhodnocení této definice postupuje jazyk Scheme v těchto dvou krocích:

- vytvoří proceduru s danými parametry a daným tělem,
- naváže symbol jména procedury na tuto nově vzniklou proceduru.

Tento proces můžeme demonstrovat na následující interakci s jazykem Scheme. v tomto případě budeme vytvářet proceduru pro výpočet druhé mocniny čísla:

```
> na2
reference to undefined identifier: na2
> (define (na2 x) (* x x))
> na2
#<procedure:na2>
> (na2 5)
25
> (na2 6)
36
> (na2 (na2 2))
16
> (+ 5 (na2 (- 8 2)))
41
```

Všimněme si výsledku vyhodnocení symbolu `na2` před a po definici procedury. Dále nepřehlédněme, že nově definovanou uživatelskou proceduru je možno používat stejně jako zabudované systémové procedury.

Průvodce studiem

Je poněkud nešťastné, že jazyk Scheme používá speciální formu `define` pro dvě odlišné činnosti: je-li první parametr symbol, naváže tento symbol na hodnotu, je-li první pa-

rametr vzor volání procedury (seznam), vytvoří proceduru a naváže na ni symbol. Mnohem vhodnější by bylo vytvořit zcela jinou speciální formu pro vytváření procedur, například:

(procedure (na2 x) (x x))*

Systém jazyka Scheme nám umožňuje zavést takovouto syntaxi pomocí tzv. makrosystému. Popis tohoto mechanismu je však mimo rozsah tohoto textu.

1.3.2 Pretty-printing

Pretty-printing slouží k zpřehlednění kódu.

Procedura na výpočet ceny výrobku včetně DPH z kapitoly 1.3.1 je zapsána na jediném řádku programu. Překladači jazyka Scheme je jedno, jestli je procedura zapsána na jednom nebo více řádcích – definice každé procedury je ze syntaktického hlediska seznam a překladač tak snadno pozná, kde definice končí. Proceduru proto většinou zapisujeme na více řádcích a snažíme se odsazením naznačit, které části programu logicky patří k sobě.

Takovéto rozdělení procedury na více řádků a odsazení jednotlivých řádků nazýváme *pretty-printing*. Proceduru na výpočet ceny s DPH bychom za dodržení zásad pretty-printing mohli zapsat takto:

```
(define (cena_s_dph cena dph)
   (+ cena
      (* cena (/ dph 100))))
```

První řádek procedury obsahuje pouze její hlavičku (klíčové slovo `define` a vzor volání procedury), druhý řádek obsahuje volání procedury `+`. Třetí řádek obsahuje výpočet DPH z ceny a sazby daně. Odsazení nám přitom prozrazuje, že sčítáme dva výrazy: cenu a DPH.

Pretty-printing kódu není zcela jednoznačně dán. Předchozí program bychom mohli například rozdělit jen do dvou řádků (hlavička a tělo) nebo do čtyřech či více řádků. Podstatné pro pretty-printing je dosažení maximální čitelnosti programu a dodržení takového odsazení, které odpovídá logickému členění kódu.

Průvodce studiem

Celá řada integrovaných programovacích prostředí nám se správným zarovnáním jednotlivých řádků programu pomáhá. V prostředí DrScheme můžeme nechat automaticky odsadit řádek podle struktury programu tím, že na tomto řádku stiskneme tabelátor (toto chování je převzato z editoru Emacs). Celý program pak můžeme správně odsadit stiskem Ctrl-I.

Pretty-printing je nepostradatelná pomůcka pro pochopení a ladění programů.

1.3.3 Příklad: výpočet obsahu trojúhelníka

Obsah trojúhelníka pomocí Heronova vzorce.

V tomto ukázkovém příkladě napíšeme proceduru na výpočet obsahu trojúhelníka pomocí Heronova vzorce. Obsah trojúhelníka o stranách a, b, a c lze vypočítat jako:

$$P = \sqrt{s(s-a)(s-b)(s-c)}$$

kde výraz s je roven

$$s = \frac{a+b+c}{2}.$$

Naším cílem bude vytvořit proceduru `heron`, která akceptuje parametry `a`, `b` a `c` a vrátí obsah trojúhelníka. Existuje několik možných přístupů k této úloze. Převést vzorec do prefixového tvaru by nám po absolvování kapitoly 1.2 již nemělo činit potíže. Problém však bude, jak přistoupit k parametru `s`.

Průvodce studiem

Dříve než budete pokračovat ve čtení, pokuste se problém vyřešit sami. Veškeré své pokusy provádějte přímo na počítači, v prostředí programovacího jazyka. Jen tak si můžete být jistí, že váš program funguje.

V první chvíli se můžeme pokusit programátorské řešení problému převést na řešení matematické a parametr `s` dosadit do vzorce:

$$P = \sqrt{\frac{a+b+c}{2}\left(\frac{a+b+c}{2}-a\right)\left(\frac{a+b+c}{2}-b\right)\left(\frac{a+b+c}{2}-c\right)}$$

Vznikne tak objemný vzorec, který však umíme mechanicky převést do jazyka Scheme[1]:

```
(define (heron a b c)
  (sqrt (* (/ (+ a b c) 2)
           (- (/ (+ a b c) 2) a)
           (- (/ (+ a b c) 2) b)
           (- (/ (+ a b c) 2) c))))
```

Při rozdělení do více procedur se parametr počítá jen jednou.

Většina z nás ale cítí, že tento přístup není optimální. Vzorec a následně i program jsou zbytečně rozsáhlé. Výpočet je navíc neefektivní, protože součet velikostí stran vydělený dvěma je počítán celkem čtyřikrát. Vraťme se tedy k myšlence substituce a rozdělme program na dvě procedury: hlavní proceduru `heron` a pomocnou proceduru `heron-pomoc`. Pomocná procedura implementuje hlavní vzorec, akceptuje však vedle velikostí stran i parametr `s`. Hlavní procedura pouze zavolá pomocnou proceduru a dodá jí velikosti stran a vypočtenou hodnotu `s`.

```
(define (heron-pomoc a b c s)
  (sqrt (* s (- s a) (- s b) (- s c))))

(define (heron a b c)
  (heron-pomoc a b c (/ (+ a b c) 2)))
```

Tato verze je již podstatně efektivnější, parametr `s` je počítán pouze jednou a vzorec pro výpočet obsahu trojúhelníka je v programu jasně viditelný.

Průvodce studiem

Vraťte se ke kapitole 1.2.2 a znovu si připomeňte základní pravidlo pro vyhodnocování seznamů. První bod pravidla nám říká, že nejprve vyhodnotíme všechny prvky seznamu Výpočet parametru s tedy probíhá skutečně jen jednou, v proceduře `heron`. Procedura

[1] Všimněte si zarovnání kódu. Program si přepište do prostředí jazyka Scheme a vyzkoušejte.

heron-pomoc již pracuje s číselnou hodnotou s. *K tomuto tématu se ještě vrátíme v kapitole 1.3.5.*

Někteří čtenáři mohou jako jistou vadu na kráse vnímat rozdělení problému do dvou procedur. Ukažme proto ještě jednu možnost, která vede zřejmě k nejpřehlednějšímu a nejstručnějšímu kódu. Tato verze využívá možnost lokálních definicí symbolů.[1]

Procedura může obsahovat lokální definice symbolů.

```
(define (heron a b c)
  (define s (/ (+ a b c) 2))
  (sqrt (* s (- s a) (- s b) (- s c))))
```

Při vyvolání procedury `heron` se nejprve naváže symbol `s` na vypočtenou hodnotu. Pak se vypočte a vrátí hodnota obsahu trojúhelníka. Symbol `s` je přitom *lokální* v proceduře `heron`, není dostupný mimo tuto proceduru. Pokud by náhodou v systému existovala *globální vazba* symbolu `s` na nějakou hodnotu, *lokální vazba* parametru `s` tuto vazbu po dobu vykonávání procedury `heron` *zastíní*.

1.3.4 Příklad: řešení kvadratické rovnice

V kvadratické rovnici se vyskytuje konstanta, proměnná (nebo její násobek) a druhá mocnina proměnné. Kvadratickou rovnici zapisujeme nejčastěji jako $ax^2 + bx + c = 0$. Napišme nejprve proceduru, která na základě znalosti čísel `a`, `b`, `c` a `x` spočte hodnotu kvadratického výrazu:

Řešení kvadratické rovnice.

```
(define (kvadratik a b c x)
  (+ (* a (sqr x)) (* b x) c))
```

Tento program můžeme vyzkoušet například na těchto hodnotách:

```
> (kvadratik 1 -6 8 3)
-1
```

Řešením kvadratické rovnice jsou čísla `x1` a `x2`, pro která je kvadratický výraz roven nule. Lze je spočítat pomocí známých vzorců:

$$x_1 = \frac{-b + \sqrt{D}}{2a} \qquad \text{a} \quad x_2 = \frac{-b - \sqrt{D}}{2a},$$

kde diskriminant spočteme jako $D = b^2 - 4ac$.

Programové řešení úlohy bude spočívat v implementaci třech procedur: procedury `diskriminant`, která na základě čísel `a`, `b` a `c` spočte diskriminant rovnice, procedury `reseni1`, která vrátí první řešení rovnice a procedury `reseni2`, která vrátí druhé řešení rovnice.

```
(define (diskriminant a b c)
  (- (sqr b) (* 4 a c)))

(define (reseni1 a b c)
  (/ (+ (- b) (sqrt (diskriminant a b c)))
     (* 2 a)))

(define (reseni2 a b c)
```

[1] Lokální definice symbolů nejsou z hlediska jazyka tím nejčistějším mechanismem. Vymykají se například tzv. substitučnímu modelu volání procedury, který představíme v kapitole 1.3.5. Lokální definice jazyk povoluje pouze na začátku procedury, před uvedením samotného těla procedury. Z hlediska čistoty kódu bylo vhodnější použití speciální formy `let` nebo `letrec`. Lokální definice jsou však uživatelsky velmi přímočaré a my je proto v tomto textu budeme používat.

```
  (/ (- (- b) (sqrt (diskriminant a b c)))
     (* 2 a)))
```

Pomocí těchto procedur se můžeme například pokusit vyřešit kvadratickou rovnici $x^2 - 6x + 8 = 0$, zde a = 1, b = –6 a c = 8:

```
> (reseni1 1 -6 8)
4
> (reseni2  -6 8)
2
```

Za použití procedury `kvadratik` můžeme zároveň provést zkoušku:

```
> (kvadratik 1 -6 8 4)
0
> (kvadratik 1 -6 8 2)
0
```

Průvodce studiem

V diskusi k tomuto příkladu jsme neuvažovali možnost, že diskriminant rovnice bude záporný a rovnice tak nebude mít reálné řešení. Takováto situace by mohla být považována za chybu a náš program by ji mohl ošetřit; zatím však neznáme prostředky, jak toho dosáhnout. Ve skutečnosti si můžete ověřit, že jazyk Scheme je schopen pracovat i s komplexními čísly a záporný diskriminant nepředstavuje pro náš program žádný problém.

1.3.5 Substituční model volání procedury

Uvažujme následující deklaraci těchto třech procedur:

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

(define (sc x y)
  (+ (na2 x) (na2 y)))

(define (f a)
  (+ (sc (+ a 2) (+ a 4))
     (sc (- a 2) (- a 4))))
```

Mechanismus vyhodnocení výrazu popisuje substituční model.

Pokuste se odpovědět na otázku, co překladač vrátí jako výsledek vyhodnocení procedury `f` s parametrem `5`: `(f 5)`. Správná odpověď je 140 a většina čtenářů pravděpodobně byla schopna k tomuto číslu intuitivně dojít. Asi ne všichni čtenáři by však byli schopni srozumitelně a logicky *vysvětlit*, proč je výsledkem právě tato hodnota.

Odpověď na tuto otázku nám dává *substituční model volání procedury*: voláme-li v programu uživatelskou proceduru, nahradíme místo volání tělem procedury, přičemž substituujeme skutečné parametry na místo formálních. Máme-li tedy například kód

```
(* 2 (f 5))
```

provedeme nahrazení seznamu `(f 5)` tělem procedury `f`, přičemž parametr `a` nahradíme hodnotou `5`. Dále postupujeme stejně, a to tak dlouho, dokud nedostaneme výraz obsahující pouze primitivní (zabudované) procedury. Demonstrujme substituční model volání procedury na následujícím příkladu. Každý řádek odpovídá jednomu z kroků algoritmu substitučního modelu.

```
(* 2 (f 5))
(* 2 (+ (sc (+ 5 2) (+ 5 4)) (sc (- 5 2) (- 5 4))))

(* 2 (+ (sc 7 9) (sc 3 1)))
```

```
(* 2 (+ (sc 7 9) (sc 3 1)))
(* 2 (+ (+ (na2 7) (na2 9)) (sc 3 1)))

(* 2 (+ (+ (na2 7) (na2 9)) (sc 3 1)))
(* 2 (+ (+ (na2 7) (na2 9)) (+ (na2 3) (na2 1))))

(* 2 (+ (+ (na2 7) (na2 9)) (+ (na2 3) (na2 1))))
(* 2 (+ (+ (* 7 7) (* 9 9)) (+ (* 3 3) (* 1 1))))

280
```

Dvojice řádků označují výraz před a po substituci. Zvýrazněné části kódu označují volání procedury určené k substituci a výsledek provedené substituce. Substituci procedury `na2` v posledním kroku jsme pro jednoduchost provedli v celém výrazu najednou.

Substituční model má aplikativní a normální variantu.

Samostatně stojící řádek naznačuje, že pokud je to možné, provádí tento model zjednodušení výrazů dříve, než provede substituci. Tato varianta modelu se nazývá *aplikativní substituční model* a je v souladu s pravidlem pro vyhodnocování seznamů popsaným v kapitole 1.2.2. Pokud bychom neprováděli zjednodušení výrazu před substitucí, mohli bychom hovořit o *normálním substitučním modelu*. Zavedení normálního substitučního modelu by mělo pro náš jazyk značné následky. Přestal by například platit první bod pravidla o vyhodnocování seznamů z kapitoly 1.2.2 a nepotřebovali bychom zavádět řadu speciálních forem.

Průvodce studiem

Nezapomeňme, že se jedná o pouhý ***model*** *volání procedury. Tento model nám dává možnost pochopit volání procedur zejména v okamžicích, kdy si nebudeme zcela jisti výsledkem. Neznamená to však, že takto v programovacím jazyku volání procedur opravdu probíhá. Bylo by asi značně neefektivní nahrazovat v programu jeden řetězec jiným a ten pak ještě prohledávat na výskyt formálních parametrů.*

I když jazyk řeší ve skutečnosti volání procedur jinak, zřejmě složitěji a efektivněji, pro naše potřeby bude tento model dostačující.

Shrnutí

Uživatelské procedury vytváříme pomocí speciální formy `define`, namísto symbolu však jako první parametr uvedeme vzor volání procedury a jako druhý parametr tělo procedury. Proceduru většinou zapisujeme na více řádků a jednotlivé části odsazujeme. Procedura může obsahovat interní definice symbolů. Volání procedury lze modelovat pomocí substitučního modelu, kdy místo volání procedury nahradíme jejím tělem a formální parametry nahradíme skutečnými.

Pojmy k zapamatování

- Uživatelská procedura,
- speciální forma `define`,
- pretty-printing,
- interní definice,
- aplikační substituční model volání procedury,
- normální substituční model volání procedury.

Kontrolní otázky

1. *Proč potřebujeme vytvářet uživatelské procedury?*

2. *K čemu slouží pretty-printing a co je jeho podstatou?*
3. *Kde se v těle procedury může vyskytovat interní definice symbolu?*
4. *Proč se substituční model volání procedury nazývá substituční?*
5. *Používá Scheme normální nebo aplikativní verzi modelu?*

Cvičení

1. Za pomoci procedury `na2` pro výpočet druhé mocniny čísla napište procedury `na4`, `na6` a `na8` pro výpočet čtvrté, šesté a osmé mocniny čísla.

2. Napište proceduru `plast-valce`, která akceptuje poloměr a výšku válce a vrací velikost obsahu pláště válce.

3. Strojírenský podnik potřebuje opatřit velké roury antikorozní úpravou. Ošetřit je třeba vnější i vnitřní povrch roury. Aby bylo možno odhadnout množství potřebného antikorozního přípravku, je potřeba zjistit součet vnitřního a vnějšího povrchu roury. Roura má daný vnitřní poloměr, délku a tloušťku stěny. Vytvořte proceduru `povrch-roury`.

Úkoly k textu

1. Napište proceduru `pocet-sekund`, která akceptuje délku časového intervalu ve dnech, hodinách, minutách a sekundách a vrátí délku tohoto intervalu v sekundách. Například 15 dnů, 4 hodiny, 25 minut a 5 sekund odpovídá 1311905 sekundám.

2. Napište proceduru `minut`, která akceptuje časovou informaci v sekundách a vrátí odpovídající množství celých minut. Použijte přitom proceduru `quotient`. Například 1311905 sekund odpovídá 21865 celým minutám.

3. Obdobně napište procedury `hodin` a `dnu`, které k informaci v sekundách vrátí počet celých hodin a počet celých dnů. 1311905 sekund odpovídá 364 celým hodinám a 15 celým dnům.

4. Vedení kina potřebuje zjistit celkový zisk za jedno představení v závislosti na počtu diváků. Každý divák zaplatí za lístek jednotnou cenu 100,-Kč. Každé představení stojí kino na režii paušálně 2500,- Kč. Úklidová forma si však účtuje náklady na úklid hlediště podle počtu přítomných diváků, a to 3,40 Kč za jednoho diváka. Napište proceduru `zisk-kina`.

5. 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 (některá zadání mohou skončit chybovým hlášením).

```
(define (sc x y) (+ (* x x) (* y y)))
(sc 2 3)
(sc)
sc
(define (ident x) x)
(ident 3)
(ident (ident 3))
((ident ident) 3)
((ident sc) (ident 2) (ident 3))
```

Řešení

1. Chceme-li maximálně využít existující procedury, může vypadat řešení úkolu například takto:

```
(define (na4 x) (na2 (na2 x)))
(define (na6 x) (* (na2 x) (na4 x)))
(define (na8 x) (na2 (na4 x)))
```

2.
```
(define (plast-valce polomer vyska)
  (* 2 pi polomer vyska))
```

3. K řešení můžeme přistoupit dvěma způsoby. Který ze způsobů vzbuzuje více důvěry?

```
(define (povrch-roury polomer delka tloustka)
  (+ (plast-valce polomer delka)
     (plast-valce (+ polomer tloustka) delka)))

(define (povrch-roury polomer delka tloustka)
  (+ (2 pi polomer delka)
     (2 pi (+ polomer tloustka) delka)))
```

1.4 Podmíněné příkazy

Studijní cíle: Po prostudování kapitoly bude studující schopen používat v programu podmíněné výrazy `if` a `cond`. Pochopí také význam podmíněných výrazů v programování.

Klíčová slova: Logické hodnoty, predikáty, podmíněné příkazy.

Potřebný čas: 2 hodiny.

1.4.1 Proč potřebujeme podmíněné příkazy

Podmíněné příkazy tvoří zásadní rys jazyka.

Počítač má oproti člověku tři základní výhody: je velmi přesný, velmi rychlý a neunaví se. Máme-li provést jediný výpočet, bude asi rychlejší sáhnout po papíru a tužce než psát složitý program pro počítač. Pokud však musíme stejný výpočet provést pro různé hodnoty tisíckrát, oceníme rychlost a neúnavnost počítače. V počítačové terminologii nazýváme opakování stejného výpočtu *cyklus*.

Průvodce studiem

Uvažujme následující proceduru:

```
(define (pokus) (pokus))
```

Co se stane, když tuto proceduru vyvoláme? Vyzkoušejte to na počítači.

Aby byl cyklus pro výpočet užitečný, měl by dříve či později skončit. Jen tak zjistíme konečný výsledek výpočtu. Počítač tedy musí obsahovat mechanismus, jak se **rozhodnout**, jestli pokračovat v dalším průběhu cyklu nebo ne. Potřebujeme mechanismus *větvení výpočtu*, kdy se na základě platnosti nějaké podmínky rozhodneme, kterou cestou se v dalším výpočtu vydáme. Tuto službu nám poskytují *podmíněné příkazy*.

Je jen málo skutečně užitečných programů, které by bylo možno napsat bez použití podmíněného příkazu. Podmíněný příkaz je tak hned po volání procedury druhou nejdůležitější řídící strukturou programu.

1.4.2 Logické hodnoty

Logické hodnoty označujeme `#t` a `#f`.

Dosud jsme v jazyce Scheme pracovali se třemi druhy veličin: s čísly, symboly a se seznamy. Abychom mohli začít hovořit o podmíněném příkazu, musíme se zmínit o dalším typu veličin, reprezentujícím logické (Booleovské) hodnoty *pravda* (*true*) a *nepravda* (*false*).

V jazyce Scheme jsou pro tyto hodnoty vyhrazeny dva speciální symboly: `#t` (pro pravdu) a `#f` (pro nepravdu). Tyto symboly se, stejně jako čísla, vyhodnocují samy na sebe. Vyhodnocením symbolu `#t` je tedy opět symbol `#t` a vyhodnocením symbolu `#f` je symbol `#f`.[1]

1.4.3 Predikáty

Predikáty jsou procedury vracející logické hodnoty.

Procedury, které jako svoji hodnotu vracejí jeden ze symbolů `#t` nebo `#f`, se nazývají *predikáty*. K nejpoužívanějším patří zřejmě relační predikáty pro práci s čísly `<`, `>` a `=`. Často se také využívají logické spojky `and`, `or` a unární predikát `not`.[2]

Použití predikátů se v ničem neliší od použití běžných procedur:

```
> (< 1 2)
#t
> (>= 4 (+ 2 3))
#f
> (and (= 4 (* 2 2)) (or (> 0.5 (cos 0)) (< (sin 1) 0)))
#f
```

Predikáty mohou být zabudované nebo uživatelské.

Kromě zabudovaných predikátů si můžeme vytvářet i predikáty uživatelské. Jako příklad uveďme predikát `trojuhelnik?`, který umožňuje rozhodnout, jestli tři čísla mohou tvořit délky stran trojúhelníka.[3] Bývá zvykem jméno predikátu zakončovat otazníkem; otazník je součástí názvu a nemá žádný speciální význam. Jde jen o signál programátorovi, že daná procedura vrací logickou hodnotu.

```
(define (trojuhelnik? a b c)
  (and (> (+ a b) c)
       (> (+ b c) a)
       (> (+ a c) b)))
```

Obdobně můžeme implementovat další predikáty. První dva z následujících predikátů používají zabudovanou proceduru `remainder` (zbytek po dělení dvou čísel) a určují jestli je dané číslo liché nebo sudé. Třetí predikát je implementace logické spojky XOR.

```
(define (odd? x)
  (= (remainder x 2) 1))

(define (even? x)
  (not (odd? x)))

(define (xor x y)
  (or (and x (not y))
      (and y (not x))))
```

---

[1] V některých implementacích jazyka lze pro lepší čitelnost programu použít i synonyma `true` a `false`.

[2] `and` a `or` ve skutečnosti nejsou procedury, ale speciální formy. Další predikáty lze nalézt v referenčním manuálu jazyka (viz [R5RS]).

[3] Tři čísla mohou tvořit délky stran trojúhelníka, pokud součet libovolných dvou čísel je vždy větší než číslo třetí.

Průvodce studiem

Všechny tyto procedury vyzkoušejte na počítači. Kromě toho, že si tím ověříte funkčnost těchto procedur, získáte i praxi v psaní a odlaďování kódu v jazyce Scheme.

I při psaní jednoduchých programů mohou programátoři udělat chyby: někdy se přepíší v názvu některého symbolu, jindy špatně umístí závorku. Program pak pochopitelně nefunguje a programátor musí zjistit proč. Hledání vlastních chyb tak tvoří značnou část programátorovy práce. Tuto dovednost lze získat pouze praxí.

1.4.4 Speciální forma `if`

`if` je nejjednodušší podmíněný výraz.

Nejjednodušším podmíněným výrazem v jazyce Scheme je speciální forma `if`. Tato speciální forma akceptuje tři parametry. Prvním parametrem je predikát, druhým parametrem je výraz, který bude vyhodnocen tehdy, je-li predikát splněn, a třetím parametrem je alternativní výraz – je vyhodnocen tehdy, když predikát není splněn. Formálně vypadá tvar speciální formy `if` takto:

```
(if <predikát> <výraz> <alt-výraz>)
```

Tuto speciální formu můžeme vyzkoušet při následující interakci s jazykem Scheme:

```
(if (> 1 2) (+ 3 4) (- 2 5))
(if (even? 4) (+ 4 1) (- 4 1))
```

V prvním případě nám Scheme vrátí hodnotu `-3`, protože jednička je větší než dvojka. v druhém případě dostaneme hodnotu `5`, protože čtyřka je sudé číslo.

Použití této speciální formy v programu demonstrujeme na proceduře na výpočet absolutní hodnoty čísla a na proceduře pro určení většího ze dvou čísel.

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

(define (maximum x y)
  (if (> x y) x y))
```

V některých případech může speciální forma `if` akceptovat jen dva výrazy – predikát a výraz:

```
(if <predikát> <výraz>)
```

Pokud je predikát splněn, vrací forma výsledek vyhodnocení výrazu. Jinak je výsledek nedefinovaný. Tato varianta má smysl pouze u procedur, které pracují na základě svého vedlejšího efektu a my ji nebudeme používat.

Průvodce studiem

I když to vypadá nepravděpodobně, chybný počet parametrů speciální formy `if` je jedním z nejčastějších syntaktických chyb v programech začínajících programátorů. Pokud programátor špatně umístí závorky, může se stát, že je speciální forma `if` volána se čtyřmi nebo více parametry. Překladač tuto situaci detekuje jako syntaktickou chybu. Horší je, pokud programátor omylem uvede ve formě `if` jen dva parametry. Syntakticky je program v pořádku a chyba se projeví až za běhu programu, nejčastěji jako chyba typové kontroly.

U složitějších programů proto důkladně dbejte na správné uzávorkování programu. Výrazně vám zde pomůže pretty-printing programu (viz kapitola 1.3.2).

1.4.5 Speciální forma cond

Speciální forma `if` umožňuje větvit výpočet na dvě alternativní větve. V některých případech však potřebujeme výpočet větvit na více alternativních větví. Tento problém lze jistě řešit pomocí sekvence do sebe vnořených příkazů `if`. Takovýto zápis je však poněkud nepřehledný a programovací jazyk Scheme proto zavádí speciální formu `cond`.

cond používáme při větvení do více větví výpočtu.

Formálně vypadá tvar speciální formy `cond` takto:

```
(cond (<predikát1> <výraz1>)
      (<predikát2> <výraz2>)
      …
      (else <alt-výraz>))
```

Vyhodnocení podmíněného výrazu `cond` probíhá tak, že překladač postupně vyhodnocuje jednotlivé predikáty dokud nenarazí na takový predikát, který se vyhodnotí na `#t`. Pak vyhodnotí odpovídající výraz a výsledek tohoto vyhodnocení vrátí jako výsledek vyhodnocení celého výrazu `cond`. Výrazů může být pro daný predikát uvedeno i více, v tom případě se vyhodnotí postupně všechny a vrátí se hodnota posledního z nich.

Je-li potenciálně splněno více predikátů, překladač bere v úvahu pouze první úspěšný predikát (další se ani nevyhodnocují). Není-li splněn žádný z predikátů, není výsledná hodnota definována. Namísto posledního predikátu je však možno použít klíčové slovo `else`, které v tomto kontextu zastupuje predikát, který je vždy splněn. I když tato alternativní větev není povinná, budeme ji vždy používat.

Speciální formu cond můžeme demonstrovat na proceduře pro výpočet signa čísla. Signum je matematická funkce nabývající hodnoty -1 pro záporné argumenty, hodnoty 1 pro kladné argumenty a hodnoty 0 pro nulu. Pro porovnání uvádíme proceduru signum zapsánu pomocí sekvence příkazů `if` a pomocí příkazu `cond`.

cond použijeme na výpočet signa čísla.

```
(define (signum x)
  (if (< x 0) -1
      (if (> x 0) 1 0)))

(define (signum x)
  (cond ((< x 0) -1)
        ((> x 0) 1)
        (else 0)))
```

Průvodce studiem

Speciální forma cond má poněkud obtížnější syntaxi než ostatní doposud představené formy. Častou chybou začátečníků je, že nevnímají logickou strukturu formy a vynechají jednu ze závorek. Zatímco v některých případech se podaří chybu odhalit na úrovni syntaktické, někdy dostaneme sémantickou chybu, která vede chybě za běhu programu nebo k nesprávnému výsledku. Pro kontrolu správnosti syntaxe a sémantiky formy cond nám vždy pomůže pretty-printing výrazu.

Shrnutí

V programu můžeme používat logické hodnoty `#t` a `#f`. Procedury, které vrací logické hodnoty, nazveme predikáty. Jejich názvy většinou končí otazníkem. Pro větvení výpočtu na dvě větve používáme speciální formu `if`. Pro větvení výpočtu na více větví používáme speciální formu `cond`.

Pojmy k zapamatování

- Logická hodnota, `true`, `false`,
- predikát,
- speciální forma `if`,
- speciální forma `cond`.

Kontrolní otázky

1. *Jak se vyhodnocují výrazy `#t` a `#f`?*
2. *Které předdefinované predikáty znáte?*
3. *Kdy je vhodnější použít speciální formu `cond` namísto `if` a proč?*

Cvičení

1. Předpokládejme, že `and` a `or` jsou obyčejné procedury. Pokuste se najít výraz, který toto tvrzení vyvrátí.

2. Obdobně vyvraťte, že `if` není procedura.

Úkoly k textu

1. Napište predikáty `kladne?`, `zaporne?` a `mezi?`, které vrátí logickou hodnotu podle toho, je-li číslo kladné, záporné a v intervalu vymezeného dvěma čísly. Například:
   `(kladne? -5)` => vyhodnotí se na `#f`
   `(zaporne? -5)` => vyhodnotí se na `#t`
   `(mezi? 4 2 5)` => vyhodnotí se na `#t`
   `(mezi? 9 2 5)` => vyhodnotí se na `#f`

2. Napište proceduru `maximum3` pro nalezení největšího ze třech čísel. Nezapomeňte otestovat všechny větve výpočtu. Například:
   `(maximum3 4 1 7)` => vyhodnotí se na `7`
   `(maximum3 0 4 2)` => vyhodnotí se na `4`
   `(maximum3 9 2 6)` => vyhodnotí se na `9`

3. Napište proceduru `soucet2vetsich`, která akceptuje tři argumenty a vrátí součet dvou větších čísel.

Řešení

1. Při volání procedury dochází vždy k vyhodnocení všech prvků seznamu. Vyhodnocení seznamu:

   ```
   (and (> 1 2) test)
   ```

   však vrátí hodnotu `#f`. Pokud by `and` byla obyčejná procedura, muselo by vyhodnocení skončit chybou, protože `test` je nenavázaný symbol. Obdobně postupujeme u formy `or`.

2. Postupujeme podobně jako v předchozím cvičení. Stačí vyhodnotit například výraz:

```
(if (> 1 0) 1 pokus)
```

1.5 Projekt: počítačová hra „Hádání čísel“

Studijní cíle: Studující si procvičí práci s podmíněnými výrazy a naučí se logicky analyzovat problémy.

Klíčová slova: Podmíněné výrazy, pomocné procedury.

Potřebný čas: 3 hodiny.

1.5.1 Úvod do hry

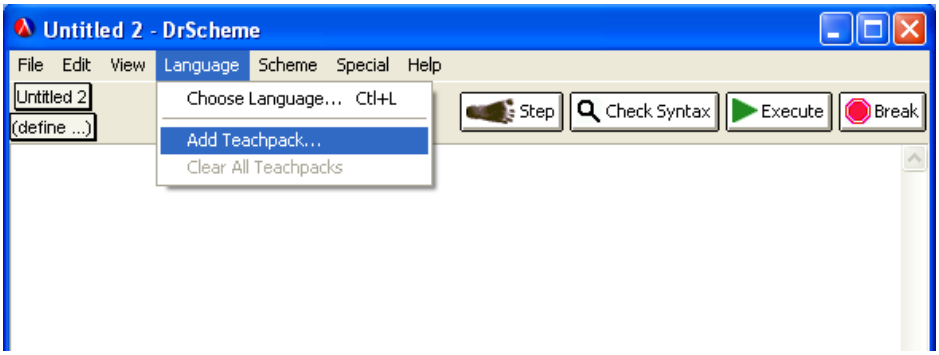
Počítačová hra „hádání čísel“ je založena na jednoduchém principu. Jeden hráč (v našem případě počítač) náhodně zvolí trojciferné číslo. V našem programu budeme toto číslo nazývat *cíl*. Druhý hráč (uživatel) se pokouší toto číslo uhodnout. První hráč přitom druhému hráči poskytuje nápovědu o tom, jak daleko respektive blízko je jeho odhad cíli.

Uživatel hádá počítačem zvolené číslo.

Jednotlivé strategie hry odpovídají různým způsobům nápovědy prvního hráče.

Pro demonstraci hry použijeme soubor `guess.scm`, který přidáme jako tzv. teachpack do jazyka Scheme. Využijeme přitom volbu menu Language/Add Teachpack.

Podpůrná funkcionalita hry je implementována v teachpacku.



Obr. 4 Přidání teachpacku

Průvodce studiem

Teachpack dodává další funkcionalitu vašemu programovacímu jazyku. Jedná se modul, který připravil instruktor kursu nebo který je dodávaný jako součást integrovaného prostředí DrScheme. Pokud čtete tento text on-line na internetu, bude nejprve potřeba teachpack „stáhnout“ a uložit na lokální disk.

Pokud nepřidáte daný teachpack do prostředí, nebude kód uvedený v této učební opoře fungovat! Seznam aktuálně používaných teachpacků lze vidět jednak v menu Language, jednak po stisku tlačítka Execute v okně interakcí. Po ukončení cvičení doporučujeme teachpack z prostředí jazyka odstranit volbou Langauage/Clear Teachpack. Touto volbou odstraníme vazbu mezi interpretrem DrScheme a daným souborem. Soubor však zůstane nadále na disku a lze jej později opět přidat.

1.5.2 Strategie „moc nízko, moc vysoko“

Jednoduchá herní strategie umožňuje najít řešení půlením intervalu.

Při této herní strategii říká první hráč (počítač) druhému hráči, jestli je jeho odhad menší než cíl nebo větší než cíl. To umožňuje hráči postupně upravovat odhad, až se strefí do cíle. Pro implementaci této herní strategie použijeme následující kód v jazyku Scheme. Kód napište do okna definic a hru spusťte stiskem tlačítka Execute.

```
(define (value d0 d1 d2)
  (+ d0 (* 10 (+ d1 (* 10 d2)))))

(define (check-guess d0 d1 d2 target)
  (cond ((< (value d0 d1 d2) target) (string->symbol "Nizko"))
        ((> (value d0 d1 d2) target) (string->symbol "Vysoko"))
        (else (string->symbol "Trefa"))))

(guess-with-gui-3 check-guess)
```

Průvodce studiem

Zkuste si hru několikrát přehrát. Vyzkoušejte si, jak počítač reaguje na jednotlivé vstupy. Dokázali byste navrhnout herní taktiku, která vede nejrychleji k cíli?

Kód vysvětlíme postupně. Pomocná procedura s názvem `value` akceptuje tři číslice `d0`, `d1` a `d2` a vytvoří číslo, složené z těchto číslic. Například:

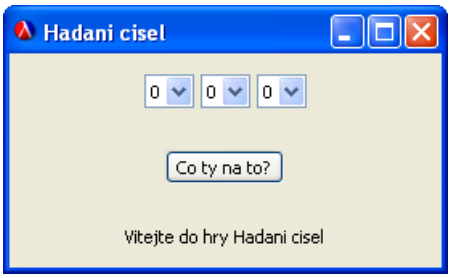
```
> (value 1 2 3)
321

> (value 4 5 6)
654
```

Proceduru používáme k tomu, abychom ze třech číslic, které uživatel zadá pomocí uživatelského rozhraní do prostředí, vytvořili číslo, které můžeme porovnat s cílem.

Druhá procedura s názvem `check-guess` je jádrem celé herní strategie. Tuto proceduru volá systém (reprezentovaný teachpakem) při každém stisku potvrzovacího tlačítka „Co ty nato?“. Vstupem do této procedury jsou číslice `d0`, `d1` a `d2`, které uživatel zadal pomocí stahovacího menu a číslo `target` – cíl, který má uživatel uhodnout. Porovnáním čísla, vzniklého složením číslic, a cíle program určí, je-li odhad příliš nízký, příliš vysoký nebo jedná-li se „trefu“. Vrátí-li procedura symbol `trefa`, hra končí.

Celou hru odstartujeme voláním procedury `guess-with-gui-3`, která jako parametr akceptuje právě proceduru implementující herní strategii.[1] Hrací okno vidíme na obrázku.



Obr. Hrací okno hry hádání čísel

[1] Procedury v jazyku Scheme mohou akceptovat jako parametr i jiné procedury. Takovéto procedury pak označujeme jako *procedury vysokého řádu* (*high order procedures*).

1.5.3 Úprava rozhodovací procedury

Navržená procedura je ideální pro koncového uživatele, není však vhodná pro programátora, který bude upravovat rozhodovací proceduru. Při odlaďování a testování bychom narazili na problém, že neznáme cíl, který je třeba uhodnout. Uchýlíme se proto k drobnému triku, který nám umožní „nahlédnout do karet“ počítači a zobrazí nám cílené číslo, které má uživatel hádat. Po odladění procedury samozřejmě tento kód opět odstraníme. Upravená procedura `check-guess` může vypadat například takto.

Programátor musí vědět, jaké číslo počítač vygeneroval.

```
(define (check-guess d0 d1 d2 target)
  ; For debugging only
  (display "Uzivatel zadal: ")
  (display d2)
  (display d1)
  (display d0)
  (newline)
  (display "Cil je roven: ")
  (display target)
  (newline)
  ; Game logic starts here
  (cond ((< (value d0 d1 d2) target) (string->symbol "Nizko"))
        ((> (value d0 d1 d2) target) (string->symbol "Vysoko"))
        (else (string->symbol "Trefa"))))
```

Spustíme-li hru, uvidíme po stisku potvrzovacího tlačítka v okně interakcí například takovýto výpis:

```
Uzivatel zadal: 000
Cil je roven: 397
```

1.5.4 Strategie „samá voda, přihořívá, hoří“

Prvním úkolem bude změnit herní strategii tak, aby namísto informací „nízko“ a „vysoko“ počítač uživatele **slovně** informoval, jak je od cíle daleko. Uživatel však nebude vědět, kterým směrem se cíl nachází. Pro slovní informaci můžeme například použít tvrzení známé dětské hry; tvrzení si nadefinujeme takto:

Jiné strategie jsou obtížnější pro hráče.

- „Samá voda“ – uživatel je od cíle vzdálen alespoň 100 jednotek,
- „Zima“ – uživatel je od cíle vzdálen 50 až 99 jednotek,
- „Teplo“ – uživatel je od cíle vzdálen 25 až 49 jednotek,
- „Přihořívá“ – uživatel je od cíle vzdálen 1 až 24 jednotek,
- „Trefa“ – („Hoří! “) uživatel uhodnul číslo.

Úkoly k textu

1. Nadefinujte proceduru `distance`, která bude akceptovat dvě čísla a určí jejich vzdálenost. Vzdálenost musí být vždy nezáporná a na pořadí čísel nezáleží. Například:

   ```
   > (distance 100 120)
   20
   > (distance 120 100)
   20
   ```

2. Za použití procedury `distance` nadefinujte novou verzi procedury `check-guess`, která implementuje navrženou herní strategii. Namísto podmíněného příkazu `if` použijte podmíněný příkaz `cond`.

Cvičení

1. Navrhněte sadu výrazů, které podle dané herní strategie úplně otestují funkčnost vaší procedury `check-guess`.

2. Stane se něco, když změníte pořadí podmínek ve vašem programu?

Řešení

1. Při použití metody White-box testování musíme otestovat každou větev procedury. Měli bychom proto vyzkoušet systém například s následujícími vstupy:

```
> (check-guess 0 2 0 500) ; cil je vice nez 100 jednotek vpravo
|Sama voda|
> (check-guess 0 2 9 500) ; cil je vice nez 100 jednotek vlevo
|Sama voda|
> (check-guess 0 2 4 500) ; cil je vice nez 50 jednotek vpravo
Zima
> (check-guess 0 8 5 500) ; cil je vice nez 50 jednotek vlevo
Zima
> (check-guess 0 7 4 500) ; cil je vice nez 25 jednotek vpravo
Teplo
> (check-guess 0 3 5 500) ; cil je vice nez 25 jednotek vlevo
Teplo
> (check-guess 0 9 4 500) ; cil je mene nez 25 jednotek vpravo
Prihoriva
> (check-guess 0 1 5 500) ; cil je mene nez 25 jednotek vlevo
Prihoriva
> (check-guess 0 0 5 500) ; zasah
Trefa
```

2. Pokud jste ve vašem programu použili formu `and` pro testování obou podmínek vzdálenosti, na pořadí pravděpodobně nezáleží. Pokud jste nicméně použili jen jednu podmínku, která se postupně v jednotlivých řádcích formy `cond` uvolňovala, na pořadí záleží. v takovémto případě byste nejprve měli testovat, jestli je číslo rovno cíli, v druhé podmínce jestli je vzdálenost menší jak 25, ve třetí podmínce jestli je vzdálenost menší jak 50 atd.

Průvodce studiem

Testování programů tohoto typu je velmi důležité. Mnohdy se stane, že při chybě v jedné z podmínek nebo při jejich špatném pořadí program funguje pro některé vstupy, ale selhává pro jiné kombinace vstupů.

1.5.5 Strategie „trefa do cifry“

Uživatel má uhodnout jednotlivé cifry cíle. Při této strategii počítač uživateli sdělí, kolik cifer cílového čísla již uhodnul. Uživatel se však nedozví, které z cifer vlastně uhodnul. Je-li cíl například roven 173 a uživatel zadal `d0` = 3, `d1` = 9 a `d2` = 1, uhodl první a poslední cifru. Prostřední cifra však neodpovídá cíli. Počítač proto odpoví „Dvě cifry“. Strategie by tedy vypadala takto:

Namísto porovnávání čísel můžeme porovnávat jednotlivé cifry.

- „Žádná cifra“ – uživatel neuhodnul žádné ze tří cifer,
- „Jedna cifra“ – uživatel uhodnul pouze jednu ze tří cifer,

- „Dvě cifry“ – uživatel uhodnul právě dvě ze tří cifer,
- „Trefa“ – uživatel uhodnul všechny tři cifry.

Pro řešení řídící procedury této herní strategie vzniká potřeba dalších pomocných procedur. Jádrem problému je, že do procedury vstupují čísla `d0`, `d1` a `d2`, které je třeba porovnávat s jednotlivými **ciframi (číslicemi)** čísla `target`. Zatímco v minulých dvou herních strategiích jsme pomocí procedury `value` skládali z číslic výsledné číslo, nyní máme opačný problém. Z čísla `target` potřebujeme zjistit jednotlivé cifry.

Pro řešení tohoto problému použijeme procedury `quotient` a `remainder` jazyka Scheme. První z procedur provádí celočíselné dělení, druhá vrací zbytek po celočíselném dělení. Vytvoříme procedury `digit0`, `digit1` a `digit2`, které budou vracet jednotlivé cifry trojciferného čísla. První ze zmíněných procedur může vypadat takto:

```
(define (digit0 target)
   (remainder target 10))
```

Za pomoci těchto procedur naprogramujte novou verzi procedury `check-guess`, která odpovídá za strategii hry. Schéma procedury ukazuje následující fragment kódu.

```
(define (check-guess d0 d1 d2 target)
   (cond ((= (value d0 d1 d2) target) (string->symbol "Trefa"))
         (( ... ) ...)
         (( ... ) ...)
         (else ...)))
```

Úkoly k textu

3. Naprogramujte procedury `digit1` a `digit2`. Otestujte tyto procedury.
4. Doplňte chybějící místa v proceduře `check-guess`. Pro jednotlivé podmínky použijte predikáty `and` a `or`.

1.5.6 Strategie „uhodnutá cifra“

Strategii lze zobecnit, aby nezáleželo na pořadí cifer.

Předchozí strategii je možno mírně pozměnit tak, že při informování uživatele nebudeme brát ohled na pořadí uhodnutých cifer. Uživatel proto může uhodnout všechny tři cifry, ale jeho odhad stále nebude správný, pokud uvede špatnou permutaci cifer. Odpovědi počítače lze shrnout do těchto bodů:

- „Žádná cifra“ – uživatel neuhodnul žádné ze tří cifer,
- „Jedna cifra“ – uživatel uhodnul pouze jednu ze tří cifer,
- „Dvě cifry“ – uživatel uhodnul právě dvě ze tří cifer,
- „Tři cifry“ – uživatel uhodnul všechny tři cifry, ale umístil je ve špatném pořadí,
- „Trefa“ – uživatel uhodnul cílové číslo.

Pokud bychom chtěli v proceduře `check-guess` vypsat všechny možné kombinace uhodnutých cifer, program by se značně zkomplikoval. Zavedeme proto pomocné procedury, které otestují, zdali se v cíli `t` objevuje jedna cifra, dvě cifry nebo dokonce všechny tři cifry uvedené jako parametry procedury. Následující fragment programu v jazyku Scheme ukazuje implementaci první ze zmíněných procedur a demonstruje použití všech tří procedur.

```
(define (1-digit-in d t)
  (or (= (digit0 t) d)
```

```
      (= (digit1 t) d)
      (= (digit2 t) d)))

> (1-digit-in 4 274)
#t

> (1-digit-in 8 274)
#f

> (2-digit-in 2 4 274)
#t

> (2-digit-in 4 7 274)
#t

> (3-digit-in 4 2 7 274)
#t
```

Úkoly k textu

5. Implementujte procedury `2-digit-in` a `3-digit-in`. Tyto procedury mohou využívat proceduru `1-digit-in`.

6. Implementujte novou logiku hry do další verze procedury `check-guess`.

Shrnutí

V této kapitole jsme se zabývali několika verzemi jednoduché hry na hádání čísel. Implementovali jsme procedury realizující různé hrací plány. Cílem bylo procvičit tvorbu jednoduchých procedur a využití rozsáhlejších podmíněných výrazů.

2 Rekurze

2.1 Rekurzivní procedury

Studijní cíle: Po prostudování kapitoly bude studující schopen navrhnout a implementovat jednoduché rekurzivní procedury. Bude umět identifikovat základní části rekurzivních procedur a vysvětlit mechanismus výpočtu rekurzivních procedur.

Klíčová slova: Rekurzivní procedura, základní případ, rekurzivní případ, mezní podmínka rekurze, rekurzivní předpis, trasování.

Potřebný čas: 4 hodiny.

Rekurzivní volání nahrazuje podporu cyklů.

Na rozdíl od mnoha jiných programovacích jazyků nemá jazyk Scheme zabudovanou silnou podporu cyklů. Úlohu iterativního příkazu zde na sebe přebírá jako jednu z vedlejších úloh podstatně mocnější aparát: systém *rekurzivního volání procedury*.

Tuto kapitolu bychom mohli jednoduše ukončit konstatováním, že s rekurzivními procedurami se v jazyce Scheme pracuje zcela stejně jako s ostatními procedurami. Důvodem, proč věnujeme rekurzivním procedurám tento prostor, je klíčová úloha, kterou hraje rekurze ve funkcionálním programování, a z toho vyplývající snaha objasnit tuto problematiku všem těm, kteří se s rekurzí v programování dosud nesetkali.

2.1.1 Principy rekurze

Faktoriál čísla.

Proceduru nazveme *rekurzivní* tehdy, pokud ve svém těle vyvolává alespoň jednu aktivaci sebe samé. Typickým příkladem rekurzivní procedury může být procedura na výpočet faktoriálu, jejíž matematická definice zní:

$$n! = \begin{cases} 1, n = 0 \\ n.(n-1)!, n > 0 \end{cases}$$

Matematická definice se rozpadá na dva případy: případ, kdy číslo n je rovno nule a případ, kdy n je větší než nula. Tento kód lze téměř mechanicky převést na funkční program v jazyce Scheme.

```
(define (fakt n)
  (if (= n 0) 1
      (* n (fakt (- n 1)))))
```

Rekurzivní procedura má vždy dvě části.

Všimněme si, že tato rekurzivní procedura se skládá ze dvou částí:

- *Základního případu* (*base case*), který obsahuje *limitní podmínku rekurze*. Limitní (mezní) podmínka řeší daný problém pro triviální případ bez použití rekurze. V našem případě jde o konstatování, že 0!=1.
- *Rekurzivního případu* (*recursive case*), který obsahuje *rekurzivní předpis*. Rekurzivní předpis redukuje daný problém na jeden (nebo více) problémů jednodušších. V našem případě jde o konstatování, že n!=n(n-1)!.

Tato struktura je společná pro téměř všechny rekurzivní procedury.[1] Při programování libovolného rekurzivního algoritmu bychom měli vždy začít tím, že si jasně formulujeme limitní podmínku rekurze a předpis rekurze.

Průvodce studiem

V mnoha programovacích jazycích může programátor psát kód bez toho, že by měl zcela ujasněn algoritmus. Kód pak postupně zkouší, upravuje a časem někdy dospěje k uspokojivému výsledku. Tento způsob tvorby softwaru je nicméně trestuhodný. Programovací jazyk Scheme jsme zvolili mimo jiné i proto, že takovýto styl programování zde téměř nikdy nevede k cíli. U rekurzivních procedur to pak platí dvojnásob. Dříve než začneme programovat, musíme mít zcela vyjasněný algoritmus rekurzivního předpisu, který nás postupně dovede až k limitní podmínce rekurze.

Rekurzivní výpočet probíhá ve dvou fázích.

Výpočet popsaný rekurzivní procedurou probíhá vždy ve dvou fázích:

- *fáze navíjení*, kdy postupně aplikujeme předpis rekurze a kdy se problém redukuje na problémy jednodušší; tato fáze končí dosažením limitní podmínky rekurze, kdy lze problém vyřešit bez použití rekurze,
- *fáze odvíjení*, kdy se postupně vrací řízení až do místa první aktivace procedury; v této fázi se mnohdy konstruuje výsledek volání procedury.

Důležitou vlastností rekurzivních procedur je, že neprobíhají v konstantním paměťovém prostoru. Při každé aktivaci procedury mají totiž jednotlivé symboly, použité jako parametry procedury, jinou hodnotu. Původní hodnoty však musí být zachovány, neboť mohou být použity při návratu z rekurzivních volání – ve fázi odvíjení. Při každé aktivaci procedury proto systém vytváří nový prostor, který použije k uschování aktuálních vazeb jednotlivých symbolů. Část paměti vyhrazená tomuto účelu se nazývá *zásobník*.

2.1.2 Trasování výpočtu

Trasování je důležité pro ladění programů.

Odlaďování rekurzivních procedur je mnohdy náročnější než odlaďování procedur, které nepoužívají rekurzi. Proto je dobré vědět, jak lze v prostředí Scheme sledovat průběh výpočtu.

Většina překladačů jazyka dovoluje zapnout sledování libovolné procedury pomocí procedury `trace`. Vypnutí sledování provádíme vyvoláním procedury `untrace`. Zapneme-li například sledování naší procedury `fakt`, získáme po jejím vyvolání takovýto výstup:[2]

```
> (require (lib "trace.ss"))
> (trace fakt)
(fakt)
> (fakt 3)
|(fakt 3)
| (fakt 2)
| |(fakt 1)
| | (fakt 0)
| | 1
```

[1] V systému tzv. *líného vyhodnocování* mohou existovat rekurzivní procedury, které nemají mezní podmínku rekurze a přesto nezpůsobí nekonečný cyklus programu. Datové struktury, které líné vyhodnocování využívají, se nazývají *proudy* a jsou mimo rozsah tohoto textu.

[2] Musíme zároveň k programu přiložit knihovnu, která obsahuje podporu trasování výpočtu. V případě prostředí DrScheme to provádíme příkazem `require`.

```
| |1
| 2
|6
6
```

Systém nás informuje výpisem o každém vyvolání procedury `fakt` (v tomto příkladě byla procedura `fakt` postupně vyvolávána s parametry 3, 2, 1 a 0) a o každém výstupu z procedury `fakt` s uvedením návratové hodnoty (návratové hodnoty byly 1, 1, 2 a 6). Na tomto záznamu je dobře vidět fáze navíjení, dosažení limitní podmínky rekurze a fáze odvíjení. Za pozornost stojí, že fáze navíjení je pouze „přípravná“, výsledek celé procedury je konstruován až v průběhu fáze odvíjení.

2.1.3 Příklad: součet čísel v intervalu

Součet čísel v intervalu.

Rekurzi si dále demonstrujeme na jednoduchém příkladě. Cílem je napsat proceduru, která sečte všechna celá čísla na určeném intervalu. Pokud `a` a `b` jsou dvě celá čísla, výsledkem procedury bude hodnota `a + (a+1) + (a+2) + (a+3) + … + (b-1) + b`.

Průvodce studiem

*Nebudeme předstírat, že tento problém neumíme vyřešit elegantněji. S originální myšlenkou údajně přišel na konci 18. století školák jménem Karl Gauss. Když učitel ve třídě zadal žákům úkol sečíst všechna celá čísla od 1 do 100, domníval se, že má od třídy alespoň na půl hodiny pokoj. Malý Karl ale přišel s jednoduchou myšlenkou: vytvořme tabulku, kde do prvního řádku napíšme všechna čísla od 1 do 100. Do druhého řádku pak pod ně napíšme čísla od 100 do 1. Pod sebou tak budou čísla 1 a 100, 2 a 99, 3 a 98 atd. Součet každého sloupce je přesně 101, sloupců je přitom 100. Výsledek úkolu je tedy (101 * 100) / 2 = 5050.*

My k problému přistoupíme tak jako všichni Karlovi spolužáci: postupně mechanicky sečteme všechna čísla. Tato procedura nám bude sloužit jako vynikající vzor pro řadu dalších, složitějších procedur.

Rekurzivní proceduru implementuje následující kód. Mezní podmínkou rekurze je situace, kdy `a` je větší než `b`. Na takovémto intervalu žádná celá čísla neleží a výsledek je tedy nula. Pokud `a` je menší (nebo rovno) číslu `b`, potom lze problém převést na jednodušší: stačí vzít číslo `a` a přičíst ho k součtu všech čísel na intervalu od `a+1` do `b`. Tím nám vzniká rekurzivní předpis. Všimněme si, že v každém dalším volání procedury se nám parametr `a` zvětšuje o jedničku. Nutně tedy v konečné době dosáhne a převýší parametr `b`, který se nemění.

```
(define (soucet-od-do a b)
  (if (> a b) 0
      (+ a (soucet-od-do (+ a 1) b))))
```

Program můžeme spustit na počítači a můžeme si ověřit jeho běh trasováním výpočtu:

```
> (soucet-od-do 1 100)
5050
> (trace soucet-od-do)
(soucet-od-do)
> (soucet-od-do 1 5)
|(soucet-od-do 1 5)
| (soucet-od-do 2 5)
| |(soucet-od-do 3 5)
| | (soucet-od-do 4 5)
| | |(soucet-od-do 5 5)
| | | (soucet-od-do 6 5)
| | | 0
| | |5
```

```
| | 9
| |12
| 14
|15
15
```

Vytvoříme si nyní dvě modifikace této procedury. V první variantě si pozměníme zadání tak, že budeme sčítat pouze každé druhé číslo na intervalu. V tomto případě stačí jednoduše zkopírovat předchozí kód, změnit název procedury (v hlavičce i v rekurzivním volání) a upravit velikost inkrementu: namísto, abychom v každém kroku přičítali k číslu a hodnotu jedna, budeme přičítat dvojku. Dosažení mezní podmínky rekurze přitom není nijak ohroženo.

Součet čísel na intervalu představuje vzor typické rekurzivní procedury.

```
(define (soucet2-od-do a b)
  (if (> a b) 0
      (+ a (soucet2-od-do (+ a 2) b))))
```

```
> (soucet2-od-do 1 100)
2500
> (trace soucet2-od-do)
(soucet2-od-do)
> (soucet2-od-do 1 5)
|(soucet2-od-do 1 5)
| (soucet2-od-do 3 5)
| |(soucet2-od-do 5 5)
| | (soucet2-od-do 7 5)
| | 0
| |5
| 8
|9
9
```

Průvodce studiem

Při tvorbě mnoha procedur budeme často vycházet z jednoduchého vzoru, který modifikujeme. Příkazy editoru „kopíruj“ a „vlož“ tak patří mezi základní pracovní nástroje každého programátora.

Další modifikací procedury `soucet-od-do` bude procedura na součet všech sudých čísel na intervalu od `a` do `b`. Opět nejprve zkopírujeme původní proceduru a změníme její název. V tomto případě ještě formálně přepíšeme příkaz `if` na `cond` – za chvíli budeme potřebovat přidat další podmínku, aby procedura sčítala pouze sudá čísla. Přepsaný program vypadá takto (pozor, jedná se stále o proceduru `soucet-od-do`, změněné je jen jméno a forma zápisu):

Rekurzivní procedura může mít více rekurzivních předpisů.

```
(define (soucet-sude a b)
  (cond ((> a b) 0)
        (else (+ a (soucet-sude (+ a 1) b)))))
```

Chceme-li sčítat pouze sudá čísla, potřebujeme rozlišit mezi dvěma případy: když je číslo `a` sudé (můžeme použít zabudovaný predikát `even?`) a když není. V prvním případě chceme číslo přičíst k výsledku tak, jak jsme to dělali dosud ve větvi `else`, v druhém případě číslo přičíst nechceme a rádi bychom pokračovali ve sčítání dalších čísel od `a+1` do `b`.

```
(define (soucet-sude a b)
  (cond ((> a b) 0)
        ((even? a) (+ a (soucet-sude (+ a 1) b)))
        (else (soucet-sude (+ a 1) b))))
```

Tato rekurzivní procedura má dva rekurzivní předpisy. Který z nich se zvolí záleží na tom, jakou hodnotu má číslo `a`. Je-li to číslo sudé, přičte se k výsledku, jinak se výpočet jednoduše převede na další případ.

Průvodce studiem

Při tvorbě rekurzivních procedur se nám často stane, že nechtíc vytvoříme tzv. nekonečný cyklus. To se projeví tak, že výpočet běží podezřele dlouho a někdy také ubývá paměti, takže počítač se může dostat do problémů. Výpočet v takovémto případě ukončíme stiskem tlačítka „Break" a v programu musíme nalézt chybu. Uvědomme si, že cyklus může vznikat pouze tak, že procedura rekurzivně zavolá samu sebe. Chyba tedy musí být v parametrech rekurzivního volání nebo v mezní podmínce rekurze.

Parametry rekurzivního volání musí přibližovat výpočet mezní podmínce rekurze. Je-li mezní podmínkou test na `n=0`*, musí se parametr* `n` *v každém volání snižovat. Je-li mezní podmínkou test na* `a>b`*, musí se buď* `a` *zvětšovat nebo* `b` *zmenšovat.*

Shrnutí

Rekurzivní procedura ve svém těle volá samu sebe. Každá rekurzivní procedura má dvě základní části: mezní podmínku rekurze a rekurzivní předpis. Rekurzivní předpis obsahuje rekurzivní volání procedury, parametry volání se přitom musí změnit tak, aby bylo nové volání bližší mezní podmínce rekurze. V opačném případě skončíme v nekonečném cyklu. Průběh rekurzivního výpočtu můžeme monitorovat trasováním.

Pojmy k zapamatování

- Rekurzivní procedura,
- základní případ, mezní podmínka rekurze,
- rekurzivní případ, rekurzivní předpis,
- trasování výpočtu.

Kontrolní otázky

1. *Kdy skončí volání rekurzivní procedury nekonečným cyklem?*
2. *Co je to zásobník programu a jaká data tam jsou ukládána?*
3. *Jaké dvě základní fáze má rekurzivní výpočet?*

Cvičení

1. Napište proceduru `pocet-cifer`, která spočte počet cifer kladného čísla zapsaného v desítkové soustavě. V programu můžete použít zabudovanou proceduru `quotient`, která provádí celočíselné dělení dvou čísel. Například:

```
> (pocet-cifer 528)
3
> (pocet-cifer 4)
1
> (pocet-cifer 48426)
5
```

2. Harmonický součet n prvků je definován vzorcem:

$$1+\frac{1}{2}+\frac{1}{3}+\frac{1}{4}+\frac{1}{5}+\frac{1}{6}+\frac{1}{7}+\frac{1}{8}+...+\frac{1}{n}$$

   Napište proceduru `harmonic`, která pro dané číslo `n` spočte tento výraz. Například:

```
>(harmonic 5)
2.2833
```

Úkoly k textu

1. Napište proceduru `generuj-cifry`, která akceptuje číslo `n` a vygeneruje náhodné číslo o `n` cifrách. Výsledné číslo generujte po jednotlivých cifrách, pro generování každé cifry použijte volání procedury `(random 10)`. Například:

   ```
   > (generuj-cifry 0)
   0
   > (generuj-cifry 10)
   4731068401      ; například
   ```

2. Ludolfovo číslo π lze podle Leibnize odhadnout jako součet této nekonečné řady:
   $$\frac{\pi}{8} = \frac{1}{1\cdot 3} + \frac{1}{5\cdot 7} + \frac{1}{9\cdot 11} + \frac{1}{13\cdot 15} + \ldots$$

   Napište proceduru `odhad-pi`, která akceptuje číslo `n` a odhadne π součtem `n` členů Leibnizovy řady.

3. Obdobně lze odhadnout číslo π součtem následující řady.
   $$\frac{\pi}{4} = 1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \frac{1}{11} + \frac{1}{13} + \ldots$$

   Napište proceduru `odhad-pi2`, která využije tento vzorec.

4. Wallisova formule tvrdí, že číslo π lze také spočítat jako součin následující řady:
   $$\frac{\pi}{4} = \frac{2\cdot 4}{3\cdot 3}\cdot\frac{4\cdot 6}{5\cdot 5}\cdot\frac{6\cdot 8}{7\cdot 7}\cdot\frac{8\cdot 10}{9\cdot 9}\cdot\ldots$$

   Napište proceduru `odhad-pi3`, která využije tento vzorec.

Řešení

1. Vydělíme-li celočíselně číslo desítkou, ztratíme vždy poslední cifru. Například:
   ```
   (quotient 6933 10)
   693
   (quotient 693 10)
   69
   (quotient 69 10)
   6
   ```
   O kladném čísle, které je menší než 10, můžeme jednoznačně říci, že má jedinou cifru. Máme tak k dispozici jak rekurzivní předpis, tak mezní podmínku rekurze a můžeme zapsat kód:

   ```
   (define (pocet-cifer n)
     (if (< n 10) 1
         (+ 1 (pocet-cifer (quotient n 10)))))
   ```

2. Nejprve si musíme položit otázku, z které strany budeme daný výraz počítat. Pokud budeme počítat zleva doprava, budeme postupně zvyšovat číslo ve jmenovateli a výpočet ukončíme mezní podmínkou rekurze, když jmenovatel překročí číslo `n`. Rozhodneme-li se počítat prvky posloupnosti zprava doleva, musíme číslo `n` postupně snižovat, až dosáhneme hodnoty jedna. Druhá metoda je v tomto případě poněkud jednodušší:

```
(define (harmonic n)
  (if (= n 1) 1.0
      (+ (/ 1 n) (harmonic (- n 1)))))
```

2.2 Projekt: Lissajousovy křivky

Studijní cíle: Studující si procvičí základy práce s prostředím jazyka Scheme a základy práce s rekurzivními procedurami. Naučí se upravovat mezní podmínku rekurze.

Klíčová slova: Lissajousovy křivky, mezní podmínka rekurze, rekurzivní předpis.

Potřebný čas: 2 hodiny.

2.2.1 Teorie Lissajousových křivek

Lissajousovy obrazce vykresluje bod, který se pohybuje harmonickým pohybem současně ve dvou na sebe kolmých směrech.

Lissajousovy křivky vykresluje bod pohybující se ve dvou směrech současně.

Harmonickým pohybem by se například pohybovala kulička, zavěšená na pružině, pokud bychom ovšem neuvažovali postupné tlumení kmitů způsobené odporem pružiny. Harmonický pohyb v čase je popsán goniometrickou funkcí sinus. V čase 0 je tak kulička v rovnovážné poloze, v čase $\pi/2$ je v horním úvratí, v čase π opět prochází rovnovážnou polohou a v čase $3\pi/2$ dosahuje dolního úvratí.

Pokud se bod harmonicky pohybuje ve dvou na sobě kolmých směrech současně, opisuje tak zvané Lissajousovy křivky. Tvar křivky přitom záleží na třech faktorech:

Tvar křivky je dán poměrem frekvencí a fázovým posunem.

- frekvenci, s jakou se bod pohybuje v jednom směru, tuto frekvenci označíme jako f_x,
- frekvenci, s jakou se bod pohybuje v druhém směru, tuto frekvenci označíme jako f_y,
- fázovém posunu, který vyjadřuje, jak se pohyb v jednom směru „předbíhá“ oproti pohybu v druhém směru, fázový posun označíme jako ϕ.

Lissajousovy obrazce byly poprvé popsány francouzským fyzikem 19. století Julesem Lissajousem.

Obr. 5 Jules Antoine Lissajous

Průvodce studiem

Představte si, že se bod pohybuje harmonicky směrem zleva-doprava a stejně rychle ve směru nahoru-dolů. Obě frekvence jsou tedy stejné. Dále si představme, že pohyb „odstartujeme" z rovnovážné polohy (polohy „uprostřed"), fázový posun je tedy nulový. Bod se začne pohybovat současně doprava a nahoru, po jisté době dosáhne v obou směrech maxima a začne se pohybovat zpět. Na stínítku nám tak vykreslí diagonální úsečku.

Pro změnu předpokládejme, že pohyb bodu „odstartujeme" tak, že ve směru horizontálním bude bod v nulové, rovnovážné poloze, ale ve směru vertikálním bod „zatlačíme" do horní polohy. Rozdíl mezi fázemi obou pohybů tak bude roven čtvrtině cyklu, tedy hodnotě polovina π. Bod se začne pohybovat od středu doprava a současně z horní polohy směrem dolů. Když ve vodorovném směru dorazí zcela vpravo, bude ve svislém směru uprostřed stínítka. Postupně nám tak vykreslí na obrazovce kružnici.

Souřadnice bodu na stínítku lze matematicky vyjádřit pomocí rovnic:

$$x(t) = \sin(f_x t) \text{ a } y(t) = \sin(f_y t + \phi)$$

Parametr t přitom představuje čas. Vzhledem k periodicitě funkce sinus z toho plyne, že když t dosáhne hodnoty 2π, bude křivka, kterou bod opisuje, zcela vykreslena.

2.2.2 Vykreslování křivek na obrazovce

Za využití grafické knihovny lze pohyb bodu na obrazovce popsat následujícím kódem. Před spuštěním kódu je třeba přidat knihovnu `canvas` jako teachpack do jazyka Scheme (viz kapitola 1.5, přidání teachpacku). Kód napište do okna definic.[1]

```
(define (lissajous-loop t fx fy phi)
  (graphics-line-to (sin (* fx t))
                    (sin (+ (* fy t) phi)))
  (lissajous-loop (+ t 0.0001) fx fy phi))

(define (lissajous fx fy phi)
  (graphics-init 500 500 -1 -1 1 1)
```

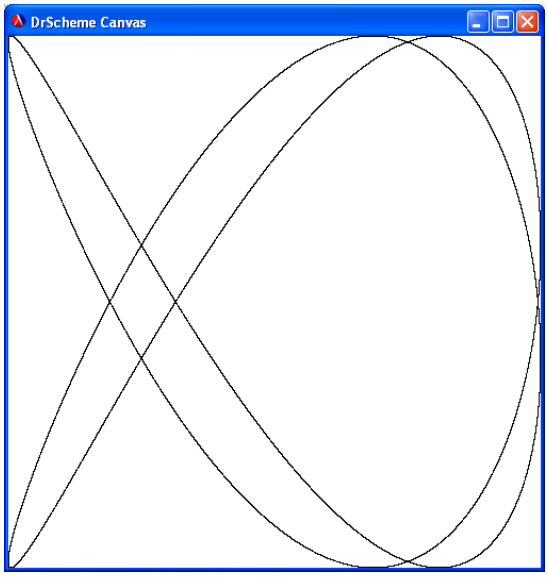
[1] Tyto procedury pracují pomocí tzv. vedlejšího efektu. Rozdíl mezi hlavním efektem a vedlejším efektem procedury vysvětlíme podrobněji v kapitole 2.3.1. Dále se zde využívá fakt, že tělo procedury může obsahovat více než jeden výraz. V tuto chvíli přistoupíme k těmto faktům intuitivně a nebudeme jim připisovat výrazný význam.

```
(graphics-move-to 0 (sin phi))
(lissajous-loop 0 fx fy phi))
```

Procedura `lissajous` akceptuje tři parametry: hodnoty frekvencí f_x a f_y a fázový posun ϕ. Procedura běží v nekonečném cyklu, pro zastavení budete muset použít tlačítko „Break“. Spusťte proceduru lissajous například s těmito parametry:

```
> (lissajous 2 3 1)
```

Na obrazovce byste měli vidět následující obrazec.



Obr. 6 Lissajousova křivka

2.2.3 Experimenty s Lissajousovými křivkami

Průvodce studiem

I když v tuto chvíli uvedenému programu plně nerozumíte, experimentujte chvíli s Lissajousovými obrazci. Tyto experimenty vám umožní získat představu o různých druzích obrazců a pomohou vám v řešení následujících úkolů.

Experimentujte s Lissajousovými křivkami. Vyzkoušejte následující kombinace parametrů.

Program běží v nekonečném cyklu.

fx	fy	phi
1	1	0
1	1	0.1
1	2	0
1	2	0.2
2	3	0
2	3	0.3

3	4	0
3	4	0.4
4	5	0
4	5	0.5
5	6	0
7	8	0
3	5	0

2.2.4 Zastavení nekonečného cyklu

Pokud jsou frekvence f_x a f_y celá kladná čísla, bude celá Lissajousova křivka vykreslena v *čase* t = 2π. Naším cílem bude ověřit toto tvrzení a upravit program tak, aby se nekonečný cyklus zastavil, když čas t ≥ 2π.

Průvodce studiem

Právě vyřčené tvrzení nemusí čtenáři v tuto chvíli dávat mnoho smyslu. Můžeme se ptát co je to „čas", jakou souvislost má čas s číslem π a jak můžeme zastavit nekonečný cyklus. Kód navíc může obsahovat neznámé konstrukce a volání zcela neznámých knihovních procedur. V tomto cvičení si ukážeme, že je možné pracovat i s neznámým programem a využívat konstrukce, kterým zcela nerozumíme.

V následujících odstavcích se pokusíme analyzovat problém po částech. Úkoly k textu pak shrnují kroky, které je potřeba provést.

Nekonečný cyklus zastavíme pomocí podmíněného příkazu.

Na problém je možno se podívat z pohledu programátorského nebo z pohledu fyzikálního. Začněme zcela mechanickým pohledem programátora analyzujícího neznámý počítačový kód.

Jako první krok je třeba zjistit, co v programu způsobuje nekonečný cyklus. Jak víme, jediný mechanismus, který v jazyku Scheme umožňuje opakované vykonávání programu[1], je rekurze, tedy situace, kdy procedura volá samu sebe. Podíváme-li se na dvě výše uvedené procedury, snadno zjistíme, která z nich je rekurzivní.

Každá rekurzivní procedura by měla mít dvě části: rekurzivní případ (volání sama sebe) a základní případ, kdy lze výsledek určit bez použití rekurze. v našem kódu zcela chybí základní případ. Kdykoli zavoláme proceduru `lissajous-loop`, procedura vyvolá samu sebe. Neexistuje zde žádný mechanismus, který by cyklus ukončil.

Dále si můžeme si všimnout, že při každém novém vyvolání procedury `lissajous-loop` se zvýší hodnota parametru `t` o malý přírůstek. Pokud zadání hovoří o zastavení v čase t ≥ 2π, bude se pravděpodobně jednat o tento parametr. Do programu je tudíž třeba přidat podmínku, která omezí rekurzivní volání pouze na situace, kdy t ≤ 2π, a efektivně tím zastaví nekonečný cyklus.

Z pohledu fyzikálního se můžeme zamyslet, proč parametr `t` označujeme jako čas. Lissajousova křivka je výsledek fyzikálního experimentu, který probíhá ve spojitém čase. Bod, který se pohybuje harmonicky ve dvou směrech, křivku vykreslí za jistý čas. V každém okamžiku se při-

[1] Jediný mechanismus, který byl zde uveden a který používáme.

tom bod nachází v jiném místě roviny. Počítač nám nicméně neumožňuje pracovat se spojitými veličinami. Řešením je v tomto případě tabelovat čas – „rozsekat“ čas na malé intervaly, na začátku každého z nich zjistit aktuální pozici bodu v rovině a tyto pozice na obrazovce spojit pomocí úseček. Dosáhne-li čas hodnoty 2π, což je perioda funkce sinus, musí být křivka plně vykreslena a bod opakovaně opisuje stejnou dráhu.

Úkoly k textu

1. Přidejte na začátek procedury `lissajous-loop` kód
```
(display t)
(newline)
```
a ověřte platnost tvrzení, že celá křivka bude vykreslena v čase 2π.

2. Upravte proceduru tak, aby se cyklus zastavil, pokud hodnota `t` přesáhne 2π.

Cvičení

1. Co přestavuje v programu hodnota `0.0001`? Co se stane, když tuto hodnotu snížíme například na 0.000001 nebo naopak zvýšíme na 0.1? Co když namísto této hodnoty uvedeme například hodnotu 10? Proveďte příslušné experimenty s kódem.

Řešení

1. Tato hodnota představuje velikost tabelace času. Čím menší je příslušný zlomek, tím přesněji by teoreticky měla být křivka na obrazovce vykreslena. Zvolíme-li tuto hodnotu v řádu desetin, jasně uvidíme že na obrazovce vykreslujeme lomenou čáru. Při velmi malých hodnotách nicméně snadno narazíme na zobrazovací přesnost počítače. Další snižování pak již nevede ke zvětšení přesnosti zobrazení, dosáhneme tak ale zpomalení vykreslování křivky. Při velkých hodnotách tabelace (v řádu jednotek nebo desítek) již vykreslovaný obrazec vůbec neodpovídá trajektorii bodu.

2.2.5 Zastavení překreslování křivek

Při používání modifikované verze procedury z kapitoly 2.2.4 zjistíme, že v některých případech se program zastaví ihned po vykreslení křivky zatímco jindy je program překreslí křivku více než jednou. Například při zadání `fx` = 5, `fy` = 7 a `phi` = 1 je křivka vykreslena jednou, zatímco při volbě `fx` = 6, `fy` = 8 a `phi` = 1 je křivka vykreslena dvakrát. V našich úvahách se přitom omezíme na situace, kdy frekvence jsou celá kladná čísla.

Některé křivky se vykreslí více než jednou přes sebe.

Průvodce studiem

Nejprve musíme navrhnout způsob jak určit, kolikrát byla křivka překreslena. Nejjednodušší možností je použít stopky a změřit, jak dlouho trvá počítači vykreslení celé křivky a porovnat tento čas s dobou běhu výpočtu. Pokud vykreslení křivky na vašem počítači trvá 22 sekund a program se zastaví po zhruba 44 sekundách, křivka je zřejmě vykreslena dvakrát. Je-li vykreslování na vašem počítači příliš rychlé nebo příliš pomalé a stopování doby běhu výpočtu je tak obtížné, upravte rychlost vykreslování tak, jak to bylo popsáno ve cvičení v kapitole 2.2.4.

Experimentujte se systémem a postupně vyplňte následující tabulku. Na základě provedeného pozorování se pak ve cvičení pokusíme nalézt zákonitost mezi počtem překreslení křivky a vstupními parametry procedury.

`fx`	`fy`	`phi`	Počet překreslení
3	5	1	
2	7	0	
5	7	1	
2	4	0	
6	8	0	
6	8	2	
6	9	0	
5	15	3	

Cvičení

2. Závisí počet překreslení na hodnotě fázového posunu `phi`?

3. Závisí počet překreslení na hodnotách obou frekvencí `fx` a `fy`?

4. Co se stane, když jsou obě frekvence `fx` a `fy` prvočísla?

Řešení

2. Nezávisí. Například pro `fx` = 6 a `fy` = 8 byl počet překreslení stejný pro obě testované hodnoty `phi`. Veličinu `phi` můžeme z našich úvah o zcela vypustit.

3. Ano, závisí. Změníme-li jednu z frekvencí, počet překreslení se může změnit. Například pro frekvence `fx` = 2 a `fy` = 7 byl počet překreslení roven jedné, zatímco u kombinace `fx` = 2 a `fy` = 8 byla křivka překreslena dvakrát.

4. Jsou-li obě frekvence prvočísla, je počet překreslení vždy roven jedné. Viděli jsme to na prvních třech experimentech.

Úkoly k textu

3. Odvoďte vztah mezi frekvencemi `fx` a `fy` a počtem překreslení křivky.

4. Upravte proceduru `lissajous-loop` tak, aby byla křivka vykreslena vždy jen jednou. Využijte přitom zabudovanou proceduru `gcd`, která počítá největšího společného dělitele dvou čísel.

Shrnutí

Lissajousovy křivky představují jednoduchý a graficky názorný model, s jehož pomocí jsme procvičovali analýzu problému, návrh řešení a implementaci v konkrétním programovacím jazyce. Analýzou neznámého kódu jsme zjistili, která část programu je zodpovědná za nekonečný cyklus a zavedením podmíněného příkazu jsme nekonečný cyklus odstranili. Dále jsme pomocí případové studie odhalili vztah mezi frekvencemi a počtem překreslování křivky a modifikovali jsme mezní podmínky rekurze tak, aby byla křivka vykreslena vždy jen jednou.

2.3 Vedlejší efekt procedur a želví grafika

Studijní cíle: Studující se naučí rozlišovat mezi hlavním a vedlejším efektem procedur. Pochopí princip želví grafiky a procvičí si použití želví grafiky v jednoduchých procedurách.

Klíčová slova: Želví grafika, hlavní a vedlejší efekt procedur, speciální forma begin.

Potřebný čas: 2 hodiny.

2.3.1 Hlavní a vedlejší efekt procedury

Většina doposud uvedených procedur pracovala prostřednictvím svého *hlavního efektu* (*main effect*). Hlavní efekt procedury představuje její výsledek, její návratovou hodnotu, kterou je možno dále využít ve výpočtech. Návratovou hodnotu procedury je například možno uložit do proměnné nebo ji použít jako vstup do jiné procedury. Vezmeme-li například proceduru `soucet-od-do`-do z kapitoly 2.1.3, můžeme snadno spočítat součet čísel od 1 do 5 nebo součet čísel od 10 do 15. Dále můžeme sečíst takto získané hodnoty.

```
> (soucet-od-do 1 5)
15
> (soucet-od-do 11 15)
65
> (+ (soucet-od-do 1 5) (soucet-od-do 10 15))
80
```

Procedura může mít hlavní i vedlejší efekt.

Tato procedura pracovala pomocí svého hlavního efektu.[1] Kromě svého hlavního efektu může mít procedura i *vedlejší efekt* (*side effect*). Jako svůj vedlejší efekt může procedura například vypsat zprávu na obrazovku, vykreslit obrázek, zapsat soubor na disk a podobně. Programátor by měl rozlišovat mezi hlavním a vedlejším efektem procedur a snažit se omezit použití vedlejšího efektu na nejnutnější situace. Nejběžnější případ použití vedlejšího efektu je pak obecně realizace vstupů a výstupů.

Upravme kód procedury `soucet-od-do` tak, že na její začátek přidáme příkazy `display` a `newline`. Procedura `display` akceptuje jeden parametr a vypíše jej na obrazovku. Tato procedura pracuje svým vedlejším efektem, nezajímá nás proto, co tato procedura vrací, a výsledek této procedury nevyužíváme v dalších výpočtech. Procedura `newline` vypíše na obrazovku znak nového řádku, takže další výpisy začínají opět od levého kraje okna.

```
(define (soucet-od-do a b)
  (display "Scitam cislo ")
  (display a)
  (newline)
  (if (> a b) 0
      (+ a (soucet-od-do (+ a 1) b))))
```

[1] Hlavní efekt procedury v jazyce Scheme je jednoduše dán výsledkem vyhodnocení (posledního výrazu) těla procedury. V jiných programovacích jazycích bývá zvýrazněn klíčovým slovem **`return`**.

Pokud opět vyvoláme takto upravenou proceduru, uvidíme její hlavní i vedlejší efekt.

```
> (soucet-od-do 1 5)
Scitam cislo 1
Scitam cislo 2
Scitam cislo 3
Scitam cislo 4
Scitam cislo 5
Scitam cislo 6
15
```

Hlavní efekt je návratová hodnota `15` zatímco vedlejším efektem jsou výpisy v okně prostředí interakcí jazyka Scheme. Podívejme se na efekt vyhodnocení tohoto výrazu:

```
> (+ (soucet-od-do 1 5) (soucet-od-do 11 15))
Scitam cislo 1
Scitam cislo 2
Scitam cislo 3
Scitam cislo 4
Scitam cislo 5
Scitam cislo 6
Scitam cislo 11
Scitam cislo 12
Scitam cislo 13
Scitam cislo 14
Scitam cislo 15
Scitam cislo 16
80
```

Vedlejší efekt je využíván hlavně pro výstup.

Zároveň jsme v tomto příkladu zjistili, že tělo procedury může obsahovat více než jeden výraz jazyka Scheme. Pokud takováto situace nastane, jsou jednotlivé výrazy vyhodnoceny postupně. Hlavní efekt procedury je dán jako výsledek vyhodnocení posledního výrazu. Všechny ostatní výrazy mají význam jen pro svůj vedlejší efekt. To bývá zdrojem častých chyb začátečníků, kteří uvádějí do těla procedury výrazy bez vedlejšího efektu. Tento nefunkční kód pro výpočet součtu druhých mocnin je typický příklad chybného použití.

```
(define (soucet-ctvercu x y)
  (sqr x)
  (sqr y)
  (+ x y))
```

První dva výrazy v těle procedury sice počítají hodnotu druhé mocniny `x` a `y`, ale tyto výsledné hodnoty nejsou nikde uchovány a nejsou použity pro další výpočet. Správná verze je proto tato:

```
(define (soucet-ctvercu x y)
  (+ (sqr x) (sqr y)))
```

Průvodce studiem

I když to vypadá, že hlavní a vedlejší efekt procedury spolu příliš nesouvisí, může to být v praxi často jinak. Procedura může například jako vedlejší efekt otevřít soubor pro čtení a jako svůj hlavní efekt vrátit deskriptor otevřeného souboru nebo chybový kód.

V našich příkladech budeme nicméně vedlejší efekt využívat střídmě a kombinaci hlavního a vedlejšího efektu nebudeme podporovat.

2.3.2 Speciální forma `begin`

`begin` umožňuje seskupit několik příkazů do jednoho bloku.

Pozornému čtenáři na předchozím záznamu interakce s prostředím jazyka může vadit jeden detail. Při sčítání čísel od 1 do 5 se jako poslední text vytištěný vedlejším efektem objevil řetězec "`Scitam cislo 6`". Protože 6 je větší než 5, jednalo se o dosažení mezní podmínky

rekurze a číslo 6 se ve skutečnosti **nepřičetlo** k výsledku. Jako podstatně vhodnější by se jevil například takovýto výpis:

```
> (soucet-od-do 1 5)
Scitam cislo 1
Scitam cislo 2
Scitam cislo 3
Scitam cislo 4
Scitam cislo 5
Hotovo
15
```

To znamená zásadnější zásah do programu. Zřejmě budeme chtít začít opět podmíněným výrazem s tím, že pokud je splněna mezní podmínka rekurze, budeme chtít provést dvě věci: zobrazit slovo „Hotovo“ a vrátit hodnotu 0. Pokud podmínka splněna nebude, budeme chtít provést dokonce čtyři věci najednou: zobrazit text, zobrazit číslo, odřádkovat a rekurzivně zavolat proceduru.

Pro spojování několika příkazů do jediného bloku můžeme použít speciální formu `begin`. Tato forma akceptuje libovolné množství výrazů a postupně je vyhodnocuje. Návratovou hodnotou (hlavním efektem) je potom výsledek vyhodnocení posledního výrazu. Upravený kód může vypadat takto:

```
(define (soucet-od-do a b)
  (if (> a b)
      (begin
        (display "Hotovo")
        0)
      (begin
        (display "Scitam cislo ")
        (display a)
        (newline)
        (+ a (soucet-od-do (+ a 1) b)))))
```

Průvodce studiem

I zde je třeba myslet na to, že všechny výrazy formy `begin` až na poslední mají význam pouze pro svůj vedlejší efekt. Častou chybou je pak snaha o použití formy `begin` v situaci, kdy žádný vedlejší efekt nepotřebujeme.

2.3.3 Úvod do želví grafiky

Želví grafika je založena na jednoduché analogii.

Princip želví grafiky vychází z jednoduchého modelu. Představme si pískoviště plné jemného a pečlivě uhrabaného písku. Do tohoto pískoviště položíme želvu. Jak se želva pohybuje v pískovišti, zanechává za sebou v písku stopu. Pokud bychom mohli želvě dávat příkazy a ovlivňovat tak její pohyb, mohli bychom v písku kreslit obrazce.

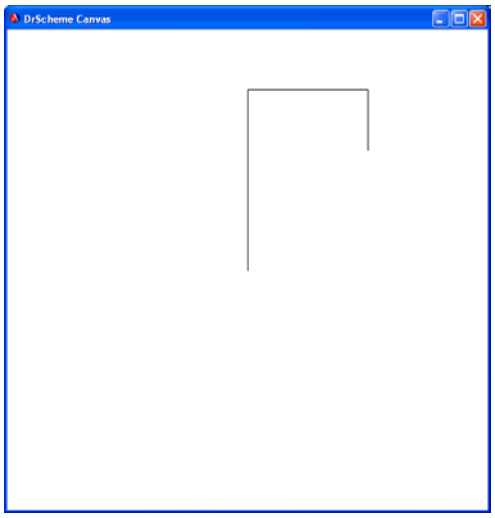
Balík želví geometrie je součástí teachpacku `turtle.scm`. Tento teachpack nám dává tyto základní příkazy pro práci s želví grafikou:

- `init-playground` – inicializace grafického okna, „pískoviště“ želví grafiky, parametrem je poloměr okna,
- `forward` – posune imaginární želvu vpřed, želva zanechá v písku stopu, parametrem je vzdálenost, o kterou se má želva posunout,
- `left` – otočí imaginární želvu vlevo, parametrem je počet stupňů,
- `right` – otočí imaginární želvu vlevo, parametrem je počet stupňů.

Jednoduchý obrázek „šibenice“ pak pomocí želví geometrie můžeme vyrobit například tímto kódem. Výsledek vidíme na následujícím obrázku.

```
(define (sibenice)
  (forward 3)
  (right 90)
  (forward 2)
  (right 90)
  (forward 1))

(init-playground 4)
(sibenice)
```



Obr. 7 Želví grafika

Průvodce studiem

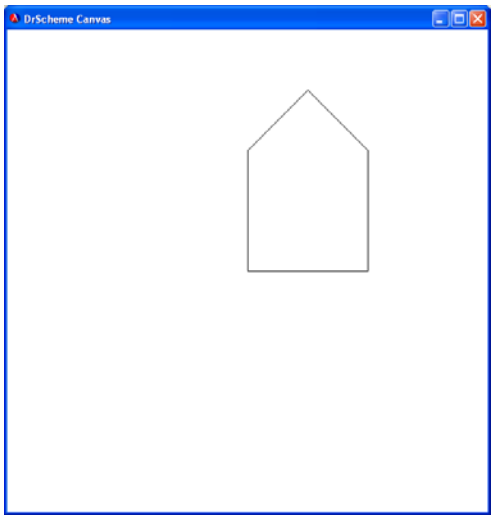
Při psaní programů pro želví grafiku je potřeba mít na mysli, že veškeré příkazy jsou relativní vzhledem k aktuální pozici a aktuálnímu natočení želvy. Tato zdánlivá nevýhoda bude mít velký význam při vytváření složitějších rekurzivních obrazců.

2.3.4 Jednoduché obrázky

Pomocí želví grafiky lze snadno kreslit jednoduché obrázky.

Následující dva obrázky ukazují jednoduché použití želví grafiky. První obrázek představuje dům. Byl vytvořen voláním procedury `dum`, velikost pískoviště byla nastavena na 2. Inicializaci pískoviště přitom ponecháváme mimo kreslící proceduru.

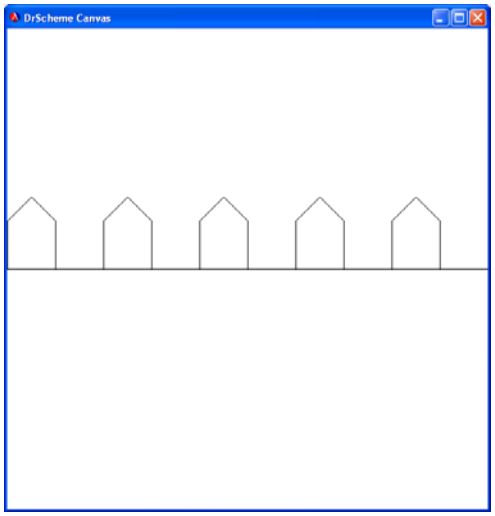
```
> (init-playground 2)
> (dum)
```



Obr. 8 Dům

Druhý obrázek představuje ulici o `n` domech. Velikost pískoviště je nastavena na `n`. Obrázek byl vytvořen touto sekvencí příkazů:

```
> (init-playground 5)
> (set-turtle -5 0 0)
> (ulice 5)
```



Obr. 9 Ulice

2.3.5 Příkazy želví grafiky

Uveďme ještě referenční manuál příkazů želví grafiky:

Základní příkazy želví grafiky.

- `init-playground` – inicializace grafického okna, „pískoviště“ želví grafiky, parametrem je poloměr okna, želva se nachází uprostřed okna a směřuje na sever,
- `erase-playground` – smazání obsahu okna, pozice želvy se nemění,
- `close-playground` – uzavření grafického okna,

- `forward` – posune imaginární želvu vpřed, želva zanechá v písku stopu, parametrem je vzdálenost o kterou se má želva posunout, nepovinným parametrem je barva stopy želvy,
- `backward` – posune imaginární želvu vzad, želva zanechá v písku stopu, parametrem je vzdálenost o kterou se má želva posunout, nepovinným parametrem je barva stopy želvy,
- `left` – otočí imaginární želvu vlevo, parametrem je počet stupňů,
- `right` – otočí imaginární želvu vlevo, parametrem je počet stupňů,
- `set-turtle` – akceptuje pozici x, y a úhel, nastaví želvu na danou pozici,
- `turtle-x` – zjistí aktuální x-souřadnici želvy, 0 je uprostřed pískoviště, záporné hodnoty jsou vlevo, kladné vpravo,
- `turtle-y` – zjistí aktuální y-souřadnici želvy, 0 je uprostřed pískoviště, záporné hodnoty jsou dole, kladné nahoře,
- `turtle-angle` – zjistí aktuální směr želvy, 0 je sever, 90 je východ, 180 jih, 270 západ,
- `playground-size` – zjistí aktuální poloměr velikosti pískoviště.

Shrnutí

Návratová hodnota volání procedury se nazývá hlavní efekt. Jakékoli jiné působení procedury na okolí se nazývá vedlejší efekt. Vedlejší efekt se nejčastěji používá pro realizaci vstupů a výstupů. Tělo procedury může obsahovat více výrazů. Potřebujeme-li více výrazů spojit do jednoho bloku, použijeme speciální formu `begin`. Želví grafika představuje jednoduchý mechanismus pro kreslení. Želvu navigujeme po kreslící ploše příkazy `forward`, `left` a `right`.

Pojmy k zapamatování

- Hlavní a vedlejší efekt procedury,
- speciální forma `begin`,
- želví grafika.

Kontrolní otázky

1. *Jaký je rozdíl mezi hlavním a vedlejším efektem procedury?*
2. *Kdy je nutno použít vedlejší efekt?*
3. *Jaké jsou základní příkazy želví grafiky?*

Cvičení

1. Co je špatného na následujícím kódu:
```
(+ (display "Hello") (display "World"))
```

2. Napište proceduru `chaos` simulující náhodný pohyb želvy po obrazovce. Procedura by měla pracovat v nekonečném cyklu. V každém kroku by měla posunout želvu vpřed o náhodnou vzdálenost a otočit o náhodný úhel. Využijte zabudovanou proceduru `random`, která akceptuje číslo n a vrací pseudo-náhodné celé číslo v rozmezí do 0 do n-1.

3. Želva v našem systému se pohybuje velmi rychle. Ve skutečnosti nejsme schopni její pohyb ani zachytit, obrázek se jednoduše objeví na obrazovce. Navrhněte proceduru `slow-forward`, která by pracovala stejně jako procedura `forward`, ale posouvala želvu po obrazovce pomalu. Proceduru pak otestujte v procedurách `dum` a `ulice`.

Úkoly k textu

1. Implementujte proceduru `dum`. Procedura musí pracovat tak, jak ukazuje obrázek Obr. 8.

2. Implementujte proceduru `ulice`. Procedura musí pracovat tak, jak ukazuje obrázek Obr. 9. Procedura musí akceptovat jako parametr počet domů v ulici a musí používat (volat) proceduru `dum` pro vykreslování jednotlivých domů.

3. Upravte proceduru `chaos` tak, aby se želva nemohla dostat mimo obrazovku. Využijte procedury `turtle-x`, `turtle-y` a `set-turtle`.

Řešení

1. Procedura `display` pracuje svým vedlejším efektem, její hlavní efekt je nedefinovaný. Nemůžeme proto sčítat výsledky volání těchto procedur.

2. Procedura může vypadat například takto:

```
(define (chaos)
  (forward (random 10))
  (left (random 180))
  (chaos))
```

3. Při řešení tohoto problému můžeme s výhodou využít omezené rychlosti počítače. Základní myšlenka pak spočívá v tom, že namísto jednoho posunutí želvy například o jednu jednotku ji posuneme tisíckrát o jednu tisícinu jednotky. Režie volání procedur přitom způsobí, že úsečka bude vykreslována pomalu. Program v jazyce Scheme pak může vypadat například takto:

```
(define (slow-forward d)
  (if (<= d 0) (void)
      (begin
        (forward 0.001)
        (slow-forward (- d 0.001)))))
```

2.4 Projekt: regulární polygony pomocí želví grafiky

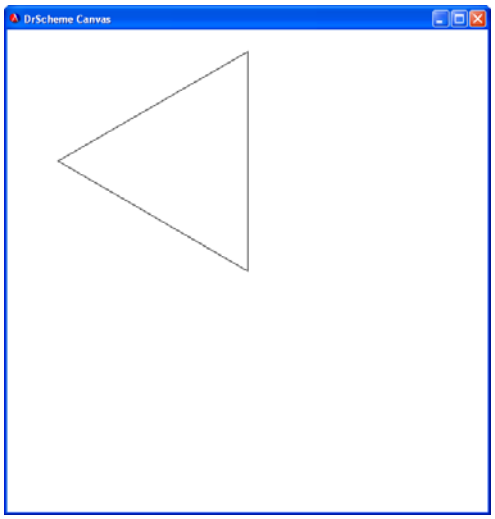
Studijní cíle: Při řešení projektu si studující vizuálním způsobem procvičí práci s želví grafikou a s rekurzivními procedurami.

Klíčová slova: Regulární polygony, rotace, želví grafika.

Potřebný čas: 2 hodiny.

2.4.1 Jednoduché polygony

Pomocí želví grafiky nakreslíme jednoduché polygony.



Obr. 10 Trojúhelník

Obrázek Obr. 10 představuje trojúhelník, nakreslený pomocí želví grafiky. Základní funkční blok procedury `trojuhelnik` spočívá v posunutí želvy o jednu jednotku vpřed a otočení o 120 stupňů, což odpovídá vnějšímu úhlu trojúhelníku. Tento blok je potřeba provést třikrát pro každou stranu trojúhelníka. Vyzkoušejte si tento kód na počítači:

```
(define (trojuhelnik)
  (forward 1)
  (left 120)
  (forward 1)
  (left 120)
  (forward 1)
  (left 120))
```

Průvodce studiem

Pozornějšímu čtenáři neujde, že poslední příkaz procedury `trojuhelnik` je v podstatě zbytečný. Způsobí pouze to, že želva zůstane po vykreslení trojúhelníku ve stejné pozici jako na začátku kreslení, což v tuto chvíli nemá žádný význam. My tento příkaz přesto uvedeme, protože nám umožní zachytit konstrukční podobnosti mezi programy pro vytváření různých polygonů.

Úkoly k textu

1. Napište obdobné procedury na vykreslení čtverce, pentagonu a hexagonu. Procedury otestujte. Všimněte si podobností mezi jednotlivými procedurami.

2.4.2 Obecný polygon

Jak jsme ukázali v předchozí kapitole, procedury na kreslení regulárních polygonů libovolného stupně mají podobnou strukturu. Skládají se z bloku, který kreslí jednu stranu polygonu. Tento blok je třeba provést tolikrát, kolik má polygon stran. Blok obsahuje příkaz na posun vpřed o jednu jednotku a příkaz na otočení o vnější úhel polygonu. Lze vysledovat, že vnější

Za využití rekurze lze napsat proceduru pro vykreslení obecného polygonu.

úhel polygonu je roven podílu 360 a počtu vrcholů. Pro trojúhelník je to 120°, pro čtverec 90°, pro pentagon 72° atd.

Naším cílem je tyto podobnosti zachytit a navrhnout obecnou proceduru `polygon`, která vykreslí na obrazovku polygon libovolného stupně. Procedura bude mít zřejmě rekurzivní charakter a pro svou práci potřebuje dva parametry – počítadlo vykreslených stran (zde označené jako `n`) a úhel natočení (`angle`). Kód může mít takovouto strukturu:

```
(define (polygon n)
  (polygon-loop n (/ 360 n)))

(define (polygon-loop n angle)
  (if (= n 0) (void)
      (begin
         ...
         ...
         ...)))
```

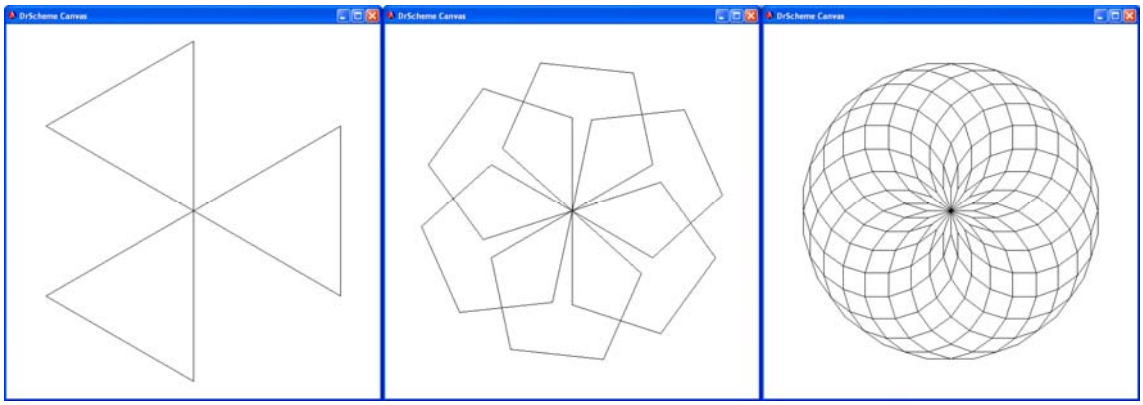
Úkoly k textu

2. Doplňte vynechaná místa v proceduře `polygon-loop`. Otestujte proceduru `polygon` na několika příkladech.

2.4.3 Rotující polygony

Polygony můžeme snadno nechat rotovat.

Zajímavých efektů dosáhneme, když do jednoho obrázku zobrazíme více polygonů stejného stupně, přičemž pokaždé změníme počáteční orientaci želvy. Obrázek Obr. 11 ukazuje výsledek některých rotací polygonů. Vlevo se nacházejí tři rovnostranné trojúhelníky, uprostřed 6 pentagonů a vpravo dvacet dvacetiúhelníků.



Obr. 11 Rotující polygony

Všechny tyto obrazce byly vytvořeny stejnou procedurou. Vytvoření takovéto procedury je velmi snadné, jednak proto, že proceduru na vykreslení jednoho polygonu již máme hotovou, a také proto, že nám tato procedura může posloužit jako vzor.

Úkoly k textu

3. Zkopírujte procedury `polygon` a `polygon-loop` a přejmenujte je na `rotate` a `rotate-loop`. Při změně jména nezapomeňte změnit i rekurzivní volání. Co se stane, když nahradíte

v proceduře `rotate-loop` příkaz `(forward 1)` příkazem `(polygon 3)`?

4. Upravte proceduru `rotate` tak, aby akceptovala jako parametry stupeň polygonu a počet požadovaných rotací. Otestujte tuto proceduru na několika příkladech. Pravděpodobně budete muset volit grafická okna různých velikostí, aby bylo možno rotující polygony zobrazit.

5. Upravte proceduru `rotate` tak, aby namísto zobrazení mnoha polygonů na obrazovce animovala rotační pohyb jednoho polygonu kolem počátku souřadnic. Navrhněte řešení. Využijte přitom toho, že druhým parametrem příkazu `forward` může být barva vykreslovaného objektu, například `(forward 1 "black")`. Dále můžete využít systémové procedury `sleep`, která pozastaví běh výpočtu na daný počet vteřin. Vzhledem k omezené rychlosti grafického systému Scheme a funkcím dostupným z teachpacku neočekávejte zcela perfektní výsledek.

Shrnutí

Polygony libovolného stupně lze jednoduše vykreslovat jedinou procedurou. Obdobnou proceduru lze použít i pro zobrazení několika polygonů stejného stupně stejnoměrně otočených kolem počátku.

2.5 Projekt: fraktální obrazce pomocí želví grafiky

Studijní cíle: Kromě praxe ve vytváření rekurzivních procedur se studující při řešení tohoto projektu zdokonalí v abstraktním myšlení a v analyzování rekurzivních závislostí.

Klíčová slova: Fraktální obrazce, želví grafika.

Potřebný čas: 4 hodiny.

Fraktální obrazce nalezneme často v přírodě.

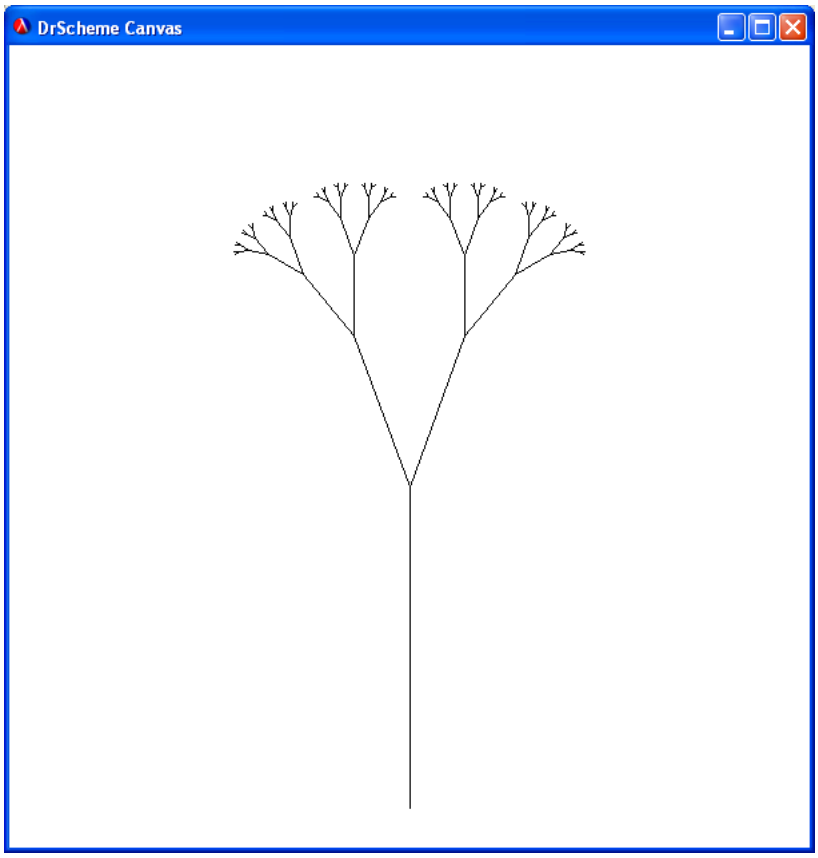
Fraktální obrazce, laicky řečeno, obsahují v sobě menší a jednodušší verze sebe samých, jsou tedy svým způsobem rekurzivní. S tímto fenoménem se velmi často setkáváme v živé i neživé přírodě. Jednotlivé části rostlin, skal, nebo krystalů vypadají podobně jako celá věc. Podíváme-li se na fotografii stromu na Obr. 12, vidíme že každá větev stromu vypadá podobně (až na natočení) jako celý strom.

Obr. 12 Strom

Naším cílem v tomto projektu bude vytvoření programů pro kreslení různých fraktálních obrazců podobných rostlinám. První vzorový příklad vyřešíme společně.

2.5.1 Plevel

Následující fraktální obrázek ukazuje fiktivní rostlinu, kterou zde pracovně nazveme jako *plevel*. Jedná se o obrázek 7. řádu.



Obr. 13 Plevel - obrazec 7. řádu

Fraktální obrazce rostlin se řídí jednoduchými pravidly.

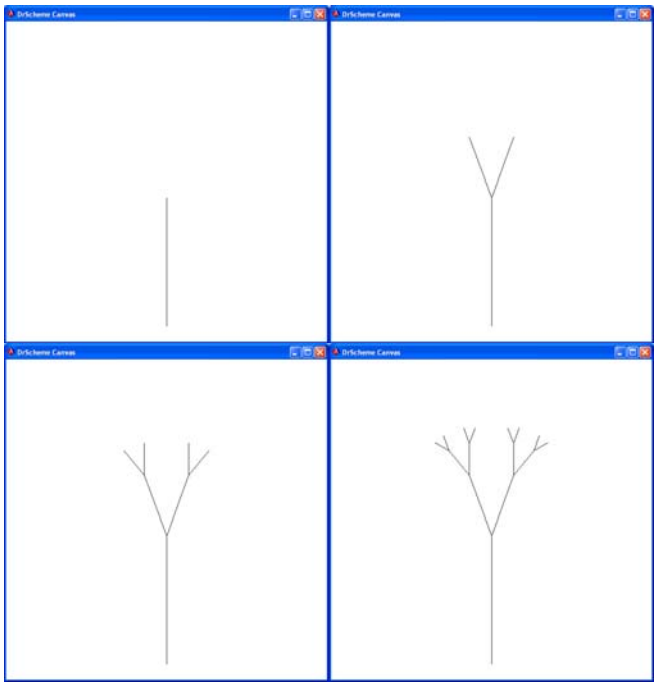
Budeme-li analyzovat tento fraktální obrazec, uvidíme, že se skládá z jedné svislé úsečky, z jejíhož konce pak „vyrůstají“ pod jistými úhly dva podobné ale menší fraktální obrazce. Každý z nich se potom opět skládá z úsečky a dvou podobných ale menších obrazců. Úhel natočení bychom mohli snadno změřit a ukázalo by se, že menší obrazce navazují na úsečku pod úhlem 20 stupňů. Velikost těchto obrazců (měřeno délkou úsečky) je přitom poloviční.

Průvodce studiem

Tyto a podobné fraktální obrazce lze elegantně popsat pomocí mechanismu tzv. L-systémů. v tomto popisu má každý obrazec svůj axiom a sadu pravidel. V našem textu nebudeme popis obrazců nijak formalizovat a přidržíme se intuitivního náhledu na rekurzivní podstatu problému.

Začneme-li přemýšlet o postupu vytváření těchto obrazců, musíme uvážit velikost a řád obrazce. Velikost udává délku kreslené úsečky a v každém kroku se zmenšuje na polovinu. Řád udává úroveň složitosti daného obrazce a v každém kroku se snižuje o jedničku. Pro vykreslení obrazce 7. řádu tak stačí nakreslit úsečku dané velikosti, otočit kreslící zařízení (želvu) vlevo a nakreslit poloviční obrazec 6. řádu, dále otočit zařízení vpravo a nakreslit další poloviční obrazec 6. řádu. Obrazce 6. řádu se dále skládají z poloviních obrazců 5. řádu atd. Obrazec nultého řádu je prázdný.

Pro snadnější analýzu bývá vhodné, podívat se, jak obrazec „roste“. Obrázek Obr. 14 ukazuje kompozici obrazců 1., 2., 3. a 4. řádu.



Obr. 14 Plevel – obrazce 1. až 4. řádu

V tuto chvíli jsme téměř připraveni na zápis kódu v jazyce Scheme. Jediným zbývajícím problémem je vyřešit návrat želvy do místa, odkud kreslení začalo. Po vykreslení „levé větve“ našeho plevele totiž potřebujeme, aby se želva vrátila do přesně stejného místa odkud jsme začali, abychom mohli plynule pokračovat pravou větví. Pro tyto účely si doplníme příkazy pro práci s želví grafikou o dva příkazy:

Pro vytvoření algoritmu je užitečné vidět různé řády obrazců.

- procedura `remember` způsobí, že si želva zapamatuje svoji aktuální pozici
- procedura `return` způsobí, že se želva vrátí do poslední zapamatované pozice.

Průvodce studiem

Tyto dva příkazy pracují nad interním zásobníkem pozic želvy. V praxi to znamená, že můžeme dát příkaz `remember` vícekrát za sebou. Po sobě zadávané příkazy `return` pak způsobí, že se želva postupně vrací na zapamatovaná místa.

Kód v jazyce Scheme pro vykreslení rekurzivního obrazce plevele pak vypadá takto:

```
(define (plevel n size)
  (if (> n 0)
      (begin
        (remember)
        (forward size)
        (left 20)
        (plevel (- n 1) (* size 0.5))
        (right 40)
        (plevel (- n 1) (* size 0.5))
        (return))))
```

Obrázek Obr. 13 byl vytvořen těmito příkazy:

```
(init-playground 500)
(set-turtle 0 -450 0)
(plevel 7 400)
```

Velikost grafického okna a počáteční pozici želvy budeme zachovávat stejnou i pro další fraktální obrazce.

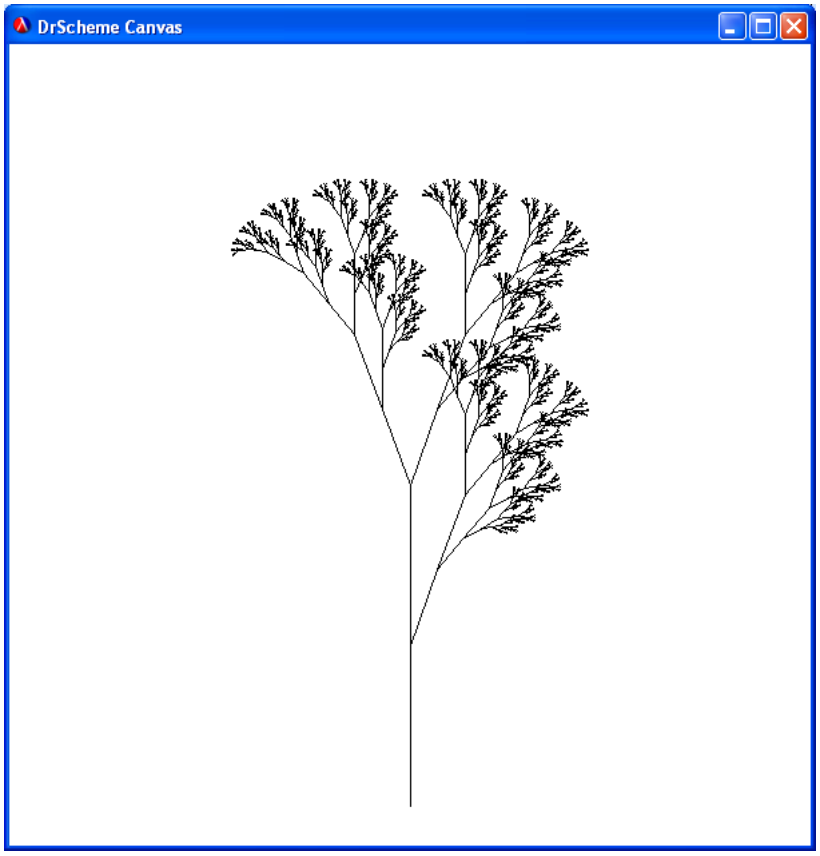
Úkoly k textu

1. Experimentujte s kódem. Změňte úhel odklonu a koeficient velikosti a pozorujte, jaký to má vliv na výsledný obrazec.

2.5.2 Tráva

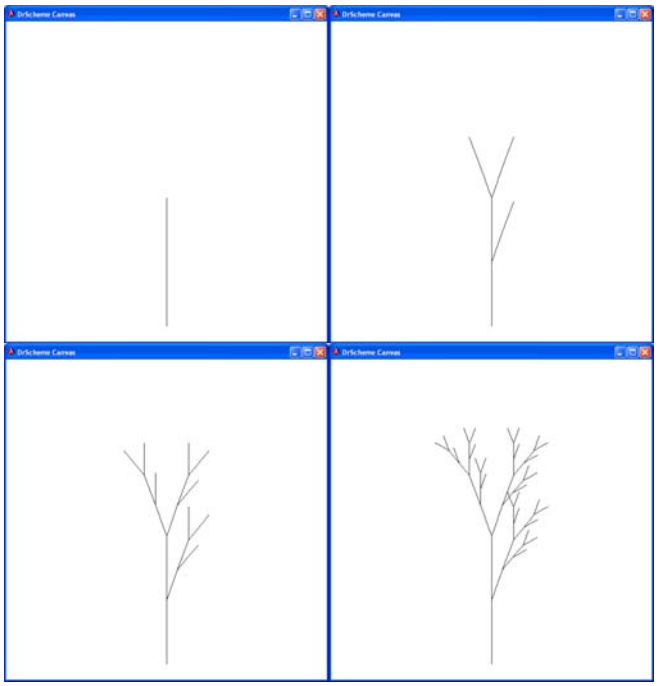
Další fraktální obrazec má pracovní název „tráva“. Následující obrázek ukazuje obrazec 8. řádu a byl vytvořen pomocí volání procedury `(trava 8 200)`.

Fraktální obrazce lze modifikovat co do velikosti, úhlů i algoritmu.



Obr. 15 Tráva – obrazec 8. řádu

Pro snazší analýzu problému ukážeme kompozici obrazců 1. až 4. řádu. Řešení problému je obdobné jako v předchozím případě. Jediný koncepční rozdíl je v tom, že každý obrazec trávy obsahuje tři instance sebe samého namísto dvou a dvě úsečky namísto jedné.



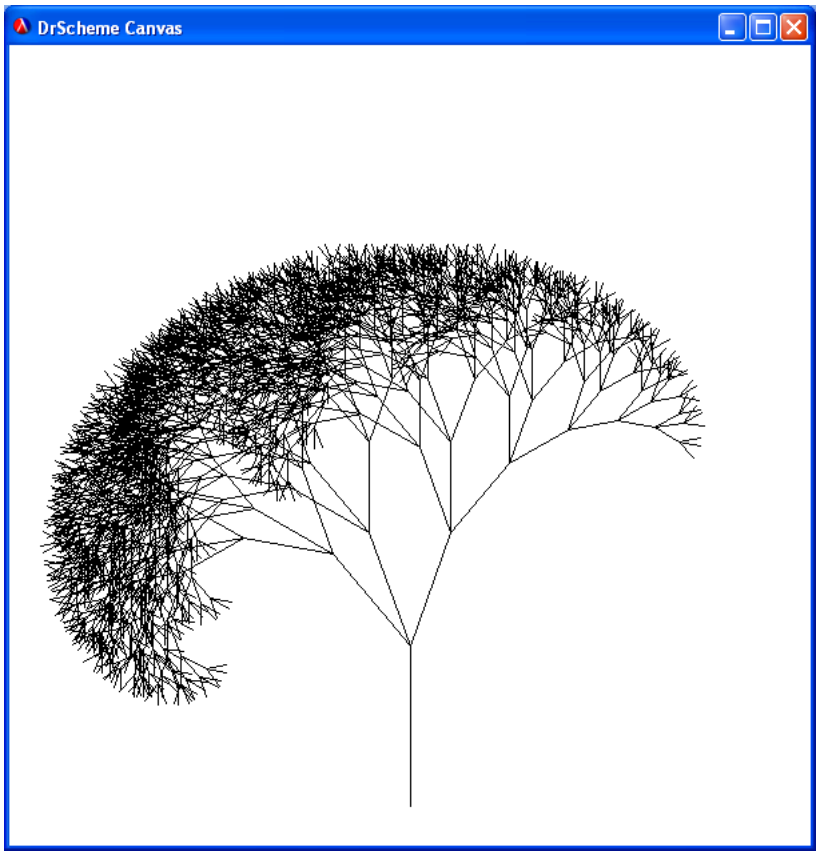
Obr. 16 Tráva – obrazce 1. až 4. řádu

Úkoly k textu

2. Navrhněte proceduru `trava`, která akceptuje řád a velikost a nakreslí fraktální obrazec.

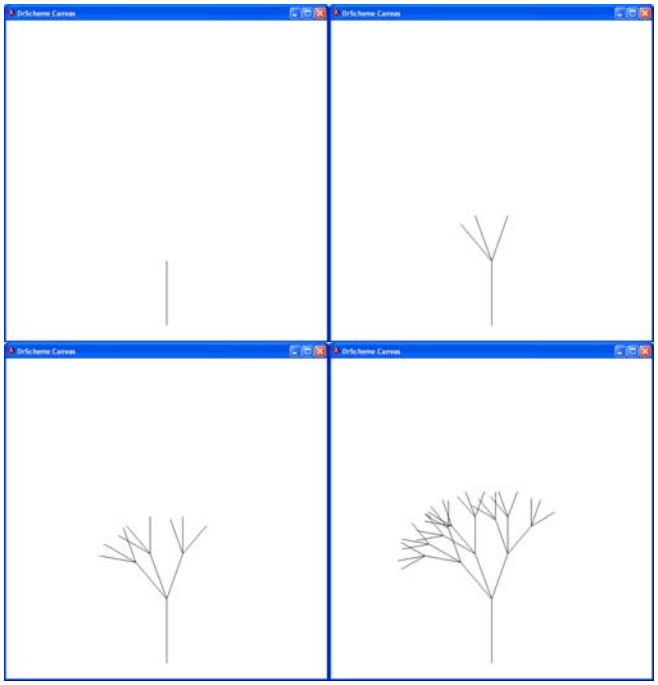
2.5.3 Strom

Další fraktální obrazec má pracovní název „strom“. Následující obrázek ukazuje obrazec 8. řádu a byl vytvořen pomocí volání procedury `(strom 8 200)`.



Obr. 17 Strom – obrazec 8. řádu

Pro snazší analýzu problému opět ukážeme kompozici obrazců 1. až 4. řádu.



Obr. 18 Strom – obrazce 1. až 4. řádu

Zde je program ještě bližší prvnímu řešenému případu než v předchozí podkapitole. Máme zde pouze jednu úsečku a tři instance sama sebe realizované rekurzivním voláním. Úhly náklonu vzhledem k vertikále jsou 40° vlevo, 20° vlevo a 20° vpravo. Velikost každé „větve stromu“ je rovna třem čtvrtinám původní velikosti.

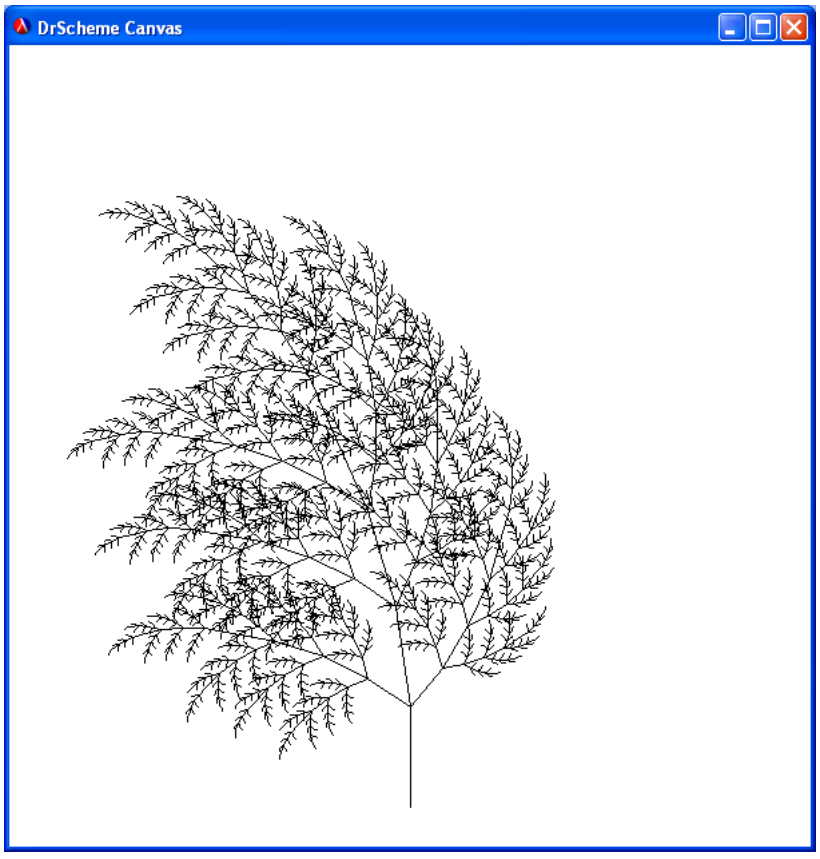
Úkoly k textu

3. Navrhněte proceduru `strom`, která akceptuje řád a velikost a nakreslí fraktální obrazec.

2.5.4 Kapradí

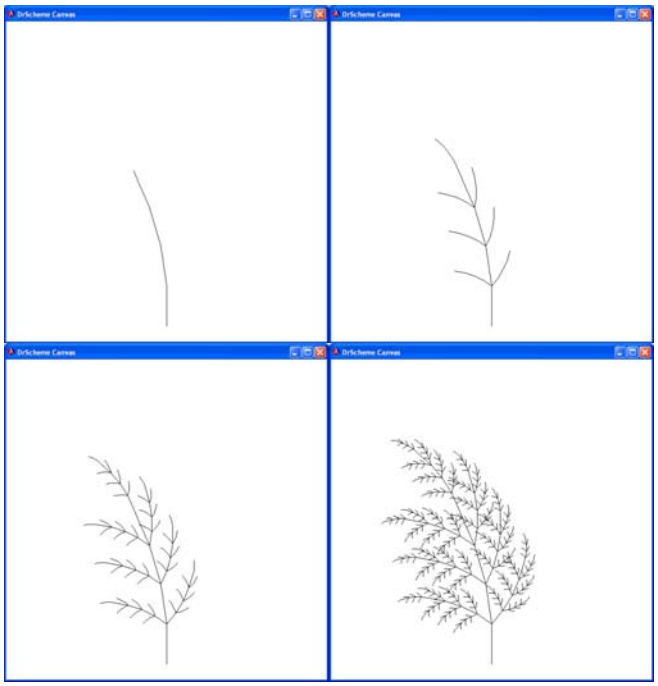
Další fraktální obrazec má pracovní název „kapradí“. Následující obrázek ukazuje obrazec 5. řádu a byl vytvořen pomocí volání procedury `(kapradi 5 120)`.

Některé fraktální obrace jsou velmi komplexní.



Obr. 19 Kapradí – obrazec 5. řádu

Pro snazší analýzu problému opět ukážeme kompozici obrazců 1. až 4. řádu.



Obr. 20 Kapradí – obrazce 1. až 4. řádu

Program na vykreslení tohoto obrazce je výrazně komplikovanější než v předchozích případech. Podíváme-li se na obrazec prvního řádu pozorně, vidíme, že větvička se skládá ze čtyř úseček, každá úsečka je od předchozí úsečky odkloněna o 8°. Z obrazce druhého řádu navíc vidíme, že na konci každé úsečky vyrážejí vlevo i vpravo malé větvičky – obrazce nižšího řádu. Náklony větviček od předchozí úsečky jsou obtížně zjistitelné a proto je uvádíme v tomto přehledu:

- první větvení: vpravo 40°, vlevo 56°,
- druhé větvení: vpravo 32°, vlevo 48°,
- třetí větvení: vpravo 24°, vlevo 40°[1].

Podíváme-li se na obrazec třetího řádu, vidíme, že celá větvička je navíc vždy zakončena malou větvičkou – obrazcem nižšího řádu.

Komplikovaná je i velikost obrazců jednotlivých řádů. Tato velikost totiž neklesá rovnoměrně jako v předchozích příkladech, ale je závislá na řádu obrazce. Pokud obrazec n-tého řádu má úsečku velikosti x, pak obrazec n-1 řádu používá úsečku velikosti $0.25x\sqrt{n-1}$.

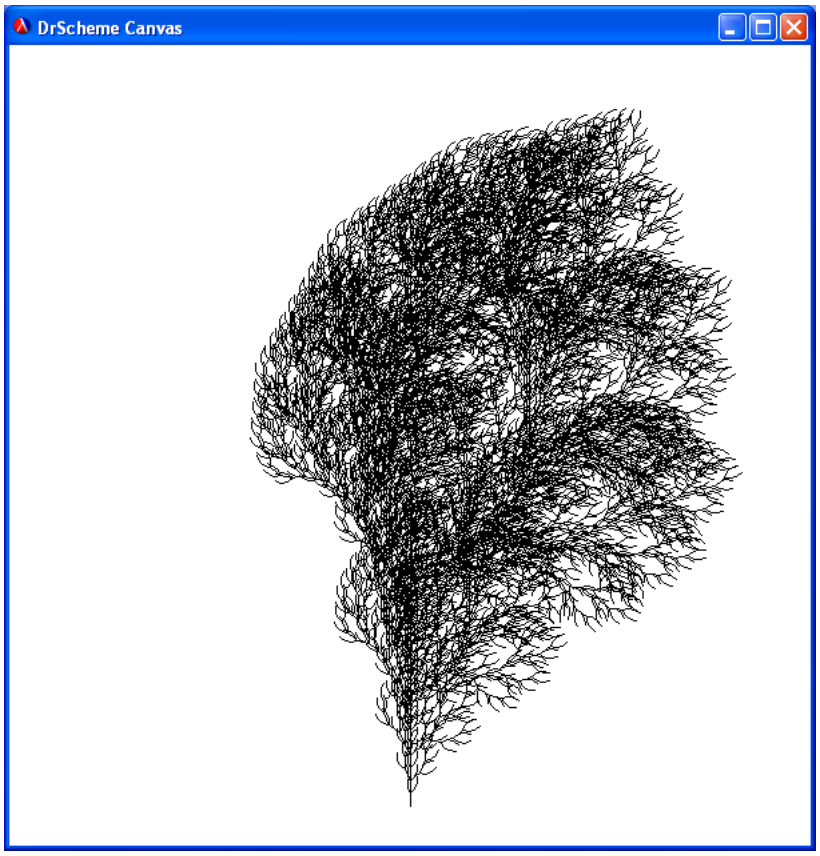
Úkoly k textu

4. Navrhněte proceduru `kapradi`, která akceptuje řád a velikost a nakreslí fraktální obrazec.

2.5.5 Keř

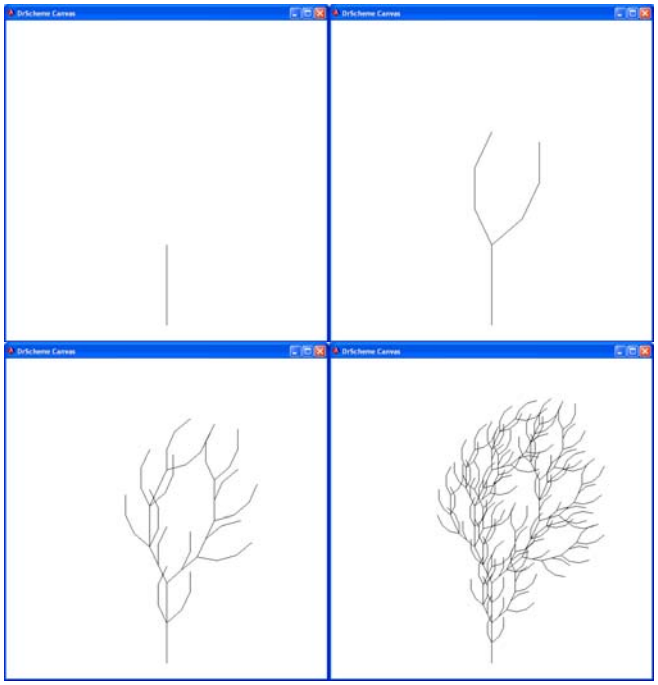
Další fraktální obrazec má pracovní název „keř“. Následující obrázek ukazuje obrazec 6. řádu a byl vytvořen pomocí volání procedury `(ker 6 250)`.

[1] Všimněte si, že jednotlivé náklony jsou vždy násobkem čísla 8.



Obr. 21 Keř – obrazec 6. řádu

Pro snazší analýzu problému opět ukážeme kompozici obrazců 1. až 4. řádu.



Obr. 22 Keř – obrazce 1. až 4. řádu

Program na vykreslení tohoto obrazce je ještě komplikovanější než v předchozí příkladě. Celý program má zcela jinou stavbu. Základní strukturu zde proto uvedeme a čtenářům zbývá doplnit pouze levou a pravou „větev“ keře. Lze snadno vysledovat, že každá z větví se skládá ze tří keřů nižšího řádu. Úhel náklonu mezi jednotlivými keři ve větvi je přitom 25°.

```
(define (ker n size)
  (if (= n 1)
      (forward size)
      (begin
        (ker (- n 1) (/ size 2.0))
        (ker (- n 1) (/ size 2.0))

        ; leva vetev
        (remember)
         ……….
         ……….
         ……….
         ……….
        (return)

        ; prava vetev
        (remember)
         ……….
         ……….
         ……….
         ……….
        (return))))
```

Úkoly k textu

5. Navrhněte proceduru `ker`, která akceptuje řád a velikost a nakreslí fraktální obrazec.

Shrnutí

Schematický vzhled mnoha rostlin má rekurzivní charakter a lze jej popsat jednoduchým programem. Na tomto projektu si studující kromě tvorby rekurzivních procedur procvičili analýzu rekurzivních obrazců a hledání rekurzivních závislostí.

3 Modularita

3.1 Modulární programování

Studijní cíle: Po zvládnutí kapitoly budou studující schopni členit rozsáhlejší kód na několik jednodušších procedur a využívat lokální procedury pro zjednodušení a zapouzdření kódu.

Klíčová slova: Lokální procedura.

Potřebný čas: 1 hodina 30 minut.

Problém rozkladu většího úkolu na samostatné procedury – moduly a zavedení lokálních procedur budeme demonstrovat na modelovém příkladě výpočtu druhé odmocniny čísla.

K výpočtu druhé odmocniny využijeme Heronův algoritmus.

I když lze druhou odmocninu čísla velmi jednoduše matematicky nadefinovat, není zcela evidentní, jak ji pomocí elementárních aritmetických operátorů vypočítat. S prvním algoritmem přišel Heron Alexandrijský, pravděpodobně v prvním století před naším letopočtem.[1] Algoritmus výpočtu druhé odmocniny z čísla `a` lze jednoduše vyjádřit v těchto krocích:

1. Vytvořme odhad x (i když velmi nepřesný) druhé odmocniny z čísla a
2. Je-li tento odhad dostatečně přesný, pak číslo x je cílená hodnota
3. Pokud je odhad nepřesný, můžeme jej zlepšit tak, že vypočteme průměr čísla x a čísla a/x. To je náš nový odhad, můžeme pokračovat bodem 2.

3.1.1 Rozklad problému

Problém řešíme pomocí několika procedur.

V představeném algoritmu Heron tvrdí, že aritmetický průměr čísla `x` a `a/x` je vždy lepším odhadem druhé odmocniny z `a` než samotné číslo `x`. Jako první tedy vytvoříme proceduru na výpočet průměru dvou čísel. Za pomoci této procedury pak můžeme vytvořit proceduru pro zlepšení odhadu za pomoci Heronova tvrzení.

```
(define (prumer a b)
  (/ (+ a b) 2))

(define (zlepsi x a)
  (prumer x (/ a x)))
```

Pokud Heronovu tvrzení nevěříme, můžeme použít překladač jazyka Scheme a toto tvrzení podrobit experimentu. Řekněme, že chceme počítat druhou odmocninu ze 2 a náš počáteční naivní odhad je číslo 1.

```
> (zlepsi 1.0 2)
1.5
> (zlepsi 1.5 2)
1.4166666666666665
> (zlepsi 1.4166666666666665 2)
1.4142156862745097
```

Heronovo tvrzení obstálo. Další otázkou je, jak otestovat, že náš odhad je „dostatečně přesný“. Samotný pojem je poněkud kontroverzní. Tyto a podobné problémy podrobněji řeší odvětví zvané numerická matematika. My se omezíme na konstatování, že odhad `x` je dostatečně přesný

[1] Později se ukázalo, že tento algoritmus je jen zvláštním případem tzv. Newtonovy metody. Newtonova metoda se opírá o infinitezimální počet, který Heron neznal.

odhad druhé odmocniny z `a`, pokud výraz $\left|x^2 - a\right|$ je menší, než nějaké hodně malé číslo, například 0.001.

```
(define (presny? x a)
  (< (abs (- (sqr x) a)) 0.001))
```

Průvodce studiem

Předpokládejme, že chceme nalézt druhou odmocninu z čísla 25000000 a druhou odmocninu z čísla 0,00000025. Na jaké problémy narazíme s takto navrženým algoritmem na určení dostatečné přesnosti výpočtu? Dokážete navrhnout lepší metodu?

Používáme "obalující" proceduru.

Nyní již můžeme napsat proceduru `sqrt-iter` na výpočet druhé odmocniny. Tato procedura musí akceptovat vždy dva parametry: odhad druhé odmocniny a odmocňované číslo. To je pro cílového uživatele poněkud nešťastné, lepší by bylo mít proceduru, která akceptuje jediný parametr. Tento problém řeší poslední procedura, `sqrt`, která volá proceduru `sqrt-iter` a jako počáteční odhad uvádí číslo jedna.

```
(define (sqrt-iter x a)
  (if (presny? x a) x
      (sqrt-iter (improve x a) a)))

(define (sqrt a)
  (sqrt-iter 1.0 a))
```

Na tomto příkladě vidíme, že původně složitou a obtížně uchopitelnou úlohu bylo možno rozložit na části. Každá část má svůj přesně vymezený úkol a je nezávisle testovatelná. Některé z procedur mohou být bez problémů využity i v jiných projektech.

3.1.2 Lokální jména

Pomocné procedury můžeme definovat lokálně.

I když je rozklad na podproblémy jistě užitečný, pozorný čtenář se nezbaví jisté rozpačitosti. Proč máme pět procedur, když ve skutečnosti potřebujeme jen jedinou proceduru – `sqrt`? Jak lze poznat, že procedura `presny?` patří k výpočtu druhé odmocniny? Co když si programátor bude chtít ve svém programu pojmenovat nějakou jinou proceduru názvem `zlepsi`? Na tyto všechny otázky dává odpověď zavedení *lokálních procedur*. Na podobný problém jsme již narazili v kapitole 1.3.3, když jsme do procedury zaváděli lokální definici symbolu. Obdobným způsobem můžeme definovat lokální procedury. Celý kód tak můžeme přepsat takto:

```
(define (sqrt a)

  (define (prumer a b)
    (/ (+ a b) 2))

  (define (zlepsi x a)
    (prumer x (/ a x)))

  (define (presny? x a)
    (< (abs (- (sqr x) a)) 0.001))

  (define (sqrt-iter x a)
    (if (presny? x a) x
        (sqrt-iter (zlepsi x a) a)))

  (sqrt-iter 1.0 a))
```

Navenek je tak viditelná jediná procedura: `sqrt`. Uvnitř této procedury jsou umístěné jednotlivé pomocné procedury `prumer`, `zlepsi`, `presny?` a `sqrt-iter`.

Lokální definice mohou vést ke zjednodušení kódu.

Toto řešení má ještě jednu výhodu. Všechny pomocné procedury (kromě procedury `prumer`) akceptují parametry `x` a `a`. Zatímco parametr `x` představuje počítaný odhad druhé odmocniny a postupně se mění, odmocňované číslo `a` zůstává stejné. Protože tyto procedury jsou lokální v proceduře `sqrt`, mají přístup i k parametru `a` procedury `sqrt`. Hodnota `a` proto nemusí být předávána jako parametr a kód se nám dále zjednoduší.

```
(define (sqrt a)

  (define (prumer a b)
    (/ (+ a b) 2))

  (define (zlepsi x)
    (prumer x (/ a x)))

  (define (presny? x)
    (< (abs (- (sqr x) a)) 0.001))

  (define (sqrt-iter x)
    (if (presny? x) x
        (sqrt-iter (zlepsi x))))

  (sqrt-iter 1.0))
```

Průvodce studiem

Lokální procedury budeme často využívat u komplikovanějších programů. Zatímco modularita a přehlednost výsledného programu se s použitím lokálních procedur výrazně zvýší, vytváření programů se komplikuje. Nepříjemné je zejména to, že lokální proceduru nemůžeme samostatně otestovat. Doporučujeme tedy volit stejný postup, jako jsme zvolili v této kapitole: všechny pomocné procedury nejprve napsat a otestovat zvlášť a teprve pak je umístit do těla hlavní procedury.

Shrnutí

Rozsáhlejší problémy je třeba rozkládat na menší části – procedury. Každou část je pak možno řešit a testovat samostatně. Pomocné procedury lze vkládat do hlavních procedur.

Pojmy k zapamatování

- Lokální procedura.

Kontrolní otázky

1. *Jaké výhody a nevýhody přináší rozklad úlohy na pomocné procedury?*
2. *Jaké výhody a nevýhody přináší lokální deklarace pomocných procedur?*

Úkoly k textu

1. Abyste opravdu ocenili rozklad problému na jednotlivé procedury, přepište proceduru `sqrt-iter` tak, aby počítala odhad druhé odmocniny bez využití pomocných procedur.

2. Heronův algoritmus je jen speciálním případem Newtonovy metody hledání kořene rovnice. Newtonova metoda říká, že pokud x je odhad řešení rovnice $f(x) = 0$, potom lepší odhad lze získat pomocí výrazu
   $$x - \frac{f(x)}{f'(x)}$$
   Ve jmenovateli zlomku je první derivace funkce. Heronův vzorec na zlepšení odhadu druhé odmocniny vznikne pro funkci $f(x) = x^2 - a$. Vzorec na zlepšení odhadu třetí odmocniny by vzniknul pro funkci $f(x) = x^3 - a$. Odvoďte vzorec a napište proceduru `qurt` na výpočet třetí odmocniny.

3.2 Projekt: Buffonova jehla

Studijní cíle: Při řešení projektu si studující procvičí práci s jazykem Scheme, tvorbu rekurzivních procedur a využívání lokálních deklarací pomocných procedur. Zároveň zvládnou principy metody Monte Carlo.

Klíčová slova: Buffonova jehla, rekurze, metoda Monte Carlo.

Potřebný čas: 1 hodina.

Buffonova jehla označuje jednoduchý stochastický způsob odhadu hodnoty Ludolfova čísla π. Je připisován francouzskému matematikovi 18. století Georges Buffonovi (viz obrázek Obr. 23).

Obr. 23 Georges Louis Leclerc Comte de Buffon

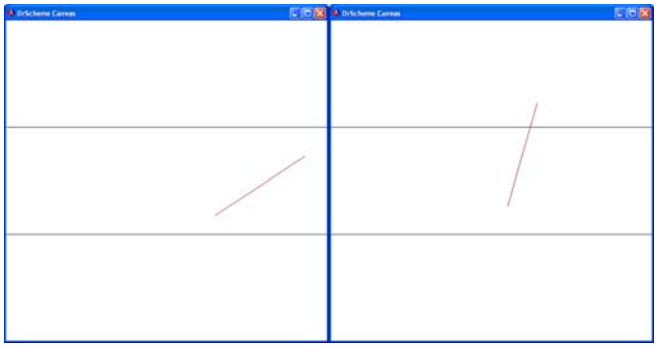
3.2.1 Princip experimentu

Základní myšlenka je velmi jednoduchá. Představme si plochu stolu, na které jsou nakresleny rovnoběžné linky. Linky jsou rozmístěny stejnoměrně například 5 centimetrů od sebe. Dále si představme, že máme jehlu nebo špendlík stejné délky jako je vzdálenost mezi linkami.

Pokud náhodně hodíme jehlu na stůl, mohou nastat dva případy:

- jehla spadne na stůl a zůstane ležet tak, že nekříží žádnou linku
- jehla zůstane ležet tak, že kříží jednu z linek.

Obě možné situace schematicky znázorňuje obrázek Obr. 24.



Obr. 24 Pozice Buffonovy jehly

Principem projektu je házet jehly na stůl a zaznamenávat si statistiku. Potřebujeme přitom dvě základní veličiny: celkový počet hodů n a celkový počet zásahů x, tedy situací, kdy jehla zkřížila linku. Když hodíme na stůl dostatečně mnoho jehel, zjistíme, že podíl 2n a x se blíží číslu π, tedy: $\frac{2n}{x} \to \pi$

3.2.2 Odhad čísla π

K tomuto projektu potřebujeme teachpack `needle.scm`. Teachpack nám poskytuje dvě procedury, které neakceptují žádné parametry:

- `initialize` – inicializuje grafické okno Buffonovy jehly
- `drop-once` – shodí jednu jehlu do grafického okna; vrátí hodnotu `#t`, pokud jehla padla na linku a hodnotu `#f` pokud jehla padla mimo linku.

Úkoly k textu

1. Napište proceduru `pocet-zasahu`, která akceptuje parametr `n`, shodí jehlu n-krát do inicializovaného grafického okna a vrátí počet zásahů. Tato procedura musí využívat proceduru `drop-once`.

2. Za pomoci procedury `pocet-zasahu` napište proceduru `odhad-pi-buffon`, která také akceptuje parametr `n` a implementuje vzorec uvedený v podkapitole 3.2.1. Procedura `pocet-zasahu` bude lokální v proceduře `odhad-pi-buffon`.

Průvodce studiem

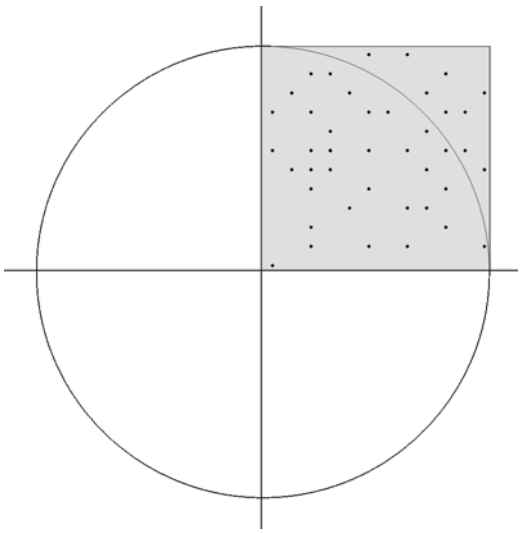
Tento experiment vám pravděpodobně nedá velmi přesný odhad Ludolfova čísla ani pro relativně velké počty experimentů. Typicky dosahujeme přesnosti na jedno desetinné místo. Přesnost je omezena jednak použitým algoritmem, ale hlavně zabudovaným generátorem pseudo-náhodných čísel. Pro běžné potřeby nám pseudo-náhodná čísla generovaná počítačem dostačují, používáme-li však velké množství takovýchto čísel ve výpočtech, začínají se objevovat nerovnoměrnosti v distribuci, které ovlivňují výsledek.

3.2.3 Metoda Monte Carlo

Buffonova jehla je jedním z rodiny stochastických projektů označovaných jako *metody Monte Carlo*. Principem je vždy experiment, který může dopadnout pozitivně nebo negativně. Experiment opakujeme mnohokrát po sobě a zaznamenáváme počet pozitivních pokusů v poměru k počtu celkových pokusu. Tento podíl má pak nějakou vypovídající hodnotu.

Představme si jednotkovou kružnici (kružnici s poloměrem 1) opsanou kolem počátku souřadnic. Plocha jednotkové kružnice je π. Pro jednoduchost uvažme pouze tu výseč kružnice, která leží v prvním kvadrantu. Plocha této výseče je π/4. Nyní orámujme tuto výseč čtvercem o délce strany 1. Plocha tohoto čtverce je 1. Pokud budeme nyní náhodně generovat body se souřadnicemi [x, y] v tomto čtverci, mohou nastat dvě situace: bod bude ležet uvnitř kruhové výseče nebo bude ležet mimo kruhovou výseč. První situace nastane, když $\sqrt{x^2 + y^2} < 1$.

Vygenerujeme-li tolik bodů, že zcela „pokryjí“ plochu čtverce, není těžké si uvědomit, že poměr počtu bodů, které padnou do kruhové výseče, vzhledem k celkovému počtu bodů, musí být stejný jako poměr ploch kruhové výseče a čtverce. Tento poměr je π/4. Viz obrázek Obr. 25.



Obr. 25 Metoda Monte Carlo

Experiment je možno jednoduše vyjádřit touto procedurou v jazyce Scheme:

```
(define (experiment)
  (define x (/ (random 1000) 1000))
  (define y (/ (random 1000) 1000))
  (< (sqrt (+ (sqr x) (sqr y))) 1))
```

Úkoly k textu

3. Upravte proceduru `pocet-zasahu` z předchozího cvičení tak, aby namísto procedury `drop-once` používala proceduru `experiment`.

4. Za pomoci procedury `pocet-zasahu` napište proceduru `odhad-pi-kruh`, která také akceptuje parametr `n` a implementuje logiku uvedenou v této podkapitole. Procedury `experiment` a `pocet-zasahu` musí být lokální v proceduře `odhad-pi-kruh`.

Shrnutí

Metody Monte Carlo představují jednoduchý a elegantní způsob odhadu některých přírodních veličin. Na tomto projektu si studující procvičili tvorbu jednoduchých rekurzivních procedur a využití lokálních deklarací pomocných procedur.

4 Závěr

V této publikaci jsme se pokusili přiblížit studujícím programovací jazyk Scheme a naučit základy programování v tomto jazyku. Primárním cílem byl přitom technický popis základních prvků jazyka, které budou potřeba pro implementaci složitějších a rozsáhlejších programů. Výběr představených prvků jazyka byl přitom pojat minimalisticky. Mezi popsanými rysy jazyka například nenajdete řadu speciálních forem (`let`, `let*`, `letrec`, `case`), podporu cyklů nebo využití přiřazovacího příkazu. Na publikaci navazují další texty věnující se problematice výpočetního procesu, složitosti výpočtu a datových struktur.

Předpokládáme proto, že po absolvování tohoto textu sáhne studující po dalších materiálech, které pokryjí tyto i další oblasti programovacích paradigmat.

5 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.

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

[Skoupil04C] SKOUPIL, D. *Programy a projekty v jazyku Scheme III*.

[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

6 Seznam obrázků

7 Rejstřík