KATEDRA INFORMATIKY
PŘÍRODOVĚDECKÁ FAKULTA
UNIVERZITA PALACKÉHO

ÚVOD DO PARADIGMAT PROGRAMOVÁNÍ

DAVID SKOUPIL

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

Olomouc 2007

Abstrakt

Tento text distančního vzdělávání seznamuje s významem struktury počítačových programů a s programovacími jazyky, jakožto prostředky, které strukturu programů určují. Studující jsou konfrontováni s historií vzniku programovacích jazyků, se vznikem a charakteristikou jednotlivých paradigmat a se způsoby popisu programovacích jazyků. Zvláštní důraz je kladen na dva podstatné aspekty programovacích jazyků: procedury, jakožto elementární moduly programu, a typové systémy.

Cílová skupina

Text je primárně určen pro posluchače prvního ročníku bakalářského studijního programu Aplikovaná informatika na Přírodovědecké fakultě Univerzity Palackého v Olomouci. Může však sloužit komukoli se zájmem o počítače a programování. Text nepředpokládá žádné vstupní znalosti.

Obsah

1 Úloha strukturalizace v programování ... 9
1.1 Programovací jazyk, paradigma a struktura programu ... 9
1.1.1 Programování „ve velkém“ ... 9
1.1.2 Testování programů ... 11
1.1.3 Matematická verifikace programů ... 11
1.1.4 Faktory kvality softwaru ... 11
1.1.5 Paradigma programování ... 12
1.2 Programovací jazyky ... 13
1.2.1 John von Neumannův stroj ... 14
1.2.2 Jazyk stroje ... 15
1.2.3 Zdrojový a binární kód ... 15
1.2.4 K čemu jsou jazyky vyšší úrovně? ... 16
1.2.5 Překladače ... 17
1.2.6 Černobílý svět ... 18
1.3 Přehled základních programovacích paradigmat ... 19
1.3.1 Naivní paradigma ... 19
1.3.2 Procedurální paradigma ... 20
1.3.3 Objektově orientované paradigma ... 20
1.3.4 Funkcionální paradigma ... 20
1.3.5 Logické paradigma ... 21
1.3.6 Speciální paradigmata ... 21
1.3.7 Trendy jednotlivých programovacích paradigmat ... 22
1.4 Syntaxe a sémantika programovacích jazyků ... 23
1.4.1 BNF ... 24
1.4.2 EBNF ... 25
1.4.3 Syntaktické diagramy ... 26
1.4.4 Další způsoby popisu jazyka ... 27
2 Struktura programu ... 30
2.1 Základní rysy modulů ... 30
2.1.1 Pochopitelnost modulů ... 30
2.1.2 Samostatnost modulů ... 30
2.1.3 Kombinovatelnost modulů ... 30
2.1.4 Zapouzdření modulů ... 31
2.1.5 Explicitní rozhraní modulů ... 31
2.1.6 Syntaktická podpora modulů ... 31

2.2 Procedury jako jednoduché moduly 33
2.2.1 Procedury, matematické funkce a algoritmus..33
2.2.2 Struktura procedury 34
2.2.3 Hlavní a vedlejší efekt procedur..36
2.3 Procedury v programu 38
2.3.1 Zásobník programu..38
2.3.2 Aktivační strom 40
2.3.3 Rekurzivní procedury 41
2.3.4 Koncově rekurzivní procedury 43
2.3.5 Rozsah platnosti proměnných..45
2.4 Předávání parametrů procedurám...48
2.4.1 L-value a R-value 48
2.4.2 Předávání parametrů..49
2.4.3 Předávání parametrů hodnotou..50
2.4.4 Předávání parametrů odkazem 50
2.4.5 Předávání parametrů hodnotou-výsledkem 51
2.4.6 Předávání parametrů jménem 52
2.5 Styl programu 53
2.5.1 Volba vhodných jmen 54
2.5.2 Zavedení jmenných konvencí..54
2.5.3 Vkládání poznámek...54
2.5.4 Zarovnávání kódu..55
2.5.5 Příklad 55
3 Struktura dat 57
3.1 Typy dat v programovacích jazycích 57
3.1.1 Datové typy 57
3.1.2 Klasifikace datových typů 58
3.1.3 Jednoduché datové typy 58
3.1.4 Strukturované datové typy...58
3.1.5 Typový systém jazyka 59
3.1.6 Statická typová kontrola a manifestované typy..60
3.1.7 Dynamická typová kontrola a implicitní typy 60
3.1.8 Průběh typové kontroly 61
3.2 Techniky spojené s typovým systémem 62
3.2.1 Přetížení...62
3.2.2 Implicitní parametry procedur...63
3.2.3 Polymorfismus...64

3.2.4 Genericita 64
3.2.5 Přetypování a koerce 65
4 Závěr 68
5 Seznam literatury 69
6 Seznam obrázků 70
7 Rejstřík 71

1 Úloha strukturalizace v programování

Každý začátečník v oboru programování počítačů je konfrontován s pojmem programovací jazyk. Každý se jej chce co nejlépe naučit a mistrně jej používat. Pouze zasvěcení dokáží přimět počítač, aby realizoval jejich myšlenky. Programovací jazyk tak v očích začátečníka představuje pomyslný šém ke Golemovi zvanému počítač.

Jistým překvapením pak bývá zjištění, kolik programovacích jazyků již bylo ve světě vyvinuto. Proč vlastně potřebujeme tolik programovacích jazyků, jak se od sebe liší? O co jde těm podivínům, kteří vytvářejí stále nové a nové jazyky, které se mnohdy ani nedočkají komerční implementace? Nestačilo by několik programovacích jazyků, s jejichž pomocí by bylo možno efektivně psát počítačové programy?

1.1 Programovací jazyk, paradigma a struktura programu

Studijní cíle: Po prostudování kapitoly bude studující schopen zdůvodnit význam struktury v programování a objasnit faktory kvality softwaru.

Klíčová slova: Správnost a robustnost, programovací jazyk, programovací paradigma

Potřebný čas: 30 minut.

Programovací jazyky skutečně původně vznikly pouze jako pomůcka pro urychlení psaní počítačového kódu. Brzy se však ukázalo, že takovéto chápání jazyků nebude dostatečné. Odborníci se začali zamýšlet nad otázkou samotného programování a nad vztahem programovacích jazyků a programování.

1.1.1 Programování „ve velkém"

Programování velkých projektů je kvalitativně odlišné od vytváření malých programů.

Důvodem k úvahám o principech programování byla rostoucí komplexnost vytvářených programů, která s sebou nesla stále větší množství drobných, těžce odhalitelných programátorských chyb. v roce 1962 dokonce jedna „drobná" programátorská chyba způsobila zkázu americké kosmické sondy Mariner I a finanční ztrátu mnoha milionů dolarů[1]. Sondu vidíme na obrázku Obr. 1

[1] v kódu kosmické sondy Mariner 1 tehdy chyběl jen jeden jediný znak: „-" (mínus).

Obr. 1 Mariner 1

Chyby v programech jsou nevyhnutelné – lze je omezit ale ne zcela odstranit.

Původní předpoklad přirovnával chyby programátorské k chybám pravopisným. Stačí se tedy pořádně naučit pravopis a máme vyhráno: chyby vymizí! Ukázalo se však, že tato analogie při zvětšující se velikosti projektu přestává platit. Programátorské chyby ve velkých projektech musí být zcela jiného druhu než běžné pravopisné chyby. Nelze se jim vyvarovat a dělají je stejně tak programátoři zkušení i nezkušení. Prohřešky pravopisu navíc málo kdy učiní text nesrozumitelným a nemívají velký dopad (nepočítáme-li sníženou známku z diktátu). Počítače, na rozdíl od lidí, mají však nulovou toleranci a sebemenší chyba v programu může mít katastrofální důsledky.

Vytváření rozsáhlých programů, nazývané „*programming-in-large*“ je značně odlišné od programů drobných. Rozdíl není jen kvantitativní, ale i kvalitativní: nelze jednoduše vzít nástroje a metodologii pro vytváření malých programů a uplatnit je na rozsáhlý projekt.

Průvodce studiem

V mnoha akčních sci-fi filmech je důvěřivý divák konfrontován s následujícím scénářem: hmyz, pokud možno nepříjemně bodavý až jedovatý, geneticky zmutuje a mnohonásobně zvětší svoji velikost. Svou přítomností pak začne terorizovat nějaké malé a poklidné městečko na středozápadě, a to až do doby, než se objeví svalnatý a sympatický zachránce.

Autorům těchto scénářů však uniká jedna podstatná věc. Hmyzí tělo je podporováno pouze chrupavčitým obalem, tzv. kutikulou. Zvětšujeme-li mechanicky takovéhoto živočicha, bez podstatných strukturálních změn, roste jeho hmotnost strměji, než nosnost podpůrných tkání. Geneticky „nafouklá“ vosa by tak bezmocně ležela na zemi a těžko by se divoce proháněla nad hlavami obyvatel městečka.

S programováním a programovacími technikami je to podobně: co skvěle funguje v malém se při „nafouknutí“ stává těžkopádným a nepoužitelným.

V případě sondy Mariner 1 američtí kongresmani požadovali *úplné* ověření spolehlivosti všech použitých programů, aby se situace již nikdy neopakovala. Bylo proto nezbytné začít přemýšlet

o tom, jak chyby z programů efektivně odstraňovat a jak programy vytvářet, aby riziko vzniku chyby nebo její dopad byly co nejmenší. Ověřování programů se vydalo dvěma nezávislými směry.

1.1.2 Testování programů

První možností, která se sama nabízela, bylo zdokonalení testování programů. Testování bylo a dodnes je důležitou součástí nasazení každého programu. Testování můžeme rozdělit na dva způsoby:

Testování programů se musí zaměřit na vnitřní strukturu..

- *Black box* testování. k programu se stavíme jako k černé skříňce, kdy nevíme nic o jeho vnitřní struktuře. Snažíme se pak otestovat všechny funkce, které po programu požadujeme.
- *White box* testování. Při testování známe vnitřní strukturu programu a snažíme se cílenými testy ověřit, že všechny větve a části programu pracují tak, jak očekáváme.

Testování však není všemocné. Výše zmíněný program pro kosmickou misi k planetě Venuši však byl testován na 100 průchodů v trenažéru bez nejmenší zjištěné chyby a byl čtyřikrát úspěšně použit pro lunární expedice! Není tak těžké spočítat, že program, který obsahuje řádově miliony řádků a který se může nacházet ve stovkách milionů stavů, není zrovna jednoduché úplně otestovat v relativně krátké době.

1.1.3 Matematická verifikace programů

Nově vzniklou informatickou disciplinou byla matematická verifikace programů. Pomocí nástrojů matematické logiky se snažíme dokázat (ve smyslu matematického důkazu), že program provádí potřebnou činnost. I když je matematická verifikace důležitou součástí teoretické informatiky, její použití pro větší programy je velmi pracné. Nasazení na komplexní projekty je pak téměř nemožné.

Matematická verifikace velkých programů je nemožná.

Verifikace programů se tak používá zejména na ověření funkčnosti některých klíčových algoritmů. Význam má zejména při ověření algoritmů, které budou implementovány hardwarově.

1.1.4 Faktory kvality softwaru

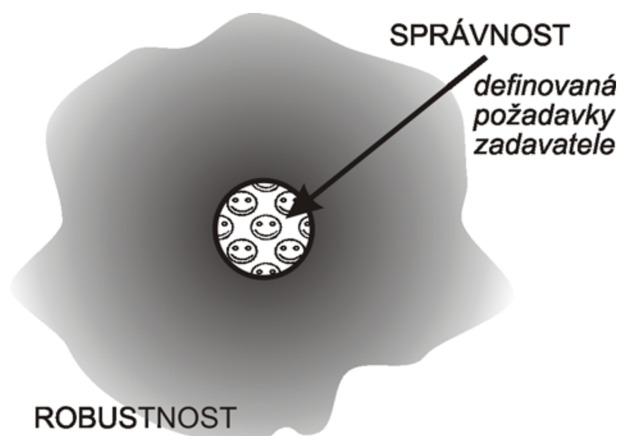
Původně naivně postavenou otázku „spolehlivosti" programů lze rozvést například podle [Meyer01] následujícím způsobem. Rozlišujeme v zásadě dva faktory, určující kvalitu programu: *vnější* a *vnitřní*. Vnější faktory jsou viditelné uživateli programu, vnitřní faktory zůstávají před uživatelem skryty, jsou viditelné pouze počítačovým profesionálům.

Robustnosti programu se dosahuje hůře než správnosti.

Jako příklad vnějších faktorů kvality můžeme uvést:

- *Správnost*, tj. schopnost programů přesně vykonávat svou úlohu tak, jak je definována v požadavcích zadavatele.
- *Robustnost*, tj. schopnost programů reagovat na abnormální podmínky.
- Další faktory, jako rychlost, efektivnost, rozšiřitelnost, kompatibilitu, cenu programu a další.

Podstatným je zejména vztah robustnosti a správnosti programu. Zatímco *správný* program reaguje správně v situaci popsané zadáním, *robustní* program by měl být schopen ignorovat evidentně špatná data a nesmí na základě neočekávaných údajů způsobit katastrofu. Robustnost je jistě nadmnožinou správnosti programu. Lze ji však mnohem obtížněji definovat a testovat – nikde totiž nemáme přesně vymezeno, jaké situace mohou nastat. Vztah správnosti a robustnosti je schematicky znázorněn na obrázku Obr. 2.



Obr. 2 Vztah správnosti a robustnosti

1.1.5 Paradigma programování

Programovací paradigma určuje vnitřní strukturu programu.

Vnitřní faktory kvality nelze tak snadno vyjmenovat, už proto, že jsou viditelné jen odborníkům. Souvisí zřejmě s vnitřní strukturou programu. Nový vítr do celé problematiky vnáší E. W. Dijkstra v knize „Structured Programming" ([Dijkstra72]): kvalitní program lze vytvořit pouze tehdy, zaměříme-li se na jeho vnitřní strukturu.

Vnějších faktorů kvality je možno dosáhnou pouze na základě existence faktorů vnitřních, tedy tehdy, bude-li mít program svoji jasně definovanou *vnitřní strukturu*, podléhající určitému stylu. Tento styl vnitřní struktury programu se nazývá *programovací paradigma*.

Dostáváme se tak k odpovědi na položenou otázku opodstatněnosti existence takového množství programovacích jazyků. Je jasné, že vnitřní struktura programu je velmi úzce spojena s použitým programovacím jazykem. A jsou to právě programovací jazyky, které nám umožňují (nebo dokonce nutí či naopak zabraňují) vytvářet programy se strukturou, odpovídající některému z paradigmat. Každý z nově vytvářených programovacích jazyků se snaží lépe nebo alespoň „jinak" podporovat jedno nebo kombinaci několika programovacích paradigmat.

Průvodce studiem

Přehled paradigmat programování a jejich základních vlastností uvedeme v kapitole1.3. Paradigma však skutečně oceníme a pochopíme až tehdy, když v duchu daného paradigmatu napíšeme několik programů

Shrnutí

Počítače jsou zcela přesné a jakákoli chyba v programu způsobí nefunkčnost celého programu.
Při řešení rozsáhlého projektu nemůžeme použít metody vhodné pro malé projekty.
Testování i matematická verifikace programů jsou u větších systémů obtížné.
Programy lze testovat pomocí white box testování a black box testování. Obě metody je třeba kombinovat.
Robustnost je mnohem složitější požadavek než správnost.
Pro tvorbu dobrých systémů je třeba dbát na strukturu programu – programovací paradigma.

Pojmy k zapamatování

- Externí a interní faktory softwaru.

- správnost a robustnost,
- Black box a White box testování,
- verifikace programů,
- paradigma programování.

Kontrolní otázky

1. *Vyjmenujte externí faktory kvality softwaru*
2. *Vysvětlete, v čem spočívá rozdíl mezi robustností a správností programu.*
3. *Popište pojem paradigma programování*

Cvičení

1. Algoritmus na výpočet největšího ze třech čísel lze formálně zapsat takto:

$$\max(a,b,c) = \begin{cases} a, a > b \wedge a > c \\ b, b > a \wedge b > c \\ c, c > a \wedge c > b \end{cases}$$

   Jak byste metodou white-box testovali program, který implementuje tento algoritmus?

Úkoly k textu

1. Představte si automatickou pračku jako jednoduchý program. Zkuste si formulovat uživatelské požadavky na **správnost** fungování pračky.

   Má-li být pračka robustní, musí odpovídajícím reagovat i na abnormální situace. Pokuste se vyjmenovat alespoň pět nestandardních situací, na které by pračka měla reagovat. Jednou z takovýchto situací může být například stav, kdy není puštěna voda. Dokážete vyjmenovat více než pět situací?

Řešení

1. Algoritmus rozlišuje mezi třemi případy. Je proto potřeba program otestovat tak, abychom prošli všemi možnými větvemi programu. Například:

```
max(12, 5, 7)
max(5, 12, 7)
max(5, 7, 12).
```

1.2 Programovací jazyky

Studijní cíle: Po prostudování kapitoly bude čtenář rozumět rozdílu mezi jazykem stroje a programovacím jazykem. Bude umět popsat používané způsoby překladu.

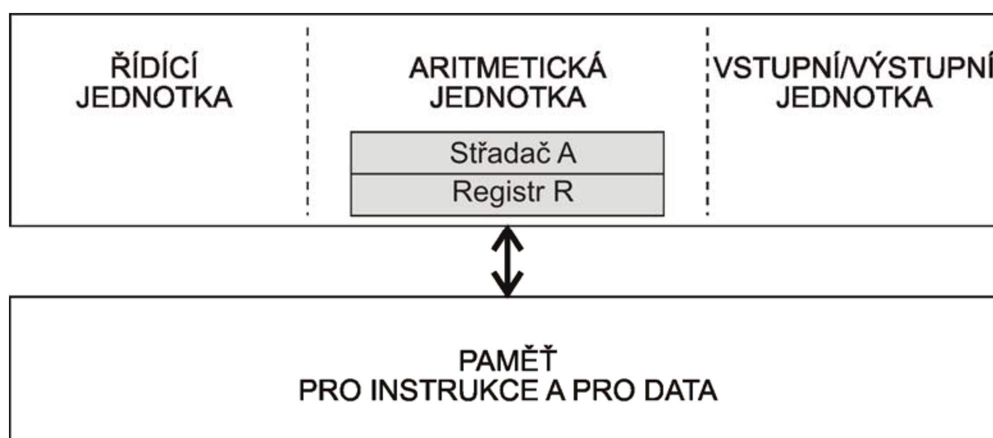
Klíčová slova: Von Neumannův stroj, jazyk stroje, zdrojový kód, binární kód, překladač, kompilátor, interpret.

Potřebný čas: 1 hodina 30 minut.

1.2.1 John von Neumannův stroj

Podoba programování, podoba programovacích jazyků a někdy i podoba programátorů je dána základním principem práce počítače. Model dnešních počítačů je znám jako *von Neumannův stroj*[1] a byl poprvé popsán v drobné práci Burkse, Goldstina a von Neumanna z roku 1947. Jednou ze základních vlastností von Neumannova stroje je, že data a instrukce sdílejí tutéž paměť. John von Neumann ve své době asi netušil, že tímto návrhem na mnoho desetiletí ovlivní vývoj informatiky.

Von Neumannův stroj určuje výpočetní model soudobých počítačů.



Obr. 3 Organizace John von Neumannova stroje

Stroj se skládal z řídící jednotky, aritmetické jednotky, jednotky vstupů a výstupů a z paměti. v paměti byly uloženy jednak instrukce programu, jednak data. Jednotlivé instrukce byly postupně brány z paměti a vykonávány, výsledky byly ukládány zpět do paměti. Řízení výpočtu probíhalo pomocí instrukce GOTO, kdy byla následující instrukce vzata z paměťové buňky zmíněné v instrukci GOTO.

Data a program sdílejí stejnou paměť.

Von Neumann charakterizoval počítač tak, že: „Využitelnost počítače je dána tím, že se některé části programu mohou vykonávat vícekrát – počet opakování je buď předem dán nebo závisí na průběhu výpočtu."

Průvodce studiem

Vliv Johna von Neumanna na informatiku je zásadní. Jeho model výpočtu ovlivnil konstrukci počítačů, programovacích jazyků, překladačů. Zkusme si však položit otázku, jestli je tento model výpočtu ideální?

Jako problém dnes mnohdy vidíme oddělení velmi rychlého procesoru od paměti (jsou spojeny tzv. sběrnicí, která má omezenou přenosovou rychlost). Možná bychom dosáhli lepších nebo alespoň kvalitativně jiných výsledků, kdyby konstrukce počítačů blíže modelovala nervový systém živočichů. „Výpočetní model" nervového systému není založen na rychlosti a komplexnosti procesorů (neuronů), ale na jejich množství a komplexitě propojení. I když vědci už dnes spekulují např. o kvantových počítačích, jedná se pořád více o vědeckou fikci než blízkou realitu.

[1] Někdy také hovoříme o von Neumannově principu. Von Neumann byl matematik maďarského původu narozený v roce 1903.

Obr. 4 John von Neumann

1.2.2 Jazyk stroje

Strojový kód je nepřehledný a nelze v něm efektivně programovat.

Program, který řídil tento stroj (a se kterým se setkáváme i u dnešních počítačů) se nazýval *jazyk stroje* nebo též *strojový kód*. Byl jen velmi těžce čitelný a obtížně srozumitelný. k jistému zlepšení došlo po zavedení tzv. *jazyka symbolických adres*, který je též nepřesně znám pod názvem *assembler*. Assembler umožnil nahradit kódy instrukcí mnemotechnickými zkratkami a jednotlivá paměťová místa bylo možno označit symbolem. Jazyk stroje spolu s jazykem symbolických adres dnes řadíme mezi jazyky nízké úrovně a v našem textu je nebudeme považovat za skutečné programovací jazyky. Od programovacích jazyků – jazyků vyšší úrovně – očekáváme možnost vyšší úrovně abstrakce.

V 50. letech minulého století panovalo přesvědčení, že dobré programy mohou být vytvořeny pouze v jazyku symbolických adres. Tvorba takovýchto programů však byla velmi nákladná a pomalá. Proto neustávala snaha o automatizaci tvorby assemblerovských programů – vytváření jazyků vyšší úrovně.

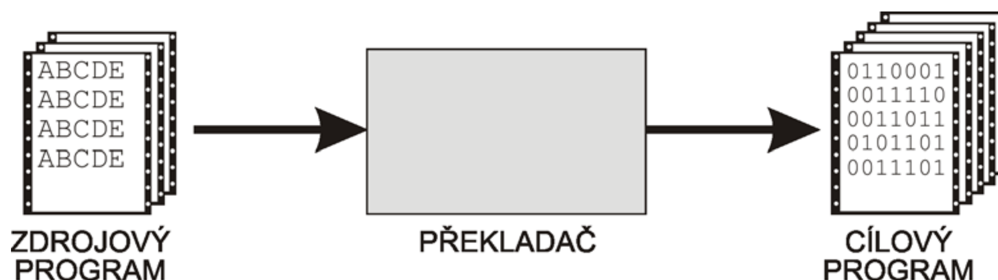
```
0000001010111001010     LOAD I
0000001011111001000     ADD J        k := i + j
0000001100111010100 0   STORE K
```

Obr. 5 Segment programu zapsaný ve strojovém kódu, v jazyku symbolických adres a ve vyšším programovacím jazyku

1.2.3 Zdrojový a binární kód

Prvním úspěchem, který přetrvává až dodnes, byl jazyk pro překlad matematických výrazů (FORmula TRANslation) zvaný FORTRAN. Program napsaný ve Fortranu nemohl být přímo vykonávaný strojem, musel být nejprve přeložen do jazyka stroje. To ostatně platí pro všechny ostatní vyšší programovací jazyky.[1] Program zapsaný ve vyšším programovacím tak nazýváme *zdrojový program* nebo též *zdrojový kód*. Výsledek překladu – cílový program v jazyce stroje – nazýváme *binární kód*.

Jazyky vyšší úrovně vyžadují překladač pro převedení do jazyka stroje.



Obr. 6 Překlad programu ze zdrojového do cílového jazyka

Mechanismus, který zajišťuje překlad ze zdrojového do cílového jazyka nazýváme překladač.

1.2.4 K čemu jsou jazyky vyšší úrovně?

Jako vždy, když vzniklo něco nového, měl i programovací jazyk Fortran mnoho nepřátel. Následující text se snaží sumarizovat argumenty odpůrců a příznivců vyšších programovacích jazyků obecně.

Jazyky vyšší úrovně umožňují efektivní vytváření programů.

Odpůrci:

- Program neběží přímo na stroji pro který je určen. Je třeba další program, čas a paměť na překlad do jazyka stroje.
- Program není tak efektivní jako program „ručně ušitý“ ani co do rychlosti, ani co do velikosti kódu.

Příznivci:

- Programy jsou přehledné, snadno pochopitelné, opravitelné a rozšiřitelné.
- Programy jsou přenositelné z jedné hardwarové platformy na druhou (což je původně neplánovaný vedlejší efekt). Stačí, aby byly počítače všech typů vybaveny překladačem téhož vyššího jazyka.
- Spousta programů (zejména knihoven) by vůbec nevznikla, kdyby byli programátoři nuceni psát ve strojovém kódu.

Triumfem vyšších programovacích jazyků bylo, když Dennis Ritchie použil jeden z nich na vytvoření operačního systému. Operační systémy, jakožto základní programy ovládající hardware počítače, byly totiž vždy výsadní doménou programátorů píšících v jazyku stroje. Zmíněným programovacím jazykem byl jazyk C a jednalo se o operační systém UNIX.

---

[1] v jistém smyslu to platí i pro programy v jazyku symbolických adres. Programy v tomto jazyce musely být „sestaveny“ ("assemblovány") pomocí překladače, zvaného assembler. Měli bychom proto správně říkat místo „programy psané v assembleru“ „programy psané pro assembler“. Zatímco u jazyka symbolických adres spočíval překlad ve více méně formálním nahrazení kódů instrukcí a adres jejich číselnými ekvivalenty, jde u vyšších programovacích jazyků o kvalitativně zcela jinou formu konverze do nižšího jazyka.

Dnes si již programování bez jazyků vyšší úrovně neumíme vůbec představit a assembler se používá již jen pro velmi specifické úkoly.

Průvodce studiem

Programovacích jazyků je mnoho a v tomto textu jsme zatím zmínili jazyk Fortran a jazyk C. Později se v průběhu studia ještě setkáte s jazykem Scheme, jazykem Visual Basic, jazykem C# (čtete [si: ša:rp], v hudební notaci označení pro Cis) a možná ještě s dalšími. Je jistě nutno seznámit se s různými jazyky, abychom byli schopni podívat se na programování počítačů z různých úhlů. Současně je však třeba mít na paměti, že jazyk je jen technickou pomůckou pro zápis programu – cílem je naučit se programovat, nikoli zvládnout programovací jazyk.

1.2.5 Překladače

S uvedením vyšších programovacích jazyků vznikla také nová disciplina v oblasti počítačové vědy: konstrukce *překladačů*, speciálních programů, jejichž účelem je transformovat program z jednoho jazyka do druhého. Ve většině případů je zdrojovým jazykem překladače nějaký vyšší programovací jazyk a cílovým jazykem je jazyk stroje.

Překladače dělíme na interprety a kompilátory.

O překladačích, jejichž cílovým jazykem je jazyk stroje, říkáme, že jsou *nativní* nebo že generují *nativní kód* (*native code generation)*. Mnohé překladače experimentálních jazyků však mohou mít za cílový program nějaký jiný programovací jazyk (nejčastěji jazyk C). Překlad pak probíhá ve dvou fázích: ze zdrojového jazyka do jazyka C a z dále z jazyka C do strojového kódu. Tento přístup volí autoři překladačů zejména tehdy, když by se vzhledem k využití daného jazyka nevyplatilo vytvářet mnoho nativních překladačů pro nejrůznější hardwarové a softwarové platformy. Pro překlad z jazyka C do strojového kódu lze totiž využít na jednotlivých platformách již existující překladače.

Zároveň se vytvořily dva základní přístupy k překladu kódu: přístup kompilační a přístup interpretační. Podle toho dnes dělíme překladače na *kompilátory* a *interprety*.

- Kompilátory jsou překladače, které přeloží celý kód ze zdrojové formy do jazyku stroje najednou. Překlad celého kódu tedy předchází jeho spuštění.
- Interprety jsou překladače, obcházející přímý překlad do jazyka stroje. Jednotlivé výrazy ze zdrojového programu jsou těmito překladači postupně interpretovány a cílový program jako celek nevzniká. Překlad tak probíhá vlastně souběžně s během programu.

Oba přístupy mají své výhody i nevýhody. Zatímco kompilátory jsou trvale v oblibě, interprety zažívají svá období rozvoje i úpadku.

Základní výhodou kompilátoru je, že výsledný kód je velmi rychlý. Tato výhoda je vykoupena faktem, že libovolná změna programu vyžaduje rekompilaci celého programu. Vedlejším, ale dost podstatným efektem kompilace je také celková kontrola syntaktické správnosti programu. Interprety jsou obecně pomalejší, náročnější na paměť a program není schopen fungovat bez přítomnosti překladače. Mnoho chyb u interpretovaných programů zjistíme až za běhu. Na druhé straně je možno kód snadno vytvářet i modifikovat a stejný program může běžet na různých platformách.

Průvodce studiem

Mezi populární interpretované programovací jazyky dnes patří například jazyk PHP, který se využívá hlavně pro programování WWW aplikací. Vzhledem k interpretaci je

program napsaný v PHP platformově nezávislý; můžete tedy program vyvíjet například na osobním počítači s MS Windows a po odladění jej nakopírovat na webový server se systémem UNIX. Výhody interpretování se projeví i v tom, že části programu lze jednoduše vložit do těla webové stránky a webový server program spustí před zasláním stránky po internetu.

Dalším známým interpretovaným jazykem používaným na internetu je jazyk JavaScript. Zatímco PHP je spouštěný již na webovém serveru, JavaScript je interpretovaný až WWW prohlížečem.

1.2.6 Černobílý svět

Výsledkem kompilace nemusí být vždy strojový kód, kompilovat lze i do mezilehlého kódu.

Protože svět není jen černobílý, i u překladačů se setkáváme s přístupy, které jsou v „šedé zóně". Kompromis mezi kompilací a interpretací programu přinesl jazyk Java, vyvinutý firmou Sun Microsystems. Program v Javě je kompilovaný to tzv. *byte kódu*. Při překladu dojde ke kontrole správnosti programu a k jeho optimalizaci. Byte kód však není jazykem stroje, je platformově nezávislý a na počítačích musí být interpretován tzv. *virtuálním strojem* (*Java virtual machine*).

S dále vylepšeným přístupem přišla o několik let později firma Microsoft. Programy v prostředí .NET se překládají do *mezilehlého jazyka* nazývaného *Microsoft intermediate language* (*MSIL*). Těsně před spuštěním je potom MSIL přeložen nativním překladačem, který je součástí .NET Framework, do strojového kódu. Konečný překlad je prováděn „na poslední chvíli" a získal proto název „*just-in-time*" neboli *JIT* překad.

Ačkoliv všeobecně platí, že některé jazyky preferují spíše interprety a některé jazyky dávají přednost kompilátorům, jedná se především o otázku technickou a na strukturu jazyka a strukturu programu, kterými se především zabýváme, to nemá příliš velký vliv.

Shrnutí

Počítače jsou založeny na modelu von Neumannova stroje, řídí se strojovým kódem.
Zdrojový kód píšeme v programovacích jazycích vyšší úrovně, které poskytují vyšší úroveň abstrakce.
Překlad zdrojového kódu do strojového kódu zajišťují překladače.
Základní typy překladu jsou kompilace a interpretace, existují však i kombinace obou postupů.

Kontrolní otázky

1. *Charakterizujte princip práce von Neumannova stroje.*
2. *Definujte pojmy zdrojový kód a strojový kód.*
3. *Popište základní dvě formy překladu programu.*
4. *Vyjmenujte výhody a nevýhody kompilace programů.*
5. *Kde se setkáváme s pojmy byte-code a MSIL.*

Úkoly k textu

1. Pokuste se na internetu najít jména 10 různých programovacích jazyků, které typicky využívají kompilační překladače a 10 jazyků využívajících interprety. Ke každému z nich zjistěte, k čemu se typicky využívá.

2. Na WWW snadno poznáte, jestli je stránka tzv. statická nebo jestli je výsledkem spuštění nějakého programu. Adresa statických stránek končí typicky příponou htm, adresa stránek psaná ve jazyku Visual Basic končí na `asp` a adresy stránek psaných v PHP končí na `php`. Projděte své oblíbené webové stránky a všimněte si, které jsou statické a které jsou výsledkem spuštění programu.

Pojmy k zapamatování

- Von Neumannův stroj,
- jazyk stroje,
- zdrojový kód, binární kód,
- překladač, kompilátor, interpret ,
- nativní překladač,
- byte-code, MSIL.

1.3 Přehled základních programovacích paradigmat

Studijní cíle: Po prostudování kapitoly bude mít studující přehled o konkrétně používaných programovacích paradigmatech. Bude umět navrhnout použití jednotlivých paradigmat a znát jazyky podporující jednotlivá paradigmata.

Klíčová slova: Naivní paradigma, procedurální paradigma, objektově orientované paradigma, funkcionální paradigma, logické paradigma.

Potřebný čas: 1 hodina 30 minut.

Uveďme si přehled základních programovacích paradigmat – vyzkoušených a ověřených programovacích stylů, vedoucích k tvorbě kvalitního softwaru. Tento výčet si neklade nároky ani na úplnost, ani na časovou neměnnost. v průběhu času jistě mnoho zažitých paradigmat vymizí a objeví se paradigmata nová.

Průvodce studiem

Pokud jste doposud nikdy aktivně neprogramovali, možná vám výčet jednotlivých paradigmat přijde poněkud nesrozumitelný. Není divu. Jednotlivá paradigmata plně pochopíte teprve tehdy, až si v nějakém konkrétním programovacím jazyce, podporujícím dané paradigma, napíšete alespoň středně velký projekt. Proto doporučujeme, abyste se k této kapitole časem vrátili a zkusili najít ve svém programu rysy konkrétního paradigmatu.

Seznam paradigmat však přesto uvádíme už teď. Jinak byste snadno mohli nabýt dojmu, že váš první čerstvě zvládnutý programovací styl je ten jediný možný.

1.3.1 Naivní paradigma

Programovací jazyky: Klasický jazyk BASIC.

Použití: Toto paradigma je radno nepoužívat.

Naivní programovací paradigma je typické pro počítačové laiky a začátečníky. Vyznačuje se nekoncepčností a chaotičností a proto bychom jej vlastně ani neměli nazývat paradigmatem.

Programovací jazyky, podporující toto paradigma, jsou minimálně strukturované, neumožňují modularitu a poskytují minimální prostředky pro datovou abstrakci.

Například ve standardním BASICu byly řádky číslovány pořadovým číslem. Při přidání kódu bylo nutné provádět přečíslování řádků. Často používanou konstrukcí byl skok (příkaz GOTO) na jiný programovací řádek.[1] Programy byly těžko pochopitelné a nepřehledné.

1.3.2 Procedurální paradigma

Procedurální paradigma je považováno za klasické.

Programovací jazyky: Klasické jazyky jako Fortran, C, Modula2 nebo Pascal.

Použití: Toto paradigma má univerzální použití, rozšířené je obzvláště v komerční sféře. Je vhodné zejména pro malé projekty.

Procedurální paradigma se též někdy nazývá *klasické* nebo *imperativní*. z prvního synonyma lze vyčíst, že jde o jedno z nejstarších paradigmat[2], druhé synonymum napovídá, že základní úlohu v tomto paradigmatu hrají příkazy. Typické pro procedurální paradigma je, že průběh výpočtu je dán *sekvencí* po sobě jdoucích instrukcí (příkazů), přičemž rozhodující roli zde hraje přiřazovací příkaz. Výrazné využití mají také cykly.

V procedurálním paradigmatu je program strukturalizován v závislosti na funkcionalitě – procedurách. Můžeme tedy říci, že jeden modul programu *vykonává* jednu akci, zatímco druhý modul vykovává jinou akci. Stejné akce přitom mohou být vykonávány nad různými daty.

1.3.3 Objektově orientované paradigma

Objektově-orientované paradigma je soudobým průmyslovým standardem.

Programovací jazyky: k historickým jazykům patří např. Simula a Smalltalk, v dnešní době snad komerčně nejvyužívanějším jazykem je objektově orientované rozšíření jazyka C zvané C++. k dalším perspektivním objektovým jazykům patří Java, C# nebo třeba Eiffel.

Použití: Původní použití bylo soustředěno do oblasti simulace a umělé inteligence. v dnešní době má univerzální použití, vhodné je zejména pro rozsáhlé a komerční projekty

Základními programovacími prvky v objektově orientovaném programování jsou útvary zvané *objekty*. Tyto útvary v zásadě modelují objekty reálného světa: osoby, předměty, dokumenty, události. Každý objekt je nositelem jistých informací o sobě samém (tzv. stavu) a má schopnost na požádání tento stav měnit. Objekt „faktura“ z podnikového informačního systému má svoje číslo a platební informace. Svůj stav může změnit, například z neproplacené faktury se může stát faktura proplacená.

V objektově orientovaném paradigmatu je program strukturalizován nikoli podle procedur, které se vykonávají, ale podle objektů, které se v systému vyskytují. Objekt v sobě *zapouzdřuje* jak data, tak procedury pracující nad těmito daty. Průběh výpočtu je pak určen *posíláním zpráv* mezi jednotlivými objekty. Pomocí zpráv objektům říkáme, jak mají změnit svůj stav.

Objektově orientované programování přineslo zásadní posun v kvalitě programování velkých systémů a umožnilo rychlejší vývoj programů.

1.3.4 Funkcionální paradigma

Programovací jazyky: Typickým reprezentantem je Lisp nebo Scheme (dialog Lispu), z novějších experimentálních jazyků lze jmenovat ML, Miranda či Haskell.

Funkcionální a logické paradigma nacházejí použití zejména ve výzkumu a výuce.

---

[1] v programátorském slangu se někdy hovořilo o „hop-sem-hop-tam“ programování.

[2] Procedurální paradigma soupeří co do data vzniku s paradigmatem funkcionálním. Klasické se nazývá možná spíše proto, že se nejméně odlišuje od dobře známého jazyka symbolických adres.

Použití: Původní použití paradigmatu vzešlo z oblasti umělé inteligence, dnes je používáno především ve výuce a výzkumu.

V případě funkcionálního paradigmatu je průběh výpočtu založen na postupném *aplikování funkcí*. Funkce zde bývají aplikovány na výsledky jiných funkcí. Pro toto paradigma je typické, že se zde nepoužívá přiřazovacího příkazu a silné místo zde zaujímají tzv. funkce vysoké úrovně a rekurze. Po procedurálním paradigmatu jde o druhé nejstarší paradigma.

Rozsáhlé programy psané funkcionálním stylem mohou působit poněkud nepřehledně. Krátké programy, naopak, mohou využít tradiční expresívnosti tohoto paradigmatu a s minimálním množstvím použitých prostředků vyjádřit základní algoritmy.

1.3.5 Logické paradigma

Programovací jazyky: Prolog (PROgramming in LOGic) a jeho dialekty.

Použití: Značného rozšíření se dočkal v oblasti umělé inteligence (zejména znalostních systémů). Komerční využití tohoto paradigmatu jsou zřídká.

V logickém programování je program počítači předložen ve formě množiny *faktů* (známých skutečností) a *pravidel* - tzv. *klauzulí* (implikací, umožňujících z platnosti jednoho faktu odvodit jiný). Průběh výpočtu založen na metodě *unifikace* a *zpětného řetězení*: programátor předloží systému nějaké tvrzení (cílovou hypotézu) a systém se na základě faktů a klauzulí programu snaží dané tvrzení dokázat.

1.3.6 Speciální paradigmata

Často se setkáváme s jazyky, které vycházejí z některého paradigmatu a přidávají k němu podporu pro další potřebnou činnost. Vznikne-li celá skupina takovýchto jazyků a vytvoří se programovací styl, můžeme hovořit o vzniku nového paradigmatu. z mnoha možností uvedeme alespoň následující.

Paralelní a distribuované programování

S existencí více-procesových operačních systémů a počítačů propojených rychlými sítěmi bylo zapotřebí vyvinout jazyky, umožňující efektivní paralelizaci[1] a distribuci[2] programu. Klíčovým pojmem je tu komunikace a synchronizace mezi procesy. k jazykům patří například jazyk MPD, s podporou se stále více setkáváme i u klasických jazyků (Java, C#).

Datové programování

Počítače jsou tradičně využívány pro práci s velkými objemy dat. Vnikaly tak jazyky, umožňující snadnou manipulaci s daty, přímé napojení na databázi a snadnou tvorbu tiskových sestav. Tyto jazyky většinou vycházejí z imperativního paradigmatu, mohou v sobě ale zahrnovat i objektové prvky. Historickým příkladem je programovací jazyk COBOL.[3]

Webové programování

Programování WWW aplikací v sobě nese jistá specifika. Potřebná je možnost integrovat kód s WWW stránkami, přistupovat do databáze, pracovat snadno s textem atd. Typickými jazyky jsou PHP a Visual Basic Script.

Textové programování

---

[1]Dekompozici programu na úlohy, které mohou být souběžně zpracovávány různými procesory sdílejícími stejnou paměť.

[2] Dekompozici programu na úlohy, které mohou být souběžně zpracovávány různými počítači.

[3] I když COBOL patří mezi dědečky mezi programovacími jazyky, jedná se o jeden z nejpoužívanějších jazyků na světě z hlediska množství dosud běžících programů, které jsou v něm napsány.

V některých případech potřebujeme snadno a rychle vytvářet programy na práci s textem: vyhledávání výrazů, extrakce řetězců nebo zpracovávání regulárních výrazů. Jazyky, které to dovolují, můžeme zařadit do oblasti textového programování. Reprezentantem může být jazyk PERL.

Průvodce studiem

I když se může zdát, že programovací paradigma je jednoznačně dáno použitým jazykem, v praxi je situace komplexnější. Různé jazyky různě kvalitně podporují některé paradigma. Je dřina programovat v jazyku C objektově orientovaně. Stejně tak je téměř nemožné napsat v PROLOGu imperativní kód.

Jedna věc jde ale vždy snadno: v jakémkoli jazyku lze vystřihnout nejlepší ukázku naivního paradigmatu.

1.3.7 Trendy jednotlivých programovacích paradigmat

Programovací paradigmata se vyvíjejí. Některá paradigmata zanikají, jiná se slučují nebo vznikají zcela nová.

Předem je třeba poznamenat, že tento odstavec je čistě subjektivním hodnocením autora a že jiní odborníci mohou zastávat jiné názory. Nemá cenu diskutovat naivní paradigma. I v evoluci programátorů (homo computeris) platí známá poučka o tom, že ontogeneze je zkrácená fylogeneze, takže s naivním přístupem se budeme setkávat asi stále. Lze jen doufat, že produkty tohoto paradigmatu nedojdou komerčního úspěchu, aby pak po léta ztrpčovaly život zcela nevinným uživatelům.

Procedurální paradigma ve své čisté podobě zaniká a stále více se směšuje s ostatními paradigmaty, čehož nejlepším příkladem jsou jazyky z rodiny C++ (C++, C#, Java). v současné době snad všechny moderní jazyky univerzálního určení nesou stopu procedurálního paradigmatu. Toto paradigma tak často plní úlohu skutečného „klasického“ paradigmatu, ze kterého ostatní jazyky vycházejí a ke kterému přidávají další specifické rysy.

Objektově orientované programování se stalo stálicí poslední doby a zdá se, že z programátorského nebe jen tak nezmizí. v dnešní době téměř každý jazyk s ambicemi na komerční použití nese znaky objektové orientovanosti. Toto paradigma se promítá i do jiných oblastí informatiky: vznikají objektově orientované databáze, objektově orientované návrhy systémů atd.

Funkcionální programování bylo znovu objeveno jakožto výborné prostředí pro výuku algoritmizace a pro výzkum v oblasti informatiky. Příkladem zde může být zejména jazyk Scheme. Praktické využití je zatím nevelké, což se může změnit s příchodem masivně paralelních strojů.

Logické programování představuje jedno z historických zklamání. Po velmi úspěšném startu v oblasti umělé inteligence v Evropě a Japonsku a po vyhlášení projektu počítače páté generace založeném právě na myšlenkách logického programování se výrazně nevyvíjí. Je možné, že jeho doba teprve přijde.

Shrnutí

Rozlišujeme několik základních paradigmat: naivní, procedurální, objektové, funkcionální a logické.
Paradigmata se vyvíjejí a různé jazyky poskytují podporu pro jedno nebo více paradigmat

Pojmy k zapamatování

- Naivní paradigma,
- procedurální paradigma, sekvence příkazů, přiřazovací příkaz,
- objektově orientované paradigma, zapouzdření, zasílání zpráv,
- funkcionální paradigma, funkce, rekurze,
- logické paradigma, fakta, pravidla, unifikace a zpětné řetězení.

Kontrolní otázky :

1. *Charakterizujte základní princip procedurálního programování.*
2. *Co znamená zasílání zpráv a v jakém paradigmatu se používá?*
3. *K jakým paradigmatům byste zařadili jazyky C, C#, Scheme a COBOL?*
4. *Používá PROLOG přiřazovací příkaz? Jaký je princip práce PROLOGU?*

Úkoly k textu

1. Vraťte se k seznamu jazyků z úkolu v kapitole 1.2. Ke každému jazyku zkuste najít podporované paradigma.

1.4 Syntaxe a sémantika programovacích jazyků

Studijní cíle: Po prostudování kapitoly bude čtenář schopen číst a zapisovat syntaxi výrazů za pomocí různých metod.

Klíčová slova: Syntaxe, sémantika, BNF, EBNF, terminál, neterminál, syntaktický diagram, tutoriál, referenční manuál.

Potřebný čas: 4 hodiny

K pojmům, se kterými se v oblasti programovacích jazyků a počítačů setkáváme téměř denně, patří pojmy syntaxe a sémantika.

Syntaxe popisuje formu, sémantika popisuje význam.

Syntaxe programovacího jazyka popisuje *formální strukturu programu*. Definuje klíčová slova, identifikátory, čísla a další programové entity a určuje způsob, jak je lze kombinovat.

Sémantika programovacího jazyka určuje *logický význam* jednotlivých výrazů jazyka.

Příkladem nám může být formát zápisu data, tj. dne, měsíce a roku. Datum může mít syntaktický tvar:

```
DD/DD/DD
```

`D` je symbol pro číslici. Den, který toto datum označuje, ale není syntaxí jednoznačně určen:

- Výraz `01/A4-90Z` neodpovídá syntaktické definici
- Výraz `10/11/12` odpovídá syntaktické definici. Může označovat buď 10. listopad 1912 (evropský standard) nebo 11. říjen 1912 (americký standard) nebo 12. listopad 1910. Popřípadě to může být i 10. listopad 2012, atd.

Chceme-li vyjádřit definici data včetně sémantického popisu, musíme napsat:

```
DD/MM/YY
```

a specifikovat, že `DD` označuje den v měsíci, `MM` označuje měsíc v roce a `YY` označuje poslední dvojčíslí roku. Navíc můžeme specifikovat, že pokud je `YY` menší než 15, označujeme rok ve 21. století, je-li `YY` větší nebo rovno 15, označujeme rok v 20. století.

Výše uvedený výraz `10/11/12` lze takto jednoznačně interpretovat jako 10. listopad 2012. Naproti tomu výraz `29/02/09` je sice syntakticky správný, v rámci zvolené sémantiky je však chybný neboť rok 2009 není přestupným rokem.

Výraz může být syntakticky správný a sémanticky chybný.

Průvodce studiem

Lze říci, že syntaxe popisuje objekt povrchně; popisuje jeho vnější vzhled bez jakékoli snahy přiřadit objektu význam. v přirozených jazycích plní tuto úlohu gramatika. Věta „Disketa barví betonovou myšlenku." je gramaticky zcela správná, ve střízlivém stavu však této větě jen stěží přiřadíme význam.

1.4.1 BNF

Syntaxi programovacích jazyků je možno popsat několika způsoby. Nejčastější z nich je *Backus-Naurova forma* (BNF).[1]

BNF představuje vžitý způsob popisu syntaxe výrazů a příkazů v informatice.

Nejlépe uvedeme BNF na jednoduchém příkladu. Uvažujme syntaktický zápis reálných čísel typu 3.142 (tj. čísel, která obsahují celou část, desetinnou tečku a desetinnou část). Syntaxi takovýchto čísel lze vyjádřit pomocí tří pravidel takto:

```
<real-number> ::= <digit-sequence>.<digit-sequence>
<digit-sequence> ::= <digit> | <digit><digit-sequence>
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

K tomuto příkladu je třeba poznamenat:

- Symboly **.** (tečka) a číslice **0** až **9**, které se v tisku vyskytují napsané tučně, představují tak zvané *terminální* symboly. Jde o symboly, které se vyskytují v popisovaných výrazech.
- Symboly `<real-number>`, `<digit-sequence>` a `<digit>`, uzavřené ve špičatých závorkách, označují proměnné, abstrakce nebo symbolické konstrukce (záleží na nás, jaké pojmenování si vybereme), které se v popisovaných výrazech nebudou vyskytovat. Takovéto symboly slouží pouze k odvozování správných řetězců terminálních symbolů a nazývají se symboly *neterminální*.
- Symbol `::=` je třeba číst jako „je"
- Symbol `|` je třeba číst jako „nebo"

Třetí pravidlo tedy přečteme jako: „Číslice je 0 nebo 1 nebo 2 nebo ... nebo 9. Nic jiného není číslice."

Druhé pravidlo přečteme jako: „Sekvence čísel je číslice nebo číslice následovaná sekvencí číslic. Nic jiného není sekvence čísel."

První pravidlo přeložíme jako: „Reálné číslo je sekvence číslic následovaná tečkou a další sekvencí číslic. Nic jiného není reálné číslo."

[1] Pro účely konstrukce překladačů programovacích jazyků se jako popis používá tzv. gramatik. Ačkoliv lze nalézt mnoho podobného mezi BNF a gramatikami, popis gramatik přesahuje rozsah tohoto textu.

V druhém a třetím pravidle může být každá z možností oddělených symbolem | zapsána jako samostatné pravidlo. Například druhé pravidlo by se tak rozpadlo na pravidla

```
<digit-sequence> ::= <digit>
<digit-sequence> ::= <digit><digit-sequence>
```

Význam však zůstává v obou případech stejný.

Průvodce studiem

V praxi popisujeme pomocí BNF nejen syntaxi programovacích jazyků,ale i tvar názvů souborů v operačních systému, tvar adresy počítačů v síti, formální tvar emailových adres, tvar adres WWW stránek a podobně.

Jako příklad můžeme popsat pomocí BNF syntaxi jména souboru v operačním systému MS Windows.[1]

Používaný symbol Λ (lambda) zde označuje redukci na prázdný řetězec – tedy možnost, že uvedený neterminál zcela chybí. Například druhé pravidlo je možno číst jako: „Drive je buď nic (tj. není uveden) nebo je to písmeno následované dvojtečkou."

```
<file> ::= <drive> <path> <filename>
<drive> ::= Λ | <letter> :
<path>::= <absolute> | <relative>
<absolute> ::= Λ | \ <relative>
<relative> ::= Λ | <filename> \ <relative>
<filename> ::= <name> | <name> . <filename>
<name> ::= <char> | <char> <name>
<char> ::= <letter> | <number> | <special>
<letter>::= A | … | Z
<number> ::= 0 | … | 9
<special> ::= ! | @ | # | $ | % | _
```

1.4.2 EBNF

Pod zkratkou EBNF rozumíme *rozšířenou* Backus-Naurovu formu (Extended Backus-Naur Form). Od BNF se liší v těchto rysech:

- Odstraňuje používání špičatých závorek pro neterminály. Zavádí přitom konvenci, že neterminály jsou psány vždy velkými písmeny, terminály vždy malými písmeny a v tisku tučně, speciální terminální znaky jako +, -, | atd., jsou uzavírány do apostrofů, tj. píšeme '+', '-', '|'.
- Umožňuje používat kulatých závorek pro shromažďování výrazů.
- Zavádí symbol složených závorek `{expr}` pro žádné, jedno nebo více opakování výrazu `expr`.
- Zavádí symbol hranatých závorek `[expr]` pro nepovinné konstrukce.

Rozšířenou Backus-Naurovu formu lze demonstrovat na formálním přepisu příkladu ze začátku kapitoly. v EBNF by bylo možné popis reálného čísla shrnout do dvou pravidel:

EBNF poskytuje vyšší komfort popisu syntaxe než BNF. Zavádí konstrukce pro nepovinné výrazy a opakování výrazu.

[1] Tento příklad není zrovna učebnicovým příkladem efektivního popisu. Měl by být proto chápán jen jako demonstrace použití BNF na všeobecně známé výrazy.

```
Real-number ::= Digit {Digit} '.'{Digit}
Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Průvodce studiem

Rozšířená BNF (EBNF) je pro tvůrce syntaktického zápisu podstatně komfortnější než její „klasická“ forma. Proto jí většinou dáváme přednost před jinými formami popisu.

Mechanismy, které používá EBNF v mnoha případech „zlidověly“ a používají se i jiných souvislostech. Například při pohledu na manuálové stránky příkazů systému UNIX zjistíte, že nepovinné parametry příkazů jsou uváděny v hranatých závorkách a jednotlivé alternativy odděleny svislou čárkou. I když se zde nejedná o „čistokrevnou EBNF“, používají se zde stejné řídící konstrukce a odborníci v oblasti IT jim přirozeně rozumí.

Jako procvičení celé problematiky můžeme pomocí EBNF popsat reálné číslo, u něhož může chybět reálná část nebo desetinná část, ale nemohou chybět obě současně, tj. výrazy `20.45`, `.17` a `20.` jsou platné zápisy pro reálná čísla 20.45, 0.17 a 20.0, avšak tečka sama o sobě není platný zápis čísla.

```
Real-number ::= Digit {Digit} '.' {Digit}
Real-number ::= {Digit} '.' Digit {Digit}
Digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Na první pohled se zdá, že EBNF má vyšší popisovací schopnost než BNF, tj. poskytuje nám stejné možnosti jako BNF a k tomu i něco navíc. Tento dojem je však poněkud falešný. Ve skutečnosti je popisovací schopnost obou gramatik identická. Jakýkoli výraz popsaný EBNF lze převést na ekvivalentní výraz v BNF. Formální důkaz ponecháme posluchačům a převod z EBNF do BNF si ukážeme pouze na předchozím příkladě.

Začneme třetím pravidlem, které stačí jen formálně přepsat:

```
<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Pro převedení prvního a druhého pravidla zavedeme následující substituci:

```
A ::= {Digit}
```

Obě pravidla lze pak přepsat do tvaru:

```
Real-number ::= Digit A '.' A
Real-number ::= A '.' Digit A
```

Přepis pravidel v tomto tvaru nám již nebude dělat problémy. Ukažme si tedy jak budou pravidla vypadat a současně ukažme, jak přepsat do BNF substituci `A`.

```
<real-number> ::= <digit> <a> . <a>
<real-number> ::= <a> . <digit> <a>
<a> ::= Λ | <digit> <a>
```

1.4.3 Syntaktické diagramy

Syntaktické diagramy představují další způsob, jak popisovat syntaxi programovacího jazyka. Na rozdíl od předchozích textových popisů se zde jedná o grafické znázornění syntaxe. Principy tvorby syntaktických diagramů lze shrnout do těchto bodů:

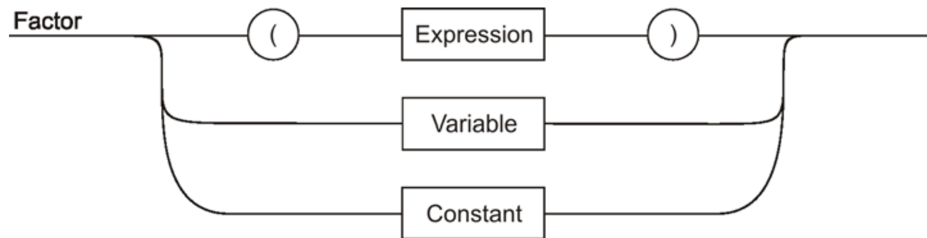
Syntaktické diagramy se vyznačují přehledností a intuitivitou, snadno je pochopí i laik.

- Pro každý neterminál existuje jeden diagram, název neterminálu je uveden v levé části diagramu,
- každé pravidlo BNF s daným neterminálem na levé straně generuje jednu cestu v diagramu,
- terminály jsou psány v kroužcích, neterminály ve čtverečcích,

- možnost opakování je v diagramu znázorněna smyčkou.

Obrázek Obr. 7 představuje syntaktický diagram pro výraz, zapsaný pomocí EBNF jako:

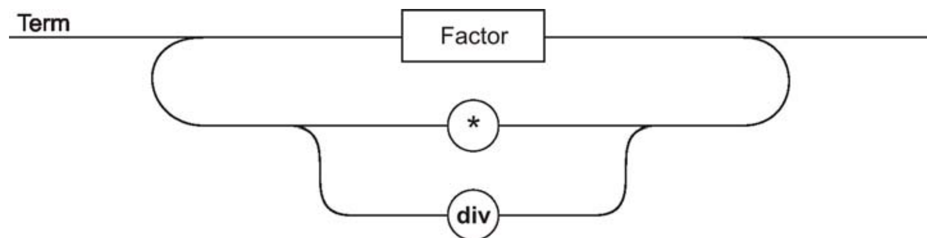
```
Factor ::= '(' Expression ')' | Variable | Constant
```



Obr. 7 Příklad syntaktického diagramu

Obrázek Obr. 8 představuje syntaktický diagram pro EBNF výraz:

```
Term ::= Faktor { ('*' | 'div') Faktor }
```



Obr. 8 Příklad syntaktického diagramu.

> **Průvodce studiem**
>
> *Syntaktické diagramy nejsou využívány příliš často. U větších projektů mohou být značně rozsáhlé a nepřehledné. Pro popis jednoduchých konstrukcí jsou však syntaktické diagramy velmi intuitivní a snadno je pochopí i laik.*

1.4.4 Další způsoby popisu jazyka

U konkrétních implementací programovacích jazyků se setkáváme s různými druhy publikací a příruček, které více či méně formálním způsobem popisují syntaxi a sémantiku jazyka. Mezi nejběžnější patří:

- **Tutoriál:** základní průvodce daným programovacím jazykem a překladačem. Tutoriál nám dává základní představu o hlavních rysech a konstrukcích jazyka. Problematika je typicky vysvětlována na příkladech. Syntaxe a sémantika jazyka je uváděna postupně, podle potřeby. Slouží k základnímu seznámení s daným programovacím jazykem.
- **Uživatelský manuál:** dokument, popisující práci s příslušným překladačem z uživatelského hlediska. Důraz je tedy zejména kladen na technické otázky jako jsou instalace produktu, jeho spouštění, práce v menu atd.
- **Referenční manuál:** dokument, popisující vyčerpávajícím syntaxi a sémantiku programovacího jazyka. Je tradičně tříděn abecedně, podle syntaxe jazyka. Jednotlivé

Referenční manuál potřebuje programátor ke své práci průběžně.

příkazy jazyka jsou popsány přirozeným jazykem (tj. většinou anglicky). Referenční manuál slouží zejména programátorům pro rychlé vyhledání podrobností o jednotlivých příkazech.

- **Formální definice:** precizní definice jazyka, určená pro profesionály v daném oboru. Používá formalismus BNF nebo syntaktických diagramů.

Průvodce studiem

Budete-li vytvářet dokumentaci k vašim programům, je potřeba důsledně oddělit jednotlivé typy manuálů. Za běžných podmínek potřebuje uživatel nejprve uživatelský manuál, aby mohl produkt instalovat, spustit a provést základní nastavení. Jde-li o komplexnější systém, sáhne asi i po tutoriálu, kde mu na příkladech a za využití didaktických technik vysvětlíte ovládání programu. Teprve zkušený uživatel potřebuje referenční manuál. Zatímco uživatelské manuály a tutoriály časem na polici zapadají prachem, referenční manuál potřebuje uživatel stále.

Je proto špatné, obtěžovat uživatele v referenčním manuálu poznámkami o instalaci a triviálními příklady.

Shrnutí

Syntaxe a sémantika jsou zásadní pojmy v oblasti programovacích jazyků a programování.
Syntaxi popisujeme pomocí BNF nebo EBNF. Tyto formy využívají sekvence pravidel pro postupné odvození syntaxe výrazu. v pravidlech se vyskytují terminální a neterminální symboly a speciální konstrukce pro alternativy, opakování apod.
Programovací jazyk nebo jiný produkt lze dále popsat pomocí tutoriálu, uživatelského manuálu nebo referenčního manuálu.

Pojmy k zapamatování

- Syntaxe a sémantika,
- BNF a EBNF, pravidla, terminály a neterminál,
- syntaktické diagramy,
- tutoriál, uživatelský manuál, referenční manuál, formální definice.

Kontrolní otázky

1. *Jaký je rozdíl mezi terminálním a neterminálním symbolem?*
2. *Co přesně vyjadřuje výskyt výrazu ve složené závorce na pravé straně pravidla v EBNF?*
3. *Co přesně vyjadřuje výskyt výrazu v hranaté závorce na pravé straně pravidla v EBNF?*
4. *V jakém typu manuálu najdete instalační instrukce?*

Cvičení

1. Emailová adresa má v nejjednodušším tvaru podobu uživatel@doména2.doména1, například jan.novak@seznam.cz. Před zavináčem je jméno uživatele, které často obsahuje i tečku. Za zavináčem musí adresa obsahovat alespoň dva názvy domén, oddělených tečkou. Popište syntaxi emailové adresy pomocí EBNF.
2. Konvertujte tuto syntaxi do podoby BNF.

Úkoly k textu

1. Adresa WWW stránky je popsána pomocí tzv. URL schématu například takto: `http://www.domena.cz:8080/stranka.asp?par1=a&par2=b&par3=c`, kde
`http` označuje použitý přenosový protokol,
`www.domena.cz` je jméno serveru v systému DNS,
`8080` je číslo portu na kterém je server dostupný,
`/stranka.asp` je název WWW stránky,
`par1=a`, `par2=b` atd. jsou názvy a hodnoty jednotlivých proměnných, zasílaných serveru pomocí metody GET protokolu HTTP; parametry jsou odděleny symbolem `&`.

   Popište tuto syntaxi v EBNF, předpokládejte přitom, že máte k dispozici neterminál `aplha-num`, který označuje libovolný neprázdný alfanumerický řetězec znaků.

   Dále nakreslete k této syntaxi i syntaktický diagram.

Řešení

1. Cvičení má více než jedno správné řešení, váš popis by nicméně měl vypadat zhruba takto:

```
Email-adresa ::= Uzivatel '@' Domena
Uzivatel ::= Retezec ['.' Retezec]
Domena :: = {Zona} Zona Retezec
Zona ::= Retezec '.'
Retezec ::= {Znak}Znak
Znak ::= 0 | … | 9 | a | … | z
```

2. Převod do BNF by měl být přibližně následující. Všimněte si obecné techniky nahrazení konstrukcí [] a {}.

```
<email-adresa> ::= <uzivatel> @ <domena>
<uzivatel> ::= <retezec> | <retezec> . <retezec>
<domena> :: = <zony> <zona> <retezec>
<zony> ::= Λ | <zona> <zony>
<zona> ::= <retezec> .
<retezec> ::= <znaky><znak>
<znaky> ::= Λ | <znak><znaky>
<znak> ::= 0 | … | 9 | a | … | z
```

2 Struktura programu

V kapitole 1 jsme ukázali, že pro dosažení kvality je nutné, aby programy měly jistou přesně definovanou a známou *vnitřní strukturu*. Tato struktura je dána použitým programovacím paradigmatem. Ve většině případů je prvním cílem strukturalizace programu rozklad rozsáhlého problému na jisté malé, snadno pochopitelné a otestovatelné části. Tyto části obecně nazýváme *moduly*.

2.1 Základní rysy modulů

Studijní cíle: Po prostudování kapitoly bude studující schopen popsat základní rysy modulů a identifikovat moduly v reálných systémech.

Klíčová slova: Modul.

Potřebný čas: 1 hodina.

Informatika nepatří mezi přísně exaktní vědy a mnoho pojmů, které používá, není přesně vymezeno. Uveďme si alespoň několik základních rysů, které charakterizují moduly programu.

Moduly slouží k rozčlenění velkého celku na menší, samostatně zpracovatelné díly.

2.1.1 Pochopitelnost modulů

Modul by měl vykonávat jednu jasně definovanou a pochopitelnou úlohu, popřípadě několik jasně definovaných úloh. Jen tehdy má dělení systému na moduly smysl.

Příklad: v programu můžeme mít modul na práci s počítačovou grafikou. Tento modul nám umožňuje kreslit na obrazovku nerůznější grafické obrazce (úsečky, kružnice, čtverce) nastavovat barvy atp. Jeho úloha je zcela jasná.

Protipříklad: Nebylo by asi vhodné rozdělit jinak kompaktní program na moduly s tím, že každých 20 řádků bude představovat jeden modul. Asi těžko bychom totiž hledali jednoznačný logický popis činnosti pro libovolně vybraný úsek 20-ti řádků programu.

2.1.2 Samostatnost modulů

Každý modul musí být relativně samostatný, a měl by mít co možná nejmenší počet vazeb na ostatní moduly. Nebylo by vhodné, aby byly všechny moduly programu navzájem propojené a na sobě závislé. Moduly bychom tak nemohli individuálně otestovat, pochopit ani přenést do jiného projektu.

Příklad: Výše zmíněný modul na práci s grafikou můžeme použít v různých projektech.

Protipříklad: Častou chybou může být vytvoření tzv. inicializačního modulu, tedy modulu, který inicializuje všechny ostatní moduly v systému.

2.1.3 Kombinovatelnost modulů

Moduly musí být navzájem kombinovatelné. Musí být možno modul vzít a použít jej v jiném kontextu nebo třeba i v jiném projektu.

Příklad: v numerické matematice jsou dlouhodobě úspěšné knihovny na numerické výpočty. Tyto knihovny lze nezávisle použít v různých projektech.

Protipříklad: Typografický systém TeX využívá pro zvýšení komfortu uživatelů a pro dodání funkcionality systém maker. Jednotlivé balíky maker však mnohdy nelze kombinovat a uživatel si tak musí často vybrat, jestli chce například používat komfortní systém LaTeX nebo pohodlně kreslit obrázky.

2.1.4 Zapouzdření modulů

Moduly musí mít právo na jisté soukromí: je přípustné a žádoucí, aby veškeré informace, které nejsou potřebné pro klienty modulů, zůstaly skryté uvnitř modulu. Hovoříme o *zapouzdření* modulu. v praxi se pak ukazuje, že většina funkcionality modulu je skryta a jen malá část je viditelná navenek. Skryté části říkáme *implementace* modulu a veřejné části říkáme *rozhraní* (*interface*) modulu. Tomuto principu také někdy říkáme *ledovcový* princip (viz obrázek Obr. 9).

Příklad: Řídíme-li automobil, používáme volant, řadící páku, pedály atd. Mnoho z nás však neví a ani nepotřebuje vědět, jak funguje posilovač řízení nebo systém vstřikování paliva. Nemůže také tyto systémy ovlivňovat jinak, než přes poskytnuté rozhraní.

Protipříklad: Operační systém Windows tradičně používá adresář `system32` pro ukládání sdílených knihoven programů. Většina softwarových produktů, které instalujeme, si do tohoto adresáře ukládá soubory. Informace o těchto souborech a jiných nastavení se dále ukládají do systémových registrů. Další soubory se pak nacházejí v adresáři `Program Files`. Nesprávný zásah na kterémkoli z těchto míst způsobí nefunkčnost celého produktu.

2.1.5 Explicitní rozhraní modulů

Z definice modulu musí být všeobecně zřejmé, jaké předpoklady pro vykonávání své úlohy potřebuje. Tento princip se odborně nazývá principem explicitního rozhraní.

Příklad: Hovoříme-li například o modulu, který provádí řešení soustavy lineárních rovnic, musí být jasně stanoveno, v jaká data a v jakém formátu modul vyžaduje, aby mohl vykonávat svoji funkci.

Protipříklad: Někteří zkoušející na vysoké škole nedávají studentům k dispozici požadavky na zkoušky, nebo jsou požadavky příliš obecné. Studenti pak neví přesně na co se připravit, aby zkoušku úspěšně složili.

2.1.6 Syntaktická podpora modulů

Moduly počítačového programu musí být jasně vymezeny syntaktickými jednotkami programu. Ze zápisu programovacího jazyka musí být zcela evidentní, kde začíná a kde končí zápis jednoho modulu. Není například možné, aby v kompaktním programu patřily modulu všechny sudé řádky, zatímco všechny liché řádky modulu nepatřily.

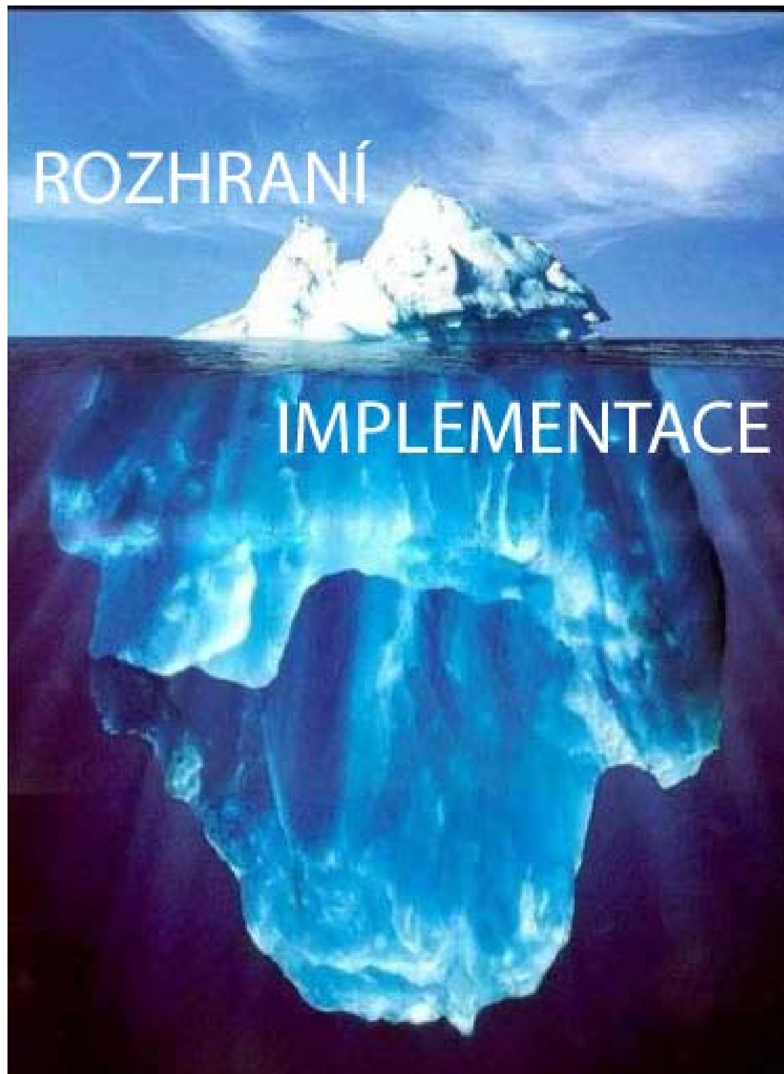
Příklad: Většina programovacích jazyků obsahuje podporu pro tvorbu procedur, které tvoří elementární moduly programu.

Protipříklad: v jazycích BASIC bylo zvykem, že moduly (podprogramy) vždy začínaly na jistém řádku a končily na řádku, který obsahoval návratový příkaz. Programátor modul aktivoval tak, že uvedl číslo řádku na který je třeba předat řízení. z programu ale nebylo nijak patrné, že zde nějaký modul začíná.

Průvodce studiem

Modul lze z tohoto hlediska chápat jako malou soukromou firmu, specializovanou na určitou práci. Programátor musí být schopen zjistit, čím se firma zabývá a za jakých okolností je schopna provést inzerovanou práci. Udělá-li programátor s touto firmou

kontrakt, znamená to, že jí dodá patřičná data (stejně tak jako operační paměť, diskovou paměť a strojový čas) a firma provede inzerovanou službu. Programátora-zadavatele přitom nemusí zajímat vnitřní organizace firmy (například počet a platy zaměstnanců firmy). Je taktéž možné, aby části zakázky firma zadala jako zakázku sub-dodavatelům.



Obr. 9 Ledovcový efekt

Shrnutí

Moduly slouží k rozdělení velkého a nepřehledného celku na menší části. Cílem je tyto menší části zpracovávat a testovat samostatně. Aby bylo rozdělení efektivní a splnilo svůj účel, musí moduly splňovat některé základní rysy.

Pojmy k zapamatování

- Modul,
- rozhraní a implementace modulu, ledovcový efekt,
- zapouzdření.

Kontrolní otázky

1. *Co rozumíme pod pojmy zapouzdření modulu a ledovcový efekt?*
2. *Z jakého důvodu trváme na zachování samostatnosti jednotlivých modulů?*

Úkoly k textu

1. Pokuste se najít ve vašem okolí pět příkladů modulů. Demonstrujte na těchto příkladech základní rysy modulů.

2.2 Procedury jako jednoduché moduly

Většina programovacích jazyků podporuje vytváření procedur – nejjednodušších modulů.

Studijní cíle: Studující se seznámí s pojmem procedura a pochopí vztah mezi procedurou programovacího jazyka, obecnou matematickou funkcí, algoritmem a modulem. Pochopí model vyhodnocování procedur.

Klíčová slova: procedura, algoritmus, deklarace a definice procedury, formální a skutečné parametry, hlavička a tělo procedury, volání procedury.

Potřebný čas: 3 hodiny.

V různých programovacích paradigmatech jsou moduly tvořeny různými způsoby. Jedním z nejběžnějším a nejjednodušších modulů, vyskytujících se v různých obměnách téměř ve všech paradigmatech, jsou *procedury*.[1]

2.2.1 Procedury, matematické funkce a algoritmus

Pojem procedura jakožto název pro jednoduchý modul se do informatiky dostal na základě jisté analogie s matematickými funkcemi. Ukažme si podstatné rozdíly mezi pojetím funkcí, jak je chápou matematici a procedur jak je chápeme my.

Matematici chápou funkci jako zvláštní případ zobrazení. Totální funkce f je zobrazení, které přiřazuje každému prvku množiny A právě jeden prvek množiny B. Tento fakt zapisujeme:

$$f : A \to B$$

nebo

$$f : x \mapsto y, x \in A, y \in B$$

a píšeme

$$f(x) = y$$

Množinu A pak nazýváme doménou funkce (definičním oborem), množinu B nazýváme rozsahem (oborem hodnot) funkce.

Zásadní rozdíl mezi matematických chápáním funkce a naším chápáním procedury je, že v matematické definici nemusí být vždy řečeno, jakým způsobem získáme ze vstupních hodnot hodnoty výstupní. Je například zcela korektní definovat funkci *sqrt*[2] jako:

---

[1] Setkáváme se též s názvy jako funkce, podprogram, routina, subroutina, metoda atd. v tomto textu se přidržíme pojmu procedura.

[2] Funkce na výpočet druhé odmocniny čísla.

$sqrt(x) = y, y^2 = x$

Matematickou notací zde říkáme, že druhou odmocninou z čísla *x* rozumíme takové číslo *y*, jehož druhá mocnina je rovna číslu *x*. Tato definice funkce splňuje všechna kritéria injektivního zobrazení (a tedy funkce v matematickém pojetí), nepředstavuje však proceduru ve smyslu informatickém.

Na rozdíl od definice matematické funkce, procedura vždy implementuje algoritmus.

V informatice můžeme také definovat proceduru, která jako vstup akceptuje číslo a vypočítá jeho druhou odmocninu. s takto uvedenou matematickou definicí bychom však neuspěli. Informatika chápe vždy procedury jako *algoritmy* popisující výpočet výsledku na základě vstupů. Navíc procedury v našem chápání splňují i požadavky kladené na modul, čímž se ještě více liší od matematického pojetí.

Průvodce studiem

Pod pojmem algoritmus si představujeme podrobný, do jednotlivých kroků rozebraný postup, kterým se na základě vstupů dobereme výstupů. v běžném životě se algoritmům asi nejvíce blíží recepty z kuchařek. Recept nám popisuje jak na základě vstupů (surovin) získáme výstup (pokrm).

Předchozí matematická definice druhé odmocniny je zcela korektní. Máme-li odmocninu takto nadefinovanou, můžeme ji začít používat v matematických konstrukcích, větách a důkazech. Definice nám však nedává ani ten nejmenší návod, jak druhou odmocninu z čísla vlastně spočítat. Máme-li se přidržet předchozí analogie, matematická definice nám říká jak daný pokrm vypadá a jaké má vlastnosti, takže jej zaručeně poznáme, neříká nám ale, jak jídlo uvařit.

2.2.2 Struktura procedury

Část programu, která obsahuje zápis kódu procedury nazýváme *deklarace* procedury.[1] Deklarace má klasicky alespoň tři části: jméno procedury, seznam tzv. formálních parametrů procedury a tělo procedury. v některých jazycích obsahuje deklarace procedury i tzv. návratový typ (typům bude věnována kapitola 3.1). Deklarace jednoduché procedury následovníka může v některém fiktivním programovacím jazyku vypadat takto[2]:

Deklarace procedury se skládá z hlavičky a těla. v hlavičce je obsažen název a seznam formálních parametrů.

```
procedure succ(x) {
  return (x + 1)
}
```

`succ` představuje *jméno procedury*, `(x)` je seznam *formálních parametrů*, `x` je tedy (v tomto případě jediným) formálním parametrem procedury. Můžeme také říci, že procedura `succ` akceptuje parametr `x`. Všechny tyto prvky dohromady pak tvoří *hlavičku procedury*. Kód, uvedený mezi složenými závorkami, je tak zvané *tělo procedury*. v mnoha programovacích jazycích je možné v těle procedury deklarovat lokální proměnné a dokonce i lokální procedury. Pro tuto chvíli však tyto další možnosti nejsou důležité, vystačíme s tímto jednoduchým pojetím.

[1] v některých jazycích se setkáváme s pojmy *deklarace* procedury a *definice* procedury. v takovém případě se deklarací rozumí zveřejnění jména a seznamu formálních parametrů procedury a definicí uvedení těla procedury. Toto řešení má jisté výhody pro možnost odděleného překladu jednotlivých částí programu (modulů). Překlad pak musí být následován tzv. spojováním programu.

[2] v tomto textu budeme používat fiktivní meta-jazyk. Syntaxe tohoto jazyka je částečně inspirována jazykem C++ ale není s ním totožná. Protože se jedná o fiktivní jazyk, nepředpokládáme, že by uvedené fragmenty kódu byly přeložitelné a spustitelné. v dalších textech budeme využívat programovací jazyk Scheme, jehož syntaxe je poněkud odlišná.

Samotná deklarace ještě nestačí k tomu, aby byla procedura využita. Použití procedury v programu nazveme *volání*[1] procedury. Parametry, uvedené při volání procedury, nazveme *aktuální* (též *skutečné*) parametry procedury.

```
succ(2 + 1)
```

Výsledkem tohoto volání procedury `succ` je hodnota `4`. Jak vlastně k této hodnotě dospějeme? Vyhodnocování volání procedur lze shrnout do několika bodů:[2]

- vyhodnotí se výrazy, uvedené jako aktuální parametry,
- výsledek vyhodnocení jednotlivých aktuálních parametrů přiřadíme do formálních parametrů,
- tělo procedury vyhodnotíme, výsledek vyhodnocení vrátíme jako výsledek aplikace procedury. Mnohdy bývá výsledek procedury označen klíčovým slovem `return`.

Výše uvedená aplikace procedury `succ` na skutečný parametr `2 + 1` proběhne tedy takto:

- vyhodnotí se hodnota `2 + 1` na číslo `3`,
- hodnota `3` se přiřadí za formální parametr `x`,
- vyhodnotí se výraz `3 + 1`,
- vrátí se hodnota 4.

V zájmu jednoduchosti jsme se v tomto příkladě dopustili několika závažných zjednodušení. k mnohým aspektům se ještě v podrobněji vrátíme v dalších kapitolách, jiné ponecháme na další učební opory. Nyní je nutno učinit pouze několik poznámek.

Procedury mohou akceptovat více než jeden parametr; v tom případě se vyhodnotí jednotlivé parametry v nějakém, blíže nespecifikovaném, pořadí (pořadí vyhodnocení je dáno implementací jazyka). Nelze v zásadě ani vyloučit, že u paralelních systémů bude vyhodnocování všech parametrů probíhat současně.

Výsledek aktivace procedury nejspíše přiřadíme do nějaké proměnné, vytiskneme jej na obrazovku nebo zapíšeme do souboru. Můžeme však také použít výsledek volání jedné procedury jako parametr volání jiné procedury. Lze tedy například použít i zápis:

```
succ(succ(2))
```

Výsledkem takovéhoto příkazu je hodnota 4.

Průvodce studiem

Uvědomme si, že procedury představují jednoduché moduly počítačového programu. Určitě splňují kriterium zapouzdřenosti (dělí se na hlavičku a tělo) a explicity rozhraní (známe formální parametry). Jsou navzájem kombinovatelné (výsledek jedné procedury může vstoupit do jiné procedury jako parametr) a v jazycích pro ně máme syntaktickou podporu.

Je na programátorovi, aby zajistil relativní samostatnost a pochopitelnost procedur.

[1] Namísto pojmu volání procedury (procedure call) také někdy používáme termíny jako: aplikace procedury, aktivace procedury nebo odskok do procedury. Všechny tyto termíny mají stejný význam.

[2] Tento způsob vyhodnocování volání procedur nazýváme aplikativní. Existují i jiné způsoby, jejich popis však poněkud přesahuje rozsah tohoto textu a proto se jim nebudeme věnovat.

2.2.3 Hlavní a vedlejší efekt procedur

Až doposud jsme se zabývali procedurami, které pracovaly prostřednictvím *hlavního efektu*. Takového chování procedur je typické například pro funkcionální programovací paradigma a představuje obdobu chování matematické funkce. Je-li procedura aktivována, spočte ze seznamu skutečných parametrů a dalších dostupných proměnných a konstant výslednou hodnotu a tu vrátí jako svůj výsledek. Působí-li procedura pouze hlavním efektem, nic v našem programovacím prostředí nemění. Zavoláme-li takovouto proceduru mnohokrát po sobě, dostaneme pokaždé stejný výsledek.

Návratová hodnota procedury představuje hlavní efekt.

V některých případech působení prostřednictvím hlavního efektu nedostačuje a procedury mají vedle hlavního i *vedlejší efekt* (*side effect*). To se týká především procedur, které potřebují něco vypsat na obrazovku, zapsat do souboru a podobně. Vypočtená návratová hodnota tedy není to jediné, co po zbude po jedné aktivaci procedury. Vyvoláme-li takovouto proceduru vícekrát po sobě, můžeme dostat různé výsledky nebo můžeme postupně měnit prostředí, například postupně vypisujeme další a další řádky na obrazovku. Příkladem může být takto upravená procedura `succ`, využívající proceduru `display`, která vypíše zprávu na obrazovku počítače.

```
procedure succ(x) {
  display("Vstupem je cislo ", x)
  return (x + 1)
}
```

Některá programovací paradigmata využívají vedlejší efekt zásadním způsobem, pro jiná je to spíše okrajový rys nebo vedlejší efekt nepodporují vůbec. O legitimitu existence vedlejšího efektu procedur v některých paradigmatech se dosud vedou spory. Problém představuje zejména v logickém a funkcionálním paradigmatu. v zásadě lze říci, že vedlejší efekt je na závadu teoretické čistotě jazyka, zhoršuje odladitelnost, komplikuje paralelní zpracování úloh. Na druhé straně zjednodušuje problematiku vstupu a výstupu, řešení chybových situací a v některých případech značně zmenšuje velikost kódu.

Procedurální programování často využívá i vedlejší efekt procedur.

Průvodce studiem

V praktickém životě využívají programátoři vždy takové nástroje a mechanismy, které jim maximálně ulehčí práci. Ideální se proto jeví využívat, pokud je to možno, hlavního efektu procedur a vedlejší efekt omezit na zdůvodněné případy.

Použijeme-li vedlejší efekt procedury, je to v kódu nutno vždy řádně dokumentovat. Vedlejší efekt totiž není ihned patrný z hlavičky procedury a pokud s ním nepočítáme, může vést k chybám a nepředvídanému chování programu. Výhodnou taktikou je také rozdělit si procedury tak, aby buď působily jen hlavním efektem, nebo měly jen vedlejší efekt a návratovou hodnotou byla například chybová zpráva.

Shrnutí

Procedury představují jednoduché moduly programu. Od matematických funkcí se liší tím, že implementují algoritmus. Deklarace procedur se skládá ze jména procedury, seznamu formálních parametrů a z těla procedury. Při volání procedury se formální parametry zamění za skutečné a provede se tělo procedury. Procedury mají svůj hlavní a vedlejší efekt.

Pojmy k zapamatování

- Deklarace a volání procedury,
- hlavička a tělo procedury,

- formální a skutečné parametry,
- hlavní a vedlejší efekt procedur.

Kontrolní otázky

1. *Jaký je rozdíl mezi deklarací a voláním procedury?*
2. *Jak poznáme, že má procedura vedlejší efekt?*
3. *Kdy je vhodné použít vedlejší efekt a kdy použijeme hlavní efekt procedury?*
4. *Jaký je rozdíl mezi formálními a skutečnými parametry procedury?*

Cvičení

1. Co bude výsledkem následujícího volání procedury `test`? Co bude hlavním efektem a co bude vedlejším efektem?

```
procedure test(x, y) {
   z = x + y
   display(z)
   return z
}

test(2, 3)
```

2. Co bude hlavním a vedlejším efektem tohoto volání procedury `test`?

```
test(test(1, 2), test(3, 4))
```

3. Co bude hlavním a vedlejším efektem tohoto volání procedury `test`?

```
test(test(test(test(1, 1), 1), 1), 1)
```

Úkoly k textu

1. Napište program na výpočet kořenů kvadratické rovnice. Analyzujte problém, nalezněte jednotlivé moduly a navrhněte procedury na řešení tohoto problému. Využijte přitom náš fiktivní meta-jazyk.

Řešení

1. Procedura `test` akceptuje dva parametry, `x` a `y`. Ve svém těle nastaví proměnnou `z` na výsledek součtu `x` a `y`. Tuto hodnotu jednak vytiskne na obrazovku a dále pak vrátí jako výsledek. Vedlejším efektem je vytištěná číslice „5“ na obrazovce, hlavním efektem je hodnota 5.
2. Hlavním efektem je hodnota 10. Na obrazovce bude vytištěn řetězec „3 7 10“ nebo řetězec „7 3 10“. Záleží to na tom, v jakém pořadí bude daný programovací jazyk vyhodnocovat parametry volání procedury.
3. Hlavním efektem je hodnota 5. Na obrazovce bude vytištěn řetězec „2 3 4 5“.

2.3 Procedury v programu

Studijní cíle: Studující po prostudování této kapitoly porozumí mechanismu předávání řízení mezi procedurami. Pochopí princip rekurzivních procedur a jejich vztah k iterativnímu výpočtu. Bude schopen určit hodnoty proměnných v závislosti na použitém způsobu rozsahu platnosti proměnných.

Klíčová slova: datový segment, hromada a zásobník programu, aktivační strom, rekurze, koncová rekurze, lexikální a dynamický rozsah platnosti proměnných.

Potřebný čas: 4 hodiny

V této podkapitole se budeme věnovat velice častým případům, kdy jedna procedura vyvolává ve svém těle jinou proceduru.

2.3.1 Zásobník programu

Předávání řízení mezi procedurami (*control flow*) probíhá tak, že pokud procedura `P` volá proceduru `Q` a procedura `Q` volá `R`, řízení běhu programu se vrací po ukončení práce procedury `R` zpět do `Q` a z `Q` zpět do `P`. Návrat do volající procedury probíhá těsně za to místo programu, odkud byla procedura vyvolána. Jednotlivé aktivace můžeme chápat jako kostky dětské stavebnice, které, jak dochází k aktivacím, stavíme na sebe a jak aktivace končí, postupně odebíráme.

Volá-li jedna procedura druhou, předávají si mezi sebou řízení programu.

```
procedure P(x) {
   return 2*Q(x+2)
}

procedure Q(x) {
   return 3*R(x+3)
}

procedure R(x) {
   return 4*x
}
```

Průvodce studiem

Tento kód ukazuje situaci, kdy se procedury `P`, `Q` a `R` postupně volají. Dokážete již teď určit, ***co*** *a hlavně* ***proč*** *bude výsledkem následujícího volání?*

```
P(2)
```

Neprozradíme moc, když řekneme že výsledkem je 168. Dokážete odůvodnit, proč dostaneme tento výsledek?

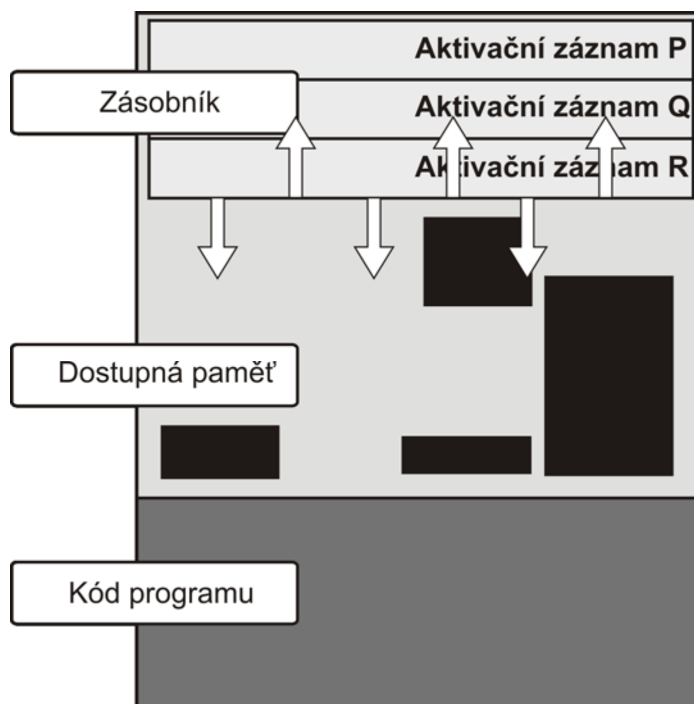
Paměť procesu se dělí na datový segment, hromadu a zásobník.

V těchto úvahách je třeba důsledně rozlišit mezi procedurami (tj. kousíčky programu, moduly) a jejich aktivacemi. Zatímco procedura existuje v programu trvale, její aktivace je iniciována, proběhne a skončí mnohokrát v rámci daného běhu programu. Každá aktivace procedury přitom „vlastní“ svoji jedinečnou kopii aktuálních parametrů. Pokud by tedy procedura během své aktivace aktivovala další proceduru, budou skutečné parametry obou aktivací existovat v paměti počítače nezávisle na sobě – a to i v případě, budou-li mít stejná jména![1]

Aktivace procedur velmi úzce souvisí s organizací paměti programu. Tradičně je paměť programu (procesu) rozdělena na tři části.

- První částí je *datový segment* (*data segment*), paměť vyhrazená *kódu* a *globálním proměnným*. Programátor s touto částí paměti nemá příliš starostí: v průběhu vykonávání programu se tato část paměti ani nezvětšuje ani nezmenšuje, modifikovat je možno pouze hodnoty globálních proměnných. Globální proměnné jsou takové, které jsou dostupné vždy ve všech procedurách.
- Druhou částí je datová oblast zvaná *hromada* (*heap*). To je místo, kde se může programátor plně realizovat. v této oblasti může požádat systém o přidělení libovolně velkého místa a v rámci přidělené paměti dělat, co uzná za vhodné.
- Poslední částí paměti je *zásobník* (*stack*). Jde o oblast paměti, která se zvětšuje a zmenšuje principem LIFO (Last In First Out) podle toho, jak probíhá vykonávání programu. z jakýchsi důvodů ve většině programovacích jazyků zásobník roste směrem dolů (na úkor velikosti hromady).

Příklad konfigurace paměti procesu můžeme vidět na obrázku Obr. 10.



Obr. 10 Schematické znázornění paměti procesu

[1] Speciálně to platí i pro případ, kdy procedura ve svém těle aktivuje sama sebe. K tzv. rekurzivním procedurám se vrátíme podrobněji v kapitole 2.3.3.

Průvodce studiem

Se zásobníkem systém pracuje systémem „Last In First Out“. Jde o stejný princip, kterým se řídí nastupování a vystupování do výtahu: ti, kteří do výtahu nastoupili jako poslední a jsou tak nejblíže dveřím, vystoupí z něj jako první. Stejně tak u zásobníku je záznam, který byl vytvořen jako poslední, odstraněn jako první. Některé záznamy tak na zásobníku pobudou jen relativně krátkou dobu, zatímco záznamy v hloubce zásobníku tam vydrží i velmi dlouho.

Každé vyvolání procedury přidá jeden záznam na vrchol zásobníku. Po ukončení dané aktivace procedury se záznam z vrcholu zásobníku opět smaže. Vyvolá-li procedura ve svém těle jinou proceduru, dojde k tomu, že se záznamy kladou na sebe. Jak aktivace procedur končí a řízení programu se vrací na místa, odkud byly procedury volány, záznamy ze zásobníku se umazávají. Tyto zásobníkové záznamy se také někdy nazývají *aktivační záznamy*. Na obrázku Obr. 10 je znázorněna situace, kdy je prováděna procedura `R`. Vidíme, že procedura `R` byla aktivována uvnitř procedury `Q` a procedura `Q` během aktivace procedury `P`.

Při aktivaci procedury vzniká na zásobníku aktivační záznam.

Co tyto záznamy obsahují? Nejpodstatnější částí záznamu jsou paměťová místa pro uložení aktuálních parametrů procedury. Vzhledem k tomu, že každá aktivace procedury má vlastní záznam na zásobníku, má každá procedura i svoji jedinečnou kopii svých skutečných parametrů, nehledě na to, že v danou chvíli může být aktivováno více procedur a jejich formální parametry mohou mít stejná jména. Obsahuje-li procedura deklaraci lokálních proměnných, jsou taktéž uloženy v záznamu dané aktivace.[1] K dalším položkám se dostaneme v kapitole číslo 2.3.5 věnované rozsahům platnosti proměnných.

2.3.2 Aktivační strom

Podívejme se na pozměněný kód předchozího příkladu.

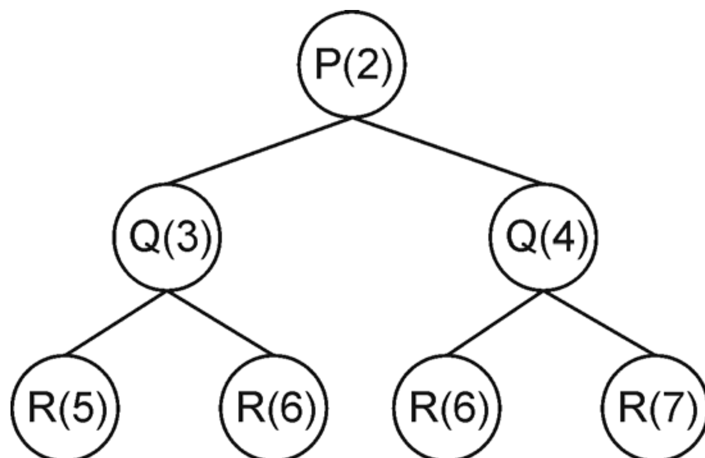
```
procedure P(x) {
   return Q(x+1) + Q(x+2)
}

procedure Q(x) {
   return R(x+2) + R(x+3)
}

procedure R(x) {
   return 4*x
}
```

Předávání řízení mezi procedurami je v tomto případě ještě komplikovanější. Přehledně je lze zobrazit pomocí tzv. *aktivačního stromu*.[2] Aktivační strom je strom, jehož uzly představují jednotlivé aktivace procedur. Pokud se v rámci aktivace procedury `P` vyvolá aktivace procedury `Q`, potom v aktivačním stromu je `P` rodičem `Q`. Průběh výpočtu zjistíme tak, že prohledáme celý strom do hloubky. Aktivační strom pro volání procedury `P` s parametrem `2` vidíme na následujícím obrázku.

Aktivační strom demonstruje průběh výpočtu.

[1] Problém je o něco složitější.

[2] Strom je pojem z teorie grafů s širokým uplatněním v celé informatice.



Obr. 11 Aktivační strom

2.3.3 Rekurzivní procedury

Proceduru nazýváme *rekurzivní,* pokud může ve svém těle aktivovat sama sebe, a to třeba i nepřímo prostřednictvím jiných procedur.

Rekurzivní procedura volá sama sebe..

Jako typický příklad rekurzivní procedury nám může sloužit procedura na výpočet `n`-tého prvku Fibonacciho posloupnosti čísel. Prvek Fibonacciho posloupnosti je v matematické notaci definován formulí:

$$Fib(n) = \begin{cases} n, & n \leq 1 \\ Fib(n-2) + Fib(n-1), & n > 1 \end{cases}$$

Prvních 6 členů Fibonacciho posloupnosti tedy tvoří čísla1, 1, 2, 3, 5 a 8. Kód takovéto procedury může mít v nějakém programovacím jazyce tvar:[1]

```
procedure Fib(n) {
   if n <= 1 return n
   else return Fib(n - 2) + Fib(n - 1)
}
```

Průvodce studiem

Rekurzivní procedury mají u studentů informatiky podobný efekt jako výuka couvání u některých žáků autoškoly. Často vyvolávají nechápavé výrazy na tvářích a obavy. Fakt, že procedura je definována pomocí sama sebe aniž by vznikala definice kruhem, je zdrojem mnohých rozpaků. Podíváme se proto podrobněji, jak rekurzivní procedury pracují a jaká je jejich struktura.

Rekurzivním procedury hrají v programování specifickou roli. Budeme se jimi zabývat podrobněji i v jiných studijních oporách. Pomocí rekurzivních procedur můžeme přimět počítač, aby danou akci opakoval několikrát po sobě, přičemž počet opakování nemusí být předem pevně dán.

V mnoha programovacích jazycích se namísto rekurze setkáváme s tzv. cykly. I když cyklus může připadat začínajícímu programátorovi jako jednodušší nástroj než rekurze, je méně obecný. Cyklus lze snadno implementovat pomocí rekurze, obrácený postup však není možný. Podrobněji se k této otázce vrátíme v kapitole 2.3.4.

---

[1] v příkladu předpokládáme, že je čtenář srozuměn s významem podmíněného výrazu, kdy se na základě hodnoty nějaké podmínky provede jedna nebo druhá část výpočtu.

Každá rekurzivní procedura je složena ze dvou částí. První část se nazývá *základní případ* (*base case* nebo též *mezní podmínka* rekurze). Základní případ řeší situaci pro nejmenší možnou velikost problému a to **bez** použití rekurze. v našem případě jde o řádek programu

Rekurzivní procedura má dvě části:základní případ a rekurzivní volání.

```
if n <= 1 return n
```

Tento kód tedy řeší problém pro dvě nejmenší přípustné hodnoty čísla `n` – hodnoty `0` a `1`. v těchto případech procedura `Fib` vrátí přímo hodnotu čísla `n`.

Druhou částí, kterou obsahuje každá rekurzivní procedura, je *rekurzivní volání* (*recursive call*). Jde o kód, který převádí daný problém na jeden nebo i více problémů, z nichž každý je jednodušší než problém původní. v našem případě jde o řádek programu

```
else return Fib(n - 2) + Fib(n - 1)
```

který převádí výpočet Fibonacciho čísla pro hodnotu `n` na výpočet dvou Fibonacciho čísel pro hodnoty `n - 2` a `n - 1`. Podstatné je zde slovo jednodušší. Tím, že redukujeme problém na problémy jednodušší máme zaručeno, že dříve či později dojdeme k nejmenší možné velikosti problému a tedy k základnímu případu, který umíme vyřešit bez použití rekurze. Pokud bychom opomněli v rekurzivní funkci uvést základní případ, vznikla by skutečná definice kruhem – výpočet by buď skončil chybou nebo probíhal až do přeplnění zásobníku programu.

Průvodce studiem

Výraz „jednodušší" je třeba chápat tak, že nové rekurzivní volání nás více přibližuje k mezní podmínce rekurze a zajišťuje, že mezní podmínky rekurze dosáhneme. Pokud by tomu tak nebylo, získali bychom teoreticky nekonečný cyklus, výpočet by běžel bez omezení dál a dál.V praxi by po jisté době došlo k vyčerpání dostupné paměti a k pádu programu.

Proceduru `f` nazýváme *lineárně rekurzivní*, pokud aktivace `f` vyvolá nejvýše jednu novou aktivaci `f`. Příkladem takovéto lineárně rekurzivní procedury může být procedura faktoriál, kterou si uvedeme za chvilku. Naopak procedura pro výpočet Fibonacciho čísel není lineárně rekurzivní, neboť v těle procedury `Fib` jsou vyvolány dvě aktivace procedury `Fib`. Takovéto rekurzivní procedury nazýváme *stromově rekurzivní*.[1]

Funkci faktoriálu `Fact` je možno definovat za použití matematické notace jako:

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

Kód pro výpočet faktoriálu v našem fiktivním metajazyku může vypadat takto:

```
procedure Fact(n) {
   if n = 0 return 1
   else return n * Fact(n - 1)
}
```

Jak vypadá běh programu při vykonávání lineárně rekurzivní procedury? Lze jej rozdělit do dvou po sobě následujících fází:

Běh lineárně rekurzivní procedury dělíme na fázi navíjení a fázi odvíjení.

[1] Pokud bychom chtěli být zcela korektní, měli bychom namísto lineárně a stromově rekurzivních procedurách hovořit o lineárně a stromově rekurzivních *výpočetních procesech*, generovaných těmito procedurami. Zavedení výpočetních procesů by znamenalo zavedení složitosti výpočtů a je mimo rozsah tohoto textu.

- *fáze navíjení* (*winding phase*) – v rámci fáze navíjení jsou aktivovány nové aktivace rekurzivní procedury,
- *fáze odvíjení* (*unwinding phase*) – řízení se postupně vrací jednotlivým aktivacím.

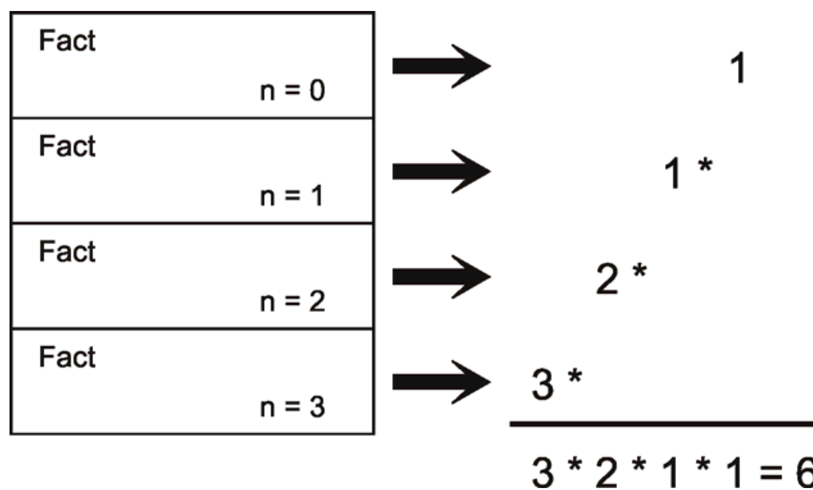
Ukažme si na příkladu, jak probíhá výpočet pro faktoriál čísla 3:

```
Fact(3) = 3 *  Fact(2)
        = 3 * (2 *  Fact(1))
        = 3 * (2 * (1 * Fact(0)))
        = 3 * (2 * (1 * 1))
```

V tuto chvíli končí fáze navíjení. Celý výpočet je rozpracován – cílem této fáze bylo v prvé řadě uložení jednotlivých hodnot do patřičných vrstev zásobníku. Ve fázi odvíjení se z jednotlivých čísel v zásobníku spočítá výsledná hodnota:

```
        = 3 * (2 * 1)
        = 3 * 2
        = 6
```

Obrázek Obr. 12 znázorňuje stav zásobníku ve chvíli, kdy končí fáze navíjení.



Obr. 12 Stav zásobníku programu při provádění procedury `Fact` pro výpočet faktoriálu. Předpokládáme, že zásobník roste směrem k hornímu okraji strany.

> **Průvodce studiem**
>
> *U lineárně rekurzivní procedury probíhá nejprve fáze navíjení a pak fáze odvíjení. U stromově rekurzivních procedur se výpočet vyvíjí po jednotlivých větvích aktivačního stromu, fáze navíjení a odvíjení se střídají.*

2.3.4 Koncově rekurzivní procedury

Zvláštním případem lineárně rekurzivních procedur jsou procedury *koncově rekurzivní* (*tail-recursive*). Koncově rekurzivní proceduru poznáme podle toho, že jako výsledek vrací hodnotu rekurzivní aktivace.

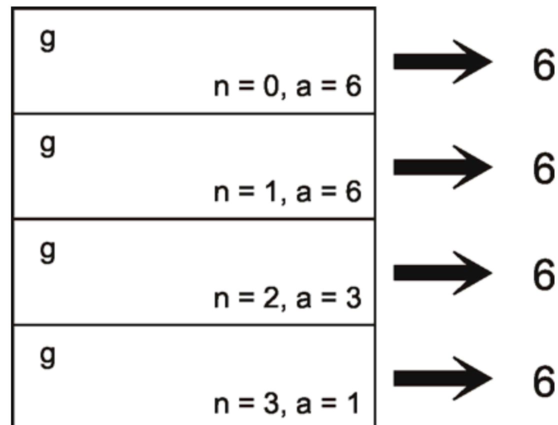
Jako příklad může sloužit speciálně upravená procedura, pro výpočet faktoriálu:

```
procedure g(n, a) {
   if n = 0 return a
   else return fact(n - 1, n * a)
}
```

Aktivace této procedury se skutečnými parametry 3 a 1 vypadá takto:

```
g(3, 1) = g(2, 3)
        = g(1, 6)
        = g(0, 6)
        = 6
        = 6
        = 6
```

Co je na koncově rekurzivních procedurách zajímavé je, že veškerá činnost je vykonávána ve fázi navíjení. v okamžiku ukončení fáze navíjení již máme výsledek spočítaný a fáze odvíjení je triviální. Demonstrujme si tento fakt na stavu zásobníku ve chvíli ukončení fáze navíjení:



Obr. 13 Stav zásobníku programu při provádění procedury `g` pro koncově rekurzivní výpočet faktoriálu. Předpokládáme, že zásobník roste směrem k hornímu okraji strany.

Podíváme-li se pozorně na definici procedury `g` a na obrázek Obr. 13, vidíme, že vlastně není důvod stavět na sebe jednotlivé zásobníkové záznamy s různými hodnotami proměnných `n` a `a`. Namísto toho by stačilo použít jen jeden zásobníkový záznam a hodnoty `n` a `a` v něm **měnit**.

O takovémto výpočtu bychom řekli, že probíhá v konstantním zásobníkovém prostoru: zásobník výpočtu se v průběhu vykonávání programu nezvětšuje. Takovýto výpočet se také jinak nazývá *iterativní*.

Koncově rekurzivní procedura generuje iterativní výpočetní proces.

Některé programovací jazyky jsou schopny automaticky detekovat koncově rekurzivní procedury a převést rekurzivní volání na iterativní proces. Tím ušetří mnoho paměti a strojového času stráveného údržbou zásobníku.Takovýmto jazykům pak také říkáme koncově rekurzivní.

Průvodce studiem

Je zajímavé porovnat lineárně rekurzivní a koncově rekurzivní algoritmus na výpočet faktoriálu. z hlediska efektivity je jistě lepší algoritmus koncově rekurzivní. Mnohem srozumitelnější je ale algoritmus lineárně rekurzivní.

Nalézt k obecnému lineárně rekurzivnímu algoritmu jeho koncově rekurzivní variantu není triviální a vyžaduje to jistou praxi. Jedním z vodítek je, že procedura by měla akceptovat o jeden parametr víc – zde si algoritmus bude „pamatovat“ mezivýsledky výpočtu. v mezní podmínce výpočtu pak tuto proměnnou vrátí jako výsledek výpočtu.

Pokud máme v našem fiktivním meta-jazyku syntaktickou podporu pro vyjádření cyklů výpočtu, lze předchozí lineárně rekurzivní algoritmus převést na cyklus následujícím způsobem. Formální parametry procedury jsou zde použity jako proměnné cyklu.[1]

```
procedure g(n, a) {
   loop {
      if n = 0 return a
      else {
         a = n * a;
         n = n - 1
      }
   }
}
```

2.3.5 Rozsah platnosti proměnných

Uvažujme následující kousek kódu. Jde o proceduru s tajemným názvem `mystery`. Tato procedura akceptuje parametr `x`, deklaruje dvě lokální procedury `P` a `Q` a vyvolává proceduru `Q` s parametrem 2.

```
procedure mystery(x) {

   procedure P() {
      display(x)
   }

   procedure Q(x){
      display(x)
      P()
   }

   Q(2)
}
```

Jako základní bereme poznatek, že je-li v modulu deklarovaná nějaká proměnná (parametr procedury, lokální procedura), je v tomto modulu přístupná. Takováto proměnná se nazývá lokální danému modulu. Pokud je proměnná definovaná *vně* daného modulu (například proměnná `x` je definovaná vně procedury `P`), je v daném modulu přístupná stejně jako lokální proměnné. Výjimku tvoří případ, kdy některá lokální proměnná má stejné jméno jako proměnná deklarovaná vně modulu. v tom případě platí princip, že lokální proměnná „zastíní" proměnnou vně modulu a proměnná vně modulu se tak stává nepřístupnou.

Proměnné, které nejsou lokální, hledáme vně procedury..

Obráceně platí, že proměnná deklarovaná uvnitř modulu, je vnitřním majetkem tohoto modulu a není dostupná z vnějšku.

Naším úkolem je zjistit, jaká dvě čísla se objeví na obrazovce po vyvolání procedury `mystery` s parametrem `1`. Ukazuje se, že výsledek vyvolání této procedury závisí na tom, jak programovací jazyk zachází se jmény proměnných a procedur.

Průvodce studiem

Tělo procedury `mystery` se skládá ze dvou částí: vyskytují se zde dvě lokální procedury `P` a `Q` a volá se zde procedura `Q` s parametrem `2`. Lokální procedury jsou viditelné pouze uvnitř procedury `mystery` a zvenčí jsou nedostupné. Tuto techniku volíme v případech,

[1] Tento příklad používá symbol = jak ve smyslu přiřazovacího příkazu, tak ve smyslu predikátu rovnosti. v rámci jednoduchosti jsme nezaváděli zvláštní symbol pro přiřazovací příkaz a doufáme, že význam symbolu = je zřejmý z kontextu. Přiřazovací příkaz nastavuje hodnotu proměnné uvedené na levé straně výrazu na hodnotu výrazu uvedeného na pravé straně výrazu.

kdy chceme procedury zabezpečit tak, aby nemohly být volány „kýmkoli“ a „odkudkoli“, nebo když potřebujeme skrýt jména těchto procedur pro zbytek programu.

Vyvoláme-li proceduru `mystery` *s parametrem* `1`*, vyvolá se nejprve procedura* `Q` *s parametrem* `2`*. Proměnná* `x` *v proceduře* `Q` *zastíní hodnotu proměnné téhož jména v proceduře* `mystery`*. První číslo, které bude vypsáno na obrazovku bude tedy jistě hodnota* `2`*. V proceduře* `Q` *se dále zavolá procedura* `P`*, bez parametrů. I procedura* `P` *se pokusí o vypsání parametru* `x`*.*

Celý problém je v tom, jak jazyk definuje termín *vně* modulu, co je vlastně *vnějšek* modulu. k tomuto problému existují v zásadě dva přístupy:

Moderní jazyky používají lexikální rozsah platnosti proměnných.

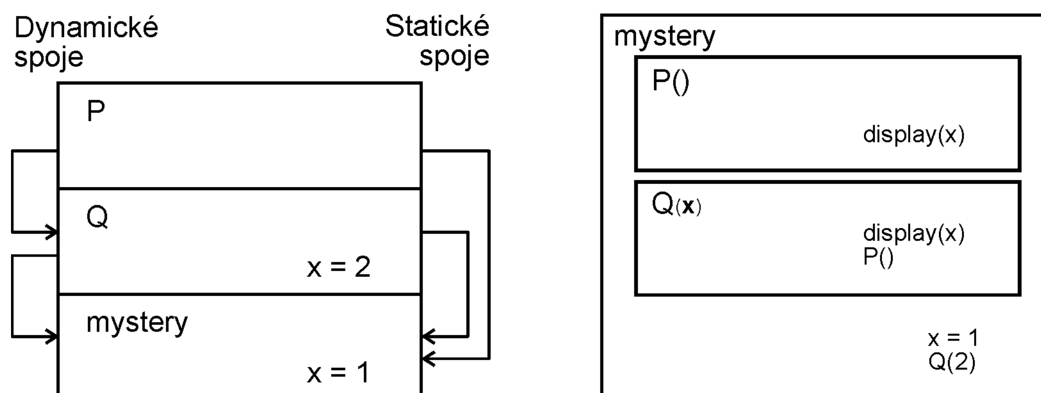
- První chápe vnějšek modulu *lexikálně*, tj. tak, jak jsou do sebe vnořeny jednotlivé deklarace ve zdrojovém programu. Vnějšek modulu je tak určen strukturou programu ještě dříve, než je program spuštěn na počítači.
- Druhý chápe vnějšek modulu *dynamicky*, tj. vnějšek je tvořen těmi aktivacemi procedur, ze kterých byl aktivován daný modul. Vnějšek modulu tak není možné určit ze zdrojového programu, je dán až průběhem výpočtu.

V prvním případě hovoříme o jazyku, podporujícím *lexikální rozsah platnosti proměnných*, ve druhém případě o jazyku, podporujícím *dynamický rozsah platnosti proměnných*.

V kapitole 2.3.1 jsme se seznámili s pojmem zásobník programu. Kromě návratové adresy, lokálních proměnných a skutečných parametrů obsahuje záznam na zásobníku tyto dvě položky, využívané k zjišťování rozsahu platnosti proměnných:

- *dynamický spoj* (*dynamic link*): ukazatel na předchozí aktivační záznam uložený v zásobníku, tedy ukazatel na aktivační záznam volající procedury.
- *statický spoj* (*static link*): ukazatel na aktivační záznam nejbližšího lexikálně nadřazeného bloku.

Obrázek Obr. 14 znázorňuje blokové schéma procedury `mystery` (určující lexikální nadřazenost bloků) a stav zásobníku v okamžiku vykonávání procedury `P`. Jsou zde schematicky znázorněny dynamické a statické spoje.



Obr. 14 Dynamické a statické spoje.

Při zjišťování hodnoty dané proměnné programovací jazyk nejprve zjistí, nejde-li o proměnnou lokální danému modulu. Není-li proměnná deklarovaná v daném modulu, hledá se modul, ve kterém je proměnná deklarovaná a kde je tedy možno zjistit její hodnotu.

U lexikálního rozsahu platnosti proměnných využívá překladač statické spoje.

Toto hledání představuje prohledávání zásobníku za využití spojů. Používá-li jazyk pro tento účel dynamické spoje, uplatňuje dynamický rozsah platnosti proměnných, používá-li statické spoje, jde o lexikální rozsah platnosti proměnných.

V našem případě tedy zřejmě výsledkem programu budou čísla `2` a `1` v případě, že jazyk používá statický rozsah platnosti proměnných a `2` a `2` při dynamickém rozsahu.

Průvodce studiem

Moderní programovací jazyky preferují statický rozsah platnosti proměnných. Vytvoření překladače pro tento typ platnosti proměnných je sice o něco obtížnější (musí se konstruovat statické vazby), pro programátory je však podstatně komfortnější. Dynamický rozsah platnosti proměnných má například jazyk Lisp.

Shrnutí

Vyvolá-li jedna procedura druhou, na zásobník programu se uloží aktivační záznam, obsahující skutečné hodnoty parametrů, lokální proměnné a dynamické a statické spoje. Kromě zásobníku obsahuje paměť programu ještě datový segment a hromadu. Předávání řízení mezi procedurami můžeme znázornit pomocí aktivačního stromu.
Volá-li procedura samu sebe, jedná se o rekurzi. Existují lineárně rekurzivní a stromově rekurzivní procedury. Rekurzivní procedury mají vždy stejnou strukturu: obsahují mezní podmínku rekurze a rekurzivní předpis. Běh rekurzivní procedury zahrnuje fázi navíjení a fázi odvíjení. Je-li procedura koncově rekurzivní, lze ji vykonávat v konstantním zásobníkovém prostoru a formálně převést na cyklus.
Rozeznáváme lexikální a dynamický rozsah platnosti proměnných. Rozsah platnosti proměnných určuje, odkud systém bere hodnoty proměnných, které nejsou v daném modulu lokální.

Pojmy k zapamatování

- Datový segment, hromada, zásobník, aktivační záznamy,
- aktivační strom programu,
- rekurzivní procedury, mezní podmínka rekurze, rekurzivní předpis, fáze navíjení a odvíjení,
- lineárně rekurzivní procedury, stromově rekurzivní procedury, koncově rekurzivní procedury, iterace
- dynamický a lexikální rozsah platnosti proměnných, dynamické a statické vazby.

Kontrolní otázky

1. *Co najdeme v datovém segmentu programu?*
2. *Co to je aktivační záznam, kde je uložen, kdy vzniká a kdy zaniká?*
3. *Jaké jsou základní dvě části každé rekurzivní procedury?*
4. *Jaký je rozdíl mezi iterací (cyklem) a koncově rekurzivní procedurou?*
5. *Jaký typ rozsahu platnosti proměnných je běžně využíván v moderních programovacích jazycích.*

Cvičení

1. Jaký bude výsledek volání procedury `P` z kapitoly 2.3.2 s parametrem `2`? Použijte aktivační strom z obrázku Obr. 11.
2. Předpokládejme, že bychom z nějakého důvodu nebo omylem implementovali proceduru `Fact` takto (tato procedura již nepočítá matematickou funkci faktoriál):

   ```
   procedure Fact(n) {
      if n = 0 return 1
      else return n * Fact(n - 2)
   }
   ```

   Vyzkoušíme-li proceduru například pro skutečný parametr `6`, procedura vrátí odpovídající výsledek. Kód je přesto chybný. Za jakých podmínek nastává problém a jaký?

Úkoly k textu

1. Nakreslete stav zásobníku v okamžiku provádění procedury `R` z kapitoly 2.3.1 s parametrem `7` na základě předchozího aktivačního stromu. Do každého aktivačního záznamu vepište, jakou obsahuje hodnotu `x`.
2. Napište novou verzi procedury na výpočet prvků Fibonacciho posloupnosti tak, aby byla koncově rekurzivní.

Řešení

1. Výsledek je hodnota 100.
2. Problém nastává tehdy, když proceduru vyvoláme s lichou hodnotou parametru. v každém kroku zmenšujeme hodnotu `n` o dvojku a v mezní podmínce rekurze testujeme situaci, kdy `n` bude rovno nule. U lichých hodnot `n` tato situace ale nikdy nenastane, `n` bude nabývat hodnot `3`, `1`, `-1`, `-3` atd. Hodnotu `0` program „přeskočí“ a skončí tak nekonečných cyklem.

2.4 Předávání parametrů procedurám

Studijní cíle: Po absolvování kapitoly studující porozumí jednotlivým variantám předávání parametrů procedurám. Bude schopen zvolit pro danou situaci vhodný způsob předávání parametrů.

Klíčová slova: L-value, R-value, předávání hodnotou, odkazem, výsledkem.

Potřebný čas: 2 hodiny.

Doposud jsme si ukázali, jak se procedury deklarují a jak je možno předávat do procedur hodnoty mechanismem předávání parametrů. v praxi, především v procedurálním paradigmatu, používáme několik variant předávání parametrů.

2.4.1 L-value a R-value

Výrazy mohou mít R-value i L-value.

V popisu možností předávání parametrů proceduře budeme používat pojmy *L-value* a *R-value*. Dříve než přistoupíme k samotnému popisu, vysvětlíme si význam těchto pojmů. Termíny L-value a R-value vznikly původně při práci s přiřazovacím příkazem, kdy termín L-value označuje hodnotu, která musí být přítomna na levé straně přiřazovacího příkazu a R-

value hodnotu stojící vpravo od přiřazovacího příkazu. Máme-li tedy dva výrazy `A` a `B`, potom, chceme-li provést přiřazení `A = B`, musí pro výraz `A` existovat L-value a pro výraz `B` R-value.

Obecně pojmem L-value označujeme místo v paměti (paměťovou buňku) do které lze zapisovat. Jakožto R-value chápeme hodnotu, která může být uložena na nějakém místě paměti. Mnoho výrazů má jak L-value, tak R-value. Demonstrujme si celou situaci na jednoduchém příkladě. Označme paměť počítače písmenem `M`, jednotlivé buňky paměti jako `M[i]`. Předpokládejme, že paměť je naplněna hodnotami tak, jak je to znázorněno na obrázku Obr. 15.

Obr. 15 Úsek paměti.

Potom L-value výrazu `M[1]` je adresa `1` v paměti stroje. R-value výrazu `M[1]` je číslo `4`. Výraz `M[0] + M[1]` nemá žádnou L-value, má však R-value rovnu hodnotě `10`. z tohoto pohledu je již zřejmá oprávněnost pojmenování: v přiřazovacím příkazu `A = B` musí mít výraz `A` na levé straně L-value, tj. musí reprezentovat nějakou adresu v paměti. Výraz `B` na pravé straně musí mít R-value, tj. musí existovat hodnota, kterou lze přiřadit.

Průvodce studiem

Některé výrazy mají pouze R-value, zatímco jiné mají jak R-value tak L-value. v našem příkladě výraz `M[0] + M[1]` nemá L-value, protože vniknul součtem dvou jiných výrazů a neoznačuje žádné konkrétní místo v paměti.

Existují však programovací jazyky, kdy výsledek aritmetických výrazů může mít v některých případech i L-value. Například v jazyku C lze sčítat a odčítat adresy v paměti a výsledný výraz má L i R-value.

2.4.2 Předávání parametrů

Předáváním parametrů budeme nadále rozumět proces navázaní skutečných parametrů na formální parametry procedury. Tento proces může být uskutečněn mnoho způsoby - základním kriteriem přitom je, co předáváme do paměťového prostoru určeného parametrům a jak s předanou hodnotou nakládáme. Často namísto „způsobů předávání parametrů“ používáme termín „způsoby volání procedur“.

Pro demonstrativní účely si nadefinujeme proceduru `swap` takto:

```
procedure swap(x, y) {
   var temp
   temp = x
   x = y
   y = temp
}
```

V této proceduře použijeme lokální proměnnou procedury nazvanou `temp`. Tato proměnná je vytvořena v zásobníku programu pokaždé, když je procedura swap aktivována. Efekt předávání parametrů si popíšeme na volání procedury `swap` s parametry `I` a `A[I]`, kde `A` je pole čísel a `A[I]` je i-tá položka v poli čísel `A`.

2.4.3 Předávání parametrů hodnotou

Volání hodnotou (*Call by value*) používá například jazyk C. s tímto způsobem předávání parametrů se však setkáváme ve většině ostatních programovacích jazyků a mnohde představuje implicitní způsob předávání parametrů.

Volání hodnotou je základní způsob předávání parametrů.

Při aktivaci procedury systém postupně vyhodnotí všechny aktuální parametry volání. Výsledné hodnoty jsou přiřazeny dočasným lokálním proměnným, odpovídajícím formálním parametrům procedury. Tyto proměnné jsou uloženy v aktivačním záznamu na zásobníku programu. s těmito proměnnými se pak v proceduře pracuje stejně jako s libovolnými jinými lokálními proměnnými.

V případě volání procedury `swap(I, A[I])` se vytvoří se dočasné proměnné `T1` a `T2`, provede se přiřazení:

```
T1 = I
T2 = A[I]
```

a provede se kód procedury ve tvaru:

```
temp = T1
T1 = T2
T2 = temp
```

Po ukončené vykonávání procedury zanikne aktivační záznam na zásobníku a zaniknou tak i hodnoty `T1` a `T2`. Důležitým poznatkem je, že vzhledem k tomu, že veškerá manipulace probíhá s dočasnými proměnnými, hodnoty výrazů `I` a `A[I]` se po provedení procedury nezmění.

2.4.4 Předávání parametrů odkazem

Volání odkazem (*Call by reference*) je požíváno méně často než volání hodnotou, vyskytuje se však také v řadě programovacích jazyků. Podporuje jej například jazyk C#.

Volání odkazem používáme například pro vedlejší efekt..

Při aktivaci procedury se nejprve se vyhodnotí aktuální parametry. Předpokládá se přitom, že všechny aktuální parametry mají L-value.[1] Do procedury pak nepošleme hodnoty aktuálních parametrů, ale jejich adresy v paměti (tedy L-values).

V případě volání procedury `swap(I, A[I])` se tedy vytvoří se dočasné proměnné `Addr1` a `Addr2` pro adresy skutečných parametrů a provede se přiřazení:

```
Addr1 = ref(I)
Addr2 = ref(A[I])
```

Předpokládáme přitom, že `ref` je zabudovaná systémová procedura, která nám k proměnné vrátí její adresu v paměti. Protějškem procedury `ref` bude procedura `deref`, která nám k dané adrese v paměti vrátí obsah paměťové buňky s touto adresou.

Dále se provede se kód procedury ve tvaru:

```
temp = deref(Addr1)
deref(Addr1) = defer(Addr2)
deref(Addr2) = temp
```

Po vykonání procedury hodnoty `Addr1` a `Addr2` zaniknou. Všimněme si však, že manipulace neprobíhala s hodnotami `Addr1` a `Addr2`, ale obsahy paměťových buněk, které leží na těchto adresách. Vedlejším efektem této procedury tedy je, že do paměťové buňky `I` se přiřadí obsah buňky `A[I]` a do původní paměťové buňky `A[I]` se přiřadí obsah buňky I.

---

[1] Pokud by některý skutečný parametr neměl L-value, je možné jeho R-value přenést do dočasné proměnné a vzít L-value této proměnné. Mnoho jazyků však v této situaci jednoduše vyhlásí chybu.

Průvodce studiem

Rozdíl mezi proměnnou, adresou paměti a obsahem dané adresy není zcela průhledný a může z počátku působit studujícím potíže. Chápejme na chvíli paměť jako řadu rodinných domků stojících vedle sebe v ulici. Adresou je pak číslo rodinného domu, obsahem adresy je pak obyvatel daného domku. Předpokládejme, že každý domek má fasádu jiné barvy. Náš kamarád Petr bydlí v zeleném domku s orientačním číslem 12. v naší zavedené syntaxi procedur `ref` *a* `deref` *platí:*

```
ref[ZelenýDomek] je 12
deref[12] je Petr
```

Používá-li jazyk systém předávání parametrů odkazem, je důležité si vždy uvědomit, že v procedurách pracujeme ve skutečnosti s adresami proměnných exitujících mimo proceduru. Tento způsob předávání parametrů je velmi výhodný, chceme-li aby procedura měla vedlejší efekt na své parametry. Často se také používá tam, kde jednotlivé výrazy představují velké úseky paměti (velké databázové tabulky, obrázky) a bylo by pomalé, nevýhodné nebo dokonce nemožné ukládat do zásobníku lokální kopie těchto výrazů.

2.4.5 Předávání parametrů hodnotou-výsledkem

Volání hodnotou-výsledkem se používá u paralelních a distribuovaných výpočtů.

Volání hodnotou-výsledkem (*Call by value-result*, *copy-restore linkage*) používají například jazyky Fortran nebo MPD.

Při volání procedury se nejprve u všech skutečných parametrů zjistí L-value i R-value. R-value jsou předány do procedury stejně jako u volání hodnotou. L-value všech parametrů jsou uschovány do tabulky. Po vykonání těla procedury, dříve než jsou dočasné proměnné odstraněny ze zásobníku, jsou hodnoty dočasných proměnných nakopírovány zpět do L-value aktuálních parametrů.

V případě volání procedury `swap(I, A[I])` se vytvoří dočasné proměnné T1 a T2 a provede se přiřazení:

```
T1 = I
T2 = A[I]
```

Do tabulky se uloží L-values jednotlivých skutečných parametrů:

```
Arg1 = ref(I)
Agr2 = ref(A[I])
```

a provede se kód procedury ve tvaru:

```
temp = T1
T1 = T2
T2 = temp
```

Po vykonání procedury se pak provede přiřazení:

```
deref(Arg1) = T1
deref(Arg2) = T2
```

Pokud některý z parametrů nemá svoji L-value, jazyk buď nahlásí chybu nebo se chová jako by daná hodnota byla posílána hodnotou a poslední fázi vynechá.

Tento způsob předávání parametrů dává v mnoha případech podobné výsledky, jako předávání parametru odkazem. Tak tomu je i v našem příkladě. v obecném případě však takovéto tvrzení neplatí. Do programovacích jazyků byl tento způsob předávání parametrů zaveden jak z důvodu jednoduché implementace (Fortran), tak pro výhody, které toto předávání parametrů má oproti volání procedur odkazem v případě paralelního nebo distribuovaného výpočtu.

Průvodce studiem

Při distribuovaném výpočtu běží volaná procedura na jiném počítači v jiném paměťovém prostoru a nelze proto použít odkazů.

U paralelního výpočtu běží současně volající a volaná procedura ve stejném paměťovém prostoru. v těchto případech může dojít i k tomu, že volající procedura terminuje dříve, než stačí terminovat procedura volaná. v tom případě bychom při volání odkazem dostali ve volané proceduře referenci na oblast paměti, jejíž obsah není definovaný a který může být využíván jiným procesem (tzv. dangling pointer). Používáme-li předávání parametrů hodnotou-výsledkem, tento problém nenastane: stačí, když systém zajistí, aby neproběhla poslední fáze.

2.4.6 Předávání parametrů jménem

Volání jménem (*Call by name*) bylo použito například v jazyku Algol68. U volání procedur jménem se aktuální parametry považují za jména a jsou opakovaně vyhodnocovány kdykoli, kdy jsou uvedeny. Tento způsob předávání parametrů je dosti komplikovaný, přináší zvláštní vedlejší efekty a v moderních programovacích jazycích se běžně nepoužívá. Předávání parametru jménem je však co do funkčnosti identické s používáním makra.[1]

V našem ukázkovém příkladě je tedy volání procedury `swap(I,A[I])` vykonáváno tak, jako by byl vykonáván kód:

```
temp = I
I = A[I]
A[I] = temp
```

Efekt tohoto kódu si čtenáři jistě zjistí sami.

Průvodce studiem

V programovacím jazyku je typicky možno využít více než jeden způsob předávání parametrů proceduře.Většina jazyků proto zavádí modifikátory, které určují jak bude daný parametr předán. Hlavička procedury pak může vypadat třeba takto:

```
procedure test(val x, ref y).
```

Klíčová slova `val` *a* `ref` *nám určují, že parametr* `x` *bude předáván hodnotou, zatímco parametr* `y` *bude předáván odkazem.*

Shrnutí

Programovací jazyk většinou umožňuje předávat parametry do procedur více způsoby. Základní dva způsoby jsou předávání hodnotou a předávání odkazem. U předávání hodnotou se vytvářejí lokální kopie skutečných parametrů a změny, provedené na formálních parametrech v proceduře, se neprojeví mimo proceduru. U předávání parametrů odkazem se do procedury přenášejí pouze odkazy (reference) na proměnné existující mimo proceduru. Tento typ předávání parametrů využíváme tam, kde potřebujeme provést změny v hodnotách proměnných mimo

[1] v případě makra je před provedením překladu každý výskyt identifikátoru makra nahrazen definovaným segmentem kódu.

proceduru nebo tam, kde hodnoty parametrů jsou příliš velké a vytváření jejich kopií na zásobníku by nebylo efektivní.
Vedle těchto základních způsobů existuje i předávání parametrů hodnotou/výsledkem a jménem. Použití těchto způsobů předávání parametrů je omezené na speciální situace.

Pojmy k zapamatování

- L-value, R-value,
- předávání parametrů hodnotou,
- předávání parametrů odkazem,
- předávání parametrů hodnotou/výsledkem,
- předávání parametrů jménem.

Kontrolní otázky

1. *Jaký je podstatný rozdíl mezi předávání parametrů hodnotou a odkazem?*
2. *Jak se pravděpodobně zachová překladač, pokud budete chtít předat odkazem parametr `x + 7`?*
3. *Za jakých okolností použijeme předávání parametrů hodnotou/výsledkem?*

Cvičení

1. Určete L-value a R-value výrazu: `M[M[M[5]]]` z kapitoly 2.4.1

Úkoly k textu

1. Zvolte si vhodně pole `A` a hodnotu proměnné `I`.Na těchto hodnotách demonstrujte efekt předávání parametrů odkazem z kapitoly 2.4.4.
2. Obdobně demonstrujte výsledek předávání parametrů jménem z kapitoly 2.4.6

Řešení

1. L-value je buňka číslo 4 pole `M`. R-value je hodnota 15.

2.5 Styl programu

Studijní cíle: Studující se po prostudování této kapitoly naučí správně formátovat kód a používat vhodný formální styl programu.

Klíčová slova: Jména v programu, jmenná konvence, zarovnání.

Potřebný čas: 30 minut

Doposud jsme hovořili o programovacím paradigmatu a o důležitosti struktury programu. Vedle strukturovanosti patří mezi základní kvalitativní charakteristiky každého programu i přehlednost. Tuto problematiky považujeme za natolik závažnou, že jí věnujeme celou podkapitolu.

Formální stránka programu je stejně důležitá jako stránka obsahová.

Kromě estetického dojmu z kódu je přehlednost nutná ke snadnějšímu odlaďování a rozšiřo-

rozšiřování programu. Programovací jazyk a programovací paradigma sice poskytuje programátorovi hojné možnosti pro vhodnou strukturaci programu, málokdy však strukturu vynucuje.

2.5.1 Volba vhodných jmen

Prvním principem správného stylu kódu je volba vhodných *názvů* procedur, parametrů procedur, proměnných a konstant. Název by měl být maximálně výstižný. Nebojme se přitom delších názvů! Vyžaduje to sice více psaní při tvorbě kódu, ale vynaložené úsilí se nám vrátí. Jméno procedury nebo parametru by mělo maximálně vystihovat jeho účel. Často je jméno složeno z několika slov, oddělených podtržítkem nebo kombinací velkých a malých písmen.

Pojmenujeme-li procedury `proc1` nebo `procedx12`, nemáme tušení k čemu tyto procedury slouží. Mnohem vhodnější názvy by byly `WriteRowToDatabase` a `CloseTCPConnection`.

Pro názvy identifikátorů v programu volíme pokud možno angličtinu. Používáme-li češtinu, pak výhradně bez diakritiky.

V některých programovacích jazycích jsou prostory jmen pro různé programové entity oddělené. Identifikátor tak může představovat současně název procedury, skalární parametr nebo parametr typu pole (k datovým typům se dostaneme v kapitole 3.1). Správný prostor jmen určí překladač z kontextu použití identifikátoru. I když to z počátku vypadá jako elegantní vlastnost jazyka, ve větším projektu se tato vlastnost blíží pokusu o sabotáž. Doporučujeme proto těchto vlastností nevyužívat a volit pro každou programovou entitu vždy jednoznačné jméno.

2.5.2 Zavedení jmenných konvencí

Důležité je také určení a důsledné dodržování *jmenných konvencí*. Můžeme si například určit, že všechna jména procedur budou začínat velkými písmeny. Je-li název složen z více slov, pak každé slovo bude opět začínat velkým písmenem (tzv. *camel casing*). Všechny procedury, které pracují s databází navíc mohou například začínat dohodnutou zkratkou. Vzniknout tak procedury jako `DBOpenConnection` nebo `DBDeleteRow`. Všechny proměnné a parametry mohou naopak začínat malými písmeny a jednotlivá slova mohou být oddělena podtržítkem. Vzniknou tak názvy jako `row_id`, `session_name` a podobně.

Jmenné konvence nám dodávají významné meta informace o programu. Pouhým pohledem na název procedury můžeme například určit, do jakého balíčku procedur patří.

2.5.3 Vkládání poznámek

Důležitou součástí každého programu jsou poznámky. Poznámky píšeme jednak ke každému modulu (proceduře) programu tak, aby každý pochopil, k čemu modul slouží. Snažíme se přitom zvlášť okomentovat hlavní účel procedury, popsat význam předávaných parametrů, návratovou hodnotu procedury a eventuální vedlejší efekt procedury. Dále se snažíme vložit komentář před každý náročnější krok v programu. Komentáři také oddělujeme jednotlivé logické části procedur.

V komentářích se stejně jako u názvů procedur a proměnných pokud možno držíme anglického jazyka. I když to na první pohled může vypadat jako nepodstatné, časem to oceníme. Programátorské týmy jsou často nadnárodní a anglické názvosloví je dnes v programování (a nejen tam) nedeklarovaným standardem.

Průvodce studiem

Používání angličtiny v programech může z počátku působit až snobsky.Proč se trápit s anglickými názvy a komentáři, když můžeme použít češtinu. Důvodů je mnoho, od zažité anglické terminologie až po situaci, kdy váš program dostane do ruky někdo jiný. V praxi

často nastávají situace, kdy programátora život postaví před cizí kód. A co když programátor tohoto kódu byl shodou okolností z Finska procedury mají jména jako třeba `AuetaAsiakaspiiri`*?*

Cílem by mělo být oddělení vnitřní logiky programu od uživatelského rozhraní. Veškerá komunikace s uživatelem jako jsou výpisy, nabídky nebo nápovědy by měla být jednoznačně v národním jazyce. Vnitřní život programu by měl fungovat v angličtině.

K češtině se uchylujeme pouze tehdy, pokud anglicky umíme jen velmi špatně a naše poznámky by tak byly nesrozumitelné. V tom případě se alespoň vyhneme se diakritice.

2.5.4 Zarovnávání kódu

Úroveň vnoření kódu ukazujeme pomocí odsazení řádků programu.

V některých programovacích jazycích byl způsob zarovnání pevně dán a jednotlivé výrazy musely být umístěny na samostatných řádcích a začínat na předepsaných sloupcích textu. v dnešní době většina moderních programovacích jazyků podporuje tzv. volný styl psaní kódu. Kód může být formátován libovolně a můžeme jej například napsat celý na jeden jediný dlouhý řádek.

I když programátorům volný styl kódu jistě uvolnil ruce, ne všichni takovouto volnost unesou. Pro překladač sice není formátování kódu podstatné, pro lidského čtenáře je stále klíčové. Při správném zarovnání by měl být kód přehledný a bloky které patří k sobě by měly být stejně odsazeny. Úroveň odsazení je určena úrovní vnoření; tradičně odsazujeme text vždy o tři mezery na jednu úroveň logického vnoření. Správné formátování nazýváme anglickým termínem *pretty-printing* a některé integrované vývojové nástroje ho dělají za nás.

Příklad správně formátovaného kódu jsme viděli v kapitole 2.3.4. Porovnejte kód z kapitoly 2.3.4 s následujícím stejným, ale chybně formátovaným kódem:

```
procedure g(n, a) {loop {
if n = 0 return a
     else {
a = n * a; n = n - 1
}}
}
```

2.5.5 Příklad

Procedura na načtení dat z databáze na základě dotazu by za dodržení pravidel stylu a jmenných konvencí mohla vypadat například takto:[1]

```
# -----------------------------------------------------------
# Fetches table from database based on user query
# uses global database connection string
# Accepts: SQL query string
# Returns: database table as 2D array or false on failure
# Modifies: opens and closes connection to database
procedure DBFetchTableFromDatabase(string SQL_query) {

   # try to open database connection
   database_handler = DBOpenConenction(ConnectionString)
   if (database_handler = false) {
      # connection failed, exiting
      return false
   }

   # database connection opened succesfully
```

[1] Tento příklad využívá některé rysy našeho meta-jazyka, které jsme dosud nezavedli. Patří mezi ně například symbol # pro poznámky nebo podmíněný příkaz. Tento příklad je třeba chápat jen jako demonstraci dodržování stylu kódu a jmenných konvencí.

```
    # perform query and fetch result
    table = DBExecuteQuery(database_handler, SQL_query)
    DBConnectionClose(database_handler)

    # return result of the query
    return table
}
```

Průvodce studiem

I v tom nejlepším programovacím jazyku, za využití nejmodernějšího vývojového prostředí, lze napsat kód který je zcela nečitelný, nepřehledný a ošklivý. I když, a někdy je třeba říci bohužel, takovýto program funguje, je to program špatný. Od programu neočekáváme jenom funkčnost, ale i přehlednost nutnou k dolaďování a pozdějšímu rozšiřování programu a stranou by neměla stát ani estetická stránka.

Většina programátorů k těmto poznatkům dojde sama. Úplně stačí, aby zákazník projevil přání doplnit do staršího programu nové rysy nebo udělat změnu. Programátor, který se po roce vrací ke svému dílu má dojem, že se dívá na zcela cizí kód. To co bylo před rokem nad slunce jasné, je nyní temnou záhadou. v těchto chvilkách je zcela neocenitelné dodržování konzistentního stylu programu, jmenných konvencí a discipliny poznámek.

Shrnutí

Styl psaní programu a dodržování základních konvencí je stejně podstatný jako dodržování programovacího paradigmatu. k základním pravidlům patří zvolení vhodných jmen procedur a proměnných, dodržování jmenné konvence, používání poznámek a správné zarovnávání kódu.

Pojmy k zapamatování

- Jmenná konvence,
- camel casing,
- pretty-printing.

Kontrolní otázky

1. *V jakém jazyce píšeme komentáře v programu?*
2. *Co by měl popisovat komentář u deklarace procedury?*
3. *Co určuje úroveň odsazení řádku v kódu?*

3 Struktura dat

3.1 Typy dat v programovacích jazycích

Studijní cíle: Po absolvování kapitoly bude studující schopen definovat a klasifikovat datové typy programovacích jazyků. Porozumí pojmům typový systém a typová kontrola.

Klíčová slova: Typ, jednoduché datové typy, strukturované datové typy, statická typová kontrola, dynamická typová kontrola.

Potřebný čas: 1 hodina 30 minut.

V programech se kromě klíčových slov a procedur, určených k řízení a strukturaci programu, vyskytují parametry procedur, proměnné a konstanty. Souhrnně nazýváme tyto prvky programu *data*.

3.1.1 Datové typy

Proměnné v programu mají přidělen datový typ.

Proměnné programu můžeme kombinovat pomocí operátorů a procedur a vytvářet tak složitější výrazy. Každá proměnná nebo složený výraz může nabývat jen jistých hodnot a lze s nimi provádět pouze jisté operace. v této souvislosti hovoříme o *typu výrazu*: typ nám označuje, jakých hodnot může výraz nabývat a jaké operace na něj mohou být aplikovány.

Většina programovacích jazyků například rozeznává typ zvaný `integer`, který reprezentuje celá čísla a operace, které je nad celými čísly možno provádět. Stejně tak bývá běžný typ `string`, reprezentující textové řetězce spolu s operacemi, prováděnými nad řetězci. Má-li nějaký výraz typ `integer`, znamená to, že můžeme očekávat jako jeho hodnoty výrazy `25`, `78`, `-2567` atd., ale nemůžeme očekávat, že tento výraz nabude hodnoty např. `"Bobr je hlodavec"`. Dále víme, že na tento výraz můžeme aplikovat operace jako sčítání, násobení, umocňování atd., ale nemůžeme výrazy tohoto typu spojovat nebo převádět na velká písmena. Operace jako spojování nebo konverze na velká písmena jsou definovány pro typ `string` a pro typ `integer` jsou neznámé.

Jedním z dalších často používaných typů je typ `bool` (nazvaný podle irského logika Georgea Boolea), označující logické hodnoty `true` (pravda) a `false` (nepravda). Pro tyto hodnoty jsou definovány operace logického součtu a součinu, mnohdy označované stejně jako operace aritmetického součtu a součinu. Na první pohled je však zřejmé, že operace logického součinu je odlišná od operace aritmetického součinu – je otázkou náhody, že jsou v daném programovacím jazyku obě operace označeny stejně symboly + a *. Je proto důležité si uvědomit, že všechny operace jsou součástí typu a nelze je zaměnit.

Průvodce studiem

Z čistě technického hlediska nám typ označuje jak interpretovat jistý úsek paměti. Paměť počítače je složená z buněk (bitů), které mohou obsahovat pouze hodnoty jedna nebo nula. Tyto buňky seskupujeme po osmi do slabik (bytů), které tvoři nejmenší adresovatelnou položku paměti. Jedna slabika může obsahovat číslo od 0 do 255, vyjádřené pomocí bitů ve dvojkové soustavě. Každá proměnná v programu nám odkazuje na jeden nebo více slabik v paměti počítače. Je otázkou typu výrazu, jak hodnotu nalezenou na tomto místě paměti interpretovat.

Nalezneme-li například ve dvou po sobě následujících slabikách paměti hodnoty `01000011 01011010`, *může to odpovídat textovému řetězci „CZ“ nebo celému číslu 8634, v závislosti na typu proměnné, která se na daném místě paměti nachází.*

Dříve než se začneme věnovat způsobům, jak programovací jazyk s typy pracuje, uveďme si alespoň ve stručném přehledu základní druhy a klasifikaci datových typů.

3.1.2 Klasifikace datových typů

Některé datové typy podporuje procesor, jiné vznikají v programu.

Datové typy můžeme klasifikovat podle dvou kriterií: podle toho na jaké úrovni vznikají a podle jejich struktury.

Podle prvního kriteria rozeznáváme typy vznikající na:

- úrovni stroje – tyto typy jsou přímo podporované procesorem. Bývají to zejména typy `char` (`byte`), `integer` a `real`.
- úrovni jazyka – tyto typy nejsou definované strojem, ale jsou simulovány programovacím jazykem. Algoritmy operací nad těmito typy nejsou integrovány v elektronických obvodech stroje, ale jsou součástí programovacího jazyka. Patří sem často typy jako `bool`, `array` nebo `struct`.
- úrovni programátora – tyto typy si v daném programovacím jazyku definuje sám programátor. Můžeme sem řadit například složitější datové typy jako `list` (seznam) nebo `tree` (strom).

Podle druhého kriteria lze typy roztřídit na dvě základní skupiny: jednoduché datové typy a strukturované datové typy.

3.1.3 Jednoduché datové typy

Datové typy mohou být jednoduché nebo strukturované.

Jednoduché datové typy jsou atomární (tj. dále již nedělitelné). Někdy také hovoříme o *skalárních* datových typech. Ze zástupců těchto datových typů vybereme (bez jakýchkoli nároků na úplnost) například:

- `char` – typ zabírající tradičně jednu slabiku paměťového místa počítače, reprezentující písmena nebo čísla v daném paměťovém rozsahu. Často též nazývaný `byte`.
- `integer` – typ zabírající jednu nebo více slabik paměti, reprezentující kladná nebo záporná celá čísla.
- `real` – typ reprezentující čísla v plovoucí desetinné čárce, uložené v paměti počítače většinou ve formě mantisy a exponentu. v některých jazycích nazývaný také `float`.
- `bool` – již zmíněný typ, nabývající hodnot `true` a `false`.
- `enum` – datový typ, u něhož je vyjmenováno, jakých hodnot může nabývat. Programátor si může například nadefinovat datový typ `Season` (roční období), jehož jedinými možnými hodnotami budou `Spring`, `Summer`, `Autumn` a `Winter`.

3.1.4 Strukturované datové typy

Strukturované datové typy nejsou atomární, zahrnují v sobě vždy několik položek stejného nebo různých typů. Každý strukturovaný datový typ musí definovat operaci selekce (tzv. selektor) pomocí něhož je možno přistupovat k jednotlivým položkám datového typu. Mezi nejběžnější typy tohoto druhu patří struktura a pole.

Struktura

`struct` (*struktura*) – odpovídá kartézskému součinu několika typů.[1] Je-li například jedním typem `string`, druhým typem `integer` a třetím typem `bool`, můžeme vytvořit datový typ `string x integer x bool`.

K základním druhům strukturovaných typů patří struktura a pole..

Definice takového datového typu v programovacím jazyku může vypadat například takto:

```
struct Person {
   string Name
   integer Age
   boolean Sex
}
```

Hodnotami výrazu tohoto typu jsou uspořádané trojice. Každá z položek trojice je pro názornost označena jménem. v našem příkladě jsme definovali typ `Person` (osoba) s položkami `Name` (jméno), `Age` (věk) a `Sex` (pohlaví), pomocí něhož můžeme zapisovat základní charakteristiky osob.

Selektorem typu struktura může být například operátor `.` (tečka). Je-li `X` proměnnou typu `Person`, potom `X.Name` odkazuje na jméno, `X.Age` na věk a `X.Sex` na pohlaví daného jedince.

Pole

`array` (*pole*) – odpovídá zobrazení z podmnožiny přirozených čísel do daného typu. Tento typ je též nazýván agregovaným typem. v některých programovacích jazycích hovoříme též o typu *vektor*.

Příkladem může být pole 5-ti reálných čísel, formálně zapsané třeba jako:

```
array Data real[0 .. 4]
```

Pole `Data` lze pak chápat jako zobrazení z množiny {0, 1, 2, 3, 4} do množiny reálných čísel. Do každé položky pole můžeme zapsat jedno reálné číslo. Selektorem typu pole bývá symbol `[i]`, kde `i` je *index* dané položky pole. Je-li `X` proměnná typu `Data`, potom `X[0]` označuje první položku pole, `X[1]` druhou položku atd.

Vedle jednorozměrných polí se v programovacích jazycích vyskytují i vícerozměrná pole. Pomocí takovýchto datových struktur pak můžeme reprezentovat například matice nebo databázové tabulky. Index pole tvoří často číslo, jak tomu bylo v uvedeném příkladě. Indexem pole však mohou být i jiné datové typy a často se setkáváme s poli indexovanými například pomocí řetězců znaků.[2]

3.1.5 Typový systém jazyka

Typovým systémem jazyka rozumíme soubor pravidel, která přiřazují výrazům v daném jazyce typ. Máme-li například výraz `2 + 3`, typový systém přidělí tomuto výrazu nejspíše typ `integer`. Výrazu `2 * (3.14 - 2)` přidělí nejspíše typ real. Pro výraz `7 + „ABCD"` však typový systém typ nenajde. Říkáme proto, že typový systém *akceptuje* výraz, pokud pro něj nalezne typ a *zamítne* výraz, pokud k němu typ nenalezne. Výrazy zamítnuté typovým systémem představují chybu a nemohou být provedeny.

Typový systém jazyka určuje typ výrazů.

Samotný proces přiřazování typu výrazům nazýváme též *typovou kontrolou* výrazů. Typová kontrola má značný význam z hlediska praxe programátora, neboť umožňuje odhalit velké množství programátorských chyb. Ve zkratce lze říci, že typovou kontrolou zjišťujeme, zda typy jednotlivých operandů jsou slučitelné s typy jednotlivých operátorů.

Důležitou úlohu v programovacím jazyce hraje fakt, kdy se typová kontrola provádí. Existují v zásadě dva přístupy: provádět typovou kontrolu hned při překladu výrazu nebo ji provádět až

---

[1] Někdy také hovoříme o záznamu (`record`).

[2] V tom případě hovoříme o tzv. asociovaném poli.

těsně před vykonáním výrazu strojem. Podle toho hovoříme o *statické* a *dynamické typové kontrole* a o *manifestačních* a *implicitních* typech.

3.1.6 Statická typová kontrola a manifestované typy

Statická typová kontrola probíhá při kompilaci a vyžaduje manifestaci typů proměnných.

Provádí-li jazyk typovou kontrolu již při překladu výrazu, potřebuje nutně již v této rané době přesně znát typy všech proměnných. Vyžaduje proto explicitní oznámení (manifestaci) typu proměnné dříve, než je proměnná použita, a to většinou hned v okamžiku vzniku proměnné. Explicitní uvedení proměnné a jejího typu nazýváme *deklarace proměnné*. Proměnná, která by byla použita aniž by byla deklarována, způsobí syntaktickou chybu. Nutnost přesné znalosti typů proměnných v době překladu a z toho vyplývající existence deklarací také efektivně znemožňují, aby se typ proměnné za běhu výpočtu měnil. Typ proměnných je tedy statický - neměnný v průběhu výpočtu. Proto tento druh typové kontroly nazýváme *statický* a typový systém označujeme jako *manifestační*.

Jistým druhem deklarace je i uvedení formálních parametrů procedury. Manifestační typové systémy proto také vyžadují, aby programátor u všech procedur explicitně uvedl typy všech formálních parametrů a typ návratové hodnoty. Je-li v programu uveden kód aktivující libovolnou proceduru, je provedena kontrola, zda typy skutečných parametrů odpovídají typům deklarovaným pro formální parametry. Je také zkontrolováno, zda typ návratové hodnoty procedury odpovídá očekávanému typu.

Statickou typovou kontrolu provádí například jazyky C, C++, C#, Java a další.

3.1.7 Dynamická typová kontrola a implicitní typy

Dynamická typová kontrola probíhá za běhu výpočtu.

Druhým přístupem je provádět typovou kontrolu až v okamžiku vykonání výrazu. Tento přístup je pružnější v tom, že nevyžaduje deklarace typů a navíc umožňuje, aby se typy proměnných v průběhu výkonu programu dynamicky měnily. Takovýto typový systém nazýváme *implicitní* a hovoříme o *dynamické* typové kontrole.

Otázkou je, kde v tomto případě vezme systém za běhu programu informace o typu jednotlivých proměnných. Řešením tohoto problému je, že typ proměnné je přidán ve formě tzv. *typové visačky* (*type tag*) k hodnotě proměnné. Každá proměnná obsahuje tedy v tomto typovém systému nejenom svoji hodnotu, ale i svůj typ, tedy jakýsi návod, jak tuto hodnotu interpretovat. Získá-li (eventuálně změní-li) proměnná svoji hodnotu, získá tím i informaci o svém typu.

Dynamickou typovou kontrolu provádí například jazyk Lisp, PHP, PERL nebo JavaScript.

Diskuse výhod a nevýhod obou přístupů není snadná. Našli bychom příznivce i odpůrce obou druhů typových systémů. Bez nároků na úplnost se pokusíme vyjmenovat alespoň některé výhody statického řešení (resp. nevýhody dynamického řešení):

- vyšší rychlost běhu výpočtu (typová kontrola se nemusí provádět při každém vyhodnocení výrazu, provede se jen jednou při překladu výrazu),
- menší nároky na paměť (proměnné obsahují pouze své hodnoty, nemusí obsahovat visačky s označením typu),
- větší přehlednost a lepší odladitelnost kódu.

Základní nevýhodou statického řešení je pak menší dynamičnost celého systému. Mnohdy je zapotřebí velkého programátorského úsilí pro vyřešení úlohy, která je z lidského hlediska velmi jednoduchá a v dynamickém systému řešitelná na několika řádcích.

Průvodce studiem

Mnoho formálních chyb v rozsáhlejších a méně přehledných projektech vzniká tak, že programátor pošle do procedury chybné parametry nebo je uvede ve špatném pořadí.

Ve větších projektech je proto výhodnější mít k dispozici statickou typovou kontrolu. Pro programátora to znamená podstatně větší úsilí při psaní kódu, protože musí všechny proměnné deklarovat, u všech procedur uvést typy parametrů a v některých případech musí svést boj s typovým systémem a přetypovat výrazy tak, aby dosáhl svého. Odměnou pak je, že velkou část těchto neobyčejně záludných chyb je možno snadno zjistit již při překladu programu.

Dynamická typová kontrola je užitečná u kratších programů, „skriptů", kde programátor nechce trávit čas deklaracemi a hlavní snahou je mít krátký program hotový co nejdříve.

3.1.8 Průběh typové kontroly

Jak tedy probíhá v praxi typová kontrola jednoduchého výrazu? Má-li systém provést typovou kontrolu výrazu `X = A + B`, provede v principu tyto kroky:

- zjistí, zda existuje v typovém systému pravidlo pro přiřazení typu výrazu `A + B`,
- zjistí, zda `X` může pojmout zjištěný typ výrazu `A + B` (při statické typové kontrole),

Máme-li výraz `X = f(A, B)`, kde `f` je procedura, provedení typové kontroly zahrnuje:

- zjištění, zda `f` je typu procedura,
- zjištění, zda `f` akceptuje dva parametry,
- zjištění, zda `A` a `B` jsou výrazy, kterým lze přiřadit typ shodný s typy formálních parametrů procedury `f` (při statické typové kontrole),
- zjištění, zda `X` může pojmout návratovou hodnotu procedury `f` (při statické typové kontrole).

Shrnutí

Typ výrazu označuje, jakých hodnot může daný výraz nabývat a jaké operace s ním mohou být prováděny. Některé typy jsou podporovány přímo procesorem, některé jsou předdefinovány v programovacím jazyku a některé si programátor vytváří sám. Typy dělíme na jednoduché a strukturované.

Pojmy k zapamatování

- Typ výrazu,
- jednoduchý (skalární) typ,
- strukturovaný typ,
- statická a dynamická typová kontrola,
- manifestovaný a implicitní typový systém,
- typová visačka,
- typová kontrola.

Kontrolní otázky

1. *Kam byste zařadili datový typ záznam?*
2. *Jaké typy indexů může mít datový typ pole?*
3. *V jakých krocích probíhá typová kontrola výrazu?*
4. *Jaký druh typové kontroly provádí jazyk C++?*

3.2 Techniky spojené s typovým systémem

Studijní cíle: Po prostudování kapitoly bude studující schopen použít v programu přetížení a přetypování. Pochopí význam koerce a porozumí polymorfním procedurám a generickým datovým typům.

Klíčová slova: Přetížení, přetypování, koerce, polymorfismus, genericita.

Potřebný čas: 2 hodiny.

V této kapitole se budeme věnovat některým drobnějším, ale důležitým efektům, které s sebou přináší používání typů v programovacích jazycích.

3.2.1 Přetížení

Identifikátor procedury nazveme *přetížený* (*overloaded*), pokud reprezentuje více než jednu proceduru. Nečiníme zde přitom rozdílu mezi uživatelskou procedurou a operátorem, tj. můžeme mít přetížený jak identifikátor procedury, tak operátor. O tom, která z procedur označených tímtéž identifikátorem bude použita, je rozhodnuto na základě počtu nebo typu parametrů procedury, eventuálně na základě deklarovaného návratového typu procedury.

Přetížené procedury mají stejné jméno, liší se v seznamu parametrů.

Typickým příkladem přetíženého operátoru může být operátor násobení *. Představme si, že počítač vykonává kód:

```
2 * 6
6.28 * 1.41
```

V obou případech je použit operátor s názvem `*`, v jednom případě však označoval algoritmus celočíselného násobení a v druhém případě algoritmus násobení reálných čísel. Ačkoliv měl operátor v obou případech stejné označení, jistě se pokaždé prováděl zcela jiný kód. Operátor `*` je tedy přetížený. Označuje jednak celočíselné, jednak reálné násobení, a o tom, který kód je vyvolán, rozhodují typy operandů.

Příkladem přetížení procedury může být procedura na výpis hodnot na obrazovku počítače.

```
display("Hello, World")
display(1024)
display(3.1415)
```

Zatímco v prvním případě je vykonána procedura, která znak po znaku zobrazí obsah řetězce, v druhém a třetím případě je obsah jistého úseku paměti nejprve převeden nějakým způsobem na reprezentaci čísla v decimální podobě a teprve potom vytištěn na obrazovku. Procedura `display` je tedy přetížená – označuje celou řadu procedur pro výpis proměnných různých typů na obrazovku. Stejně jako mohou být přetížené systémové procedury, můžeme (pokud to jazyk podporuje) přetížit i procedury uživatelské.

```
procedure InitiateOutput() { … }
procedure InitiateOutput(string type) { … }
procedure InitiateOutput(int deviceID) { … }
```
[1]

Není snad třeba podotýkat, že přetížení na základě typů parametrů se týká pouze jazyků s manifestovanými typy. U těchto jazyků navíc rozeznáváme dva druhy přetížení procedur:

- kontextově závislé přetížení, kdy procedury specifikované přetíženým identifikátorem lze rozlišit pouze pomocí návratové hodnoty procedury,
- kontextově nezávislé přetížení, kdy procedury specifikované přetíženým identifikátorem lze rozlišit pomocí počtu nebo typu parametrů.

Většina jazyků používá kontextově nezávislé přetížení.

Příkladem kontextově závislého přetížení může být trojice operátorů (zde ovšem zapsaných formálně jako procedury):

```
procedure real "/" (real x, real y)
procedure integer "/" (integer m, integer n)
procedure real "/" (integer m, integer n)
```

provádějících reálné dělení čísel, celočíselné dělení čísel a reálné dělení celých čísel. Vzhledem k tomu, že druhou a třetí proceduru lze rozpoznat pouze podle návratových typů, jedná se kontextově závislé přetížení operátoru `/`.

Kontextově závislé přetížení nebývá v programovacích jazycích často implementováno, neboť v některých případech může vést k nejednoznačným výrazům a pro překladač není jednoduché tyto nejednoznačné výrazy spolehlivě detekovat. Například předpokládáme-li, že proměnná `X` má deklarovaný typ `real`, potom přiřazení

```
X = (7 / 2) / (5 / 2)
```

může za použití výše uvedených procedur vést jednak k výsledku `1.4`, a jednak k výsledku `1.5`.

Průvodce studiem

Jeden z mála jazyků, který podporuje jistou formu kontextově závislého přetížení, je jazyk PERL. Tento jazyk se používá zejména pro rychlé skriptování a vytváření drobnějších programů. Aplikujeme-li v jazyku PERL například proceduru `split` *,určenou pro rozdělení řetězce na části oddělené separátorem, vrátí nám tato procedura buď počet nalezených částí nebo pole jednotlivých nalezených částí, podle toho, do jakého datového typu přiřazujeme výsledek.*

Zatímco kontextově závislé přetížení je považováno za matoucí a většina programovacích jazyků je nepodporuje, kontextově nezávislé přetížení patří mezi základní rysy většiny programovacích jazyků a programátorům výrazně ulehčuje práci.

3.2.2 Implicitní parametry procedur

Využití implicitních parametrů procedur zjednodušuje úpravy programu.

Podobný efekt jako přetížení poskytuje i možnost využití implicitních parametrů procedur. v programovacím jazyku, který tuto vlastnost podporuje, můžeme některé parametry procedury označit jako nepovinné a určit jejich implicitní hodnotu. Tato hodnota bude použita v případě, že programátor daný parametr při volání procedury neuvede.

---

[1] Tyto procedury mohou v některém programu sloužit pro inicializaci výstupního zařízení. Výstupem přitom může být například soubor, tiskárna nebo HTTP komunikace. Použijeme-li tuto procedury bez parametrů, inicializuje se nějaký standardní výstup. Použijeme-li parametr typu řetězec, využije se tento řetězec například jako název souboru. Použijeme-li parametr typu číslo, interpretujeme toto číslo jako některý z předdefinovaných typů výstupu.

Implicitní parametry procedury můžeme demonstrovat na následujícím příkladě; první parametr procedury je povinný a zbývající dva jsou pro programátora volitelné.

```
procedure WriteToLog(string message,
                     integer messageType = "WARNING",
                     string filename = "C:/LOGS/INFO.LOG") { … }
```
[1]

> **Průvodce studiem**
>
> *Procedury s implicitními parametry představují cennou vlastnost jazyka hlavně při dodatečných úpravách kódu. Často totiž zjistíme, že danou proceduru potřebujeme dále parametrizovat, až v situaci, kdy je daná procedura již mnohokrát v programu použita. Pokud by jazyk nepodporoval implicitní nepovinné parametry, museli bychom při přidání parametru procedury všechna volání této procedury v celém projektu nalézt a změnit.*

3.2.3 Polymorfismus

Proceduru nazveme *polymorfní*, pokud pracuje stejným způsobem nad různými typy. Jako typický příklad takovéto procedury může sloužit operátor ekvivalence (`=`), testující rovnost dvou výrazů. Je přitom jedno, jakého typu jsou jednotlivé výrazy, operátor `=` provede s libovolnými dvěma proměnnými tutéž akci: porovnání jejich hodnot.

Polymorfní procedura pracuje stejně nad různými datovými typy.

```
1024 = 1024
3.14 = 3.14
"Hello, World" = 1024
```

Dalšími příklady mohou být procedury pro práci se vstupními rozhraními (soubory, streamy). Procedura `open` nám otevře pro čtení vstupní stream, procedura `close` jej uzavře a procedura `eof` testuje, zda otevřený stream je či není vyčerpán. Tyto procedury pracují uniformně na libovolném typu vstupního streamu – ať už jde o stream celých čísel, reálných čísel, řetězců nebo libovolných jiných typů.

Je důležité si uvědomit, že problém řešený polymorfními procedurami není principiálně řešitelný přetížením. Polymorfní procedura pracuje uniformně až s nekonečně mnoha typy. Prakticky je však možné přetížit identifikátor pouze konečného množství procedur. Navíc je zde i zásadní koncepční rozdíl: v případě přetížení máme mnoho procedur označeno tímtéž identifikátorem, přičemž každá procedura vykonává jiný úkol. v případě polymorfní procedury máme jedinou proceduru, která je schopna plnit tentýž úkol pro různé typy.

V jazycích s implicitními typy lze polymorfní procedury vytvářet zcela přirozeným způsobem. v jazycích používajících manifestované typy je třeba pro implementaci polymorfních procedur implementovat speciální mechanismus, rozšiřující typovou kontrolu.

> **Průvodce studiem**
>
> *S pojmem polymorfismus se nejčastěji setkáváme u objektově-orientovaných systémů.*

3.2.4 Genericita

---

[1] Tato procedura zapíše řádek do logovacího souboru. Programátor musí vždy určit text, který má být zapsán, navíc ale může specifikovat typ záznamu (např. "INFO", "WARNING" nebo "ERROR") a umístění logovacího souboru.

Programovací jazyk podporuje genericitu, pokud umožňuje parametrizovat datové typy. Parametrizované datové typy se nazývají generické datové typy. Typickým příkladem zabudovaného generického typu je typ pole, zavedený v kapitole 3.1.4. Deklaraci:

Genericita umožňuje parametrizovat datové typy.

```
array Data real[0 .. 4]
```

lze chápat jako parametrizaci typu `array` typem `real` a intervalem `0 .. 4`. O skutečné genericitě však většinou hovoříme až tehdy, umožňuje-li jazyk parametrizovat programátorem vytvořené datové typy. To může být příklad typu `Pair`, popisující dvojici prvků stejného typu:

```
struct Pair[T] {
   T Item1
   T Item2
}
```

Máme-li takto deklarovaný typ, lze potom vytvořit různé instance daného typu dodáním parametru `T`: například `Pair[Integer]` by představoval dvojici celých čísel a `Pair[Real]` dvojici reálných čísel.

O generických datových typech lze uvažovat pouze v systémech s manifestovanými typy. Systémy s implicitními dynamickými typy jsou generické již svou povahou. Význam genericity spočívá v tom, že umožňuje vyšší formu abstrakce datových struktur. Programátor, píšící například třídící algoritmus, se nemusí zajímat o to, bude-li třídit reálná čísla, celá čísla, řetězce nebo záznamy podle zvoleného klíče. Může se soustředit výhradně na samotný třídící algoritmus. Tentýž kód je potom použitelný na práci s různými datovými typy – podle toho, jaký zvolíme parametr.

V programovacím jazyku C++ definujeme generické datové typy pomocí klíčového slova `template`.

3.2.5 Přetypování a koerce

Přetypování (*type cast*) a *koerce* (*coertion*) tvoří zajímavou kapitolu v používání datových typů.

Přetypováním programátor mění typ proměnné pomocí transformační procedury.

Přetypování je explicitní (tj. programátorem vyžádaná) změna typu výrazu, při současném přibližném zachování jeho hodnoty. Přetypování se provádí pomocí tzv. *transformačních procedur*. Klasickým příkladem je přetypování z typu `integer` na typ `real` a zpět. Je jasné, že změna typu z `integer` na `real` může proběhnou beze ztráty přesnosti, zatímco při změně opačným směrem může dojít k zaokrouhlení hodnoty výrazu směrem k nejbližšímu celému číslu. Uveďme alespoň několik příkladů přetypování za využití transformačních procedur `real` a `integer`. Není výjimkou, že transformační procedury mají stejný název, jako typ, do kterého transformují. Některé jazyky mají pro přetypování speciální syntaxi, odlišnou od syntaxe volání procedury.

```
real(3) = 3.0
integer(3.0) = 3
integer(3.72) = 4
```

Přetypování je pochopitelně možné jen tehdy, jedná-li se o „podobné“ datové typy. Asi těžko bychom v obecném případě smysluplně převáděli datový typ `string` na `integer`. Převod obráceným směrem je však myslitelný.

Je také třeba důsledně rozlišit mezi přetypováním a bitovým maskováním. Zatímco v případě přetypování jde o změnu typu výrazu pomocí jistého algoritmu transformační procedury, u bitového maskování je ta část paměti, která je alokovaná proměnnou, interpretována jako výraz jiného typu. Máme-li například proměnnou `I` typu `integer`, která zabírá v paměti počítače 2 byty, a typ `Pair` definovaný jako

```
struct Pair {
   char LoByte
   char HiByte
}
```

můžeme prostřednictvím maskování přistupovat ke spodnímu i hornímu bytu proměnné `I`:

`(Pair)I.Lo`	spodní byte proměnné `I`
`(Pair)I.Hi`	horní byte proměnné `I`

Koerce je v zásadě identická s přetypováním s tím rozdílem, že probíhá implicitně, tj. aniž by byla vyžádána programátorem. Koerce je do jazyků implementována zejména pro usnadnění práce programátorů, kteří by byli nuceni explicitně přetypovávat i takové výrazy jako

Koerce je nevyžádané přetypování.

```
3.14 + 7
```

na výraz

```
3.14 + Real(7)
```

neboť operátor `+` je přetížen pro reálná čísla a pro celá čísla, nikoli však pro kombinaci celého a reálného čísla. Ve většině programovacích jazyků je proto takováto konverze provedena automaticky, bez zásahu programátora.

Průvodce studiem

Koerce je užitečná vlastnost a v mnoha případech ji programátor ani nevnímá. v některých programovacích jazycích však může být i nebezpečná. Například v jazyku PHP, který bývá často využíván pro tvorbu dynamických WWW stránek, je zabudovaný koerční mechanismus pro změnu čísla na řetězec a řetězce číslic na číslo. Můžeme tedy zapsat výraz jako `"20" * 2`*. Tento rys při programování WWW stránek jistě oceníme, co se ale stane když použijeme proceduru* `substr` *na zjištění části řetězce, začínajícího na dané pozici, a omylem zaměníme pořadí parametrů? Při volání* `substr(6, "soubor.txt")` *programátor očekává jako výsledek řetězec* `"soubor"`*, koerce nám však převede první parametr na řetězec* `"6"` *(protože procedura očekává jako první parametr řetězec) a druhý parametr se snaží interpretovat jako číslo, což se nezdaří a výsledkem je hodnota* `0`*. Procedura se tedy vyvolá jako* `substr("5", 0)` *a výsledkem je prázdný řetězec.*

Shrnutí

Přetížení a implicitní parametry procedur představují cenný nástroj zejména při vývoji větších projektů a při dodatečných úpravách kódu. U přetížení může více procedur sdílet stejný název a systém mezi nimi rozlišuje podle počtu a typů parametrů, u implicitních parametrů lze některé parametry procedury při volání vynechat. Genericita umožňuje parametrizovat datové typy jiným datovým typem. Typ výrazu lze změnit přetypováním, nevyžádané přetypování se nazývá koerce.

Pojmy k zapamatování

- Přetížení, kontextově nezávislé a kontextově závislé přetížení,
- implicitní parametry procedur,
- polymorfismus
- genericita,
- přetypování, koerce.

Kontrolní otázky

1. *Podle čeho systém určí, která přetížená procedura se při volání použije?*
2. *Co je to přetypování výrazu a jak probíhá?*

Úkoly k textu

1. V kapitole 3.2.1 jsme ukázali, že vyhodnocení výrazu může vést při kontextově závislém přetížení ke dvěma různým výsledkům. Ukažte, jakou kombinací přetížených procedur se k těmto výsledkům dostaneme.

4 Závěr

Cílem tohoto textu bylo zprostředkovat čtenářům první vážný a systematický kontakt se světem programů a programování. I když se mlčky předpokládalo, že čtenáři budou mít za sebou již jistou „prenatální“ zkušenost v oboru programování, větší část textu by měla být přístupná i pro ty čtenáře, kteří s programováním nikdy nepřišli do styku. Některé části textu se však nevyhýbaly ani poměrně náročným otázkám, patřícím do diskutovaných oblastí. Tyto možná hůře pochopitelné problémy by měly čtenářům sloužit jako motivace k dalšímu studiu.

5 Seznam literatury

[Sethi89] SETHI, R. *Programming languages.* Reading, Massachusetts: Addison-Wesley, 1989. ISBN 0-201-10365-6.

[Meyer01] MEYER, B. *Object-oriented Sotware Construction. London:* Prentice-Hall Int, 2001. ISBN 0-13-629049-3.

[Dijsktra72] DAHL, O. J., DIJKSTRA, E. W., a HOARE, C. A. R. *Structured Programming*. London: Academic Press, 1972.

6 Seznam obrázků

Obr. 1 Mariner 1 ... 10

Obr. 2 Vztah správnosti a robustnosti ... 12

Obr. 3 Organizace John von Neumannova stroje ... 14

Obr. 4 John von Neumann ... 15

Obr. 5 Segment programu zapsaný ve strojovém kódu, v jazyku symbolických adres a ve vyšším programovacím jazyku ... 15

Obr. 6 Překlad programu ze zdrojového do cílového jazyka ... 16

Obr. 7 Příklad syntaktického diagramu ... 27

Obr. 8 Příklad syntaktického diagramu. ... 27

Obr. 9 Ledovcový efekt ... 32

Obr. 10 Schematické znázornění paměti procesu ... 39

Obr. 11 Aktivační strom ... 41

Obr. 12 Stav zásobníku programu při provádění procedury `Fact` pro výpočet faktoriálu. Předpokládáme, že zásobník roste směrem k hornímu okraji strany. ... 43

Obr. 13 Stav zásobníku programu při provádění procedury `g` pro koncově rekurzivní výpočet faktoriálu. Předpokládáme, že zásobník roste směrem k hornímu okraji strany. ... 44

Obr. 14 Dynamické a statické spoje. ... 46

Obr. 15 Úsek paměti. ... 49

7 Rejstřík

.NET Framework, 18
aktivační strom, 40
aktivační záznamy, 40
aktuální parametry, 35
algoritmus, 33
aritmetické jednotka, 14
assembler, 15
Backus-Naurova forma, 24
BASIC, 19
binární kód, 16
black box, 11
BNF, 24
bool, 57, 58
byte kód, 18
C#, 20
camel casing, 54
cílový program, 16
data, 14
datový segment, 39
deklarace procedury, 34
deklarace proměnné, 60
dynamická typová kontrola, 60
dynamický rozsah platnosti proměnných, 46
dynamický spoj, 46
EBNF, 25
efektivnost, 11
enum, 58
faktoriál, 42
faktory kvality, 11
fáze navíjení, 43
fáze odvíjení, 43
Fibonacciho čísla, 41
formální definice, 28
formálních parametr, 34
Fortran, 16, 20
funkcionální paradigma, 20
genericita, 64
GOTO, 14, 20
hlavička procedury, 34
hlavní efekt procedury, 36
hromada, 39
char, 58
imperativní paradigma, 20
implicitní parametry procedur, 63
implicitní typový systém, 60
index pole, 59
instrukce programu, 14
integer, 57, 58
interpret, 17
iterativní výpočet, 44
Java, 18, 20
jazyk stroje, 15
jazyk symbolických adres, 15
jazyky nízké úrovně, 15
jazyky vyšší úrovně, 15
jednoduché datové typy, 58
jednotka vstupů a výstupů, 14
JIT, 18
jmenná konvence, 54
John von Neumannův stroj, 14
klasické paradigma, 20
koerce, 65
kompatibilita, 11
kompilátor, 17
koncově rekurzivní procedury, 43
kontextově nezávislé přetížení, 63
kontextově závislé přetížení, 63

lexikální rozsah platnosti proměnných, 46
lineárně rekurzivní procedura, 42
Lisp, 20
logické paradigma, 21
L-value, 48
manifestační typový systém, 60
Mariner I, 9
mezilehlý jazyk, 18
mezní podmínka rekurze, 42
modul, 30
MSIL, 18
naivní paradigma, 19
nativní kód, 17
nativní překladač, 17
neterminální symboly, 24
objektově orientované paradigma, 20
ověřování programů, 11
paradigma programování, 12
pole, 59
polymorfismus, 64
posílání zpráv, 20
pretty-printing, 55
procedura, 33
procedurální paradigma, 20
programátorské chyby, 10
programming-in-large, 10
programovací jazyk, 9
programovací paradigma, 9, 12, 19
Prolog, 21
předávání parametrů, 48, 49
předávání parametrů hodnotou, 50
předávání parametrů hodnotou-výsledkem, 51
předávání parametrů jménem, 52
předávání parametrů odkazem, 50
předávání řízení, 38
překladač, 16
překladače, 17
přetížení, 62
přetypování, 65
real, 58
referenční manuál, 27
rekurzivní procedury, 41
rekurzivní volání, 42
robustnost, 9, 11
rozsah platnosti proměnných, 45
rozšiřitelnost, 11
R-value, 48
rychlost, 11
řídící jednotka, 14
sémantika, 23
Scheme, 20
Simula, 20
Smalltalk, 20
správnost, 9, 11
statická typová kontrola, 60
statický spoj, 46
string, 57
strojový kód, 15
stromově rekurzivní procedura, 42
struktura, 58
strukturované datové typy, 58
syntaktické diagramy, 26
syntaxe, 23
tělo procedury, 34
terminální symboly, 24
testování programů, 11
tutoriál, 27
typ, 57
typový systém, 59
UNIX, 16
uživatelský manuál, 27
vedlejší efekt procedury, 36
verifikace programů, 11
virtuální stroj, 18

volání hodnotou, 50
volání hodnotou-výsledkem, 51
volání jménem, 52
volání odkazem, 50
volání procedury, 35
von Neumann, 14
white box, 11
základní případ, 42
zásobník, 39
zdrojový kód, 16
zdrojový program, 16