KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

CVIČENÍ Z PARADIGMAT PROGRAMOVÁNÍ I

Lekce 1: Program, jeho syntax a sémantika

Učební materiál k přednášce 5. října 2006
(pracovní verze textu určená pro studenty)

JAN KONEČNÝ, VILÉM VYCHODIL

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

Olomouc 2006

Lekce 1: Program, jeho syntax a sémantika

Obsah lekce: Tato lekce je úvodem do paradigmat programování. Vysvětlíme jaký je rozdíl mezi programem a výpočetním procesem, co jsou programovací jazyky a čím se od sebe odlišují. Uvedeme několik základních paradigmat (stylů) programování a stručně je charakterizujeme. Dále objasníme rozdíl mezi syntaxí (tvarem zápisu) a sémantikou (významem) programů. Představené pojmy budeme demonstrovat na jazyku Scheme. Programy v jazyku Scheme zavedeme jako posloupnosti symbolických výrazů a popíšeme jejich interpretaci. Zavedeme pojem elementy jazyka a ukážeme několik důležitých typů elementů jazyka, mimo jiné procedury a speciální formy. V závěru kapitoly popíšeme dvě speciální formy sloužící k zavádění nových definic a k podmíněnému vyhodnocování výrazů.

Klíčová slova: abstraktní interpret, aplikace procedur a speciálních forem, cyklus REPL, elementy jazyka, evaluator, externí reprezentace, interní reprezentace, interpretace, jazyk Scheme, paradigma, printer, procedury, program, překlad, reader, speciální formy symbolické výrazy, syntaktická chyba, syntaxe, sémantická chyba, sémantika, výpočetní proces.

1.1 Programovací jazyk, program a výpočetní proces

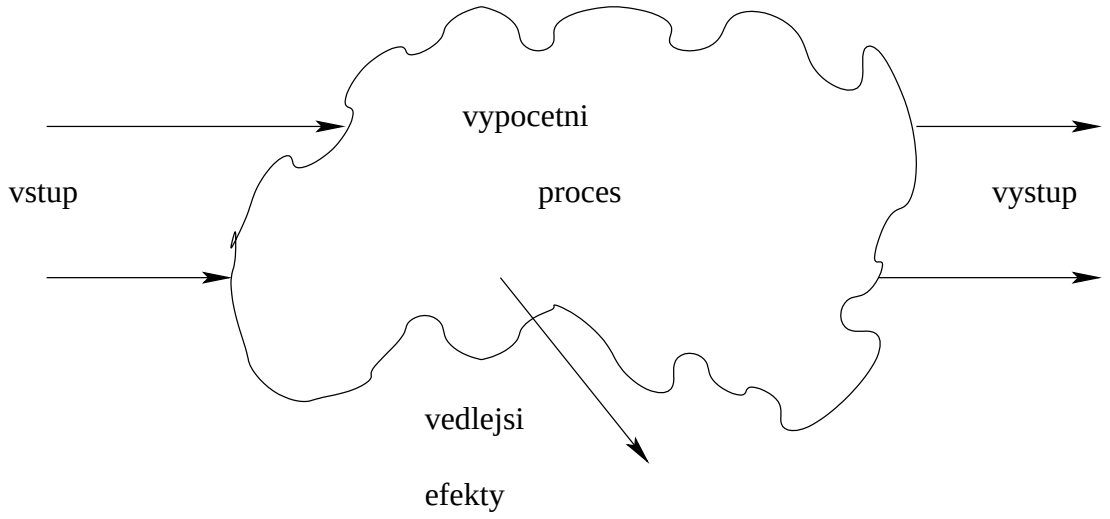
V současnosti je zřejmé, že úlohou počítačů není jen „provádět výpočty“, jak by jejich název mohl napovídat. Z dnešního pohledu se lze na počítače dívat obecně jako na *stroje zpracovávající data.* Samotný proces, při kterém jsou (nějaká) vstupní data zpracovávána nazýváme *výpočetní proces.* Výpočetní proces je dost abstraktní pojem. Pro nás však není příliš důležité rozebírat, jak si tento pojem „představit“. Důležité je chápat výpočetní proces jako proces s počátkem a koncem, probíhající v nějakých elementárních krocích, přetvářející vstupní data (vstupní parametry úlohy) na data výstupní (řešení). Takový pohled na výpočetní proces má své historické kořeny v prvopočátcích rozvoje počítačů, kde skutečně šlo jen o načtení vstupních dat, jejich zpracování a následný zápis výstupních dat. Z dnešního pohledu je tato charakterizace poněkud zjednodušená, protože uživatelé počítačů nepracují s počítači jen tímto primitivním „dávkovým způsobem“. V současné době výpočetní procesy mají celou řadu *vedlejších efektů,* jejichž účelem je například interakce s uživatelem. Viz schéma na obrázku 1.1.

Z pohledu informatika je důležité vědět, jak výpočetní procesy vznikají: výpočetní proces je důsledkem *vykonávání programu.* Každý *program* je předpisem s přesně daným tvarem a významem, podle kterého vzniká výpočetní proces. Smluvená pravidla, v souladu se kterými je program vytvořen, nazýváme *programovací jazyk.* Říkáme, že program *je napsán* v programovacím jazyku.

Na rozdíl od výpočetního procesu je tedy program „pasivní entita“ (sám o sobě nic nevykonává). V soudobých počítačích jsou programy reprezentovány soubory s přesně danou strukturou, které jsou uložené na disku počítače (ačkoliv tomu tak vždy nebylo). To jest samotné programy lze chápat jako data a obráceně (jak bude patrné především v dalších lekcích). Účelem tohoto učebního textu je seznámit studenty se základními *styly* (*paradigmaty*) *programování,* to jest styly vytváření programů, které jsou sdíleny mezi různými programovacími jazyky. Tvůrčí činnost vytváření programů se nazývá *programování,* člověk zabývající se programováním je *programátor.*

Úkolem programátora je vytvořit program tak, aby vykonáváním programu vznikl požadovaný výpočetní proces řešící přesně stanovený úkol. Kvalita programu se bezprostředně promítá do kvality výpočetního procesu a v důsledku do kvality práce člověka (uživatele) s počítačem. Ačkoliv je při počítačovém zpracování dat ústředním zájmem *výpočetní proces,* který samotné zpracování provádí, pozornost programátorů je obvykle směřována na *program,* to jest na předpis, podle kterého výpočetní proces vzniká. Program je potřeba vytvořit dostatečně kvalitně tak, aby neobsahoval chyby, aby byl efektivní, a tak dále. Ačkoliv tato kritéria zní na první poslech triviálně, s programy jejichž činnost je komplikována množstvím chyb, malou efektivitou, nebo nevhodnou komunikací s uživatelem, se bohužel setkáváme dnes a denně.

Obrázek 1.1. Výpočetní proces jako abstraktní entita (NACRT OBRAZKU)



Klíčem k vytváření kvalitních programů jenž povedou na žádoucí výpočetní procesy je pochopení pravidel daných programovacím jazykem (jak program zapisujeme, jaký význam mají konkrétní části programu, . . .) a pochopení, jak na základě vytvořeného programu vzniká výpočetní proces, a jak výpočetní proces probíhá. Těmito problematikami se budeme věnovat ve zbytku textu (a v dalších jeho dílech).

Při studiu problematiky programů a výpočetních procesů jimi generovaných budeme sledovat linii *programovacích jazyků.* Důvody jsou ryze pragmatické. Programovacích jazyků existuje celá řada (v současné době řádově v tisících), přitom různé programovací jazyky vyžadují od programátora při vytváření programů jiný styl myšlení, jiný způsob analýzy problému a jiné metody vytváření a testování programu. Není pochopitelně v našich silách zabývat se všemi možnými programovacími jazyky. Proto rozdělíme (téměř) všechny (současně používané) programovací jazyky do malého počtu (částečně se překrývajících) skupin. Každou skupinu budou tvořit jazyky s podobnými vlastnostmi – jazyky vyžadující od programátora stejnou metodu vývoje programu a myšlení nad problémem a programem samotným. Jak dále uvidíme, každá tato skupina jazyků bude korespondovat s jedním hlavním *typickým stylem programování – paradigmatem.*

Poznámka 1.1. Všimněte si, že doposud nepadl pojem *algoritmus.* Ani jsme se nezmínili o tom, jaké *typy problémů* se chystáme řešit (pomocí počítače). Důvodem je fakt, že pojmy „algoritmus“ a „problém“ lze chápat na jednu stranu neformálně (což je poněkud ošidné) a na druhou stranu formálně, což ale vyžaduje hlubší studium problematiky, kterou se v tomto kurzu nezabýváme. Studenti se budou algoritmy, problémy, řešitelností problémů a jejich složitostí zabývat v samostatném kurzu *vyčíslitelnosti a složitosti*. K této poznámce dodejme jen tolik, že výsledky dosažené v teoretické informatice ukázaly několik důležitých faktů, které je potřeba mít neustále na paměti:

(i) zdaleka ne každý problém je řešitelný,

(ii) zdaleka ne každý řešitelný problém lze řešit v přijatelné době,

(iii) všechny programovací jazyky, *které jsou Turingovsky úplné,* mají stejnou vyjadřovací sílu.

Bod (iii) potřebuje detailnější vysvětlení. Programovací jazyk nazýváme *Turingovsky úplný,* pokud je z hlediska řešitelnosti problémů ekvivalentní tak zvanému Turingovu stroji, což je jeden z formálních modelů algoritmu. Všechny běžně používané programovací jazyky jsou Turingovsky úplné (samozřejmě, že lze záměrně vytvořit i jazyk „chudší“, který Turingovsky úplný nebude, ale takovými jazyky se nebudeme zabývat). To tedy znamená, že všechny tyto jazyky jsou vzájemně ekvivalentní z hlediska své výpočetní síly. Neformálně řečeno, je-li nějaký problém možné vyřešit pomocí jednoho z těchto jazyků, je možné jej vyřešit v libovolném z nich. Všechna předchozí pozorování budou zpřesněna v kurzu *vyčíslitelnosti a složitosti.* Pro nás je v tuto chvíli důležité vědět, že z hlediska možnosti řešit problémy jsou si „všechny programovací jazyky rovny“.

Pozorování o rovnocennosti programovacích jazyků má dalekosáhlé důsledky. Říká, mimo jiné, že pokud nemůžeme problém vyřešit v jednom jazyku, pak nám přechod k jinému nikterak nepomůže, což je mimochodem dost důležitý fakt, o kterém řada „ostřílených programátorů z praxe“ *vůbec neví.* Na druhou stranu je ale potřeba zdůraznit, že mezi programovacími jazyky jsou propastné rozdíly v tom, jaký umožňují programátorovi použít aparát při psaní programu a (v důsledku) při řešení problému. Program, řešící jeden typ problému, může být v jednom programovacím jazyku napsán jednoduše, stručně a elegantně, kdežto v jiném jazyku může být napsání programu generujícího stejný výpočetní proces řádově složitější.

Nyní si stručně popíšeme, jakými programovacími jazyky se vlastně budeme zabývat. Z historického hlediska je možné rozdělit počítače (a metody programování) do dvou etap, z nichž první byla nesrovnatelně kratší než druhá:

Éra neprogramovatelných počítačů. V prvopočátcích vývoje počítačů byly počítače sestaveny jednoúčelově k vůli řešení konkrétního problému. Program byl fyzicky (hardwarově) součástí počítače. Pokud po vyřešení problémů již počítač nebyl potřeba, byl zpravidla demontován a rozebrán na součástky. Příkladem jednoúčelového neprogramovatelného počítače byly tzv. „Turingovy bomby“, což byly počítače vyráběné během válečných let 1939–1945, které byly během II. světové války používány k dešifrování německého kódovacího zařízení Enigma. Počítače společně navrhli britští vědci Alan Turing a Gordon Welchman, k jejich úplnému odtajnění došlo až v 90. letech minulého století.

Éra programovatelných počítačů. Jednorázová konstrukce počítačů byla velmi neekonomická, takže přirozeným vývojem bylo vytvořit univerzální počítač, který by mohl být programovatelný pro provádění různých úloh. Tento koncept se udržel až do současnosti, i když pochopitelně mezi programovatelností a schopnostmi počítačů v 50. letech minulého století a v současnosti je propastný rozdíl. Prvenství mezi programovatelnými počítači bývá přičítáno několika skupinám vývojářů. Snad úplně prvním počítačem v této kategorii byl počítač Z3, který uvedl do chodu v roce 1941 německý inženýr Konrad Zuse (1910–1995). Počítač byl sestrojen na zakázku koncernu BMW, technicky byl proveden pomocí relé, ale byl plně programovatelný. Původní počítač byl zničen během spojeneckého náletu na Berlín v roce 1944 (v roce 1960 byla vyrobena jeho replika). Zajímavé je, že v roce 1998 se podařilo dokázat, že Z3 (respektive jeho strojový jazyk, viz dále) byl Turingovsky úplný. Mezi další klasické zástupce patří britský počítač Colossus (1944), který byl sice konstruovaný jako jednoúčelový, ale byl omezeně programovatelný (účelem sestrojení počítače bylo dešifrování německých kódovacích systémů). Americký počítač ENIAC (1946), sestrojený k výpočtům dělostřeleckých tabulek v rámci amerického balistického výzkumného centra, byl programovatelný, pracoval však (na dnešní dobu netypicky) v desítkové soustavě. Počítač Z3 byl svou koncepcí mnohem blíže soudobým počítačům, protože pracoval v dvojkové soustavě.

Programovací jazyky lze rozdělit na *nižší* a *vyšší* podle toho, jak moc jsou vázány na konkrétní hardware počítače a podle toho, jaké *prostředky abstrakce* dávají k dispozici programátorovi.

Nižší programovací jazyky jsou těsně vázané na hardware počítače. Programové konstrukce, které lze v nižších jazycích provádět jsou spjaty s instrukční sadou procesoru. Nižší programovací jazyky neobsahují prakticky žádné prostředky abstrakce, programování v nich je zdlouhavé, často vede k zanášení chyb do programu, které se velmi špatně odhalují (a někdy i špatně opravují). Naopak výhodou programu napsaného v nižším programovacím jazyku je jeho přímá vazba na hardware počítače, což umožňuje „plně počítač využít“. V následujícím přehledu jsou uvedeni typičtí zástupci *nižších programovacích jazyků.*

Kódy stroje. Kódy stroje jsou historicky nejstarší. Představují vlastně programovací jazyky dané přímo konkrétním počítačem respektive jeho centrálním procesorem. Program v kódu stroje je de facto *sekvence jedniček a nul,* ve které jsou kódované instrukce pro procesor a data, nad kterými jsou instrukce prováděny. Kód stroje je vykonáván přímo procesorem. To jest, výpočetní proces vzniká z kódu stroje tím, že jej procesor postupně vykonává. Příklad programu v jazyku stroje (nějakého fiktivního počítače) by mohl vypadat takto:

`001`	`01000`	vlož číslo 8 do paměťové buňky
`011`	`00101`	přičti k obsahu buňky číslo 5
`100`	`00000`	vytiskni obsah buňky na obrazovku
`111`	`00000`	ukonči program

V tomto případě je každá instrukce i s hodnotou, se kterou je provedena (s tak zvaným *operandem*) kódována osmi bity. Přitom první tři bity tvoří kód instrukce a zbylých pět bitů je zakódovaný operand. Například v prvním řádku je „`001`“ kódem instrukce pro vkládání hodnoty do paměťové buňky (registru) a „`01000`“ je binární rozvoj čísla 8. Celý výše uvedený program zapsaný jako sekvence nul a jedniček tedy vypadá následovně: „`0010100001100101100000001110000`0“. Podle výše uvedeného slovního popisu se vlastně jedná o program, který sečte čísla 8 a 5 a výsledek vytiskne na obrazovku. Na to, že se jedná o primitivní program, je jeho zápis velmi nepřehledný a pouhá záměnou jedné nuly za jedničku nebo obráceně způsobí vznik chyby.

Assemblery. Assemblery vznikly jako pomůcka pro vytváření programů v kódu stroje. V programech psaných v assembleru se na rozdíl od kódu stroje používají pro zápis instrukcí jejich mnemotechnické (snadno zapamatovatelné) *zkratky.* Rovněž operandy není nutné uvádět pomocí jejich binárních rozvojů: například čísla je možné zapisovat v běžné dekadické notaci a na paměťové buňky (registry) se lze odkazovat jejich zkratkami. Dalším posunem oproti kódu stroje je možnost symbolického adresování. To znamená, že při použití instrukce skoku „skoč na jiné místo programu“ se nemusí ručně počítat adresa, na kterou má být skočeno, ale assemblery umožňují pojmenovat cíl skoku pomocí *návěstí.* Předchozí program v kódu stroje by v assembleru pro příslušný fiktivní počítač mohl vypadat následovně:

```
load  8     vlož číslo 8 do paměťové buňky
add   5     přičti k obsahu buňky číslo 5
print       vytiskni obsah buňky na obrazovku
stop        ukonči program
```

Tento program je již výrazně čitelnější než sekvence jedniček a nul. Přesto se programování v assembleru v současnosti omezuje prakticky jen na programování (částí) operačních systémů nebo programů, které jsou extrémně kritické na výkon. Stejně jako v případě kódu stroje jsou však programy v assembleru náchylné ke vzniku chyb a psaní programů v assembleru je náročné a zdlouhavé.

Programy v assembleru jsou „textové soubory“ se symbolickými názvy instrukcí. Nejedná se tedy přímo o kód stroje, takže procesor přímo nemůže program v assembleru zpracovávat. Při programování v assembleru je tedy nutné mít k dispozici překladač (kterému se říká rovněž *assembler*), který převede *program v čitelné zdrojové formě* do *kódu stroje,* viz předchozí ukázky. Vygenerovaný kód stroje je již přímo zpracovatelný počítačem. Důležité je uvědomit si, že samotný překladač je rovněž *program,* který „nějak pracuje“ a někdo jej musí vytvořit.

Autokódy, bajtkódy, a jiné. Existují další představitelé nižších programovacích jazyků, například *autokódy,* které se během vývoje počítačů příliš nerozšířily, některé z této třídy jazyků poskytují programátorovi prvky, které již lze z mnoha úhlů pohledu považovat za rysy vyšších jazyků. Stejně tak jako assemblery, programy zapsané v těchto jazycích musí být přeloženy do kódu stroje. *Bajtkódy* (anglicky *bytecode*) lze chápat jako jazyky stroje pro *virtuální procesory* (fyzicky neexistující). Abychom mohl být na základě programu v bajtkódu generován výpočetní proces, je potřeba mít opět k dispozici buď překladač bajtkódu do kódu stroje nebo program, který přímo provádí virtuální instrukce uvedené v bajtkódu.

Vyšší programovací jazyky naopak (od těch nižších) nejsou těsně vázané na hardware počítače a poskytují programátorovi výrazně vyšší stupeň abstrakce než nižší programovací jazyku. To umožňuje, aby programátor snadněji a rychleji vytvářel programy s menším rizikem vzniku chyb.

Čtyři nejstarší vyšší programovací jazyky, které doznaly většího rozšíření, byly (respektive jsou):

ALGOL První variantou byl ALGOL 58, který byl představen v roce 1958. Původně se jazyk měl jmenovat IAL (International Algorithmic Language), ale tato zkratka byla zavržena, protože se (anglicky) špatně vyslovovala. Autory návrhu jazyka byli A. Perlis a K. Samelson [PeSa58, Pe81]. Mnohem známější varianta jazyka, ALGOL 60, se objevila o dva roky později. ALGOLem bylo inspirováno mnoho dalších jazyků včetně PASCALu. Samotný ALGOL byl navržen jako obecný „algoritmický jazyk“ pro

vědecké výpočty.

COBOL Název COBOL je zkratkou pro „Common Business Oriented Language“, což naznačuje, jaký byl účel vyvinutí tohoto jazyka. Jazyk COBOL vznikl z podnětu ministerstva obrany Spojených států jako obecný jazyk pro vytváření obchodních aplikací. Jedno z hlavních setkání, na kterých byl jazyk ustaven se konalo v Pentagonu v roce 1959. COBOL je v současnosti jedním z nejpoužívanějších jazyků pro programování obchodních a ekonomických aplikací. I když je vývoj nových programů na ústupu, stále existuje velké množství programů v COBOLu, které se neustále udržují.

FORTRAN Jazyk FORTRAN (původní název jazyka byl „The IBM Mathematical Formula Translating System“) byl navržen k snadnému programování numerických výpočtů. Navrhl jej v roce 1953 J. Backus. FORTRAN doznal během svého více než padesátiletého vývoje velkou řadu změn. Poslední revize standardu jazyka byla provedena v roce 1995 (FORTRAN 95).

LISP První návrh jazyka LISP (z List Processing) vznikl jen o něco později než návrh jazyka FORTRAN. Autorem návrhu byl J. McCarthy [MC60]. LISP byl původně navržen také jako jazyk pro vědecké výpočty, ale na rozdíl od FORTRANu, který byl numericky založený a umožňoval pracovat pouze s čísly, byl LISP již od počátku orientován na práci se symbolickými daty. Na rozdíl od FORTRANu byl LISP vyvíjen v široké komunitě autorů a nebyla snaha jej standardizovat. V současnosti se pod pojmem LISP rozumí celá rodina jazyků, jejichž dva nejvýznamnější zástupci jsou jazyky Common LISP a Scheme.

Otázkou je, zda-li můžeme nějak srozumitelně podchytit, co mají vyšší programovací jazyky společného (zatím jsme se vyjadřovali v dost obecných pojmech jako byla například „abstrakce“ a „přenositelnost“). Odpověď na tuto otázku je v podstatě záporná, obecných rysů je velmi málo, protože vyšších programovacích jazyků je velké množství. Každý vyšší programovací jazyk by ale měl dát programátorovi k dispozici:

(i) *primitivní výrazy* (například čísla, symboly, . . .),

(ii) *prostředky pro kombinaci* primitivních výrazů do složitějších,

(iii) *prostředky pro abstrakci* (možnost pojmenování složených výrazů a možnost s nimi dále manipulovat).

Každý vyšší programovací jazyk má však svou vlastní technickou realizaci předchozích bodů. To co je v jednom jazyku primitivní výraz již v druhém jazyku být primitivní výraz nemusí a tak podobně. Programovací jazyky se jeden od druhého liší stylem zapisování jednotlivých výrazů i jejich významem, jak ukážeme v další sekci.

Stejně tak jako v případě assemblerů, programy napsané ve vyšších programovacích jazycích jsou psány jako textové soubory a pro vygenerování výpočetního procesu na základě daného programu je potřeba provést buďto jejich *překlad* nebo *interpretaci*:

Interpretace. Interpret programovacího jazyka je program, který čte výrazy programu a postupně je přímo vykonává. Interprety tedy (obvykle) z daného programu neprodukují kód v jazyku stroje.

Překlad přes nižší programovací jazyk. Překladač (*kompilátor*; anglicky *compiler*) *programovacího jazyka* je program, který načte celý program a poté provede jeho překlad do některého nižšího programovacího jazyka, typicky do *assembleru* nebo do nějakého *bajtkódu*. V tomto případě rozlišujeme tři situace:

(a) překlad byl proveden do *kódu stroje*, pak překlad končí, protože jeho výsledkem je již program zpracovatelný procesorem, který může být přímo spuštěn;

(b) překlad byl proveden do *assembleru*, pak je potřeba ještě dodatečný překlad do kódu stroje, který obvykle provádí rovněž překladač;

(c) překlad byl proveden do *bajtkódu*, pak překlad končí. Bajtkód ale není přímo zpracovatelný procesorem, musí tedy být ještě *interpretován* nebo *přeložen* (méně časté) dalším programem.

Z konstrukčního hlediska jsou překladače výrazně složitější než interprety, v praxi jsou však častěji používány, protože vykonávání překládaných programů je výrazně rychlejší než jejich interpretace. S neustálým nárůstem výkonu procesorů je však do budoucna možný obrat ve vývoji. Nevýhodou překladačů je, že s výjimkou překladačů do bajtkódu, generují programy přímo pro konkrétní procesory, takže překladače je o něco složitější „přenést“ mezi dvěma počítačovými platformami. U překladu do bajtkódu tento problém odpadá, ale za cenu, že výsledný bajtkód je nakonec interpretován (pomalé).

Překlad přes vyšší programovací jazyk. Situace je analogická jako při překladu do nižšího jazyka, pouze teď překladače překládají programy do některého cílového vyššího programovacího jazyka. Typickou volbou bývá překlad do některého z cílových jazyků, jejichž překladače jsou široce rozšířené. Takovým jazykem je například jazyk C. Pro použití jazyka C jako cílového jazyka překladu také hovoří jeho jednoduchost, jasná specifikace a přenositelnost. Překladače tohoto typu (překladače generující programy ve vyšším programovacím jazyku) jsou z hlediska složitosti své konstrukce srovnatelné s interprety. Oproti interpretaci je výhoda ve větší rychlosti vykonávání výsledného kódu. Nevýhodou je, že je potřeba mít k dispozici ještě překladač cílového jazyka, kterým je z programu v cílovém jazyku nakonec sestaven výsledný kód stroje.

Z pohledu předchozího rozdělení je dobré zdůraznit, že překlad ani interpretace se pevně neváže k programovacímu jazyku. Pro jeden programovací jazyk mohou existovat jak překladače, tak interprety. Rozdíly mezi interprety a překladači se někdy smývají. Některé přeložené kódy v sobě mají interpretační prvky. Naopak, některé interprety programovacích jazyků dokážou přeložit některé části programů s vykonávat je jako kód stroje. My se během dalšího výkladu budeme zabývat většinou interpretací, protože je z technického pohledu jednodušší.

V následujícím textu a v navazujících dílech cvičebnice se budeme zabývat pouze vyššími programovacími jazyky. Nižší programovací jazyky budou rozebrány v rámci kurzu *operačních systémů.* Na závěr úvodní sekce podotkněme, že přesun od nižších k vyšším programovacím jazykům nastal velmi brzy po zkonstruování programovatelných počítačů, protože jejich programátoři si dobře uvědomovali, jak bylo programování v nižších jazycích komplikované. Například již Konrad Zuse pro svůj počítač Z3 navrhl vyšší programovací jazyk zvaný Plankalkül [Gi97, Ro00, Zu43, Zu72]. Ačkoliv k jazyku ve své době neexistoval překladač, je nyní Plankalkül považován za *nejstarší vyšší programovací jazyk* (navržený pro fyzicky existující počítač). Konstrukce prvních překladačů si na počátku éry vyšších programovacích jazyků vyžádala obrovské úsilí. Z dnešního pohledu je konstrukce překladačů rutinní záležitost, kterou by měl (alespoň rámcově) znát každý informatik.

Vyšší programovací jazyky s sebou přinesly nové (a původně netušené) možnosti. Brzy se zjistilo, že části některých programů jsou napsány natolik obecně, že je lze používat v jiných programech. To vedlo velmi brzy k vytváření *knihoven* – speciálních částí programů, které obsahovaly předprogramované funkce, které mohly být programátory ihned používány. Další milník padl ve chvíli, kdy se podařilo použít vyšší programovací jazyky pro naprogramování operačních systémů (jazyk C a operační systém UNIX), protože ještě dlouho po nástupu vyšších programovacích jazyků se věřilo, že nejsou pro psaní operačních systémů (a základního softwarového vybavení počítačů) vhodné.

1.2 Syntax a sémantika programů

Při úvahách o programech zapsaných v programovacích jazycích musíme vždy důsledně odlišovat dva úhly pohledu na tyto programy. Prvním je jejich *syntaxe* čili způsob nebo tvar, ve kterém se programy zapisují. Druhým úhlem pohledu je *význam programů,* neboli jejich *sémantika.* Sémantiku někdy nazýváme *interpretace,* ačkoliv my tento pojem nebudeme používat, protože koliduje s pojmem *interpretace* představeným v sekci 1.1, jehož význam je „vykonávání programu“ (což je význam posunutý trochu někam jinam). Syntaxe a sémantika jsou dvě různé složky, které v žádném případě nelze zaměňovat nebo ztotožňovat. Důvod je v celku zřejmý, pouhý tvar, ve kterém zapisujeme programy nic neříká o jejich významu, jak blíže ukážeme v následujících příkladech. Důrazné odlišování syntaxe a sémantiky není jen věcí programovacích jazyků, ale je běžně používáno v *logice, teoretické informatice* i v aplikacích, například při konstrukci *analyzátorů textu, elektronických slovníků* nebo *překladačů.*

Syntax programovacího jazyka je souborem přesně daných pravidel definujících jak zapisujeme programy v daném programovacím jazyku. Syntax většiny programovacích jazyků je popsána v jejich standardech

pomocí takzvaných *formálních gramatik*, případně jejich vhodných rozšíření, například *Backus-Naurovy formy*. Obecný popis problematiky je obsahem kurzu *formálních jazyků a automatů*, viz také [Ho01, Ko97].

V sekci 1.1 jsme uvedli, že každý vyšší programovací jazyk obsahuje primitivní výrazy a prostředky pro jejich kombinaci. Syntaxe programovacího jazyka z tohoto pohledu popisuje, jak vypadají primitivní výrazy a vymezuje jakým způsobem lze z jednodušších výrazů konstruovat složitější.

Příklad 1.2. Uvažujme smyšlený programovací jazyk, ve kterém můžeme, kromě jiného, zapisovat datum. Můžeme provést úmluvu, že povolená syntaxe data bude ve tvaru

$$DD/DD/DDDD,$$

kde každé D představuje cifru z množiny $\{0, \ldots, 9\}$. Například tedy výrazy $26/03/1979$ a $17/11/1989$ odpovídají tomuto tvaru. Stejně tak ale třeba i výrazy $00/00/0000$, $01/99/6666$, $54/13/1002$ a podobně, odpovídají smluvenému tvaru. V tuto chvíli bychom se neměli nechat splést tím, že poslední tři výrazy nereprezentují žádné „skutečné datum“; doposud jsme se pouze bavili o syntaxi data (jeho zápisu), nikoliv jeho sémantice (jeho významu). Na tomto příkladu můžeme taky dobře vidět, že pouhý „smluvený tvar“, ve kterém datum zapisujeme nic neříká o jeho konkrétní hodnotě. Vezmeme-li si výraz $03/11/2006$, pak by bylo dost dobře možné přisoudit mu dva zcela různé významy:

(a) 3. listopadu 2006 (první údaj v $DD/DD/DDDD$ značí číslo dne),

(b) 11. března 2006 (první údaj v $DD/DD/DDDD$ značí číslo měsíce).

Tímto příkladem jsme chtěli demonstrovat, že tvar (částí) programů nelze směšovat s jeho významem, ani z něj jeho význam nelze nijak „přímočaře určit“.

Poučení, které plyne z předchozího příkladu, je v podstatě následující: každý programovací jazyk musí mít popsánu svojí syntax a sémantiku. Pokud se v daném jazyku budeme snažit psát programy, měli bychom se vždy se syntaxí a sémantikou daného jazyka seznámit. Předejdeme tak nebezpečným chybám, které by v programu mohly vzniknout díky tomu, že jsme předpokládali, že „něco se *píše/chová* jinak, než jak jsme mysleli (než na co jsme byli zvyklí u jiného jazyka)“. V následujícím příkladu ukážeme, jakých rozdílných významů nabývá tentýž výraz z pohledu sémantiky různých programovacích jazyků.

Příklad 1.3. Uvažujme výraz ve tvaru

$$x = y + 3.$$

Tento výraz lze chápat jako část programů zapsaných v různých programovacích jazycích. V následujícím přehledu uvedeme několik odlišných významů tohoto výrazu tak, jak jej definují vybrané programovací jazyky.

(a) V jazycích jakými jsou C, C++ a další by měl výraz „$x = y + 3$“ význam *příkazu přiřazení*. Význam výrazu bychom mohli konkrétně vyjádřit takto: „do paměťového místa označeného identifikátorem x je zapsána hodnota vzniklá zvětšením hodnoty uložené v paměťovém místě označeném identifikátorem y o tři.“ Symboly x a y tedy hrají úlohy *označení paměťových míst* a výše uvedený výraz je příkaz přiřazení hodnoty vypočtené pomocí obsahu jednoho paměťového místa do jiného paměťového místa.

(b) V jazycích jako jsou Algol 60 a Pascal je symbol „=“ vyhrazen pro *porovnávání číselných hodnot*. V tomto případě má „$x = y + 3$“ význam „porovnej, zda-li je hodnota uložená v x rovna hodnotě uložené v y zvětšené o tři.“ Oproti předchozímu případu je tedy výsledkem zpracování výrazu „$x = y + 3$“ pravdivostní hodnota *pravda* (a to v případě, že rovnost skutečně platí) nebo *nepravda* (rovnost neplatí). K žádnému přiřazení hodnot (mezi paměťovými místy) nedochází.

(c) V jazyku METAPOST, což je speciální jazyk vyvinutý pro kreslení vektorových schémat (takzvaných pérovek), má výraz „$x = y + 3$“ význam *příkazu pro řešení lineární rovnice* nebo soustavy (dříve uvedených) lineárních rovnic. Výraz „$x = y + 3$“ uvedený v programu má význam „pokus se řešit rovnici $x = y + 3$“. Pokud by například hodnota x byla známá, pak by po uvedení tohoto výrazu došlo k výpočtu hodnoty y, což by bylo $x - 3$. To je významný rozdíl oproti příkazu přiřazení, viz

bod (a), protože v tomto případě došlo k dopočtení hodnoty y a hodnota x (levá strana rovnosti) se nezměnila. Pokud by byly známé obě hodnoty x i y, pak by došlo ke kontrole konzistence rovnice (pokud by neplatila rovnost, výpočet by končil chybou: „neřešitelná rovnice nebo soustava rovnic“). Pokud by ani jedna z proměnných x a y nebyla známá, pak by byla rovnice pouze „zapamatována“ a byla by použita při pokusu o nalezení řešení při výskytu další rovnice.

(d) V logických programovacích jazycích, jakým je například jazyk PROLOG, by měl výraz „$x = y + 3$“ význam *deklarace,* že hodnota navázaná na proměnnou x musí být ve stejném tvaru jako $y + 3$. Taková podmínka je splněna například v případě, kdy na x bude navázána hodnota $f(z) + 3$ a na y by byla navázána hodnota $f(z)$, protože pak by po dosazení $f(z) + 3$ za x do „$x = y + 3$“ a po následném dosazení $f(z)$ za y do „$f(z) + 3 = y + 3$“ přešel původní výraz „$x = y + 3$“ do tvaru „$f(z)+3 = f(z)+3$“. Uvědomme si, že deklarace stejného tvaru dvou výrazů v tomto smyslu je něco jiného než řešení lineární rovnice z bodu (c). V jazyku PROLOG musí být přísně vzato proměnné x a y značeny velkými písmeny, ale když si odmyslíme tento detail, tak máme dohromady čtyři různé interpretace téhož výrazu.

V předchozím příkladu jsme viděli různé interpretace stejného výrazu. Platí samozřejmě i opačná situace a to ta, že konstrukce s jedním významem se v různých programovacích jazycích zapisuje různě. Tento jev je dobře pozorovatelný například na aritmetických výrazech, což ukazuje následující příklad.

Příklad 1.4. Uvažujme aritmetický výraz $2 \cdot (3 + 5)$ čili „vynásob dvěma součet tří a pěti“. V každém rozumném programovacím jazyku je výsledkem zpracování takového výrazu hodnota 16. Jazyky se ale leckdy liší v samotném zápisu výrazu $2 \cdot (3 + 5)$. Jedním ze zdrojů rozdílů je *pozice symbolů označujících operace* „$\cdot$“ a „$+$“ vůči *operandům* čili podvýrazům, se kterými operace provádíme. Následující přehled obsahuje čtyři typické zápisy výrazu $2 \cdot (3 + 5)$ v programovacích jazycích.

`2 * (3 + 5)` Tento tvar je v souladu s běžným psaním výrazů, symboly pro operace stojí mezi operandy se kterými operace chceme provádět. Proto tomuto zápisu říkáme *infixová notace.* Při psaní výrazů v infixové notaci hrají důležitou roli *priority operací* a jejich *asociativita.* Drtivá většina jazyků, například jazyky C a C++, umožňují psát pouze `2 * 3 + 4` místo `(2 * 3) + 4`, protože symbol „`*`“ označující násobení má vyšší prioritu než symbol „`+`“ označující sčítání. Stejně tak lze psát `2 + 3 + 4` místo `2 + (3 + 4)`, což je zaručeno tím, že operace „`+`“ je asociativní. Infixový zápis s sebou z hlediska konstrukce interpretů a překladačů programovacích jazyků přináší komplikace, protože u výrazů je potřeba správně rozpoznávat prioritu a asociativitu operací. Další technickou komplikací je použití operací, které mají víc jak dva argumenty, které již nelze psát „čistě infixově“, protože „jeden symbol“ nelze napsat „mezi tři operandy“. Například jazyky C, C++, PERL a další mají *ternární operátor*, který se zapisuje ve tvaru „$\langle operand_1 \rangle$ `?` $\langle operand_2 \rangle$ `:` $\langle operand_3 \rangle$“, tedy pomocí dvou různých symbolů vložených mezi operandy.

`(* 2 (+ 3 5))` Při použití této notace, takzvané *prefixové notace,* píšeme operaci před všechny operandy, samotné operandy jsou od operace a mezi sebou odděleny mezerami, nebo obecněji „bílými znaky“ (anglicky *whitespace*). Za bílé znaky se v informatickém žargonu považují libovolné sekvence mezer, tabelátorů a nových řádků (případně jiných speciálních znaků nemajících čitelnou reprezentaci). Každý výraz v prefixové notaci skládající se z operace a operandů je navíc celý uzavřen do závorek. Prefixová notace má proti infixové notaci jednu nevýhodu – a tou je počáteční nezvyk programátorů. Na druhou stranu má prefixová notace velké množství výhod, které oceníme postupně v dalších lekcích. Nyní podotkněme, že notace je velmi jednoduchá a nemá v ní například smysl řešit prioritu operací. Jednoduše lze také psát operace s libovolným počtem operandů, například výraz `(+ 2 3 5)` je „součet tří operandů“, `(+ (* 2 3) 5)` je „součet dvou operandů“, dále `(`*op*` (+ 1 2) 3 4 5)` je „operace *op* provedená se čtyřmi operandy `(+ 1 2)`, `3`, `4` a `5`“ a podobně.

`(2 (3 5 +) *)` Tato notace se nazývá *postfixová* a je analogická notaci prefixové s tím rozdílem, že operace se píše až za všechny operandy. Dále platí pravidlo o oddělování operandů od operace a mezi sebou a pravidlo o uzávorkování vnějších výrazů jako u prefixové notace. Tato notace je ze všech notací nejméně používaná.

`2 3 5 + *` Poslední uvedená notace se vyznačuje tím, že operace se píší opět za operandy, ale každá operace má pevný počet přípustných operandů, tím pádem je možné bez újmy vynechat ve výrazu

všechny závorky. Notace se nazývá *reverzní bezzávorková notace*, nebo též *polská bezzávorková notace*, protože ji proslavil polský logik Jan Lukasiewicz. Výrazy v tomto tvaru lze jednoduše načítat a vyhodnocovat. I přes svou zdánlivou těžkou čitelnost je bezzávorková notace poměrně oblíbená a je na ní založeno několik programovacích jazyků, například jazyk PostScript – specializovaný grafický jazyk, který umí interpretovat většina (lepších) tiskáren a jiných výstupních zařízení počítačů.

Při vytváření programů musí mít každý programátor neustále na paměti syntax a sémantiku jazyka. Syntax proto, aby věděl, jak má zapsat danou číst kódu. Sémantiku proto, aby vždy znal přesný význam toho, co píše a aby uměl rozhodnout, zda-li má právě vytvářený program skutečně *zamýšlený význam.*

I přes veškerou snahu se nelze vyhnout vzniku chyb. Chyby při vytváření programů lze obecně rozdělit na chyby syntaktické a sémantické. *Syntaktická chyba* je chybou v zápisu programu. Kód, který není z pohledu daného jazyka syntakticky správný, nelze v podstatě chápat ani jako program v daném jazyku. Syntaktické chyby lze odhalit poměrně snadno. Překladače programovacích jazyků odhalí syntaktické chyby již během překladu. Interprety programovacích jazyků odhalí syntaktické chyby až v momentě, kdy se interpret (neúspěšně) pokusí načíst chybný vstupní výraz. V případě překladačů i interpretů tedy k odhalení syntaktické chyby dochází ještě před započetím vykonávání programu (který v případě syntaktické chyby de facto programem není).

Příklad 1.5. Kdybychom se vrátili k příkladu 1.2, pak výraz $12/3/2003$ je syntakticky chybným protože prostřední složka „3“ má jen jednu cifru. Stejně tak by byly syntakticky chybné výrazy $24/k6/1234$, $12/06/20/03$ a $12–04–2003$. Zdůvodněte proč.

Druhou kategorií chyb jsou *sémantické chyby.* Jak již název napovídá, jedná se o chyby ve významu programu. Mezi typické sémantické chyby patří například provádění operací nad daty neslučitelných typů, například sčítání čísel a řetězců znaků (neformálně řečeno, jde o míchání jablek a hrušek, neboli o klasické Klausovské „vyrábění kočkopsa“). Sémantické chyby lze odhalit během překladu či interpretace pouze v omezené míře. Například překladače programovacího jazyka C jsou během překladu schopny zjistit všechny konflikty v typech dat se kterými program pracujeme. Technicky ale již třeba není možné při překladu vyloučit chyby způsobené „dělením nulou,“ protože hodnota dělitele může být proměnlivá a může během činnosti programu nabývat různých hodnot.

Překladače některých jazyků, natož jejich interprety, neumějí rozpoznat konflikty v typech předem (to jest, třeba během překladu nebo před započetím interpretace). Tento jev bychom však neměli chápat jako „chybu“ nebo „nedokonalost“ programovacího jazyka, ale jako jeho *rys* – v některých programovacích jazycích prostě dopředná analýza typů není možná a programátor musí vytvářet program s ohledem na to, že za něj nikdo „typovou kontrolu neprovádí “.

Chyba projevující se až za běhu programu se anglicky označují jako *run-time error*. Asi není třeba příliš zdůrazňovat, že se jedná o vůbec nejnebezpečnější typ chyb, protože chyby způsobené až za běhu programu zpravidla:

(a) mohou být delší dobu *latentní,* tedy programátor si jich nemusí vůbec všimnout,

(b) obvykle způsobí havárii či nekorektní činnost programu,

(c) jejich odstraňování je náročné z hlediska lidských i finančních zdrojů.

Chyby, které se někdy dlouho neprojeví, viz bod (a), jsou často způsobeny podceněním triviálních případů. To například znamená, že při načítání dat ze souboru programátor zapomene ošetřit případ, kdy soubor na disku sice existuje ale je prázdný. Programátor může například ihned po otevření souboru načítat data aniž by zkontroloval, zda-li tam nějaká jsou, což může posléze vést k chybě. Jako odstrašující příklad k bodům (b) a (c) můžeme uvést příklad vesmírné sondy Mars Climate Orbiter (MCO), která v září 1999 shořela v atmosféře Marsu vlivem chyby způsobené neprovedením přepočtu vzdálenosti v kilometrech na míle (škoda vzniklá touto „chybou za běhu výpočtu“ byla cca 125 milionů USD).

1.3 Přehled paradigmat programování

Vyšší programovací jazyky lze rozdělit do kategorií podle toho, jakým styl (paradigma) při programování v daném jazyku převládá. Znalost programovacího stylu je důležitá z pohledu vytváření kvalitních programů, protože pouze na základě dobré *vnitřní kvality programu* (to jest toho, jak je program napsán) lze dosáhnout požadované *vnější kvality programu* (to jest toho, jak se program chová z uživatelského pohledu). Za základní dvě kvality programů jsou někdy považovány:

- *korektnost* (program reaguje správně na správné vstupy),
- *robustnost* (program se umí vyrovnat i se špatnými vstupy).

Robustnost je tedy silnější vlastnost a zaručuje, že daný program nehavaruje v případě, kdy dostane na svůj vstup špatná data (třeba data, která nejsou v požadovaném formátu), ale umí danou situaci ošetřit (třeba požádáním o zadání nového vstupu, nebo automatickou opravou špatných dat).

Upozorněme nyní, jak je potřebovat chápat programovací paradigmata ve vztahu k programovacím jazykům. Paradigma je de facto styl, kterým lze programovat v jakémkoliv jazyku, který jej podporuje (umožňuje). Některé programovací jazyky jsou „šité na míru“ jednomu paradigmatu a nepředpokládá se, že by v nich programátoři chtěli pracovat jiným stylem (i když i to je možné). Vezmeme-li si jako příklad *procedurální paradigma* (viz dále), pak například jazyk PASCAL byl vytvořen s ohledem na to, že se v něm bude programovat procedurálním stylem. Proto taky někdy říkáme, že PASCAL je *procedurální jazyk.* Tuto terminologii budeme používat i nadále.

Základních paradigmat programování je několik. Nejprve si uvedeme skupinku tří paradigmat, ke kterým existují teoreticky dobře propracované formální modely.

procedurální Procedurální paradigma je jedním ze dvou nejstarších paradigmat programování, někdy též bývá nazýváno paradigmatem *imperativním.* Programy napsané v procedurálním stylu jsou tvořeny *sekvencemi příkazů,* které jsou postupně vykonávány. Klíčovou roli při programování v procedurálním stylu hraje *příkaz přiřazení* (zápis hodnoty na dané místo v paměti) a související *vedlejší efekt* (některým typem přiřazení může být modifikována datová struktura a podobně). Sekvenční styl vykonávání příkazů lze částečně ovlivnit pomocí *operací skoků* a *podmíněných skoků.* Formálním modelem procedurálních programovacích jazyků je *von Neumannův RAM stroj,* viz [vN46]. Z hlediska stylu programování ještě procedurální paradigma můžeme jemněji rozdělit na:

naivní Naivní paradigma bývá někdy chápané jako „samostatné paradigma“, častěji se mezi paradigmata programování ani neuvádí. Naivní procedurální jazyky se vyznačují jakousi všudypřítomnou chaotičností, mají obvykle nesystematicky navrženou syntaxi a sémantiku, v některých rysech jsou podobné nestrukturovaným procedurálním jazykům, viz dále. Mezi typické zástupce patří programovací jazyky rodiny BASIC (první jazyk BASIC navrhli J. Kemeny a T. Kurtz v roce 1968 za účelem zpřístupnit počítače a programování mimo komunitu počítačových vědců).

nestrukturované Nestrukturované procedurální paradigma je velmi blízké assemblerům, programy jsou skutečně lineární sekvence příkazů a skoky jsou v programech realizovány příkazem typu „GO TO“, to jest „jdi na řádek.“ V raných programovacích jazycích držících se tohoto stylu byly navíc všechny řádky programu číslované a skoky šlo realizovat pouze na základě uvedení konkrétního čísla řádku, což s sebou přinášelo velké komplikace. Později se objevily příkazy skoku využívající návěstí. Typickými zástupci v této kategorii byly rané verze jazyků FORTRAN a COBOL.

strukturované Ve svém článku [Di68] upozornil E. Dijkstra na nebezpečí spjaté s používáním příkazu „GO TO“. Poukázal zejména na to, že při používání příkazu „GO TO“ je prakticky nemožné ladit program, protože struktura programu nedává příliš informací o jeho vykonávání (v nějakém bodě výpočtu). Strukturované procedurální paradigma nahrazuje příkazy skoku pomocí podmíněných cyklů typu „opakuj $\sim$ dokud platí podmínka“ a jiných strukturovaných konstrukcí, které se do sebe *vnořují.* Typickými zástupci programovacích jazyků z této rodiny jsou jazyky C (D. Ritchie, B. Kernighan, 1978), PASCAL (N. Wirth, 1970), Modula-2 (N. Wirth, 1978), a ADA (J. Ichbiah a kolektiv, 1977–1983).

funkcionální Funkcionální paradigma je zhruba stejně staré jako paradigma procedurální. Programy ve funkcionálních jazycích sestávají ze symbolických výrazů. Výpočet je potom tvořen *postupnou aplikací funkcí,* kdy funkce jsou aplikovány s hodnotami, které vznikly v předchozích krocích aplikací, výsledné hodnoty aplikace jsou použity při aplikaci dalších funkcí a tak dále. Při čistém funkcionálním programování nedochází k vedlejším efektům, příkaz přiřazení se nepoužívá. Díky absenci vedlejších efektů je možné chápat programy ve funkcionálních jazycích jako *ekvacionální teorie* [Wec92]. Rovněž se nepoužívají příkazy skoku ani strukturované cykly. Cykly se nahrazují *rekurzivními funkcemi* a *funkcemi vyšších řádů.* V některých funkcionálních jazycích také hraje důležitou roli *líné vyhodnocování* (jazyk Haskell, 1987) a *makra* (zejména dialekty LISPu). Nejznámější teoretický model funkcionálního programování je λ-kalkul (A. Church, 1934, viz [Ch36, Ch41]), i když existují i modely jiné, například modely založené na uspořádaných algebrách (přehled lze nalézt ve [Wec92]). Typickými zástupci funkcionálních programovacích jazyků jsou Common LISP, Scheme, ML a Anfix (dříve Aleph). Mezi čistě funkcionální jazyky patří Miranda, Haskell, Clean a další.

logické Teoretické základy logického programování byly položeny v článcích [Ro63, Ro65] J. Robinsona popisujících *rezoluční metodu.* Později byla teorie logických jazyků dále rozpracována a zkonstruovány byly první překladače logických jazyků, viz také [Ll87, NS97, NM00, SS86]. Pro logické paradigma je charakteristický jeho *deklarativní charakter*, při psaní programů programátoři spíš popisují problém, než aby vytvářeli předpis, jak problém řešit (to samozřejmě platí jen do omezené míry). Programy v logických programovacích jazycích lze chápat jako *logické teorie* a výpočet lze chápat jako *dedukci důsledků,* které z této teorie plynou. Formálním modelem logického programování jsou speciální inferenční mechanismy, jejichž studium je předmětem *matematické logiky.* Programy se skládají ze speciálních formulí – tak zvaných *klauzulí.* Výpočet se v logických jazycích zahajuje zadáním cíle, což je existenční dotaz, pro který je během výpočtu nalezena odpověď, nebo nikoliv. Typickým představitelem logického programovacího jazyka je PROLOG (A. Colmerauer, R. Kowalski, 1972). Logické programovací jazyky rovněž nalezly uplatnění v deduktivních databázích, například jazyk DATALOG (H. Gallaire, J. Minker, 1978).

Další dvě významná paradigmata jsou:

objektově orientované Objektové paradigma je založené na myšlence vzájemné *interakce objektů.* Objekty jsou nějaké entity, které mají svůj vnitřní stav, který se během provádění výpočetního procesu může měnit. Objekty spolu komunikují pomocí *zasílání správ.* Objektově orientované paradigma není „jen jedno“, ale lze jej rozdělit na řadu podtypů. Ze všech programovacích paradigmat má objektová orientace nejvíc modifikací (které jdou leckdy myšlenkově proti sobě). Například při vývoji některých jazyků byla snaha odstranit veškeré rysy připomínající procedurální paradigma: cykly byly nahrazeny *iterátory,* podmíněné příkazy byly nahrazeny *výjimkami,* a podobně. Jiné jazyky se této striktní představy nedrží. Prvním programovacím jazykem s objektovými rysy byla Simula (O. J. Dahl a K. Nygaard, 1967). Mezi čistě objektové jazyky řadíme například Smalltalk (A. Kay a kol., 1971) a Eiffel (B. Meyer, 1985). Dalšími zástupci jazyků s objektovými rysy jsou C++, Java, Ruby, Sather, Python, Delphi a C♯.

paralelní Paralelní paradigma je styl programování, ve kterém je kladen důraz na *souběžné* (*paralelní*) *vykonávání* programu. Paralelní paradigma má opět mnoho podtříd a nejde o něm říct příliš mnoho obecně. Jednou z hlavních myšlenek je, že výpočetní proces se skládá z několika souběžně pracujících částí. Programy generující takové výpočetní procesy musí mít k dispozici prostředky pro jejich vzájemnou synchronizaci, sdílení dat, a podobně. Mezi zástupce čistých paralelních jazyků patří například MPD, Occam a SR. Paralelní prvky se jinak vyskytují ve všech moderních programovacích jazycích nebo jsou k dispozici formou dodatečných služeb (poskytovaných operačními systémy).

Předchozí dvě paradigmata jsou významná z toho pohledu, že velká většina soudobých programovacích jazyků v sobě zahrnuje jak objektové tak paralelní rysy. Zajímavé ale je, že ani objektové ani paralelní paradigma za sebou nemají silné formální modely, jako je tomu v případě předchozích tří paradigmat (formálních modelů existuje několik, jsou buď vzájemně neslučitelné, nebo pokrývají jen část problematiky). Další pozoruhodná věc je, že čistě objektové ani paralelní jazyky se (v praxi) de facto vůbec nepoužívají.

Obecně lze říct, že žádné paradigma se prakticky nepoužívá ve své čisté podobě. V dnešní době je typické používat programovací jazyky, které umožňují programátorům programovat více styly. Například jazyky jako jsou C++, Java, C♯, Delphi a jiné jsou procedurálně-objektové, jazyky Anfix a Ocaml jsou funkcionálně-objektové a podobně. Někdy proto hovoříme o *multiparadigmatech.* Čím více stylů programátor během svého života vyzkouší, tím větší bude mít potenciál při vytváření programů.

Naším cílem bude postupně projít všemi paradigmaty a ukázat si jejich hlavní rysy. Máme v zásadě několik možností, jak to udělat. Mohli bychom si pro každé zásadní paradigma zvolit jeden (typický) jazyk, popsat jeho syntax a sémantiku a na tomto jazyku si programovací paradigma představit. Z praktických důvodů se při našem výkladu vydáme jinou cestou. Zvolíme si na začátku jeden multiparadigmový jazyk s vysokým modelovacím potenciálem a na tomto jazyku budeme postupně demonstrovat všechny hlavní styly programování.

Pro naše účely si zvolíme jazyk Scheme. Scheme je dialekt jazyka LISP, který byl částečně inspirován jazykem ALGOL 60. Jazyk Scheme navrhli v 70. letech minulého století Guy L. Steele a Gerald Jay Sussman původně jako experimentální jazyk pro ověření některých vlastností teorie aktorů. Scheme byl inspirovaný LISPem, ale byl na rozdíl od něj výrazně jednodušší. I tak se ukázalo, že v jazyku se dají velmi rychle vytvářet prototypy programů – jazyk se tedy výborně hodil pro experimentální programování. Jazyk byl poprvé úplně popsán v revidovaném reportu [SS78] v roce 1978. Od té doby vznikají další revize tohoto revidovaného reportu, poslední report je „pátý revidovaný report o jazyku Scheme“. Pro revidované reporty se vžilo značení R^nRS, kde „n“ je číslo revize. Současná verze se tedy označuje jako R^5RS, viz [R5RS]. Stav návrhu dalšího revidovaného reportu lze najít v [DC06]. Další návrhy na úpravy a vylepšení jazyka Scheme se shromažďují prostřednictvím tak zvaných SRFI (zkratka pro *Scheme Requests for Implementation*), které lze nalézt na `http://srfi.schemers.org/`. Revidovaným reportům předcházela série článků, které vešly do historie pod názvem „lambda papers“, dva nejvýznamnější z nich jsou [St76, SS76].

Při našem exkurzu paradigmaty programování však nebudeme používat Jazyk Scheme podle jeho současného de facto standardu R^5RS, ale budeme uvažovat jeho vlastní *zjednodušenou variantu.* To nám umožní plně se soustředit na vlastní problematiku jazyka a jeho interpretace. Paradoxně nám jednodušší verze umožní zabývat se i problémy, kterými bychom se v plnohodnotném interpretu zabývat nemohli (nebo by to bylo obtížné). Během našeho zkoumání se na jazyk Scheme budeme dívat z *dvou různých úhlů pohledu*:

- *pohled programátora v jazyku Scheme* – z tohoto úhlu pohledu si budeme postupně představovat jazyk Scheme a budeme si ukazovat typické ukázky programů napsaných v jazyku Scheme; budeme přitom postupovat od jednoduchého ke složitějšímu;
- *pohled programátora interpretu jazyka Scheme* – z tohoto úhlu pohledu budeme vysvětlovat, jak funguje interpret jazyka Scheme. Tento úhel pohledu bývá v literatuře často opomíjen, je však důležitý i z předchozího pohledu. Detailním popisem mechanismu, kterým z programu vzniká výpočetní proces, získá programátor úplný přehled o tom, jak jsou programy vykonávány. Tyto znalosti lze pak využít v mnoha oblastech: při ladění programů, při programování vlastních vyhodnocovacích procesů (nejen v jazyku Scheme) a podobně. Tím, že budeme jazyk představovat od nejjednodušších částí, bude popis jeho interpretace stravitelný i pro čtenáře, kteří nikdy zkušenost s programováním neměli.

V této cvičebnici budeme studovat *funkcionální paradigma.* To je zároveň primární paradigma Jazyka Scheme. Po dvanácti lekcích v této cvičebnici budeme schopni (mimo jiné) naprogramovat vlastní interpret ryze funkcionální podmnožiny jazyka Scheme.

1.4 Symbolické výrazy a syntaxe jazyka Scheme

V této části lekce popíšeme syntaxi jazyka Scheme, to jest způsob, jak zapisujeme programy ve Scheme. Významem (interpretací) programů se budeme zabývat v další části lekce. Jazyk Scheme, stejně tak jako ostatní dialekty LISPu, má velmi jednoduchou syntaxi, zato má velmi bohatou sémantiku. To s sebou přináší několik příjemných efektů:

(a) zápis programů je snadno pochopitelný,

(b) při programování se dopouštíme jen minimálního množství syntaktických chyb,

(c) mechanická kontrola správného zápisu programu je přímočará.

Body (a) a (b) mají praktické dopady při tom, když v jazyku Scheme programujeme. Bod (c) je neméně podstatný, je však zajímavý spíš z pohledu konstrukce překladače/interpretu jazyka Scheme.

Programy v jazyku Scheme se skládají ze *symbolických výrazů*. Nejprve popíšeme jak tyto výrazy mohou vypadat a pak pomocí nich zavedeme pojem program. Neformálně řečeno, za symbolické výrazy budeme považovat čísla a symboly, což jsou základní symbolické výrazy a dále seznamy, což jsou složené symbolické výrazy. Přesná definice následuje:

Definice 1.6 (symbolický výraz).

(i) Každé *číslo* je symbolický výraz
(zapisujeme `12`, `-10`, `2/3`, `12.45`, `4.32e-20`, `-5i`, `2+3i`, a pod.);

(ii) každý *symbol* je symbolický výraz
(zapisujeme `sqrt`, `+`, `quotient`, `even?`, `muj-symbol`, `++?4tt`, a pod.);

(iii) jsou-li $e_1, e_2, \ldots, e_n$ symbolické výrazy (pro $n \geq 1$),
pak výraz ve tvaru `(`$e_1 \sqcup e_2 \sqcup \cdots \sqcup e_n$`)` je symbolický výraz zvaný *seznam*. ■

Poznámka 1.7. (a) Symbolickým výrazům budeme rovněž zkráceně říkat *S-výrazy* (anglicky: *symbolic expression*, případně *S-expression*). Vrátíme-li se opět k primitivním výrazům a prostředkům pro jejich kombinaci, pak čísla a symboly jsou primitivní symbolické výrazy, které lze dále kombinovat do seznamů. Pojem symbolický výraz (S-výraz) se objevuje už v prvním publikovaném návrhu jazyka LISP z roku 1960, popsaném v článku [MC60]. Ačkoliv byl původní LISP vyvinut na počítač IBM 704, popis jazyka a jeho interpretace pomocí S-výrazů umožnil odhlédnout od konkrétního hardware a později snadno implementovat další varianty LISPu na jiných počítačích.

(b) Z důvodů přehlednosti budeme symbolické výrazy zapisovat s barevným odlišením čísel (zelená), symbolů (modrá) a závorek (červená) tak, jak je to vyznačeno v definici 1.6.

(c) Za symboly budeme považovat libovolnou sekvenci znaků neobsahující závorky `(` a `)`, kterou *nelze přirozeně interpretovat jako zápis čísla*. Například tedy `-8.5` chápeme jako číslo „mínus osm celých a pět desetin“, kdežto `-8.5.`, `-8.5.6`, `8-8.5`, `1-` a podobně jsou symboly. Například `2*3` je rovněž symbol, protože se nejedná o sekvenci znaků, kterou bychom chápali jako „zápis čísla“. Pro úplný popis syntaxe čísel a symbolů bychom v tuto chvíli měli správně udělat detailní specifikaci všech přípustných tvarů čísel. Pro účely dalšího výkladu si však vystačíme s tímto intuitivním odlišením čísel a symbolů. Zájemce o podrobnou specifikaci čísel a symbolů v R^{5}RS Scheme odkazuji na specifikaci [R5RS].

(d) V seznamu `(`$e_1 \sqcup e_2 \sqcup \cdots \sqcup e_n$`)` uvedeném v definici 1.6 jsme označovali pomocí „$\sqcup$“ libovolnou, ale neprázdnou, sekvenci bílých znaků, které slouží jako oddělovače jednotlivých prvků seznamů. Číslo n budeme nazývat *délka seznamu* `(`$e_1 \sqcup e_2 \sqcup \cdots \sqcup e_n$`)`. Symbolické výrazy $e_1, \ldots, e_n$ jsou *prvky* seznamu `(`$e_1 \sqcup e_2 \sqcup \cdots \sqcup e_n$`)`. Například `(10ahoj20)` je seznam s jediným prvkem, jímž je symbol `10ahoj20`. Naproti tomu třeba seznam `(10 ahoj20)` má dva prvky, první je číslo `10` a druhým je symbol `ahoj20`. Dále třeba `(10 aho j 20)` má čtyři prvky, první je číslo `10`, druhý je symbol `aho`, třetí je symbol `j` a poslední je číslo `20`. Upozorněme také na fakt, že prvky seznamů mohou být opět seznamy. Například tedy `(a (1 2) b)` je tříprvkový seznam, jehož první prvek je symbol `a`, druhý prvek je dvouprvkový seznam `(1 2)` a třetí prvek je symbol `b`.

(e) Při psaní symbolických výrazů připustíme jednu výjimku pro vynechání bílých znaků mezi prvky seznamů a to kolem prvků, které jsou samy o sobě seznamy. Například v případě seznamu `(a (3 2) b)` bychom mohli psát rovněž `(a(3 2) b)`, `(a (3 2)b)` nebo `(a(3 2)b)` a pořád by se jednalo o týž seznam. Na druhou stranu, `(a(32)b)` už by stejný seznam nebyl, protože druhý prvek `(32)` je nyní jednoprvkový seznam obsahující číslo `(32)`.

Příklad 1.8. Následující výrazy *jsou* symbolické výrazy:

```
-2/3        3.141459      list?              (= 1 2)                 (2+ (3 + 3))
-3+2i       10.2.45       inexact->exact     (* 2 (sqrt (/ 2 3)))    ((10) (20))
-20.        1+            ahoj-svete         (1 + 2)                 (1 (ahoj (3)))
```

Podotkněme, že `-2/3` označuje racionální číslo $-\frac{2}{3}$, `-3+2i` označuje komplexní číslo $-3+2i$. Čísla tvaru `-20.0` můžeme psát zkráceně `-20.` a podobně.

Následující výrazy *nejsou* symbolické výrazy:

`)`	výraz začíná pravou závorkou,
`(+ 1 2))`	výraz má jednu pravou závorku navíc,
`(10 (20 ahoj)`	výrazu chybí jedna pravá závorka,

Všimněte si, že do předchozích výrazů lze vždy doplnit závorky tak, aby se z nich staly symbolické výrazy a to dokonce mnoha způsoby. Například do `(10 (20 ahoj)` můžeme doplnit závorky `(10 (20) ahoj)` nebo `(10 (20 ahoj))` a tak dále.

Nyní můžeme říct, co budeme mít na mysli pod pojmem *program v jazyku Scheme.*

Definice 1.9 (program). Program (v jazyku Scheme) je konečná posloupnost symbolických výrazů. ■

Programy v jazyku Scheme jsou posloupnosti symbolických výrazů. Pokud by byl v posloupnosti obsažen výraz, který by nebyl symbolickým výrazem dle definice, pak daná posloupnost není programem v jazyku Scheme. Z pohledu vzniku chyb se jedná o syntaktickou chybu – špatný zápis části programu.

V další části lekce se budeme zabývat sémantikou symbolických výrazů a s tím související sémantikou programů v jazyku Scheme. Podotkněme, že zatím jsme o významu symbolických výrazů vůbec nic neřekli, dívali jsme se na ně pouze z pohledu jejich zápisu. Na tomto místě je dobré poukázat na odlišnost programů tak, jak jsme je zavedli, od programů v R^{5}RS Scheme, viz [R5RS]. Definice R^{5}RS vymezuje syntaxi symbolických výrazů mnohem přísněji. My si ale plně vystačíme s naší definicí.

1.5 Abstraktní interpret jazyka Scheme

Nejpřesnější cestou popisu sémantiky programu v jazyku Scheme je stanovit, jak interprety jazyka Scheme, případně překladače, zpracovávají symbolické výrazy, protože programy se skládají právě ze symbolických výrazů. Jelikož se budeme během výkladu vždy zabývat především *interpretací,* k popisu sémantiky programu stačí popsat proces, jak jsou symbolické výrazy *postupně vyhodnocovány.* Pokud budeme mechanismus vyhodnocování přesně znát, pak budeme vždy schopni určit (i bez toho aniž bychom měli k dispozici konkrétní interpret jazyka Scheme), jak budou symbolické výrazy zpracovávány, co bude v důsledku daný program dělat, případně jestli při vyhodnocování programu dojde k chybě.

Z praktického hlediska není možné rozebírat zde implementaci nějakého konkrétního interpretu Scheme. Implementace i těch nejmenších interpretů by pro nás zatím byly příliš složité. Místo toho budeme uvažovat „abstraktní interpret Scheme“, který přesně popíšeme, ale při popisu (zatím) odhlédneme od jeho technické realizace. Před tím, než představíme abstraktní interpret a proces vyhodnocování symbolický výrazů, si řekneme, jaká bude naše *zamýšlená interpretace* symbolických výrazů. Tuto zamýšlenou interpretaci potom zpřesníme popisem činnosti abstraktního interpretu.

Budeme-li uvažovat nejprve *čísla* a *symboly,* pak můžeme neformálně říct, že úkolem čísel v programu je označovat „svou vlastní hodnotu“. Tedy například symbolický výraz `12` označuje hodnotu „číslo dvanáct“, symbolický výraz `10.6` označuje hodnotu „číslo deset celých šest desetin“ a tak dále. Z pohledu vyhodnocování můžeme říct, že „čísla se vyhodnocují na sebe sama“, nebo přesněji *„čísla se vyhodnocují na hodnotu, kterou označují “.*

Symboly lze chápat jako „jména hodnot“. Některé symboly, jako třeba `+` a `*` jsou svázány s hodnotami jimiž jsou operace (procedury) pro sčítání a násobení čísel. Symboly `pi` a `e` by mohly být svázány s přibližnými hodnotami čísel π a e (Eulerovo číslo). Jiné symboly, třeba jako `ahoj-svete`, nejsou svázány s žádnou hodnotou[1]. Z pohledu vyhodnocování říkáme, že *„symboly se vyhodnocují na svou vazbu“*. Tím máme na

[1] Alespoň je tomu tak až do okamžiku, kdy takovým symbolům nějakou hodnotu přiřadíme. Tímto problémem se budeme zabývat v další části této lekce. Zatím pro jednoduchost předpokládejme, že symboly buď jsou nebo nejsou svázány s hodnotami.

mysli, že pokud je například symbol `*` svázán s operací (procedurou) „násobení čísel“, pak je vyhodnocením symbolu `*` právě operace (procedura) „násobení čísel“.

Zbývá popsat zamýšlený význam seznamů. Seznamy jsou z hlediska vyhodnocovacího procesu „příkazem pro provedení operace“, přitom daná operace je vždy určena *prvním prvkem seznamu,* respektive hodnotou vzniklou jeho vyhodnocením. Hodnoty, se kterými bude operace provedena jsou dány hodnotami dalších prvků seznamu – *operandů*. Při psaní programů v jazyku Scheme bychom tedy vždy měli mít na paměti, že seznamy musíme psát ve tvaru

`(`⟨*operace*⟩ ⟨$operand_1$⟩ ⟨$operand_2$⟩ $\cdots$ ⟨$operand_n$⟩`)`

Všimněte si, že tento tvar přirozeně vede na prefixovou notaci výrazů, protože píšeme „operaci“ před „operandy“. Například seznam `(+ 10 20 30)` je z hlediska vyhodnocování příkaz pro provedení operace „sčítání čísel“ jež je hodnotou navázanou na symbol `+`. V tomto případě je operace provedena se třemi hodnotami: „deset“, „dvacet“ a „třicet“, což jsou číselné hodnoty operandů `10`, `20` a `30`. Výsledkem vyhodnocení symbolického výrazu `(+ 10 20 30)` by tedy měla být hodnota „šedesát“.

Je dobré uvědomit si rozdíl mezi symbolickým výrazem a jeho hodnotou, což jsou samozřejmě dvě různé věci. Například uvážíme-li výraz `(* 10 (+ 5 6) 20)`, pak se z hlediska vyhodnocení jedná o příkaz provedení operace „násobení čísel“ s hodnotami „deset“, „výsledek vyhodnocení symbolického výrazu `(+ 5 6)`“ a „dvacet“. To jest, druhou hodnotou při provádění operace násobení není samotný symbolický výraz `(+ 5 6)`, ale *hodnota vzniklá jeho vyhodnocením.* Výsledkem vyhodnocení celého výrazu je tedy číselná hodnota „dva tisíce dvě sta“.

V některých případech se můžeme dostat do situace, že pro některé symbolické výrazy nemůžeme uvažovat „hodnotu“ na kterou se vyhodnotí, protože pro ně vyhodnocení není popsáno. V takovém případě říkáme, že se v programu (či v daném symbolické výrazu) nachází *sémantická chyba.*

Příklad 1.10. Uvažujme symbolický výraz `(10 + 20)`. Tento výraz má na prvním místě symbolický výraz `10`, jehož hodnota je „číslo deset“, což není operace. V tomto případě není vyhodnocení symbolického výrazu `(10 + 20)` definováno, protože první prvek seznamu se nevyhodnotil na operaci – symbolický výraz `(10 + 20)` je sice syntakticky správně (jedná se o symbolický výraz), ale obsahuje sémantickou chybu.

Poznámka 1.11. Výrazy v prefixové notaci lze vzhledem k jejich zamýšlené interpretaci snadno číst:

`(+ 2 (* 3 4))`
„sečti dvojku se součinem trojky a čtyřky“

`(+ (* 2 3) 4)`
„sečti součin dvojky a trojky s číslem čtyři“

a podobně.

Při psaní symbolických výrazů a hodnot, na které se vyhodnotí, příjmeme následující konvenci. Symbolické výrazy, které budeme vyhodnocovat, budeme psát nalevo od symbolu „$\longmapsto$“ a hodnoty vzniklé vyhodnocením budeme psát napravo od tohoto symbolu. Přitom se hodnoty budeme snažit zapisovat pokud možno tak, jak je vypisuje drtivá většina interpretů jazyka Scheme. Pokud daný výraz nelze vyhodnotit v důsledku chyby, budeme za pravou stranu „$\longmapsto$“ uvádět stručný popis chyby. Samotný symbol „$\longmapsto$“ můžeme číst „se vyhodnotí na“. Viz následující ukázky.

```
-10.6          ⟼  -10.6
128            ⟼  128
*              ⟼  „procedura násobení čísel“
(+ 1 2)        ⟼  3
(+ 1 2 34)     ⟼  37
(- 3 2)        ⟼  1
(- 2 3)        ⟼  -1
```

```
(* pi 2)          ⟹  6.283185307179586
(+ 2 (* 3 4))    ⟹  14
(blah 2 3)       ⟹  chyba: symbol blah nemá vazbu
((+ 2 3) 4 6)    ⟹  chyba: hodnota „číslo pět“ není operace
```

Poznámka 1.12. Při vyhodnocení posledních dvou výrazů došlo k chybě. U předposledního výrazu zřejmě protože, že na symbol `blah` nebyla navázaná operace. U posledního výrazu si všimněte, že symbolický výraz `(+23)` se vyhodnotí na hodnotu „číslo pět“, což není operace. Vyhodnocení proto opět končí chybou, protože se očekává, že první prvek seznamu `((+ 2 3) 4 6)` se vyhodnotí na operaci, která by byla dále použita s hodnotami „čtyři“ a „šest“.

Při vyhodnocování výrazů si člověk, na rozdíl od počítače, může dopomoci svým vhledem do výrazu. Například u symbolického výrazu `(* 2 (/ 3 (+ 4 5)))` téměř okamžitě vidíme, že k jeho vyhodnocení nám stačí „sečíst čtyřku s pětkou, tímto výsledkem podělit trojku a nakonec tuto hodnotu vynásobit dvojkou.“ Tento postup je zcela v souladu s naší intuicí a s tím, jak jsme vyhodnocování výrazů zatím poněkud neformálně popsali. Interpret jazyka Scheme, což je opět program, ale tuto možnost „vhledu“ nemá. Abychom byli schopni interprety programovacích jazyků konstruovat, musíme sémantiku jazyka (v našem případě proces vyhodnocování výrazů) přesně popsat krok po kroku. Postup vyhodnocování musí být univerzální a použitelný pro libovolný symbolický výraz.

Nyní popíšeme interpretaci symbolických výrazů formálněji. Představíme zjednodušený model interpretu jazyka Scheme – *abstraktní interpret Scheme* a popíšeme vyhodnocování výrazů krok za krokem.

Interprety jazyka Scheme pracují interaktivně. Postupně načítají výrazy (buď ze souboru nebo čekají na vstup uživatele) a vyhodnocují je. Po načtení jednotlivého výrazu je výraz zpracován, vyhodnocen a výsledná hodnota je oznámena uživateli. Poté se opakuje totéž pro další výraz na řadě dokud není vyčerpán celý vstup. Interprety ovšem nepracují se symbolickými výrazy přímo, to jest ve formě sekvencí znaků tak, jak je zapisujeme (což by bylo neefektivní), ale převádějí je vždy do své vnitřní efektivní reprezentace. Této vnitřní reprezentaci symbolických výrazů budeme říkat *interní reprezentace.* Pro ilustraci, například číslo -123 je dostatečné malé na to, aby jej bylo možné efektivně kódovat čtyřbajtovým datovým typem „integer“, se kterým umějí pracovat všechny běžně používané 32-bitové procesory. Takže interní reprezentací symbolického výrazu `-123` může být třeba 32-bitová sekvence nul a jedniček kódující číslo -123. Seznamy jsou v interpretech reprezentovány zpravidla dynamickými spojovými datovými strukturami (podrobnosti uvidíme v dalších lekcích).

Konkrétní tvar interní reprezentace symbolických výrazů nás ale až na výjimky zajímat nebude. Pro nás je důležité uvědomit si, že každý výraz je na počátku svého zpracování převeden do své interní reprezentace a pak už se výhradně pracuje jen s ní. Část interpretu, která je odpovědná za načtení výrazu, případné odhalení syntaktických chyb a převedení výrazu do interní reprezentace se nazývá *reader*.

Při neformálním popisu interpretace symbolických výrazů jsme uvedli, že při vyhodnocování seznamů dochází k „provádění operací.“ Teď tento pojem poněkud zpřesníme, protože ono „provádění operací“ je ve své podstatě to, co způsobuje vytváření *kvalitativního výpočetního procesu* (vyhodnocování symbolů a čísel by samo o sobě příliš zajímavé nebylo), proto je mu potřeba věnovat patřičnou pozornost. Nejprve se umluvíme, že místo pojmu „operace“ budeme od této chvíle používat obecnější pojem *procedura.* Procedury, které budeme uvažovat, jsou v podstatě nějaké abstraktní elementy, které budou součástí interpretu. Proto jim také někdy budeme říkat *primitivní procedury.* Příklady primitivních procedur si ukážeme později.

Používání pojmu procedura namísto operace jsme zvolili ze dvou důvodů. První důvod je terminologický. V řadě programovacích jazyků je pojem operace spojován prakticky pouze jen s aritmetickými a logickými operacemi (tak je tomu třeba v jazyku C). Naopak pojem *procedura* bývá obvykle chápán obecněji. V terminologii některých jazyků, jako je například Pascal, jsou procedury chápány jako „podprogramy“, které mohou být opakovaně „volány“. V našem pojetí jsou (primitivní) procedury elementy interpretu, které nám umožňují vypočíst novou hodnotu z jiných hodnot. V terminologii

většiny funkcionálních programovacích jazyků, jsou obvykle procedury nazývány *funkce* (tak je tomu třeba v jazycích Common LISP, ML a Haskell), odtud taky plyne název funkcionálního paradigma. My pojem „funkce“ používat nebudeme, protože koliduje s matematickým pojmem funkce. O vztahu procedur a matematických funkcí (čili zobrazení) se budeme bavit v další lekci. Druhým důvodem je, že slovo „procedura“ je používáno ve standardu jazyka Scheme a v drtivé většině dostupné literatury, viz například [SICP, Dy96, R5RS, SF94].

Operace, které jsme neformálně používali v úvodu této lekce při nastínění vyhodnocování seznamů, neoperovaly se symbolickými výrazy, nýbrž s hodnotami vzniklými vyhodnocováním symbolických výrazů. Otázkou je, jak bychom se na ony „hodnoty“ měli přesně dívat. Doposud jsme vše demonstrovali pouze na aritmetických operacích, nabízelo by se tedy uvažovat jako hodnoty *interní reprezentace čísel*. Jelikož však budeme chtít, aby byly (primitivní) procedury schopny manipulovat i s jinými daty, než s čísly, zavedeme obecnější pojem *element jazyka*. Na elementy jazyka budeme pohlížet jako na přípustné hodnoty, se kterými mohou (ale nemusejí) manipulovat primitivní procedury. Za jedny z elementů budeme považovat i samotné primitivní procedury, což bude mít zajímavé důsledky jak uvidíme v dalších lekcích. Následující definice pojem element jazyka zavádí.

Definice 1.13 (elementy jazyka).

(i) Interní reprezentace každého symbolického výrazu je element jazyka,

(ii) každá primitivní procedura je element jazyka. ■

Dle definice 1.13 jsou tedy všechny interní reprezentace čísel, symbolů a seznamů elementy jazyka. Rovněž primitivní procedury jsou elementy jazyka. Množinu všech elementů jazyka budeme během výkladu postupně doplňovat o nové elementy. Z pohledu elementů *reader* převádí symbolické výrazy (posloupnosti znaků) na elementy jazyka (interní reprezentace symbolických výrazů). Naopak jedna z částí interpretu, zvaná *printer*, má opačnou úlohu. Jejím úkolem je pro daný element jazyka vrátit (třeba vytisknout na obrazovku) jeho externí reprezentaci. Pod pojmem *externí reprezentace* elementu máme na mysli pro člověka čitelnou reprezentaci daného elementu. U symbolických výrazů se smluvíme na tom, že pokud je element E interní reprezentací symbolického výrazu e, pak bude externí reprezentace elementu E právě symbolický výraz e. Převodem symbolického výrazu do jeho interní reprezentace a zpět do externí tedy získáme výchozí výraz. Navíc pokud je element E interní reprezentací symbolického výrazu e, tak potom externí reprezentace elementu E bude opět pro *reader* čitelná. U procedur je situace jiná. Procedury jsou abstraktní elementy u nichž se domluvíme na tom, že jejich externí reprezentace (at' už je jakákoliv) bude pro *reader* nečitelná. Tím zaručíme jakousi „čistotu“, protože *reader* slouží k načítání *právě* symbolických výrazů, tedy *ničeho jiného*. Během výkladu budeme při uvádění externí reprezentace procedur psát jejich stručný slovní popis. Uděláme tedy následující úmluvu.

Úmluva 1.14 (o čitelnosti externích reprezentací). Pokud neuvedeme jinak, externí reprezentace elementu E je čitelná readerem, právě když je E interní reprezentací některého symbolického výrazu. ■

Symbolický výraz je čistě syntaktický pojem. Symbolické výrazy jsou zavedeny jako sekvence znaků, splňující podmínky definice 1.6. Naproti tomu elementy jazyka jsou sémantické pojmy. Elementy jazyka jsou prvky množiny (všech elementů jazyka). Elementy jazyka budeme dále používat jako hodnoty a budou hrát klíčovou roli ve vyhodnocovacím procesu. Role elementů jazyka je úzce svázaná s jejich významem (sémantikou). *Reader* převádí symbolické výrazy na elementy jazyka, které jsou interními reprezentacemi symbolických výrazů. Naopak *printer* slouží k převodu elementů do jejich externí reprezentace.

Nyní popíšeme, co budeme mít na mysli pod pojmem *aplikace primitivní procedury*. V podstatě se jedná o formalizaci již dříve uvedeného „provádění operací“. Roli operací hrají primitivní procedury. Místo provádění „operací s danými hodnotami“ budeme hovořit o *aplikaci procedur s danými argumenty*, přitom

konkrétní *argumenty* zastupují konkrétní elementy jazyka (hodnoty), se kterými je daná operace aplikována. Výsledkem aplikace procedury je buď *výsledný element* (pokud aplikace proběhna v pořádku), nebo chybové hlášení „CHYBA: Proceduru nelze aplikovat s danými argumenty.“, pokud během aplikace došlo k chybě. Výsledný element je chápán jako *výsledná hodnota* provedení aplikace. Někdy říkáme, že operace *vrací* výslednou hodnotu. Viz následující ilustrativní příklad.

Příklad 1.15. Aplikací procedury „sečti čísla“ na argumenty jimiž jsou elementy zastupující číselné hodnoty 1, 10 a -30 je výsledný element zastupující číselnou hodnotu -19. Aplikací procedury „zjisti zda-li je dané číslo sudé“ na číselný argument 12.7 je hlášení „CHYBA: Procedura aplikována se špatným argumentem (vstupní argument musí být celé číslo).“. Aplikací procedury „vypočti druhou odmocninu“ na dva argumenty číselné hodnoty 4 a 10 je hlášení „CHYBA: Procedura aplikována se špatným počtem argumentů (vstupní argument musí být jediné číslo).“.

Je samozřejmé, že při aplikaci primitivní procedury s danými argumenty je obecně nutné specifikovat pořadí jednotlivých argumentů (představte si situaci pro proceduru „odečti dvě čísla“). U některých procedur budeme vždy uvažovat *pevný počet argumentů*, u některých procedur *proměnlivý*. Pro zjednodušení zavedeme následující

Označení 1.16. Nechť E je primitivní procedura a $E_1, \ldots, E_n$ jsou libovolné elementy jazyka. Výsledek *aplikace primitivní procedury E na argumenty $E_1, \ldots, E_n$ v tomto pořadí*, budeme značit $\mathrm{Apply}[E, E_1, \ldots, E_n]$. Pokud je výsledkem této aplikace element F, pak píšeme $\mathrm{Apply}[E, E_1, \ldots, E_n] = F$.

Z matematického pohledu je Apply zavedené v předchozím označení částečné zobrazení z množiny všech konečných posloupností elementů do množiny všech elementů. Pro některé n-tice elementů však toto zobrazení není definované (proto jej označujeme jako „částečné zobrazení“).

U primitivních procedur se nebudeme zabývat tím, jak při aplikaci (s danými argumenty) vznikají výsledné hodnoty. Na primitivní procedury se z tohoto úhlu pohledu budeme dívat jako na „černé skříňky.“ To jest, budeme mít přesně popsané, co při aplikaci konkrétní primitivní procedury vznikne, jaké argumenty jsou přípustné a podobně, ale nebudeme se zabývat samotným výpočtem (nebudeme vidět „dovnitř“ primitivních procedur – odtud taky plyne jejich název). S konceptem elementu jako „černé skříňky“ se v budoucnu budeme setkávat častěji.

Nyní můžeme upřesnit, co máme na mysli pod pojmem abstraktní interpret jazyka Scheme. Abstraktní interpret chápeme jako množinu všech elementů jazyka s pravidly jejich vyhodnocování:

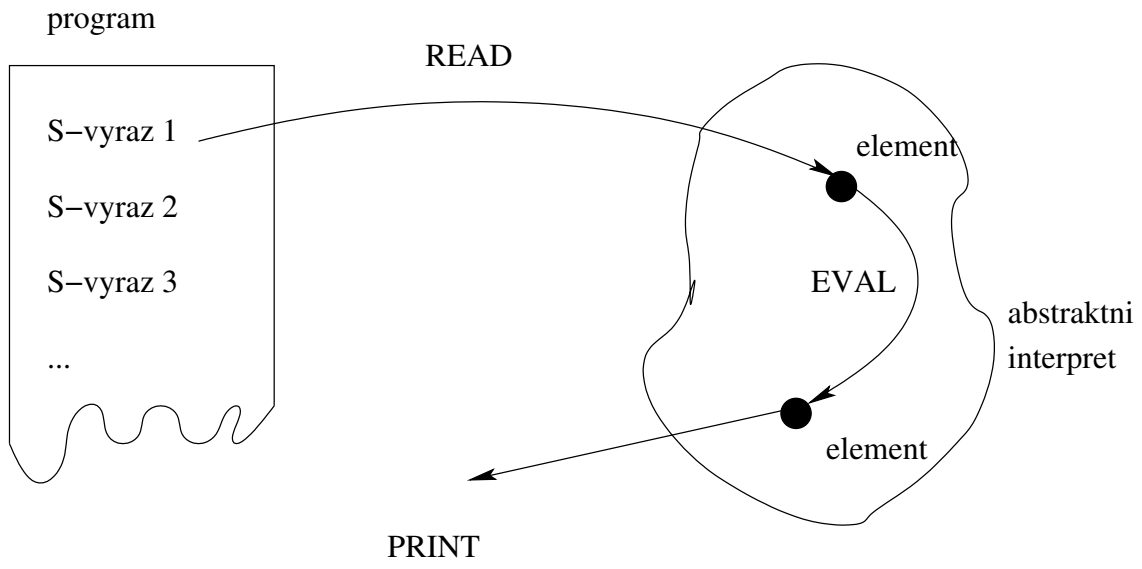
Definice 1.17 (abstraktní interpret). Množinu všech elementů jazyka spolu s pravidly jejich vyhodnocování budeme nazývat *abstraktní interpret jazyka Scheme.* ■

Posledním logickým krokem, který provedeme v této sekci je popis vyhodnocování, tím budeme mít úplně popsaný abstraktní interpret. Vyhodnocování programů, které se skládají z posloupností symbolických výrazů probíhá v cyklu, kterému říkáme REPL. Jméno cyklu je zkratka pro „Read“ (načti), „Eval“ (vyhodnoť), „Print“ (vytiskni) a „Loop“ (opakuj), což jsou jednotlivé části cyklu. Budeme-li předpokládat, že máme vstupní program, jednotlivé fáze cyklu REPL můžeme popsat následovně:

Read: Pokud je prázdný vstup, činnost interpretu končí. V opačném případě *reader* načte první vstupní symbolický výraz. Pokud by se to nepodařilo v důsledku syntaktické chyby, pak je činnost interpretu ukončena chybovým hlášením „CHYBA: Syntaktická chyba“. Pokud je symbolický výraz úspěšně načten, je převeden do své interní reprezentace. Výsledný element, označíme jej E, bude zpracován v dalším kroku.

Eval: Element E z předchozího kroku je vyhodnocen. Vyhodnocení provádí část interpretu, které říkáme *evaluator.* Pokud během vyhodnocení došlo k chybě, je napsáno chybové hlášení a ukončena činnost interpretu. Pokud vyhodnocení končí úspěchem, pak je výsledkem vyhodnocení element. Označme tento element F. Výsledný element F bude zpracován v dalším kroku.

Obrázek 1.2. Schéma cyklu REPL (NACRT OBRAZKU)



Print: Provede se převod elementu F z předchozího kroku do jeho externí reprezentace. Tuto část činnosti obstarává *printer*. Výsledná externí reprezentace je zapsána na výstup (vytištěna na obrazovku, nebo zapsána do souboru, a podobně).

Loop: Vstupní symbolický výraz, který byl právě zpracován v krocích „Read“, „Eval“ a „Print“ je odebrán ze vstupu. Dále pokračujeme krokem „Read“.

Poznámka 1.18. (a) Průběh cyklu REPL je symbolicky naznačen na obrázku 1.2.

(b) Části cyklu „Read“, „Print“ a „Loop“ jsou více méně triviální a jejich provedením se nebudeme zabývat, opět se na ně můžeme dívat jako na černé skříňky. Nutné je přesně specifikovat „Eval“, což je část odpovědná za vyhodnocování a v důsledku za vznik výpočetního procesu. Všimněte si, že vstupem pro „Eval“ není symbolický výraz, ale jeho interní reprezentace (tedy element jazyka). Výstupem „Eval“ je opět element. Při neformálním popisu vyhodnocování na začátku této sekce jsme se bavili o vyhodnocování symbolických výrazů: čísel, symbolů a seznamů; nyní vidíme, že přísně vzato vyhodnocování probíhá nad elementy jazyka. To je z hlediska oddělení syntaxe a sémantiky (a nejen z tohoto hlediska) žádoucí.

(c) V některých případech, zejména v případě reálných interpretů pracujících interaktivně, není žádoucí, aby byla interpretace při výskytu chyby úplně přerušena. Interprety obvykle dávají uživateli k dispozici řadu možností, jak blíž prozkoumat vznik chyby, což může být v mnoha případech dost složité. Místo ukončení činnosti při vzniku chyby by i náš abstraktní interpret nemusel zareagovat okamžitým ukončením činnosti. Místo toho by mohl odebrat vstupní výraz ze vstupu a pokračovat ve fázi „Read“ dalším výrazem.

(d) Práce s interpretem, která je řízená cyklem REPL má své praktické dopady při práci se skutečným interpretem. Bývá zvykem, že programy jsou postupně laděny po částech. To jest programátoři postupně vkládají symbolické výrazy, a sledují jejich vyhodnocování. Takto je obvykle postupně sestavován program, který je uložen v souboru (souborech) na disku. Poté, co je program vytvořen a odladěn, jej lze předat interpretu tak, aby cyklus REPL běžel neinteraktivně přímo nad souborem (výrazy jsou čteny přímo ze souboru). Při čtení výrazů ze souboru vede obvykle chyba rovnou k zastavení programu.

Nyní obrátíme naši pozornost na část „Eval“ cyklu REPL.

Poznámka 1.19. V dalším výkladu budeme někdy (bez dalšího upozorňování) kvůli přehlednosti ztotožňovat interní reprezentaci symbolických výrazů se samotnými symbolickými výrazy. To nám umožní stručnější vyjadřování, přitom nebude možné, abychom oba pojmy smíchali, protože se budeme zabývat vyhodnocováním elementů, kde již (vstupní) symbolické výrazy nehrají roli. Pořád je ale nutné mít na paměti, že jde o dvě různé věci. Pokud se tedy dále budeme bavit například o seznamu `(+ 2 (* 3 4))`

a jeho třetím prvku, máme tím na mysli interní reprezentaci tohoto seznamu a interní reprezentaci jeho třetího prvku a tak podobně.

Během vyhodnocování elementů budeme potřebovat vyhodnocovat symboly. Jak již jsme předeslali, symboly slouží k „pojmenování hodnot“. Při vyhodnocování výrazů je vždy k dispozici tabulka zachycující vazby mezi symboly a elementy. Tuto tabulku nazýváme *prostředí.* Prostředí si lze představit jako tabulku se dvěma sloupci, viz obrázek 1.3.

Obrázek 1.3. Prostředí jako tabulka vazeb mezi symboly a elementy

symbol	*element*
E_1	F_1
E_2	F_2
$\vdots$	$\vdots$
E_k	F_k
$\vdots$	$\vdots$

V levém sloupci prostředí se nacházejí symboly (jejich interní reprezentace), v pravém sloupci se nacházejí libovolné elementy. Pokud je symbol E uveden v tabulce v levém sloupci, pak hodnotě v pravém sloupci na témže řádku říkáme *aktuální vazba symbolu E v prostředí*. Pokud symbol E není uveden v levém sloupci, pak říkáme, že *E nemá vazbu v prostředí*, nebo případně, že *aktuální vazba symbolu E není definovaná.* Na počátku interpretace jsou v prostředí (někdy mu říkáme *počáteční prostředí*) dány *počáteční vazby symbolů*. Jedná se zejména o vazby symbolů na primitivní procedury. Například vazbou symbolu `+` v počátečním prostředí je primitivní procedura „sečti čísla“, aktuální vazbou symbolu `sqrt` je procedura „vypočti druhou odmocninu“, a podobně. Při práci s interpretem je potřeba znát základní vazby symbolů, abychom znali „jména primitivních procedur“, se kterými budeme chtít pracovat.

Označení 1.20. Pokud chceme říct, že F bude označovat výsledek vyhodnocení elementu E, budeme tento fakt zapisovat $F := \mathrm{Eval}[E]$. Pokud chceme říct, že výsledek vyhodnocení elementu E definujeme jako element F, pak tento fakt zapíšeme $\mathrm{Eval}[E] := F$.

Nyní můžeme popsat vyhodnocování elementů (v počátečním prostředí).

Definice 1.21 (vyhodnocení elementu *E*).
Výsledek vyhodnocení elementu E, značeno $\mathrm{Eval}[E]$, je definován:

(A) Pokud je *E číslo*, pak $\mathrm{Eval}[E] := E$.

(B) Pokud je *E symbol*, mohou nastat dvě situace:

(B.1) Pokud *E má aktuální vazbu F*, pak $\mathrm{Eval}[E] := F$.

(B.e) Pokud *E nemá vazbu*, pak ukončíme vyhodnocování hlášením „CHYBA: Symbol E nemá vazbu.“.

(C) Pokud je *E seznam* tvaru $(E_1\ E_2 \cdots E_n)$, pak nejprve vyhodnotíme jeho první prvek a výsledek vyhodnocení označíme F_1, formálně: $F_1 := \mathrm{Eval}[E_1]$. Vzhledem k hodnotě F_1 rozlišíme dvě situace:

(C.1) Pokud je F_1 *procedura*, pak se v nespecifikovaném pořadí vyhodnotí zbylé prvky $E_2, \ldots, E_n$ seznamu E, výsledky jejich vyhodnocení označíme $F_2, \ldots, F_n$. Formálně:

$$\begin{aligned} F_2 &:= \mathrm{Eval}[E_2], \\ F_3 &:= \mathrm{Eval}[E_3], \\ &\vdots \quad\ \vdots \\ F_n &:= \mathrm{Eval}[E_n]. \end{aligned}$$

Dále položíme $\mathrm{Eval}[E] := \mathrm{Apply}[F_1, F_2, \ldots, F_n]$, tedy výsledkem vyhodnocení E je element vzniklý *aplikací procedury F_1 na argumenty $F_2, \ldots, F_n$*.

(C.e) Pokud F_1 není *procedura*, skončíme vyhodnocování hlášením
„CHYBA: Nelze provést aplikaci: první prvek seznamu E se nevyhodnotil na proceduru.“.

(D) Ve všech ostatních případech klademe $\mathrm{Eval}[E] := E$. ■

Poznámka 1.22. (a) Všimněte si, že bod (A) je vlastně pokryt bodem (D), takže bychom jej přísně vzato mohli bez újmy vynechat. Na druhou stranu bychom mohli ponechat bod (A) a zrušit bod (D), protože elementy, které vyhodnocujeme jsou (zatím) interní formy symbolických výrazů (viz cyklus REPL). Do budoucna by to ale nebylo prozíravé, takže bod (D) ponecháme, i když momentálně nebude hrát roli (jeho úlohu uvidíme už v další sekci).

(b) Vyhodnocování symbolů jsme zavedli v souladu s naším předchozím neformálním popisem. Symboly se vyhodnocují na své aktuální vazby. Nebo končí jejich vyhodnocování chybou (v případě, když vazba neexistuje).

(c) Seznamy se rovněž vyhodnocují v souladu s předešlým popisem. Vyhodnocení seznamu probíhá tak, že je nejprve vyhodnocen jeho první prvek. V případě, že se první prvek vyhodnotí na (*primitivní*) *proceduru*, se v nespecifikovaném pořadí vyhodnotí ostatní prvky seznamu; potom je procedura aplikována na argumenty, kterými jsou právě výsledky vyhodnocení zbylých prvků seznamu – tedy všech prvků kromě prvního jímž je samotná procedura.

V této sekci jsme představili interpret primitivního funkcionálního jazyka, který budeme dále rozšiřovat až budeme mít k dispozici úplný programovací jazyk, který bude z výpočetního hlediska stejně silný jako ostatní programovací jazyky. V tuto chvíli si už ale můžeme všimnout základní ideje funkcionálního programování: *výpočetní proces* generovaný programem se skládá z po sobě jdoucích aplikací procedur. Hodnoty vzniklé aplikací jedné procedury jsou použité jako argumenty při aplikaci jiné procedury. Pořadí v jakém jsou procedury aplikovány a samotnou aplikaci řídí *evaluátor.*

Příklad 1.23. V tomto příkladu si rozebereme, jak interpret vyhodnocuje vybrané seznamy. Budeme přitom postupovat přesně podle definice 1.21.

1. Seznam `(+ 1 2)` se vyhodnotí na `3`. Podrobněji: `(+ 1 2)` je seznam, tedy vyhodnocování postupuje dle bodu (C). Vyhodnotíme první prvek seznamu, to jest stanovíme $\mathrm{Eval}[+]$. Jelikož je `+` symbol, postupujeme dle bodu (B). Na symbol `+` je navázána procedura pro sčítání. V bodu (C) tedy postupujeme po větvi (C.1) s hodnotou $F_1 =$ „procedura sčítání čísel“. Nyní v nespecifikovaném pořadí vyhodnotíme zbylé prvky seznamu `1` a `2`. Oba argumenty jsou čísla, takže $\mathrm{Eval}[1] = 1$ a $\mathrm{Eval}[2] = 2$. Nyní aplikujeme proceduru sčítání čísel na argumenty `1` a `2` a dostaneme výslednou hodnotu `3`. Výsledkem vyhodnocení `(+ 1 2)` je `3`.
2. Seznam `(+ (* 3 (+ 1 1)) 20)` se vyhodnotí na `26`. Jelikož se jedná o seznam, postupuje se krokem (C). První prvek seznamu, symbol `+`, je vyhodnocen na proceduru sčítání, takže vyhodnocení pokračuje krokem (C.1). V nespecifikovaném pořadí jsou vyhodnoceny argumenty `(* 3 (+ 1 1))` a `20`. Seznam `(* 3 (+ 1 1))` se bude vyhodnocovat následovně. První je opět vyhodnocen jeho první prvek, což je symbol `*`. Jeho vyhodnocením je „procedura násobní čísel“, pokračujeme tedy vyhodnocením zbylých prvků seznamu. `3` se vyhodnotí na sebe sama a seznam `(+ 1 1)` se opět vyhodnocuje jako seznam. Takže nejprve vyhodnotíme jeho první prvek, což je symbol `+`, jeho vyhodnocením je procedura „sčítání“. Takže vyhodnocujeme zbylé dva prvky seznamu. Jelikož jsou to dvě čísla (obě jedničky), vyhodnotí se na sebe sama. V tuto chvíli dochází k aplikaci sčítání na argumenty `1` a `1`, takže dostáváme, že vyhodnocením seznamu `(+ 1 1)` je číslo `2`. Nyní vyhodnocování pokračuje bodem, který způsobil vyhodnocování seznamu `(+ 1 1)`. To je bod, ve kterém jsme vyhodnocovali seznam `(* 3 (+ 1 1))`. Jelikož již jsme vyhodnotili všechny jeho prvky, aplikujeme proceduru násobení na hodnotu `3` a `2` (výsledek předchozího vyhodnocení). Výsledkem vyhodnocení `(* 3 (+ 1 1))` je tedy hodnota `6`. Nyní se vracíme do bodu, který způsobil vyhodnocení výrazu `(* 3 (+ 1 1))`, což je bod, ve kterém jsme vyhodnocovali postupně prvky seznamu `(+ (* 3 (+ 1 1)) 20)`. Hodnotu druhého prvku jsme právě obdrželi, to je číslo `6`. Třetí prvek seznamu je číslo `20`, které se vyhodnotí na sebe sama. V tuto chvíli aplikujeme proceduru sčítání s hodnotami `6` a `20`. Výsledkem je hodnota `26`, což je výsledek vyhodnocení původního seznamu.

3. Vyhodnocení `(10 + 20)` končí chybou, která nastane v kroku (C.e). Jelikož se jedná o seznam, je nejprve vyhodnocen jeho první prvek. To je číslo `10`, které se vyhodnotí na sebe sama. Jelikož se první prvek seznamu nevyhodnotil na proceduru, vyhodnocení končí bodem (C.e).
4. Vyhodnocení `((+ 1 2) (+ 1 3))` také končí chybou, která nastane v kroku (C.e). Při vyhodnocování seznamu `((+ 1 2) (+ 1 3))` je nejprve vyhodnocen jeho první prvek. Tím je seznam `(+ 1 2)`, který se vyhodnotí na číslo `3` (detaily jsme vynechali). Nyní jsme se dostali do bodu, kdy byl opět první prvek seznamu, konkrétně seznamu `((+ 1 2) (+ 1 3))`, vyhodnocen na něco jiného než je procedura, takže vyhodnocení končí neúspěchem v bodě (C.e).
5. Vyhodnocení `((* 2 3 4))` končí rovněž chybou, která nastane v kroku (C.e). Důvod je stejný jako v předchozím bodě. První prvek seznamu je totiž seznam `(* 2 3 4)`, jeho vyhodnocením je číslo `24`. První prvek seznamu `((* 2 3 4))` se tedy nevyhodnotil na proceduru.
6. Vyhodnocení `(+ 2 (odmocni 10))` končí chybou, která nastane v kroku (B.e), protože použitý symbol `odmocni` nemá v počátečním prostředí vazbu. Důležité je ale uvědomit si, ve kterém momentu k chybě dojde. Je to pochopitelně až při vyhodnocování prvního prvku seznamu `(odmocni 10)`. Kdybychom místo `(+ 2 (odmocni 10))` vyhodnocovali seznam `(+ (2) (odmocni 10))`, pak by rovněž došlo k chybě, ale za předpokladu, že by náš interpret vyhodnocoval prvky seznamů zleva doprava se vyhodnocování zastavilo v bodě (C.e) a k pokusu o vyhodnocení symbolu `odmocni` by vůbec nedošlo. Vysvětlete sami proč.
7. Vyhodnocení `(* (+ 1 2) (sqrt 10 20))` končí chybou při pokusu o aplikaci procedury „vypočti druhou odmocninu“, která je navázaná na symbol `sqrt`, vypsáno bude hlášení: „CHYBA: Nepřípustný počet argumentů.“

V předchozím příkladu si lze všimnout jednoho významného rysu při programování v jazyku Scheme. Tím je pouze malá libovůle v umístění závorek. Ve většině programovacích jazyků není „příliš velký hřích“, když provedeme nadbytečné uzávorkování výrazu. Například v jazyku C mají aritmetické výrazy $2 + x$, $(2 + x)$, $((2 + x))$, $((2 + (x)))$, a tak dále, stejný význam. V jazyku Scheme je z pohledu vyhodnocování každý pár závorek separátním *příkazem pro provedení aplikace procedury*. Tím pádem si nemůžeme dovolit „jen tak přidat do výrazu navíc závorky“. Srovnejte výraz `(+ 1 2)` a naproti tomu výrazy jejichž vyhodnocování končí chybami: `((+ 1 2))`, `(+ (1) 2)`, `((+) 1 2)`, a podobně.

Nyní si představíme některé základní primitivní procedury, které jsou vázány na symboly v počátečním prostředí. Pokud budeme hovořit o procedurách, budeme je pro jednoduchost nazývat jmény symbolů, na které jsou navázány. Například tedy primitivní proceduře „sečti čísla“ budeme říkat procedura `+`. Správněji bychom měli samozřejmě říkat „procedura navázaná na symbol `+`“, protože symbol není procedura. Nyní ukážeme, jak lze používat základní aritmetické procedury `+` (sčítání), `*` (násobení), `-` (odčítání) a `/` (dělení).

Procedura `+` pracuje s libovolným počtem číselných argumentů. Sčítat lze tedy libovolný počet sčítanců. Viz následující příklady:

```
(+ 1 2 3)       ⟹  6
(+ (+ 1 2) 3)   ⟹  6
(+ 1 (+ 2 3))   ⟹  6
(+ 20)          ⟹  20
(+)             ⟹  0
```

U prvních tří příkladů si všimněte toho, že díky možnosti aplikovat `+` na větší počet argumentů než dva se nám výrazně zkracuje zápis výrazu. Kdybychom tyto možnost neměli, museli bychom použít jeden z výrazů na druhém a třetím řádku. Aplikací `+` na jednu číselnou hodnotu je vrácena právě ona hodnota. Poslední řádek ukazuje aplikaci `+` bez argumentů. V tomto případě je vrácena hodnota `0`, protože se jedná o *neutrální prvek* vzhledem k operaci sčítání čísel, to jest 0 splňuje

$$0 + x = x + 0 = x.$$

To jest součet n sčítanců

$$x_1 + x_2 + \cdots + x_n$$

je ekvivalentní součtu těchto n sčítanců, ke kterým ještě přičteme nulu:

$$x_1 + x_2 + \cdots + x_n + 0.$$

Když v posledním výrazu odstraníme všechny $x_1, \ldots, x_n$, zbude nám právě nula, kterou můžeme chápat jako „součet žádných sčítanců“. Pokud se vám zdá zavedení `(+)` umělé nebo neužitečné, pak věřte, že v dalších lekcích uvidíme jeho užitečnost. Důležité je si tento *mezní případ sčítání* zapamatovat.

Analogická pravidla jako pro sčítání platí pro násobení.

```
(* 4 5 6)       ⟼  120
(* (* 4 5) 6)   ⟼  120
(* 4 (* 5 6))   ⟼  120
(* 4)           ⟼  4
(*)             ⟼  1
```

Z pochopitelných důvodů je výsledkem vyhodnocení `(*)` jednička, protože v případě násobení je jednička neutrálním prvkem (platí $1 \cdot x = x \cdot 1 = x$).

Odčítání již na rozdíl od sčítání a odčítání není komutativní ani asociativní operace, to jest obecně *neplatí* $x - y = y - x$ ani $x - (y - z) = (x - y) - z$. Pro odčítání neexistuje neutrální prvek. Primitivní procedura realizující odčítání tedy bude definovaná pro jeden a více argumentů. Na následujících příkladech si všimněte, že odčítání aplikované na jeden argument vrací jeho *opačnou hodnotu* a při odčítání aplikovaném na tři a více argumentů se prvky odčítají postupně zleva-doprava:

```
(- 1 2)         ⟼  -1
(- 1 2 3)       ⟼  -4
(- (- 1 2) 3)   ⟼  -4
(- 1 (- 2 3))   ⟼  2
(- 1)           ⟼  -1
(-)             ⟼  „CHYBA: Při odčítání je potřeba aspoň jeden argument.“
```

U dělení je situace analogická. Při dělení jednoho argumentu je výsledkem převrácená hodnota:

```
(/ 4 5)         ⟼  4/5
(/ 4 5 6)       ⟼  2/15
(/ (/ 4 5) 6)   ⟼  2/15
(/ 4 (/ 5 6))   ⟼  24/5
(/ 4)           ⟼  1/4
(/)             ⟼  „CHYBA: Při dělení je potřeba aspoň jeden argument.“
```

Všimněte si, jak interprety Scheme vypisují zlomky. Například `4/5` znamená $\frac{4}{5}$, `2/3` znamená $-\frac{2}{3}$ a podobně. Ve stejném tvaru můžeme čísla ve tvaru zlomků i zapisovat. Například:

```
(* -1/2 3/4)   ⟼  -3/8
(/ 1/2 -3/4)   ⟼  -2/3
```

Pozor ale na výraz `2/-3` což je symbol, při jeho vyhodnocení bychom dostali:

```
2/-3   ⟼  „CHYBA: symbol 2/-3 nemá vazbu“
```

Kromě dělení `/` máme ve Scheme k dispozici procedury `quotient` a `modulo` provádějící celočíselné podíl (procedura `quotient`) a vracející zbytek po celočíselném podílu (procedura `modulo`). Srovnejte:

```
(/ 13 5)          ⟼  13/5   podíl
(quotient 13 5)   ⟼  2      celočíselný podíl
(modulo 13 5)     ⟼  3      zbytek po celočíselném podílu
```

Při manipulaci s čísly, je vždy nutné mít na paměti *matematická čísla* na jedné straně a jejich *počítačovou reprezentaci* na straně druhé. Je dobře známo, že iracionální čísla jako je π a $\sqrt{2}$ v počítači nelze přesně zobrazit, protože mají nekonečný neperiodický rozvoj. Taková čísla je možné reprezentovat pouze přibližně (nepřesně). Na druhou stranu třeba číslo $0.\overline{6}$ můžeme v počítači reprezentovat přesně, protože $0.\overline{6} = \frac{2}{3}$.

Interprety jazyka Scheme se snaží, pokud je to možné, provádět všechny operace s *přesnou číselnou reprezentací*. Pouze v případě, když přesnou reprezentaci nelze dál zachovat, ji převedou na nepřesnou, pro detaily viz specifikaci přesných a nepřesných čísel uvedenou v [R5RS]. Za přesná čísla považujeme racionální zlomky (a celá čísla), za nepřesná čísla považujeme čísla zapsaná s pohyblivou řádovou tečkou. Viz příklad.

```
(/ 2 3)          ⟹  2/3                 přesná hodnota 2/3
(/ 2 3.0)        ⟹  0.6666666666666666  nereprezentuje přesnou hodnotu 2/3
(* 1.0 (/ 2 3))  ⟹  0.6666666666666666  přesná hodnota 2/3 převedena na nepřesnou
(sqrt 4)         ⟹  2                   přesná hodnota 2
(* -2e-20 4e40)  ⟹  -8e+20              nepřesná hodnota -8 · 10^20
```

Pro vzájemný převod přesných hodnot na nepřesné slouží procedury `exact->inexact` a `inexact->exact`. Samozřejmě, že takový převod je v obou směrech obecně vždy jen přibližný. Viz příklad.

```
(exact->inexact 2/3)                    ⟹  0.6666666666666666
(inexact->exact 0.6666666666666666)     ⟹  6004799503160661/9007199254740992
```

V druhém případě v předchozím výpisu bychom možná očekávali, že nám interpret vrátí přesnou hodnotu $\frac{2}{3}$. To ale není přesná hodnota vstupního čísla `0.6666666666666666`. Místo toho nám byl vrácen komplikovaný racionální zlomek. Pro převedení tohoto zlomku na zlomek jednodušší, který se od něj svou hodnotou příliš neliší, bychom mohli použít primitivní proceduru `rationalize`. Tato procedura se používá se dvěma argumenty, první z nich je *racionální číslo*, druhým parametrem je číslo vyjadřující *maximální odchylku*. Procedura vrací nové racionální číslo, které má co možná nejkratší zápis, a které se od výchozího liší maximálně o danou odchylku. Následující příklad ukazuje, jak by se pomocí `rationalize` dalo konvertovat předchozí nepřehledný zlomek na přesnou hodnotu $\frac{2}{3}$:

```
(rationalize 6004799503160661/9007199254740992 1/1000)  ⟹  2/3
(rationalize (inexact->exact 2/3) 1/1000)               ⟹  2/3
```

K přesné reprezentaci čísel dodejme, že v jazyku Scheme je možné používat libovolně velká celá čísla. V praxi to například znamená, že přičtením jedničky k danému celému číslu vždy získáme číslo větší. Ačkoliv toto tvrzení může znít triviálně, ve většině programovacích jazyků, které používají pouze čísla s pevným rozsahem hodnot, tomu tak není.

Ve Scheme je možné počítat rovněž s imaginárními a komplexními čísly a to opět v jejich přesné a nepřesné reprezentaci. Za přesnou reprezentaci komplexního čísla považujeme reprezentaci, kde je reálná a imaginární složka komplexního čísla vyjádřena přesnou reprezentací racionálních čísel. Například tedy `3+2i`, `-3+2i`, `-3-2i`, `+2i`, `-2i`, `3+1/2i`, `-2/3+4/5i` jsou přesně reprezentovaná komplexní čísla. Při zápisu imaginárních čísel je vždy potřeba začít znaménkem, například `2i` je tedy symbol, nikoliv číslo. Na druhou stranu třeba `0.4e10+2i`, `2/3-0.6i` a podobně jsou komplexní čísla v nepřesné reprezentaci. S čísly se dále operuje prostřednictvím již představených primitivních procedur:

```
(sqrt -2)                ⟹  0+1.4142135623730951i
(* +2i (+ 0.4-3i -7i))   ⟹  20.0+0.8i
```

a podobně. V interpretu jsou k dispozici další primitivní procedury, které zde již představovat nebudeme, jedná se o procedury pro výpočet goniometrických, cyklometrických a jiných funkcí, procedury pro zaokrouhlování a jiné. Zájemce tímto odkazuji na [R5RS].

Nakonec této sekce upozorněme na zápis dlouhých symbolických výrazů. Pokud uvážíme například aritmetický výraz

$$1 + \frac{2 \cdot (x + 6)}{\sqrt{x} + 10},$$

pak jej přímočaře přepisujeme ve tvaru:

```
(+ 1 (/ (* 2 (+ x 6)) (+ (sqrt x) 10)))
```

Ačkoliv je výraz pořád ještě relativně krátký, není napsán příliš přehledně, protože jsme jej celý napsali do jednoho řádku. Při psaní složitějších symbolických výrazů je zvykem je vhodně strukturovat abychom zvětšili jejich čitelnost, například takto:

```
(+ 1
   (/ (* 2 (+ x 6))
      (+ (sqrt x) 10)))
```

Při dělení symbolických výrazů do víc řádků platí úzus, že pokud to není nutné z nějakých speciálních důvodů, pak by žádný řádek neměl začínat závorkou „)“.

Z hlediska „programátorského komfortu“ jsou podstatnou součástí téměř každého programu *komentáře*. Pomocí komentářů programátor v programu slovně popisuje konkrétní části program tak, aby se v programu i po nějaké době vyznal. Komentáře jsou ze syntaktického hlediska speciální sekvence znaků vyskytující se v programu. Ze sémantického hlediska jsou komentáře části programů, které jsou zcela ignorovány. To jest překladače a interprety během své činnosti komentáře nijak nezpracovávají a program s komentáři se zpracovává jako stejný program, ve kterém by byly komentáře vynechány.

V jazyku Scheme jsou komentáře sekvence znaků začínající středníkem (znak „;“) a končící koncem řádku. Znak středník tedy hraje speciální roli v programu – označuje počátek komentáře. Následující příklad ukazuje část programu s komentáři:

```
;; toto je priklad nekolika komentaru
(+ (* 3 4)  ; prvni scitanec
   (/ 5 6)) ; druhy scitanec
;; k vyhodnoceni (+ 10 20) nedojde, protoze to je soucast tohoto komentare
```

1.6 Rozšíření Scheme o speciální formy a vytváření abstrakcí pojmenováním hodnot

V předchozí sekci jsme ukázali vyhodnocovací proces. Interpret jazyka Scheme můžeme nyní používat jako „kalkulačku“ pro vyhodnocování jednoduchých výrazů. Samozřejmě, že komplexní programovací jazyk musí umožňovat daleko víc než jen prosté vyhodnocování výrazů představené v předchozí kapitole. Důležitým prvkem každého programovacího jazyka jsou *prostředky pro abstrakci.* Abstrakcí při programování obvykle myslíme možnost vytváření obecnějších programových konstrukcí, které je možné používat ke zvýšení efektivity při programování. Efektivitou ale nemáme nutně na mysli třeba jen rychlost výpočtu, ale třeba provedení programu, které je snadno pochopitelné, rozšiřitelné a lze jej podle potřeby upravovat.

Prvním prostředkem abstrakce, který budeme používat bude *definice vazeb symbolů.* Při práci s interpretem bychom totiž leckdy potřebovali upravit vazby symbolů v prostředí. Uvažme například následující výraz:

```
(* (+ 2 3) (+ 2 3) (+ 2 3))
```

Ve výrazu se třikrát opakuje tentýž podseznam. Při vyhodnocování celého výrazu se tedy bude seznam `(+ 2 3)`. Vyhodnocovat hned třikrát. Kdyby se jednalo o výraz, jehož výpočet by byl časově náročný, pak bychom při výpočtu čekali na jeho výpočet třikrát. V takové situaci by se hodilo mít k dispozici aparát, kterým bychom mohli na nový symbol bez vazby, řekněme `value`, navázat hodnotu vzniklou vyhodnocením `(+ 2 3)`. Jakmile by byla vazba provedena, stačilo by pouze vyhodnotit:

```
(* value value value)
```

Při vyhodnocení posledního uvedeného výrazu už by se seznam `(+ 2 3)` nevyhodnocoval, takže zdržení související s vyhodnocením tohoto seznamu by nastalo pouze jednou.

Mnohem důležitějším důvodem pro možnost definice nových vazeb je zpřehlednění kódu. Ve větších programech se obvykle vyskytují nějaké hodnoty, které se během výpočtu nemění, ale během životního cyklu programu mohou nabývat různých hodnot. Takovou hodnotou může být například sazba DPH. Bylo by krajně nepraktické, kdyby programátor napsal program (například účetní program), ve kterém by byla sazba DPH vyjádřená číslem na každém místě, kde by ji bylo potřeba. Takových míst by ve větším programu byla zcela jistě celá řada a při změně sazby DPH by to vedlo k velkým problémům. Uvědomte si, že problém by zřejmě nešel vyřešit hromadnou náhradou čísla `5` za číslo `19` v programu. Co kdyby se v něm číslo `5` vyskytovalo ještě v nějakém jiném významu než je hodnota DPH? Čisté řešení v takovém případě je použít abstrakci založenou na pojmenování hodnoty symbolem. To jest, v programu je lepší místo číselné

hodnoty používat symbol pojmenovaný třeba `gdp-value` a pouze na jednom zřetelně označeném místě programu mít provedenu definici vazby čísla reprezentující konkrétní sazbu na symbol `gdp-value`.

Z toho co jsme nyní řekli je zřejmé, že potřebujeme mít k dispozici prostředky pro definici nových vazeb. Nebo předefinování již existujících vazeb novými hodnotami. Samotné „provedení definice“ by měl mít na starosti nějaký element jazyka, který bychom chtěli během programování používat. Řekněme ale již v tuto chvíli, že takovým elementem *nemůže být procedura.* Důvod je v celku prostý. Před aplikací procedury dojde vždy k vyhodnocení všech prvků seznamu. Jedním z nich by mělo být jméno symbol, na který chceme novou hodnotu nadefinovat. Pokud by ale symbol neměl vazbu, pak by vyhodnocení končilo neúspěchem. Proberme tuto situaci podrobněji. Předpokládejme, že bychom chtěli mít k dispozici proceduru navázanou na symbol `define` a chtěli bychom zavádět nové definice takto:

```
(define gdp-value 19)
```

Z pohledu vyhodnocovacího procesu je předchozí seznam vyhodnocen tak, že se nejprve vyhodnotí jeho první prvek. Tím je dle našeho předpokladu symbol, který se vyhodnotí na proceduru provádějící definice. Dále by byly vyhodnoceny v nespecifikovaném prostředí zbylé dva argumenty. `19` se vyhodnotí na sebe sama, zde žádný zádrhel neleží. Při vyhodnocování symbolu `gdp-value` by ovšem došlo k chybě v kroku (B.e), viz definici 1.21, protože symbol nemá vazbu.

Jednoduchou analýzou problému jsme dospěli do situace, kdy potřebujeme mít v jazyku Scheme nové elementy, při jejichž aplikaci se nebude vyhodnocovat zbytek seznamu tak, jako u procedur. Tyto nové elementy nyní představíme a budeme jim říkat *speciální formy.*

Nyní již známe tři typy elementů jazyka: interní reprezentace symbolických výrazů, primitivní procedury a speciální formy. Speciální formy mají podobný účel jako procedury – jejich aplikacemi lze generovat, případně modifikovat, výpočetní proces. Zcela zásadně se však od procedur liší v tom, jak probíhá jejich aplikace během vyhodnocování.

Speciální formy jsou aplikovány s argumenty jimiž jsou *elementy v jejich nevyhodnocené podobě* a každá *speciální forma si sama určuje* jaké argumenty a v jakém pořadí (zda-li vůbec) bude vyhodnocovat. Proto je nutné každou speciální formu vždy detailně popsat. Při popisu se na rozdíl od procedur musíme zaměřit i na to, jak speciální forma manipuluje se svými argumenty. Aplikaci speciální formy E na argumenty $E_1, \ldots, E_n$ budeme značit v souladu s označením aplikace primitivních procedur výrazem $\mathrm{Apply}[E, E_1, \ldots, E_n]$.

Poznámka 1.24. Všimněte si toho, že procedury jsou aplikovány s argumenty, které jsou předem získány vyhodnocením. Tedy procedura přísně vzato nemůže nijak zjistit, vyhodnocením jakých elementů dané argumenty vznikly. To je patrné pokud se podíváme na definici 1.21, bod (C.1).

Jelikož jsme zavedli nový typ elementu – speciální formy, musíme rozšířit vyhodnocovací proces tak, abychom mohli zavést aplikaci speciální formy, který nastane v případě, že první prvek seznamu se vyhodnotí na speciální formu. Pro použití speciálních forem rozšíříme bod (C) části „Eval“ cyklu REPL následovně:

Definice 1.25 (doplnění vyhodnocovacího procesu o speciální formy).
Výsledek vyhodnocení elementu E, značeno $\mathrm{Eval}[E]$, je definován:

(A) Stejné jako v případě definice 1.21.

(B) Stejné jako v případě definice 1.21.

(C) Pokud je E *seznam* tvaru $(E_1\ E_2 \cdots E_n)$, pak nejprve vyhodnotíme jeho první prvek a výsledek vyhodnocení označíme F_1, formálně: $F_1 := \mathrm{Eval}[E_1]$. Vzhledem k hodnotě F_1 rozlišíme tři situace:

(C.1) Pokud F_1 je *procedura*, pak se postup shoduje s (C.1) v definici 1.21.

(C.2) Pokud F_1 je *speciální forma*, pak $\mathrm{Eval}[E] := \mathrm{Apply}[F_1, E_2, \ldots, E_n]$.

(C.e) Pokud F_1 není *procedura ani speciální forma*, skončíme vyhodnocování hlášením „CHYBA: Nelze provést aplikaci: první prvek seznamu E se nevyhodnotil na proceduru ani na speciální formu.“.

(D) Stejné jako v případě definice 1.21. ■

V novém bodu (C.2) jasně vidíme, že výsledkem vyhodnocení E je aplikace speciální formy F_1 na elementy $E_2, \ldots, E_n$, což jsou prvky seznamu E (kromě prvního) v jejich nevyhodnocené podobě.

V jazyku Scheme budeme mít vždy snahu mít co možné nejméně (nezbytné minimum) speciálních forem, protože zavádění speciálních forem do jazyka a jejich využívání programátorem je potenciálním zdrojem chyb. Důvod je zřejmý: jelikož si každá speciální forma určuje vyhodnocování svých argumentů, na uživatele formy (to jest programátora) to na rozdíl od procedur klade další požadavek (pamatovat si, jak se argumenty zpracovávají). Druhý úhel pohledu je více méně epistemický – zařazováním nadbytečných forem do jazyka se zvyšuje jeho celková složitost.

Pokud budeme dále hovořit o speciálních formách, budeme je pro jednoduchost nazývat stejně jako symboly, na které jsou speciální formy navázány v počátečním prostředí. Symboly budeme značit tmavě modrou barvou, abychom zřetelně viděli, že na uvažovaný symbol je v počátečním prostředí navázána speciální forma. Jako první zavedeme speciální formu `define`, která slouží k definici vazby na symbol.

Definice 1.26 (speciální forma `define`). Speciální forma `define` se používá se dvěma argumenty ve tvaru:

`(define` ⟨*jméno*⟩ ⟨*výraz*⟩`)`

Při aplikaci speciální forma nejprve ověří zda-li je argument ⟨*jméno*⟩ symbol. Pokud tomu tak není, aplikace speciální formy je ukončena hlášením „CHYBA: První výraz musí být symbol.“ V opačném případě je vyhodnocen argument ⟨*výraz*⟩. Označme F výsledek vyhodnocení tohoto elementu, to jest $F := \mathrm{Eval}[\langle \textit{výraz} \rangle]$. Dále je v prostředí vytvořena nová vazba symbolu ⟨*jméno*⟩ na element F. Pokud již symbol ⟨*jméno*⟩ měl vazbu, tato původní vazba je nahrazena elementem F. Ve standardu jazyka Scheme [R5RS] se uvádí, že výsledná hodnota aplikace speciální formy `define` není definovaná. ■

Poznámka 1.27. (a) Speciální forma `define` má při své aplikaci *vedlejší efekt*: modifikuje prostředí.

(b) Specifikace jazyka Scheme R^{5}RS neupřesňuje, co je pod pojmem „nedefinovaná hodnota“ myšleno. Respektive specifikace uvádí, že výsledná hodnota může být „jakákoliv“. Jednotlivé interprety proto vracejí při použití `define` různé hodnoty. V našem abstraktním interpretu Scheme vyřešíme situaci s nedefinovanou hodnotou následovně. Budeme uvažovat speciální element jazyka *„nedefinovaná hodnota“*, tento element se bude v souladu s bodem (D) definice 1.25 vyhodnocovat na sebe sama. Externí reprezentací tohoto elementu bude „prázdný řetězec znaků“. Dodejme, že tímto způsobem je nedefinovaná hodnota řešena i v mnoha skutečných interpretech jazyka Scheme.

Příklad 1.28. Tento příklad ukazuje vyhodnocování seznamů obsahujících `define`.

1. Po vyhodnocení `(define ahoj (* 2 3))` se na symbol `ahoj` naváže hodnota `6`. Podrobněji, celý seznam se vyhodnocuje dle bodu (C), takže jako první je vyhodnocen jeho první prvek. Tím je symbol `define`, který se vyhodnotí na speciální formu. Speciální forma je aktivována s argumentem `ahoj` a druhým argumentem je seznam `(* 2 3)`. Jelikož je `ahoj` symbol, je vyhodnocen seznam `(* 2 3)`, jehož vyhodnocením získáme hodnotu `6`. Do prostředí je zavedena vazba symbolu `ahoj` na hodnotu `6`. Výsledkem aplikace speciální formy je nedefinovaná hodnota.
2. V případě seznamu `(define a (* 2 a))` je definován výsledek vyhodnocení seznamu `(* 2 a)` na symbol `a`. V případě, že by během vyhodnocování výrazu symbol `a` doposud neměl žádnou vazbu, dojde k ukončení vyhodnocování výrazu `(* 2 a)` v bodě (B.e), protože `a` nemá vazbu. Pokud by již `a` vazbu mělo a pokud by na `a` bylo navázáno číslo, pak by byla po aplikaci speciální formy hodnota navázaná na `a` nadefinovaná na dvojnásobek původní hodnoty.
3. Po vyhodnocení `(define blah +)` se na symbol `blah` naváže hodnota, která byla navázaná na symbolu `+`. Pokud měl symbol `+` doposud svou počáteční vazbu, pak by byla na symbol `blah` navázána primitivní procedura sčítání.

4. Po postupném vyhodnocení seznamů `(define tmp +)`, `(define + *)`, a `(define * tmp)` se změní vazby na `+` a `*`. Na pomocný symbol `tmp` bude navázána původní vazba symbolu `+`.
5. Vyhodnocení `(define 10 (* 2 3))` končí chybou při aplikaci speciální formy (`10` není symbol).
6. Seznam `((define a 10) 20)` se vyhodnocuje dle bodu (C), takže jako první se vyhodnotí jeho první prvek, čímž je seznam `(define a 10)`. Vyhodnocením tohoto seznamu vznikne nová vazba v prostředí a vrácena je „nedefinovaná hodnota“. Jelikož se první prvek seznamu `((define a 10) 20)` vyhodnotil na nedefinovanou hodnotu, což není ani procedura, ani speciální forma, je vyhodnocení ukončeno chybou dle bodu (C.e).
7. Po vyhodnocení `(define define 20)` dojde k redefinici symbolu `define`, na který bude navázaná hodnota `20`. Pokud bychom neměli speciální formu `define` navázanou ještě na jiném symbolu, pak bychom ji po provedené této definice nenávratně ztratili.

Nyní ukážeme několik použití `define` bez detailního rozboru vyhodnocování. Budeme předpokládat, že výrazy vyhodnocujeme postupně během jednoho spuštění interpretu.

```
(define a 10)
a               ⟼  10
(* 2 a)         ⟼  20
(sqrt (+ a 5))  ⟼  3.8729833462074
```

Redefinicí hodnoty symbolu `a` bude předešlá vazba „zapomenuta“:

```
(define a 20)
(* 2 a)         ⟼  40
(sqrt (+ a 5))  ⟼  5
```

V následujícím příkladu je nová vazba symbolu `a` vypočtena pomocí jeho aktuální vazby:

```
(define a (+ a 1))
a                   ⟼  21
```

Nyní můžeme nadefinovat vazbu symbolu `b` na hodnotu součtu čtyřky s aktuální vazbou `a`. Samotná aktuální vazba symbolu `a` se po provedení následující definice nijak nemění:

```
(define b (+ 4 a))
a                   ⟼  21
b                   ⟼  25
(sqrt b)            ⟼  5
```

Pokud nyní změníme vazbu symbolu `a`, vazba symbolu `b` nebude změněna. Nic na tom nemění ani fakt, že hodnota navázaná na `b` byla v jednom z předchozích kroků stanovena pomocí hodnoty `a`.

```
(define a 666)
a               ⟼  666
b               ⟼  25
```

Následující definicí bude na symbol `plus` navázána primitivní procedura, která je navázána na symbol `+`:

```
(define plus +)
(plus 1 2)      ⟼  3
(plus 1 b)      ⟼  26
(plus 1 2 3)    ⟼  6
```

Nyní bychom mohli předefinovat vazbu symbolu `+` třeba takto:

```
(define + *)
(+ 1 1)      ⟼  1
(+ 2 3)      ⟼  6
```

Primitivní proceduru sčítání, však máme neustále navázanou na symbolu `plus`, takže jsme ji „neztratili“.

Poznámka 1.29. Kdybychom v našem abstraktním interpretu dali vyhodnotit symbol `define`, interpret by nám měl odpovědět jeho vazbou. Tedy měl by napsat něco ve smyslu „speciální forma pro vytváření definic“. Skutečné interprety jazyka Scheme na vyhodnocení symbolu `define` reagují obvykle chybovým hlášením, protože nepředpokládají vyhodnocení symbolu `define` na jiné pozici než na prvním místě seznamu. Například interprety Guile a Elk však reagují stejně jako náš abstraktní interpret.

1.7 Pravdivostní hodnoty a podmíněné výrazy

V předchozích sekcích jsme pracovali pouze s číselnými hodnotami a ukázali jsme princip vazby hodnoty na symbol. V praxi je pochopitelně potřeba pracovat i s jinými daty než s čísly, například s pravdivostními hodnotami. Velmi často je totiž v programech potřeba provádět podmíněné vyhodnocování v závislosti na vyhodnocení jiných elementů. Podmíněným vyhodnocováním se budeme zabývat v této části lekce. Abychom mohli uvažovat podmínky a jejich pravdivost, musíme do našeho interpretu zavést nové elementy jazyka reprezentující *pravdivostní hodnoty – pravda* a *nepravda*.

V počátečním prostředí bude element „pravda“ navázaný na symbol `#t` („*t*“ z anglického *true*) a element „nepravda“ bude navázaný na symbol `#f` („*f*“ z anglického *false*). Samotné elementy „pravda“ a „nepravda“ se budou vyhodnocovat na sebe sama dle bodu (D) definice 1.25, vyhodnocovací proces tedy nemusíme nijak upravovat. Pravdivostní hodnoty budeme zapisovat ve tvaru `#t` a `#f` a podle potřeby je budeme ztotožňovat se samotnými symboly `#t` a `#f`, které budou jejich externí reprezentací. V tomto případě jsme tedy udělali výjimku proti obecné úmluvě 1.14 na straně 23.

Pravdivostní hodnoty „pravda“ a „nepravda“ jsou speciální elementy, které se vyhodnocují na sebe sama. Pro jednoduchost uvažujeme, že pravdivostní hodnoty jsou v počátečním prostředí navázány na symboly `#t` (pravda) a `#f` (nepravda). Specifikace R^{5}RS jazyka Scheme zachází s pravdivostními hodnotami odlišně: `#t` a `#f` nejsou symboly, ale speciální sekvence znaků rozpoznatelné readerem, který je při načítání převádí na pravdivostní hodnoty. My tuto koncepci pro jednoduchost nebudeme přebírat, protože bychom museli rozšiřovat pojem *symbolický výraz*, což nechceme.

Abychom mohli formulovat netriviální podmínky, potřebujeme mít k dispozici procedury, které budou při aplikaci vracet pravdivostní hodnoty. Procedury, výsledkem jejichž aplikace jsou pravdivostní hodnoty, nazýváme *predikáty*. Tento pojem se používá ve standardu jazyka Scheme [R5RS], budeme jej používat i my, i když poněkud nevhodně koliduje s pojmem predikát, který je užívaný v *matematické logice* (v jiném smyslu než zde). Z praktického hlediska jsou ale predikáty obyčejné procedury (tak, jak jsme je představili v předchozích sekcích), pouze vracející pravdivostní hodnoty.

V počátečním prostředí je definováno několik predikátů, kterými můžeme porovnávat číselné hodnoty. Jsou to predikáty `<` (ostře menší), `<=` (menší rovno), `=` (rovno), `>=` (větší rovno), `>` (ostře větší), jejichž zamyšlený význam je zřejmě jasný. Všechny tyto predikáty pracují se dvěma argumenty, viz příklady.

```
#t              ⟼  #t
#f              ⟼  #f
(<= 2 3)        ⟼  #t
(< 2 3)         ⟼  #t
(= 2 3)         ⟼  #f
(= 2 2.0)       ⟼  #t
(= 0.5 1/2)     ⟼  #t
(>= 3 3)        ⟼  #t
(> 3 3)         ⟼  #f
```

Příklad 1.30. Kdybychom si vzali například seznam `(+ 2 (< 3 3))`, pak při jeho vyhodnocení dojde k chybě. Všimněte si totiž, že seznam `(< 3 3)` se vyhodnotí na `#f`. Po tomto kroku by měla být aplikována procedura sčítání s číselnou hodnotou `2` a druhým argumentem, kterým by byla pravdivostní hodnota „nepravda“. Aplikace procedury sčítání by tedy končila chybovým hlášením „CHYBA: Druhý argument při aplikaci sčítání není číslo.“.

Nyní zavedeme speciální formu `if`, pomocí niž budeme moci provádět podmíněné vyhodnocování.

Definice 1.31 (speciální forma `if`). Speciální forma `if` se používá se dvěma argumenty ve tvaru:

`(if` ⟨*test*⟩ ⟨*důsledek*⟩ ⟨*náhradník*⟩`)`,

přitom ⟨*náhradník*⟩ je *nepovinný argument* a nemusí být uveden. Při aplikaci speciální formy `if` je nejprve vyhodnocen argument ⟨*test*⟩. Pokud je hodnota vzniklá jeho vyhodnocením různá od elementu „nepravda“ (to jest elementu navázaného na `#f`), pak je výsledkem aplikace speciální formy `if` výsledek vyhodnocení argumentu ⟨*důsledek*⟩. V opačném případě, to jest v případě, kdy se ⟨*test*⟩ vyhodnotil na nepravda, rozlišujeme dvě situace. Pokud je přítomen argument ⟨*náhradník*⟩, pak je výsledkem aplikace speciální formy `if` výsledek vyhodnocení argumentu ⟨*náhradník*⟩. Pokud ⟨*náhradník*⟩ není přítomen, pak je výsledkem aplikace speciální formy `if` nedefinovaná hodnota, viz poznámku 1.27 (b). ■

Poznámka 1.32. Všimněte si, že speciální forma `if` zachází při vyhodnocování *testu* s pravdivostními hodnotami ještě o něco obecněji, než jak jsme předeslali. *Náhradník*, což je argument vyhodnocující se v případě, kdy vyhodnocení *testu* skončilo nepravdou, je skutečně vyhodnocen pouze v případě, kdy se *test* vyhodnotí na „nepravda“ (to jest `#f`). V ostatních případech je vždy vyhodnocen *důsledek*. To zahrnuje i případy, kdy výsledkem *testu* není pravdivostní hodnota, ale jakýkoliv jiný element kromě elementu „nepravda“.

Speciální forma `if` slouží k podmíněnému vyhodnocování. Seznamy, ve kterých se používá forma `if` lze přirozeně číst. Například `(if (<= a b) a b)` můžeme číst „pokud je `a` menší nebo rovno `b`, pak vrať `a`, v opačném případě vrať `b`“. Tudíž při vyhodnocení tohoto seznamu obdržíme jako výsledek vždy menší z hodnot navázaných na `a` a `b`. Speciální forma `if` zachází s pravdivostními hodnotami v zobecněné podobě. Jediný element, který je považován za „nepravdu“ je element navázaný na `#f`.

Otázkou je, proč trváme na tom, aby byla speciální forma `if` právě speciální forma. Nabízí se otázka, zda-li by nebylo dostačující chápat ji jako proceduru. Odpověď je dvojí. Na jednu stranu bychom ji *zatím* mohli chápat jako proceduru. S tímto konceptem „`if` jako procedury“ bychom si však nevystačili příliš dlouho. Druhým úhlem pohledu je samotná myšlenka podmíněného vyhodnocení. U něj totiž nejde jen o to, hodnota kterého z výrazů ⟨*důsledek*⟩ a ⟨*náhradník*⟩ je vrácena, ale o to, aby byl vždy *vyhodnocen pouze jeden z nich.* Kdybychom se chtěli v interpretu přesvědčit, že `if` je skutečně speciální forma, mohli bychom zkusit vyhodnotit následující seznam:

```
(if (= 0 0) #t nepouzity-symbol)
```

Pokud by `if` byla procedura, pak by byly vyhodnoceny všechny její argumenty a vyhodnocování by skončilo chybou v bode (B.e), protože symbol `nepouzity-symbol` by neměl vazbu (pokud bychom ji předtím nevytvořili, což nepředpokládáme). V případě `if` jako speciální formy je nejprve vyhodnocena podmínka `(= 0 0)`, která je (vždy) pravdivá. Tím pádem je výsledkem aplikace `if` výsledek vyhodnocení druhého argumentu, což je argument `#t`, který se vyhodnotí na „pravda“. Symbol `nepouzity-symbol` v tomto případě nebude nikdy vyhodnocen.

Nyní uvedeme příklady použití speciální formy `if`.

```
(define a 10)
(define b 13)
(if (> a b) a b)                  ⟼  13
(+ 1 (if (> a b) a b))            ⟼  14
(if (<= a b) (+ a b) (- a b))     ⟼  23
(if (<= a b) (- b a) (- a b))     ⟼  3
```

Pokud jsou na symboly `a` a `b` navázána nezáporná čísla, pak se poslední z uvedených podmíněných výrazů vyhodnotí na absolutní hodnotu rozdílu číselných hodnot navázaných na `a` a `b` (rozmyslete si proč). Podmíněné výrazy se někdy k vůli přehlednosti píší tak, že všechny tři argumenty jsou pod sebou:

```
(if (<= a b)
    (- b a)
    (- a b))  ⟼  3
```

Přehlednost zápisu vynikne teprve v případě, kdy máme několik podmínek vnořených. Například:

```
(if (<= a b)
    (if (= a b)
        a
        (- a))
    #f)  ⟼  -10
```

Zkuste si rozebrat, jak by se předchozí výraz vyhodnocoval v případě různých hodnot navázaných na `a` a `b`. Které všechny případy musíme v takové situaci uvažovat?

I když to není příliš praktické, v podmíněných výrazech můžeme uvádět podmínky, které jsou vždy pravdivé (v obecném smyslu) nebo vždy nepravdivé. Viz následující ukázku.

```
(if #t 10 20)  ⟼  10
(if 1 2 3)     ⟼  2
(if #f 10 20)  ⟼  20
(if #f 10)         nedefinovaná hodnota
```

Doteď jsme se při vyhodnocování seznamů setkávali prakticky výhradně se seznamy, které měly na prvním místě symbol, jenž se vyhodnotil na primitivní proceduru nebo na speciální formu. V následujících ukázkách demonstrujeme, že tomu tak obecně být nemusí a přitom vyhodnocovací proces nekončí chybou (jak tomu bylo ve všech předcházejících případech, kdy první prvek seznamu nebyl symbol).

```
(if #t + -)           ⟼  procedura „sčítání čísel“
((if #t + -) 10 20)   ⟼  30
((if #f + -) 10 20)   ⟼  -10
```

V obou posledních případech je nejprve vyhodnocen první prvek celého seznamu, tím je opět seznam, který způsobí aplikaci speciální formy `if`. Po testu podmínky, která v prvním případě dopadne pozitivně a v druhém negativně, je vyhodnocen symbol `+` nebo symbol `-`. Jejich vyhodnocením vznikají primitivní procedury. V tomto okamžiku se tedy první prvek celého seznamu vyhodnotí na proceduru. Dále jsou v nespecifikovaném pořadí vyhodnoceny ostatní argumenty (čísla `10` a `20`) a procedura získaná vyhodnocením prvního výrazu je s nimi aplikována. To vede buď na jejich součet nebo na jejich rozdíl.

Jako další příklad tohoto fenoménu si můžeme ukázat seznam, jehož vyhodnocením získáme absolutní hodnotu čísla navázaného na symbol `a`. Přímočaře problém vyřešíme třeba takto:

```
(if (<= a 0) (- a) a)
```

Na druhou stranu bychom ale mohli použít následující kód:

```
((if (<= a 0) - +) a)
```

V tomto případě se první prvek seznamu opět vyhodnotí na proceduru sčítání nebo odčítání v závislosti na tom, zda-li byla na `a` navázaná hodnota menší nebo rovna nule. Procedury sčítání nebo odčítání jsou pak aplikovány s jediným argumentem jímž je hodnota vázaná na `a`. Výsledná hodnota je tedy buď tatáž jako hodnota navázaná na `a` nebo její obrácená hodnota.

Shrnutí

V této úvodní lekci jsme ukázali, co je program a jaký je jeho vztah k výpočetnímu procesu. Program je pasivní entita, lze si jej představit jako soubor uložený na disku. Výpočetní procesy jsou generované programy během jejich zpracování počítačem. Výpočetní procesy manipulují s daty, mění vstupy na výstupy a mají vedlejší efekty. Programovací jazyky dělíme na nižší (kódy stroje, assemblery a autokódy) a vyšší. Vyšší programovací jazyky umožňují větší komfort při programování díky možnostem abstrakce, které

dávají uživatelům (programátorům). Programy napsané ve vyšších programovacích jazycích jsou buď překládané (kompilované), nebo interpretované. U programovacích jazyků a programů rozeznáváme jejich syntaxi (tvar) a sémantiku (význam), které nelze zaměňovat. Programování v daném jazyku vždy podléhá programovacímu stylu nebo více stylům, kterým říkáme paradigmata programování. Mezi základní paradigmata patří: funkcionální, procedurální, objektové, paralelní a logické paradigma. Syntax a interpretaci v případě konkrétního jazyka jsme demonstrovali na jazyku Scheme. Programy v jazyku Scheme jsme definovali jako sekvence symbolických výrazů, které dělíme na čísla, symboly a seznamy. Dále jsme ukázali model interpretu jazyka Scheme a popsali vyhodnocování. Zavedli jsme pojem element jazyka. Základními elementy pro nás byly primitivní procedury zastupující operace a interní formy symbolických výrazů. Na elementy jazyka se lze dívat jako na hodnoty, se kterými je možné provádět výpočty. Popsali jsme aplikaci primitivních procedur, prostředí a vazby symbolů v počátečním prostředí. Dále jsme poukázali na nutnost vybavit náš interpret novým typem elementů – speciálními formami kvůli možnosti definovat nové vazby symbolů. Nakonec jsme se zabývali podmíněným vyhodnocováním a představili jsme další typy elementů jazyka – pravdivostní hodnoty a nedefinovanou hodnotu.

Pojmy k zapamatování

- program, výpočetní proces,
- nižší programovací jazyk, kód stroje, autokód, assembler, bajtkód,
- vyšší programovací jazyk, překladač, interpret,
- primitivní výrazy, prostředky kombinace, prostředky abstrakce,
- syntax programu, sémantika programu,
- operace, operandy,
- infixová/prefixová/postfixová notace, polská (reverzní) bezzávorková notace
- syntaktická chyba, sémantická chyba,
- paradigmata programování: funkcionální, procedurální, objektové, paralelní, logické
- jazyk Scheme, symbolické výrazy, S-výrazy,
- čísla, symboly, seznamy, program,
- vyhodnocovací proces, abstraktní interpret Scheme,
- primitivní procedury, elementy jazyka
- interní reprezentace symbolických výrazů,
- externí reprezentace elementů,
- reader, printer, evaluator, cyklus REPL,
- aplikace primitivní procedury, argumenty,
- prostředí, počáteční prostředí, vazba symbolu, aktuální vazba,
- speciální formy, definice vazeb, podmíněné vyhodnocování,
- pravdivostní hodnoty, predikáty, nedefinovaná hodnota.

Nově představené prvky jazyka Scheme

- speciální formy: `define` a `if`,
- aritmetické procedury: `+`, `-`, `*`, `/`, `quotient`, `modulo`, `sqrt`,
- procedury pro konverzi číselné reprezentace: `exact->inexact`, `inexact->exact`, `rationalize`,
- pravdivostní hodnoty: `#t`, `#f`,
- predikáty: `<`, `<=`, `=`, `>=`, `>`.

Kontrolní otázky

1. *Jaký je rozdíl mezi syntaxí a sémantikou programu?*
2. *Jak v jazyku Scheme zapisujeme seznamy?*
3. *Z čeho se skládají programy v jazyku Scheme?*

4. *Jak probíhá vyhodnocení symbolických výrazů?*
5. *Co máme na mysli pod pojmem aplikace procedur?*
6. *Jaký je rozdíl mezi procedurami a speciálními formami?*
7. *Proč nemůže být* `define` *procedura?*
8. *Co je aktuální vazba symbolu?*
9. *Jaké znáte typy chyb a jaký je mezi nimi rozdíl?*
10. *Co jsou to predikáty?*
11. *Z jakých částí se skládá cyklus REPL?*
12. *Jaké má výhody prefixová notace výrazů?*

Cvičení

1. Napište, jak interpret Scheme vyhodnotí následující symbolické výrazy:
```
(/ 2 3)          (2 + 3)    (+ -)            (1 2 3 4)       (modulo 10 3)
(+ (* 2 3) 4)    (+1 2)     (+ (*))          ((+ 1 2))       (rationalize 2 1/10)
(- (- 2))        (+)        (* (+) 2 (+))    (quotient 5)    (sqrt (* 1/2 0.5))
```
2. Rozeberte vyhodnocení následujících výrazů krok po kroku. Pokud při vyhodnocení dojde k chybě, zdůvodněte proč se tak stalo.
   (a) `(+ (* 2 3) (/ 2 8))`
   (b) `(if (= 0 (+ (- (+ 10 20) 30))) 7 8)`
   (c) `(* 2 sqrt (4))`
3. Napište, jak se vyhodnotí následující výrazy a zdůvodněte proč.
```
(define a 10)
(define b (+ a 1))
(define a 20)
b   ⟹   ???
```
4. Napište, jak dopadne vyhodnocení následujících výrazů a zdůvodněte proč.
```
(define define 1)
(define a 2)
a
```
5. Napište výsledky vyhodnocení následujících výrazů.
```
(if + 1 2)
(sqrt (if + 4 16))
(+ ((if - - +) 20))
(if (if (> 1 0) #f #t) 10 20)
```
6. Následující výraz
```
(define nedef ···)
```
doplňte tak, aby po jeho vyhodnocení byla na symbol `nedef` navázána nedefinovaná hodnota.

Úkoly k textu

1. Uvažujme následující výraz:
```
(if (if a
        (> a b)
        -666)
    (if b
        (- b)
        #t)
    10)
```

Proveďte analýzu předchozího výrazu a napište jak dopadne jeho vyhodnocení v závislosti na různých vazbách symbolů `a` a `b`. Soustřeďte se přitom na případy, kdy vyhodnocení končí úspěchem (tedy nikoliv chybou). Kolik musíme při našich úvahách rozlišit případů?

2. Předpokládejte, že máme na symbolech `a`, `b` a `c` navázány tři číselné hodnoty. Napište symbolický výraz, jehož vyhodnocením je vrácena *největší hodnota z* `a`, `b` *a* `c`. U výrazu důkladně otestujte jeho správnost pro vzájemně různé hodnoty navázané na symboly `a`, `b` a `c`. Je pochopitelné, že na dané symboly můžeme navázat nekonečně mnoho vzájemně odlišných číselných hodnot, ale z hlediska testování správnost výrazu stačí probrat jen konečně mnoho testovacích možností. Kolik jich bude? Zdůvodněte správnost vaší úvahy.

3. Prohlédněte si následující matematické výrazy a rozmyslete si, jak byste je vyjádřili pomocí symbolických výrazů v jazyku Scheme.

$$\frac{x+\sqrt{9}\cdot\dfrac{\log(z+5)}{\log(y)}}{x+y+z+4}, \qquad y\cdot\sqrt{\sin\left(\frac{4\cdot\operatorname{atan}(x)}{y}\right)}, \qquad \left(e^{y\cdot\log(z)}\right)^2\cdot\frac{x}{z\cdot\left(y+\frac{4+5^2-x}{4}\right)}.$$

Proveďte přepis výrazů a pomocí interpretu jazyka Scheme zjistěte, jaké budou jejich hodnoty v případě, že za x, y a z dosadíme číselné hodnoty 1, 2 a 3. Při přepisu použijte procedury navázané na symboly `log`, `exp`, `sqrt` a `atan`.

Řešení ke cvičením

1. Řešení po řadách: `2/3`, chyba, chyba, chyba, `1`; `10`, chyba, `1`, chyba, `2`; `2`, `1`, `2`, chyba, `0.5`.
2. Výsledky: (a) `25/4`; (b) `7`; (c) chyba (při vyhodnocování `(4)`). Podrobný rozbor udělejte dle definice 1.25.
3. Symbol `b` se na konci vyhodnotí na `11`. Protože vazba `b` byla definována jako hodnota první vazby symbolu `a` (což bylo `10`) zvětšená o jedna.
4. Vyhodnocení končí chybou při pokusu vyhodnotit druhý výraz. Při vyhodnocení prvního výrazu se totiž na symbol `define` naváže jednička, takže při vyhodnocení druhého výrazu se první prvek seznamu nevyhodnotí ani na proceduru ani na speciální formu.
5. Výsledky vyhodnocení: `1`, `2`, `-20`, `20`.
6. Například: `(define nedef (if #f #f))`