Mendelova univerzita v Brně
Provozně ekonomická fakulta

Programovací jazyk Lua a jeho možnosti využití v předmětu Programovací techniky

Bakalářská práce

Petr Vévoda

Vedoucí práce: RNDr. Tomáš Hála, Ph. D.

Brno 2015

Chtěl bych poděkovat svému vedoucímu bakalářské práce RNDr. Tomáši Hálovi, Ph. D. za odborné vedení, za pomoc a rady při zpracování této práce.

Čestné prohlášení

Prohlašuji, že jsem práci *Programovací jazyk Lua a jeho možnosti využití v předmětu Programovací techniky* vypracoval samostatně a veškeré použité prameny a informace uvádím v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách, ve znění pozdějších předpisů a v souladu s platnou Směrnicí o zveřejňování vysokoškolských závěrečných prací.
Jsem si vědom, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.
Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

V Brně dne 22. května 2015 .....
podpis

Abstract

Vévoda, Petr. *Programming Language Lua and Possibilities of Its Usage in the Course Programming Techniques.* Bachelor thesis. Brno, 2015.

This bachelor thesis compares programming languages Pascal and Lua. This work starts with a brief history and description of both languages. It focuses on a comparison and description of key topics of programming languages in the scope of the course Programming Techniques, which is taught at the Mendel University in Brno, and covers the following topics: specific data types, creation of abstract data types and modules, object-oriented programming and interaction with operating system. Implementations of examples in both programming languages, which are then reviewed from didactic points of view, have been presented too.
The main contribution consists of a definition of an advantages and disadvantages of using specific language as educational programming language. This text emphasises critical principles on which each of these two specific languages are based on. Conclusions of this work can be used for creating a course syllabus or as a way to make transition from one language to another easier for programmers.

Key words

programming languages, Lua, Pascal, comparison, didactic points of view

Abstrakt

Vévoda, Petr. *Programovací jazyk Lua a jeho možnosti využití v předmětu Programovací techniky.* Bakalářská práce. Brno, 2015.

Bakalářská práce porovnává programovací jazyky Pascal a Lua. Tato práce začíná stručnou historií a popisem obou srovnávaných jazyků. Následuje popis a porovnání klíčových témat programovacích jazyků v rozsahu předmětu Programovací techniky vyučovaného na Mendelově univerzitě v Brně. Mezi tato témata patří konkrétní datové typy, tvorba abstraktních datových typů, tvorba modulů, objektové programování a práce s operačním systémem. Součástí této práce jsou implementace příkladů v obou programovacích jazycích, které jsou následně zhodnoceny z didaktických hledisek. Hlavní přínos této práce spočívá v definici výhod a nevýhod použití konkrétního jazyka ve výuce programování. V textu jsou zdůrazněny kritické principy, na kterých je konkrétní jazyk založen. Práce může být využita při vytváření studijního plánu předmětu nebo jako usnadnění přechodu programátora z jednoho jazyka do druhého.

Klíčová slova

programovací jazyky, Lua, Pascal, srovnání, didaktická hlediska

Obsah

1 Úvod a cíl

Každý programovací jazyk má své odlišnosti a je vhodný k jinému účelu. Výběr vhodného jazyka tvoří důležitou část přípravy programátora při výkonu povolání. Totéž platí, pokud vybíráme, který z nich se nejlépe hodí pro výukové účely – tj. takový, kde zápis kódu by měl být přehledný a snadný, logika algoritmů co nejvíce transparentní. Volbu nám usnadňují různé odborné texty, které se zabývají srovnáním dvou (nebo i více) programovacích jazyků. Mezi tyto texty patří i tato bakalářská práce srovnávající jazyk Pascal s jazykem Lua.

Předmět Programovací techniky, vyučovaný na Provozně ekonomické fakultě Mendelovy univerzity v Brně, navazuje na problematiku řešenou v předmětu Algoritmizace, kde jsou probírány základní programátorské obraty v jazyce Pascal. V předmětu Programovací techniky je tato probraná problematika rozšířena. Absolvent předmětu by měl být schopen profesionálně zhodnotit zadanou úlohu, znát běžné programovací techniky při řešení problémů, vybrat a implementovat optimální řešení úlohy, tvořit bezpečné a snadno udržovatelné programy. Student také získá schopnost vytvářet kolektivní díla po částech s definovým rozhraním. Současným vyučovacím jazykem je Pascal.

V jednotlivých kapitolách budou porovnána klíčová témata programovacích jazyků – konkrétní datové typy, dále tvorba abstraktních datových typů, modulů, objektů a práce s operačním systémem. Kapitoly přibližně pokryjí látku, vyučovanou v předmětu Programovací techniky (témata překračující rámec předmětu nebudou zkoumána). Témata jako rekurze nebo řadící algoritmy budou vynechána z důvodu shodného přístupu obou jazyků. Přístupy jazyků ke zkoumaným tématům budou podrobně popsané, doplněné implementací úloh řešených v předmětu Programovací techniky (nebo pouze ilustračními příklady). Tato řešení poslouží pro porovnání a zhodnocení z několika didaktických hledisek. Mezi tato hlediska patří: syntaxe, sémantika, časová složitost. Dále délka, náročnost a pochopitelnost zápisu algoritmu. Kromě těchto hledisek budou v textu zdůrazňovány a vysvětlovány části programového kódu, kde nepochopení jejich logiky může být příčinou navržení neefektivního nebo dokonce chybného algoritmu.

Cílem této práce je srovnat oba zkoumané jazyky (Pascal a Lua) a definovat výhody a nevýhody zvolení tohoto jazyka jako výukového v předmětu Programovací techniky. Práce by tedy měla poskytovat doporučení, na základě kterých se může vyučující rozhodnout o změně výukového programovacího jazyka z Pascalu na Lua, o zavedení jazyka Lua do výuky nebo o doplnění výuky stávajícího jazyka příklady napsanými v jazyce Lua.

2 Zkoumané jazyky

2.1 Programovací jazyk Pascal

Ve 20. století nastal velký rozvoj techniky v oblasti počítačů. Vzhledem k jejich rozšiřování mimo skupiny odborníků, bylo zapotřebí vyvinout programovací jazyk vhodný pro výuku programování. Tehdejší jazyky (Fortran, COBOL a další) nenabízely dostatečně průzračné konstrukce, které by umožnily jednoduchou prezentaci potřebných principů.

Začátkem 70. let byl navržen programovací jazyk Pascal. Autorem jazyka je profesor Niklaus Wirth z Vysoké školy technické v Curychu, který se snažil vytvořit jazyk vhodný k výuce programování a byl snadno implementovaný na většině počítačů. Počátkem 80. let se stal jazyk Pascal výukovým jazykem na vysokých školách a používal se jako systémový programovací jazyk pro IBM PC a Apple. Na rozšíření jazyka Pascal pro systém MS DOS měla hlavní zásluhu firma Borland a její překladač Borland Turbo Pascal. (Základy programování v jazyce Pascal, 2004)

Překladač Turbo Pascal byl navržen Andersem Hejsbergem. Překladač a vývojového prostředí spatřili poprvé světlo světa v roce 1983. Software mohl pracovat pod operačními systémy CP/M, CP/M-86 a MS-DOS. Jednalo se o vylepšenou verzi standardního Pascalu, který prošel několika úpravami, které pomohly dosáhnout na tehdejší dobu velké rychlosti kompilace oproti konkurenčním jazykům. Postupně se vyvíjely nové verze (od 1.0 do 7.0), kdy ve verzi 4.0 se poprvé objevily moduly a možnost kompilovat jednotlivé moduly zvlášť, od verze 5.5 obsahuje nástroje pro objektové programování. (Object Pascal History, 2014)

Free Pascal se poprvé objevil v devadesátých letech minulého století, v době, kdy se společnost Borland rozhodla, že se vývoj Borland Pascal zastaví na verzi 7, aby byl nahrazen jiným produktem (později známým jako Delphi). Za vývojem Free Pascalu stojí Florian Paul Klämpfl. Původně byl překladač 16-ti bitový spustitelný soubor v MS-DOS kompilovaný pomocí Turbo Pascalu. Po přibližně dvou letech už byl schopný se sám zkompilovat a byl publikován na internetu, kde se stal velice populární jak v podnicích, tak ve školách, protože se na rozdíl od Turbo Pascalu jednalo o freeware. Krátce nato provedl Michael van Canneyt úspěšný port Free Pascalu na operační systém Linux. K Free Pascalu existuje řada vývojových prostředí, mezi nejznámější patří Lazarus IDE. (Free Pascal History, 2015)

Pascal je imperativní, strukturovaný programovací jazyk, primárně uzpůsobený pro výukové účely – zápis kódu je snadný, přehledný a rychlý. Používá statickou typovou kontrolu, jedná se o kompilovaný jazyk s manuální správou paměti.

2.2 Programovací jazyk Lua

V devadesátých letech minulého století potřebovala brazilská ropná společnost PETROBRAS vyvinout programovací jazyk pro zpracování datových souborů určených pro velmi časté simulace. Vzhledem k faktu, že měla firma velmi různorodé hardwarové vybavení, byl kladen požadavek nejenom na úspornost a efektivitu, ale i na přenositelnost jazyka. Na vývoji pracovali Roberto Ierusalimschy, Luiz Henrique de Figueiredo a Waldemar Celes, studenti Papežské univerzity v Rio de Janeiro v Brazílii, kteří v roce 1993 vytvořili programovací jazyk Lua (Lua 5.3 Reference Manual, 2015).

Interpret jazyka Lua je napsaný v jazyce C, který je podporovaný širokým spektrem různých zařízení – to umožňuje jazyku Lua běžet jak na různých malých zařízeních, tak i velkých síťových serverech. Vzhledem k relativně krátké době existence jsou stále vyvíjené nové verze (od verze 1.0 po současnou verzi 5.3). Od verze 2.1 je dostupný pod softwarovou licencí freeware. Oproti Pascalu se jedná o open–source software.

Lua je odlehčený, procedurální programovací jazyk určený pro integraci se softwarem napsaném v C/C++ a v dalších běžných programovacích jazycích. Jedná se o dynamicky typovaný, interpretovaný programovací jazyk s automatickou správou paměti využívající garbage collector, což z něj dělá ideální nástroj pro snadnou konfiguraci, skriptování a rychlý vývoj aplikací. Mezi základní znaky jazyka patří: rozšiřitelnost, jednoduchost, efektivita a přenositelnost (Lua 5.3 Reference Manual, 2015).

Klíčovým prvkem jazyka Lua je využití datového typu tabulka a příslušných metatabulek a jejich metametod. Pomocí tabulek se vytváří všechny strukturované datové typy – záznam, pole, množina a další. Metametody umožňují předefinovat chování samotného jazyka – můžeme změnit reakci jazyka například na přiřazení do proměnné, volání funkce nebo matematické operátory. Zároveň jsou užitečné v případech, kdy chceme ošetřit potenciální nežádoucí stav nebo požadujeme nestandardní funkcionalitu (např. sčítání dvou záznamů), bez existence metametod by takový program skončil chybou při interpretaci kódu programu.

3 Konkrétní datové typy

3.1 Úvod

Datový typ je určen množinou přípustných hodnot proměnné (nebo konstanty) a operacemi, které je možné s hodnotami tohoto typu provádět.

Programovací jazyk Pascal patří mezi staticky typované jazyky, což znamená, že všechny proměnné musí mít deklarovaný datový typ před provedením kompilace zdrojového kódu. V průběhu kompilace se pak provádí typová kontrola, která vyhodnocuje korektnost jednotlivých výrazů a odhaluje případné nesrovnalosti (např. nekompatibilní datové typy na levé a pravé straně přiřazovacího příkazu, použití operátoru, který není pro datový typ definován, …).

Naproti tomu programovací jazyk Lua se řadí mezi dynamicky typované jazyky, tj. typy nejsou přiřazené proměnným, ale jejich hodnotám. To mimo jiné také znamená, že se při vytváření proměnných neuvádí jejich typ, překladač použitý datový typ rozpozná přímo z toho, jakým způsobem je hodnota proměnné zapsána (Tišnovský, 2009). Typ hodnoty můžeme zjistit pomocí funkce **type**. Do proměnné tedy můžeme opakovaně přiřazovat libovolné hodnoty, přičemž datové typy těchto hodnot mohou být odlišné:

```
--Vytvoření nové proměnné (prvotní přiřazení hodnoty)
A = 5
print(type(A))       --vypíše se "number"

--Přiřazení hodnoty jiného typu
A = "text"
print(type(A))       --vypíše se "string"
```

Z výše uvedeného příkladu vyplývá určitý problém – programátor (např. student) si musí plně uvědomovat, které proměnné už používá a k jakému účelu. V proměnné **A** si uchovával číslo, v momentě, kdy do ní přiřadil řetězec, ztratil hodnotu původního čísla a změnil účel této proměnné. Z pohledu jazyka Lua se jedná o validní postup, nebude ohlášena žádná chyba.

Výhodou dynamicky typovaného jazyka je jeho pružnost – odpadá nutnost kompletně předvídat typy dat, se kterými bude program pracovat. Stejně tak pro ně nemusíme předem deklarovat proměnné. Mezi nevýhodu tohoto přístupu patří vyšší riziko, že hodnota nabude neočekávaného datového typu, který pak při dalším zpra-

cování způsobí neočekávané výsledky programu nebo jeho pád. Takové chyby, které nastávají až za běhu programu, se mohou projevit daleko od místa jejich vzniku a velice obtížně se hledají.

3.2 Jednoduché datové typy

Základním znakem proměnných jednoduchých datových typů je absence vnitřní struktury. Tyto proměnné obsahují jedinou hodnotu – například znak, číslo, logickou hodnotu apod. Hodnoty typů lze mezi sebou pomocí relačních operátorů porovnávat (tj. na množinách jejich hodnot existuje relace uspořádání).

3.2.1 Číslo

V jazyce Pascal se nachází spousta číselných datových typů, které se liší svou velikostí, rozsahem hodnot a podporou čísel se znaménky. Běžné implementace používají následující datové typy:

Tabulka 3.1 Celočíselné datové typy v jazyce Pascal (Rybička a Čačková, 2012)

Datový typ	Velikost	Rozsah
byte	1 B	0 až 255
word	2 B	0 až 65535
longword	4 B	0 až $2^{32} - 1$
cardinal	4 B	jako longword
qword	8 B	0 až $2^{64} - 1$
shortint	1 B	-128 až 127
smallint	2 B	-2^{15} až $2^{15} - 1$
integer	2–4 B	smallint nebo longint
longint	4 B	-2^{31} až $2^{31} - 1$
int64	8 B	-2^{63} až $2^{63} - 1$

Tabulka 3.2 Reálné datové typy v jazyce Pascal (Rybička a Čačková, 2012)

Datový typ	Velikost	Rozsah
real	4 nebo 8 B	jako single nebo double
single	4 B	$1,5 * 10^{-39}$ až $3,4 * 10^{38}$
double	8 B	$5,5 * 10^{-324}$ až $1,7 * 10^{308}$
extended	10 B	$1,9 * 10^{-4932}$ až $1,1 * 10^{4932}$
comp	8 B	2^{-63} až $2^{63} - 1$
currency	8 B	2^{-63} až $2^{63} - 1$

V jazyce Lua existuje pouze jeden číselný datový typ **number**. Tento typ je vnitřně reprezentován dvěma způsoby – jako **integer** nebo jako **float**. Jazyk Lua má pevně daná pravidla, kdy bude která reprezentace využita, ale zároveň mezi nimi automaticky provádí typovou konverzi, když je to potřeba (viz podkapitola 3.4 na straně 33). Programátor se může ve většině případů rozhodnout rozdíly mezi těmito typy ignorovat, ale má i možnost převzít úplnou kontrolu nad způsobem, jakým budou čísla reprezentována. Standardní Lua používá 64 bitů (tj. 8 B) jak pro celá čísla, tak pro čísla s plovoucí desetinnou čárkou (Lua 5.3 Reference Manual, 2015). Tyto reprezentace odpovídají v Pascalu datovým typům **int64** a **double**.

Na množině celých čísel jsou v obou jazycích definované běžné aritmetické operátory pro sčítání, odečítání, násobení, dělení, umocnění, unární mínus, celočíselné dělení a zbytek po celočíselném dělení. Liší se však zápis posledních dvou operátorů, jak ukazuje následující tabulka:

Tabulka 3.3 Operátory celočíselného dělení

Operace	Operátor v Pascalu	Operátor v Lua
Celočíselné dělení	div	//
Zbytek po celočíselném dělení	mod	%

Kromě způsobu zápisu se liší i používaná matematická definice pro tyto dvě operace v případě, že jsou odlišná znaménka operandů, v ostatních případech se tato odlišnost neprojeví. Rozdíl je vidět na následujícím příkladu:

```
//v jazyce Pascal
A := -10 div 3;              //proměnná A je rovna -3
B := -10 mod 3;              //proměnná B je rovna -1
```

```
--v jazyce Lua
A = -10 // 3          --proměnná A je rovna -4
B = -10 \% 3          --proměnná B je rovna 2
```

Jazyk Lua totiž na rozdíl od jazyka Pascal, provede neceločíselné dělení a následně zaokrouhlí podíl k nejbližšímu menšímu číslu. U obou variant však platí:

$$\text{podíl} * \text{dělitel} + \text{zbytek} = \text{dělenec}$$

3.2.2 Boolean

Typ boolean může nabývat pouze dvou hodnot – **false** a **true** v obou jazycích. Jazyk Lua však umožňuje, aby se v podmínce vyskytoval libovolný výraz (v Pascalu nemůže podmínka obsahovat jiný než booleovský výraz). V jazyce Lua jsou za nepravdu považovány hodnoty **false** a **nil**, všechno ostatní je považováno za pravdivé. Velikost proměnné tohoto typu je v obou jazycích jeden bajt.

3.2.3 Logické operace

Jazyk Lua obsahuje pět logických operátorů: =, ~= (nonekvivalence), **and**, **or**, **not**. Oba jazyky mají implementováno tzv. zkrácené vyhodnocování – druhý operand operace **and** vyhodnocen pouze v případě, kdy první operand má hodnotu **True**, v opačném případě druhý operand není vyhodnocován výsledek operace má hodnotu **False**. U operace **or** je druhý operand operace vyhodnocen pouze, když první nabývá hodnoty **False**, jinak není vyhodnocován a výsledkem operace je hodnota **True**. (Polách, 2006)

Na rozdíl od jazyka Pascal není vždy výsledkem logických operací **and** a **or** hodnota typu **boolean**, jak můžeme vidět na následujících příkladech:

```
--operace and
nil and true          --vrátí nil
false and "text1"     --vrátí false
true and 5            --vrátí 5
```

Operace **and** v jazyce Lua vrací první argument, pokud byl nepravdivý (tj. nabýval hodnoty **false** nebo **nil**). V opačném případě vrací druhý argument.

```
--operace or
nil or true          --vrátí nil
false or 5           --vrátí 5
"text1" or "text2"   --vrátí text1
```

Operace **or** v jazyce Lua vrací první argument, pokud byl pravdivý (tj. nenabýval hodnoty **false** nebo **nil**). V opačném případě vrací druhý argument.

3.2.4 Bitové operace

Standardně rozlišují programovací jazyky datový blok o nejmenší velikosti jeden bajt. V některých případech však může být užitečné pracovat se samotnými bity. Například při požadavku uložení informací na minimálním prostoru se často využívá možnosti zakódovat tyto informace do jednotlivých bitů. Většina jazyků pro tyto příležitosti nabízí programátorovi operátory pro provádění bitových operací. V obou jazycích tyto operace fungují stejně, liší se pouze ve značení jednotlivých operátorů, viz následující tabulka:

Tabulka 3.4 Logické operace

Operace	Operátor v Pascalu	Operátor v Lua
Logický součin	and	&
Logický součet	or	|
Výlučný součet	xor	~
Posuv vlevo	shl	<<
Posuv vpravo	shr	>>
Negace	not	~

3.2.5 Znak

Pascal reprezentuje znaky datovým typem **char** – množina hodnot tohoto typu nabývá 256 hodnot – to znamená, že může obsahovat jakýkoliv znak ASCII tabulky (zobrazitelné i řídící znaky). Velikost proměnné tohoto typu je jeden bajt. Free Pascal navíc podporuje datový typ **WideChar**, velikost proměnné je dva bajty – obsahuje

jeden znak v kódování UTF-16 (Free Pascal, 1993–2010).

V jazyce Lua není pro znaky implementován žádný datový typ. Jako náhrada se používá strukturovaný datový typ **string** – to s sebou přináší nárůst režie.

3.2.6 Nil

V jazyce Lua se vyskytuje datový typ **Nil**. Nabývá pouze jediné hodnoty – **nil**. Hlavní vlastností tohoto typu je odlišnost od všech ostatních datových typů, jazyk Lua hodnotu **nil** používá pro vyjádření nepřítomnosti hodnoty. Každá globální proměnná nabývá před prvním přiřazením této hodnoty. Navíc je možné přiřazením hodnoty **nil** smazat libovolnou proměnnou. (Lua 5.3 Reference Manual, 2015).

V Pascalu existuje **nil** pouze jako hodnota datového typu **pointer** – využívá se k inicializaci ukazatelů. Ukazatel s hodnotou **nil** neodkazuje na žádná data. Turbo Pascal (na rozdíl od Free Pascalu) umožňuje takový ukazatel dereferencovat.

3.2.7 Ukazatel

Kromě statických typů existují také typy dynamické, které mají oproti prvním zmíněným tyto výhody:

- Proměnná vzniká po zavolání speciálního příkazu (kdykoliv za běhu programu), během kterého dojde k obsazení místa v paměti.
- Místo v paměti, na kterou proměnná odkazuje, lze pomocí speciálního příkazu uvolnit.
- Proměnné může být přidělena různě velká část paměti.
- Umožňují vytvářet vlastní datové typy.

Základním prostředkem pro vytváření dynamických datových struktur je datový typ ukazatel. Neobsahuje přímo data, ale pouhou adresu na místo v paměti, kde se nacházejí (tzv. referenci). Přístupu k datům, na která ukazuje, se říká dereference. V obou jazycích je velikost proměnné typu ukazatel 4 B nebo 8 B (záleží na architektuře počítače).

V Pascalu existují dva typy ukazatelů – typové a netypové. Druhé zmíněné se používají v Programovacích technikách pro uložení řetězců v konvenci jazyka Pascal na minimálním prostoru a v programových modulech pro zajištění univerzálnosti (tj. aby modul nebyl pevně vázán na určitý typ dat, se kterými umí pracovat). Veškeré vyhrazování a uvolňování paměti, stejně jako operace s daty musí programátor explicitně zapsat, což mu umožňuje důkladně se seznámit s principem fungování dyna-

mických datových struktur. Nicméně se tím také otevírají možnosti pro tvorbu chyb (ztráta ukazatele, porovnávání adres ukazatelů namísto dat, …) a zmatečných zápisů programu.

V jazyce Lua jsou všechny datové typy kromě typu **boolean** a **number** jsou ukazatele do paměti. Tyto ukazatele jazyk automaticky dereferencuje, kdykoli je to potřeba. V jazyce Lua neexistují žádné operátory pro referenci nebo dereferenci ukazatelů (v Pascalu symboly **@** a ^).

3.3 Strukturované datové typy

Proměnné strukturovaných typů mohou obsahovat více hodnot. Na rozdíl od jednoduchých typů mají svou vnitřní strukturu. U neskalárních nejsou hodnoty uspořádány – nelze tedy použít standardní relační operátory. Na rozdíl od jazyka Pascal má Lua pouze jeden strukturovaný typ **table**, pomocí kterého se tvoří všechny ostatní struktury.

3.3.1 Pole

V předmětu Programovací techniky se pro reprezentaci n-rozměrného pole používá jeho statická varianta. V jazyce Pascal se vyznačuje těmito vlastnostmi:

- jedná se o homogenní strukturu – všechny jeho prvky mají stejný datový typ,
- je statické – počet prvků odpovídá deklaraci, za běhu velikost nelze změnit,
- je indexované – pouze hodnoty bázového ordinálního typu mohou být indexy,
- je jednorozměrné – ale prvkem může být další pole.

Programovací jazyk Lua využívá k tomuto účelu dynamickou datovou strukturu **table**. Mezi její nesporné výhody patří:

- prvky pole mohou nabývat libovolného datového typu (struktura je nehomogenní),
- je dynamická – počet prvků lze za běhu programu měnit,
- její indexy nemusejí být pouze ordinálního typu – indexem můžou například i řetězce, pak se jedná o tzv. asociativní pole (v Pascalu není implementováno).

Uvažujme následující příklad: Na vstupu je zadána hodnota, určující počet čísel na standardním vstupu. Načtěte tato čísla do pole.

```
//v jazyce Pascal
program pole01;
var pole: array[1..3] of byte;
    // n je počet čísel na standardním vstupu
    n, i: byte;
begin
  write('Zadejte počet čísel: ');
  readln(n);
  for i := 1 to n do
    readln(pole[i]);
end.
```

Co se stane s proměnnou **n** nebo prvkem **pole[i]**, když uživatel zadá číslo větší než 255? Při výchozím nastavení překladače dojde k přetečení a do proměnné se uloží 8 nejméně významných bitů zadaného čísla. Další problém může nastat v případě, že hodnota proměnné **n** bude větší než počet prvků pole – potom v cyklu dojde k přístupu do paměti, která pro nás nebyla alokována, výsledkem bude chyba za běhu programu. Tento nežádoucí stav může programátor ošetřit pomocí funkcí **low** a **high**.

```
--v jazyce Lua:
pole = {}
print("Zadejte pocet cisel: ")
n = io.read("*n")
for i = 1, n do
  pole[#pole + 1] = io.read("*n")
end
```

V jazyce Lua nehrozí přetečení dokud je hodnota **n** v rozsahu od -2^{63} do $2^{63}-1$. Ve **for** cyklu za běhu přidáváme do pole podle potřeby další prvky. Tento algoritmus je v porovnání s předchozím univerzálnější – nemusíme odhadovat velikost načítaných dat, nejsme limitovaní deklarovanou velikostí pole a máme k dispozici mnohem větší rozsah hodnot, které jsme schopni ukládat.

Free Pascal zná od verze 1.1 také dynamické pole. Jedná se o pole, kde je při deklaraci vynechaný rozsah indexů – je zadáván (a případně změněn) až za běhu programu pomocí funkce **SetLength**, která na haldě alokuje potřebnou paměť. Dynamické pole vždy začíná indexovat prvky od nuly. Největší index takového pole zjišťujeme pomocí funkce **High**. V případě, že chceme přestat s tímto polem pracovat, stačí zavolat funkci **SetLength** s druhým parametrem 0 a dojde k uvolnění paměti. Stále se však nejedná o heterogenní strukturu (kterou má jazyk Lua). Tento typ také umožňuje předávat procedurám a funkcím jako parametr tzv. otevřená pole – při deklaraci parametru jako dynamické pole mohou tyto operace pracovat s polem libovolného rozsahu, které je stejného bázového typu:

```
program pole01;

procedure soucet(pole:array of byte);
var i, soucet:integer;
begin
  soucet := 0;
  for i:=0 to high(pole) do
    soucet := soucet + pole[i];
  writeln(soucet);
end;

    //Rozsah je vynechán.
var pole: array of byte;
    // n je počet čísel na standardním vstupu.
    n, i : byte;

begin
  write('Zadejte pocet cisel: ');
  readln(n);
  setlength(pole, n);
  for i := 0 to high(pole) do
    readln(pole[i]);
  //Vypíše součet první poloviny prvků pole.
  soucet(pole[0..high(pole) div 2]);
  //Uvolnění paměti.
  setlength(pole, 0);
end.
```

3.3.2 Asociativní pole

Asociativní pole nebo také hashovací tabulka je datová struktura, která slouží k optimalizaci ukládání a vyhledávání dat pomocí zvolených klíčů. Klíčem se rozumí část ukládaných údajů, která slouží k nalezení indexu, na který bude údaj uložen. Index je spočítán z klíče pomocí tzv. hashovací funkce.

Hashovací funkce musí splňovat tyto vlastnosti:

- pro každá vstupní data vrací hodnotu indexu,
- pro stejná vstupní data vrací stejný index,
- pravděpodobnost všech indexů je stejná,
- pro různá vstupní data může vrátit stejný index – pak vzniká tzv. kolize,

– výpočet probíhá velmi rychle. (Vyhledávání …, 2003–2014)

Programovací jazyk Lua využívá k reprezentaci všech pokročilejších datových struktur (záznam, pole, seznam, …) datový typ **tabulka**. Jedná se o heterogenní asociativní pole, ve kterém jsou uloženy dvojice klíč–hodnota. Na rozdíl od datového typu pole v programovacím jazyce Pascal může být klíčem, stejně jako hodnotou jakýkoliv datový typ.

V případě jazyka Lua můžeme k indexaci použít libovolný datový typ, tudíž můžeme použít přímou indexaci. V jazyce Pascal můžeme simulovat asociativní pole tak, že si uložíme jednotlivé klíče do proměnné typu string a na základě jejich pozice je převedeme na celočíselné indexy proměnné typu pole. Tento způsob implementace však předpokládá konstantní délku klíčů a jejich omezený počet – v praxi tento případ nastává výjimečně. Proto je za následujícím příkladem uvedena další možnost simulace asociativního pole, kde klíče mohou být různé délky a jejich počet není omezený.

Příklad: Chceme ukládat kurzy různých měn vůči české koruně.

```
--v jazyce Lua
tab = {}              --vytvoření prázdné tabulky
print('Zadejte nazev meny a kurz: ')
klic = io.read('*line')
tab[klic] = io.read('*n')   --vytvoření prvku a vložení hodnoty
```

Rozbor pro Pascal: pozice jednotlivých měn v textové proměnné jsou rovny prvkům aritmetické posloupnosti, kde platí:

a_n	=	$a_1 + (n-1) * d$, kde
a_n	...	n-tý prvek posloupnosti
a_1	...	první prvek posloupnosti
n	...	pořadí hledaného prvku posloupnosti
d	...	rozdíl mezi dvěma sousedními prvky posloupnosti

V našem případě, kdy měny ukládáme jako třípísmenné zkratky oddělené čárkou je vzorec následující:

$$\text{poziceHledanéMěny} = 1 + (\text{hledanýIndex} - 1) * 4$$

```
program kurzymen;
const maxindex = 3;
      meny = 'gbp,usd,frf';
var mena: string[3];
    kurz: integer;
    kurzy: array[1..maxindex] of integer;
```

```
begin
  read(mena, kurz);
  kurzy[(pos(mena, meny) - 1) div 4 + 1] := kurz;
end.
```

Následující implementace simuluje v jazyce Pascal pole indexovatelné pomocí řetězců různé délky:

```
program asocpole0;
type
  THodnota = byte;
  UkPrvek = ^Prvek;
  Prvek = record
    klic: pointer;
    hodnota: THodnota;
    levy, pravy: UkPrvek;
  end;

  AsocPole = UkPrvek;

procedure init (var p: AsocPole);
begin
  p := nil;
end;

procedure UlozPrvek (var p: AsocPole; klic: string;
hodnota: THodnota);
begin
  if p = nil then
  begin
    new(p);
    GetMem(p^.klic, length(klic) + 1);
    string(p^.klic^) := klic;
    p^.hodnota := hodnota;
  end
  {v případě, že již existuje prvek se zadaným klíčem, bude
pouze aktualizována jeho hodnota}
  else if klic = string(p^.klic^) then p^.hodnota := hodnota
    else if klic < string(p^.klic^) then
           UlozPrvek(p^.levy, klic, hodnota)
      else UlozPrvek(p^.pravy, klic, hodnota);
end;
```

```
function ZiskejPrvek (p: AsocPole; klic: string): UkPrvek;
begin
  while (p <> nil) AND (string(p^.klic^) <> klic) do
  begin
    if klic < string(p^.klic^) then p := p^.levy
       else p := p^.pravy;
  end;
  ZiskejPrvek := p;
end;

procedure Vypis (p: AsocPole);
begin
  if p <> nil then
  begin
    vypis(p^.levy);
    writeln(string(p^.klic^), '=', p^.hodnota);
    vypis(p^.pravy);
  end;
end;
```

Jak demonstruje tento příklad, dosáhli jsme větší obecnosti za cenu zhoršení časové složitosti při vkládání a přístupu k prvkům pole (z konstantní na lineárně logaritmickou). Vlivem použití binárního stromu došlo k zápisu delšího kódu a zvýšení jeho zápisové náročnosti. Stále však nedosahujeme obecnosti datového typu table v jazyce Lua – jak klíče, tak jejich hodnoty nejsou heterogenní.

V případě výskytu duplicitních klíčů můžeme v jazyce Lua hashovací tabulku reprezentovat jako **tabulku**, kde jednotlivé prvky budou také **tabulky** obsahující synonyma. Příklad:

```
maxindex = 20
hashtab = {}
for i = 1, maxindex do
  hashtab[i] = {}
end
...
function vloz (hashtab, data)
  table.insert(hashtab[hash(data)], hodnota)
end
```

Nejběžnější variantou hashovací tabulky v jazyce Pascal bývá tabulka s explicitně zřetězenými synonymy, což znamená, že pole obsahuje pouze ukazatele na jednotlivé seznamy synonym a každý prvek seznamu ukazuje na svého následníka. Příklad:

```
program hashtabulka;
const maxindex = 20;
type tdata = integer;
     ukprvek = ^prvek;
     prvek = record
               data: tdata;
               dalsi: ukprvek;
             end;
var hashtab: array[1..maxindex] of ukprvek;
...
procedure vloz (var hashtabulka: hashtab; data: tdata);
var pom, i: ukprvek;
begin
  i := hash(data);
  new(pom);
  pom^.data := data;
  pom^.dalsi := hashtabulka^[i];
  hashtabulka^[i] := pom;
end;
```

Programovací jazyk Lua má velikou výhodu v případě hashovacích tabulek s unikátními klíči, kdy na rozdíl od Pascalu umožňuje dynamicky přidávat nebo odebírat prvky tabulky a nevyžaduje od uživatele explicitně definovat funkci pro přepočet klíčů na indexy.

U hashovacích tabulek s duplicitními klíči již nelze ani v jazyce Lua použít přímé indexace, nicméně seznamy synonym lze procházet pomocí cyklu **for**, není potřeba definovat vlastní funkci pro průchod seznamem jako v jazyce Pascal.

3.3.3 Tabulka

Jak již bylo psáno v kapitole Asociativní pole, **tabulka** je v jazyce Lua heterogenní asociativní pole, ve kterém jsou uloženy dvojice klíč–hodnota. Tato struktura reprezentuje všechny datové struktury (záznamy, pole, seznamy, …). **Tabulka** je dynamicky alokovaný objekt, program manipuluje pouze s ukazateli na tyto objekty. Jazyk Lua nikdy nevytváří skryté ani jiné kopie, o kterých by programátor neměl ponětí. Pro vytvoření se zavolá konstruktor, v nejjednodušší formě zapsaný jako **{}**.

Indexem do tabulky může být libovolný datový typ (není omezená délka přípustného indexu ani jejich počet) – klíčem do této struktury může být na rozdíl od Pascalu další strukturovaný typ (např. tabulka), funkce nebo řetězec. Jedinou hodnotou, které

nemůže klíč ani hodnota prvku tabulky nabývat, je **nil**. Při řešení reálných problémů vyvstává požadavek na znalost délky pole – Lua má pro tento účel operátor #, který tuto hodnotu zjistí, nicméně tabulka se chová jako pole pouze v případě, že jsou její indexy číselné. Další velice důležitou podmínkou je, aby klíče nabývaly všech možných hodnot v intervalu od hodnoty 1 do hodnoty největšího indexu – musí tvořit nepřerušenou sekvenci. Pole se v jazyce Lua standardně číslují od 1, operátor # očekává tento počátek číslování a prochází postupně všechny indexy do doby, než narazí na položku, která má hodnotu **nil**. V případě, že dojde k porušení sekvence, přestává být struktura indexovaným polem a stává se polem asociativním. Neznalost tohoto principu může způsobit, že začínající programátor (student) bude zmatený chováním jeho programu a stráví velké množství času hledáním příčiny takového chování.

Programovací jazyk Lua, ačkoliv minimalistický, nabízí nepřeberné množství možností, jak rozšířit jeho funkcionalitu. Můžeme vytvářet vlastní abstraktní datové typy, přidat objektové programování, předefinovat stávající nebo definovat nové operátory a podobně. Pro tyto případy obsahuje Lua velice silný nástroj – metatabulky. Každé tabulce můžeme přiřadit právě jednu metatabulku, která nám umožňuje změnit chování hodnoty v případě, že narazíme na nedefinovanou operaci (např. vynásobení tabulky číselným koeficientem). Při každém pokusu o tuto operaci Lua zkontroluje, zda má tabulka přiřazenou metatabulku, v ní se pokusí vyhledat metametodu pod indexem **__mul**. Při pozitivním nálezu provede funkci, která je v ní definována.

Příklad: Předpokládejme, že máme uchované záznamy o mzdách. U každé záznamu mzdy evidujeme jméno a příjmení příjemce a výši mzdy. Dojde například k neočekávanému výpadku zakázek a potřebujeme snížit mzdy.

```
//v jazyce Pascal
program mzdy01;
const maxindex = 100;
type tmzda = record
       jmeno:string[50];
       prijmeni:string[50];
       castka:word;
     end;
     tmzdy = array[1..maxindex] of tmzda;

{Pascal nenabízí možnost předefinovat operátory, musíme použít
proceduru}
procedure snizitmzdy (var mzdy: tmzdy; koef: real);
var i: 1..maxindex;
begin
  for i := 1 to maxindex do
    mzdy[i].castka := round(mzdy[i].castka *  koef);
end;
```

```
var mzdy: tmzdy;
    koef: real;
begin
  //načtení dat
  nacti(mzdy);

  //rozhodli jsme se snížit mzdy o 5 procent
  koef := 0.95;
  snizitmzdy(mzdy, koef);
end.
```

```
--v jazyce Lua
mt_mzdy = {
  --metametoda, která definuje chování operátoru *
  __mul = function(mzdy, koef)
      for i in ipairs(mzdy) do
        mzdy[i].castka = math.floor(mzdy[i].castka * koef)
      end
      return mzdy
    end
  }

function tmzdy()
  local mzdy = {}
  setmetatable(mzdy, mt_mzdy)
  return mzdy
end

mzdy = tmzdy()
--načtení dat
nacti(mzdy)

--rozhodli jsme se snížit mzdy o 5 procent
koef = 0.95
mzdy = mzdy * koef
```

Samozřejmě i v jazyce Lua bychom mohli použít proceduru, která by vedla ke stejnému výsledku, avšak na tomto příkladu bylo účelem demonstrovat, jak snadno lze v jazyce Lua předefinovat operátor a tak vytvořit naprosto novou funkcionalitu podle potřeby programátora v případě, že stávající z různých důvodů nevyhovuje jeho požadavkům. Metatabulky a jejich metametody nabízejí nepřeberné množství

možností, jak ovlivnit chování tabulky reprezentující abstraktní datovou strukturu. Vyvolaná metametoda získává jako parametry operandy konkrétní operace, pro kterou je určena. Kompletní výčet klíčů metametod je uveden v následujícím výčtu:

- **__index** – v případě, že tabulka nemá hledaný klíč, pokusí se Lua vyhledat tento klíč v metatabulce (používá se pro vytvoření dědičnosti)
- **__newindex** – zavolá se v případě přiřazení do tabulky
- **__mode** – kontroluje tzv. slabé reference
- **__call** – vyvolá se v případě, že je tabulka následována kulatými závorkami (volání funkce)
- **__metatable** – umožňuje skrýt metatabulku
- **__tostring** – zavolá se při volání zabudované funkce tostring
- **__len** – reaguje na použití operátoru # na tabulku
- **__gc** – ovlivňuje garbage collector při práci s datovým typem **userdata**
- **__unm** – vyvolá se při použití unárního mínusu před tabulkou
- **__add, __sub, __mul, __div, __mod** – vyvolají se při sčítání, odečítání, násobení, dělení a operace modulo s tabulkou
- **__pow** – metametoda pro umocňování tabulky
- **__concat** – užitečná v případě pokusu o zřetězení tabulky
- **__eq, __lt, __le** – zavolají se při použití relačních operátorů =, < a <=

(Metatable Events, 2014)

3.3.4 Řetězec

Datový typ řetězec je představován jako posloupnost znaků. V obou jazycích se standardně předpokládá kódování na jednom bytu. Základním řetězcovým typem obou jazyků je **string**, liší se však způsob jeho implementace.

V Pascalu může datový typ **string** obsahovat maximálně 255 znaků. V řetězci se udržuje informace o počtu platných znaků. Tato informace se nazývá okamžitá délka a je fyzicky uložena v nultém bytu řetězce (Rybička a Čačková, 2012). Pro řetězce delší než 255 znaků lze použít datový typ **PChar** (nulou zakončený řetězec). Ačkoliv se v případě typu **PChar** jedná o ukazatel, Free Pascal podporuje inicializaci konstantou a přímé přiřazení.

V jazyce Lua jsou řetězce ukládány tak, že nemají délkový bajt na začátku, ale jsou zakončeny zvláštním bajtem s hodnotou nula (stejně jako **PChar** v Pascalu). Stejnou strukturu mají například i řetězce v jazyce C, avšak v jazyce Lua nelze hodnotu typu **string** měnit – při pokusu o změnu dojde k vytvoření nového řetězce a k vrácení ukazatele na místo do paměti, kde se nachází. Na rozdíl od jazyka C využívá Lua automatickou správu paměti – alokace a dealokace řetězců probíhá automaticky.

Každý **string** je v jazyce Lua uložen v paměti pouze jednou. Všechny ostatní výskyty jsou pouhé reference na toto místo, čímž se v případě ukládání dat v podobě záznamů snižuje režie potřebná k jejich uložení do paměti. Rozdíly mezi implementací řetězců obou jazyků se mohou výrazně projevit už i v primitivních příkladech, kdy často používaný algoritmus jednoho jazyka je v druhém naprosto neefektivní. Tento rozdíl je patrný na příkladu, kde načítáme data ze vstupu po řádcích (popřípadě znacích) a řetězíme je do proměnné (uvedena pouze část kódu):

```
//v jazyce Pascal
radek := '';
while not eof do
begin
  readln(radek);
  ret := ret + radek;
end;
```

V jazyce Lua by obdobně zapsaný algoritmus vedl k dlouhému provádění cyklu a ohromnému množství zpracovaných dat – řetězce jsou v jazyce Lua nezměnitelné, tedy při každém dalším zřetězení by se vytvořil nový řetězec zvětšený o řádek.

Pro jednoduchost přepokládejme, že velikost každého načítaného řádku je stejná. Potom můžeme velikost přesunutých dat v jazyce Lua (při použití stejného algoritmu jako v jazyce Pascal) vyjádřit jako součet aritmetické řady podle vzorce:

s_n	$=$	$((a_1 + a_n) * n)/2$, kde
s_n	...	velikost přesunutých dat v bajtech
a_1	...	velikost řetězce po načtení 1. řádku v bajtech
a_n	...	velikost řetězce po načtení všech řádků v bajtech
n	...	počet načtených řádků

V případě, že se na vstupu vyskytovalo 10 řádků a velikost každého řádku byla 100 B (délka výsledného řetězce byla 1 000 B), muselo v jazyce Lua dojít podle vzorce k přesunu celých 55 000 B dat!

Mnohem efektivnějším algoritmem je načítání řádků do tabulky a zavolání funkce **table.concat** po skončení načítání:

```
--v jazyce Lua
ret = {}
for radek in io.lines() do
  ret[#ret + 1] = radek end
ret = table.concat(ret)
```

3.3.5 Množina

Pro práci s množinami implementuje Pascal třídu množinových typů. Pro specifikace typu prvků množiny slouží tzv. bázový typ množiny. Aby mohl být datový typ bázovým, musí splňovat dvě podmínky – musí být ordinálním typem a musí nabývat hodnot s ordinálními čísly v rozmezí 0 až 255 (množina tedy může obsahovat maximálně 256 různých hodnot). Mezi hodnoty deklarovaného typu pro množinu pak patří prázdná množina a množina prvků bázového typu.

V jazyce Pascal se pro práci s množinou používá datový typ **set**. Vytvoření množiny probíhá zavoláním konstruktoru (syntakticky zapsaného jako hranaté závorky, které mohou být prázdné nebo obsahovat výčet hodnot). Množina je v Pascalu implementována jako bitové pole. Množina je tedy vyjádřena jako posloupnost bitů, přičemž každému bitu přísluší právě jedna hodnota bázového typu. Přítomnost prvku v množině je vyjádřena pomocí hodnoty příslušného bitu – hodnota 1 značí přítomnost, hodnota 0 nepřítomnost prvku.

Velikost proměnné datového typu **set** závisí na počtu prvků. Ve Free Pascalu je standardně množina uložena na 32 bajtech, v případě, že počet prvků nepřesahuje 32, uloží kompilátor jazyka množinu na 4 bajty. (Free Pascal, 1993–2010)

Turbo Pascal nabízí efektivnější implementaci množiny, kdy pro velikost platí:

$$\text{Velikost} = (\text{MaxOrdHodnota div } 8) - (\text{MinOrdHodnota div } 8) + 1$$

Jazyk Lua nemá množinu ve formě bitového pole implementovanou, pro reprezentaci tohoto typu používá strukturu **table**, kde jednotlivé prvky množiny jsou klíči do tabulky. Zároveň nenabízí množinové operace, které má Pascal: `+` (sjednocení), `*` (průnik), `-` (rozdíl). Stejně tak chybí relační množinové operátory. Následující příklad ukazuje, jak snadno můžeme tuto chybějící funkcionalitu naprogramovat pomocí metametod (v příkladu přidány pouze množinové operace):

```
--v jazyce Lua
mt_mnozina = {
  __add = function(a,b)
    local mnozina = Set()
    for key in pairs(a) do mnozina[key] = true end
    for key in pairs(b) do mnozina[key] = true end
    return mnozina
  end,

  __sub = function(a,b)
    local mnozina = Set()
```

```
    for key in pairs(a) do
      if b[key] then mnozina[key] = nil
        else mnozina[key] = true
      end
    end
    return mnozina
  end,

  __mul = function(a,b)
    local mnozina = Set()
    for key in pairs(a) do mnozina[key] = b[key] end
    return mnozina
  end
  }

--konstruktor množiny
function Set(...)
  local mnozina = {}
  for _, value in ipairs({...}) do
    mnozina[value] = true
  end
  setmetatable(mnozina, mt_mnozina)
  return mnozina
end

--vytvoření a inicializace množiny
a = Set('a', 'b', 'c')
b = Set('d', 'e', 'a', 'b')

c = a + b     --v množině c je součet množin a, b
c = a * b     --v množině c je průnik množin a, b
c = a - b     --v množině c je rozdíl množin a, b
```

Z didaktického hlediska je rychlejší použití množin v jazyce Pascal (nemusíme programovat množinové operace), zápis této nové funkcionality také mimo jiné vyžaduje znalost používání metatabulek a principu množinových operací. Mezi výhody tohoto programu v jazyce Lua patří neomezená velikost (jiným slovem mohutnost) množiny a možnost ukládání prvků libovolného datového typu.

3.3.6 Záznam

Záznam je v jazyce Pascal jediný datový typ, který může obsahovat položky různých typů (tj. heterogenní data). Umožňuje seskupovat jednotlivé datové položky do logických celků a vytvářet tak hierarchická uspořádání, která usnadňují tvorbu a práci s abstraktními datovými typy. Definice typu záznam začíná klíčovým slovem **record**, za kterým následuje seznam položek, který je potřeba ukončit klíčovým slovem **end**. Pro přístup k jednotlivým položkám se využívá tzv. tečková notace – syntaktický zápis ve tvaru **záznam.položka**.

V jazyku Pascal se vyskytuje i tzv. variantní záznam. Jedná se o speciální verzi záznamu, která umožňuje v záznamu ukládat různé varianty údajů podle hodnoty přiřazené v rozlišovací položce (ta musí nabývat pouze hodnot ordinálního typu). Překladač nekontroluje přístup k variantním složkám – vždy můžeme pracovat se všemi variantami (bez ohledu na hodnotu rozlišovací položky). Velikost záznamu odpovídá velikosti jeho největší varianty.

```
//v jazyce Pascal
type Zaznam = record
       //pevná část
       polozka: byte;
       //variantní část
       case varianta: boolean of
         true: (polozka1: byte);
         false: (polozka2: longint);
     end;
```

Pro reprezentaci záznamu v jazyce Lua se používá datová struktura **table**, kde názvy položek záznamu představují indexy do tabulky. Vzhledem k zažité konvenci spousty dalších jazyků umožňuje i jazyk Lua používat tečkovou notaci pro přístup k prvkům záznamu **tabulka.nazev** – jedná se pouze o jinou variantu syntaktického zápisu **tabulka["nazev"]**. Z didaktického pohledu zde lze spatřovat riziko v tom, že začátečníci zamění **tabulka.nazev** za **tabulka[nazev]**, kdy u první varianty je indexem řetězec **nazev**, u druhé varianty obsah proměnné **nazev**.

3.3.7 Podprogram

Datový typ podprogram je užitečný v případech, kdy potřebujeme za běhu činnost jednoho podprogramu ovlivnit různými procedurami nebo funkcemi (tzv. vnějšími podprogramy). Tyto podprogramy jsou pak často předávány jako parametry operacím, jejichž činnost chceme upravit. V jazyce Pascal se pro podprogram definuje datový typ, který odpovídá požadované hlavičce procedury nebo funkce, ale je vynechán její identifikátor. Za proměnnou tohoto typu je pak možné dosadit ukazatel na libovolný podprogram, jehož počet parametrů (a v případě funkcí i návratová hodnota) odpovídá definici typu, například:

```
//v jazyce Pascal
program podprogram;
type tfunkce = function (X, Y: byte) : byte;

function rozdil (X, Y: byte) : byte;
begin
  rozdil := X - Y;
end;

var funkce : tfunkce;
begin
  funkce := @rozdil;
end.
```

V jazyce Lua je funkce jeden z osmi podporovaných datových typů – stejně jako v Pascalu je možné podprogramy přiřazovat proměnným, předávat je jako parametry jiným funkcím, ukládat podprogramy do jiných struktur a podobně. Oproti jazyku Pascal však umožňuje vytváření lokálních anonymních funkcí (používá se pro ně též termín uzávěry), které je možné vracet jako návratový typ.

Funkce může pomocí příkazu **return** vracet libovolné množství návratových hodnot. Tyto vrácené hodnoty můžeme v jazyce Lua díky operátoru vícenásobného přiřazení přiřadit několika proměnným zároveň. V jazyce Pascal mohou funkce vracet pouze jednu hodnotu (v případě požadavku na vrácení více hodnot je musíme uložit do strukturovaného datového typu, který také může být návratovou hodnotou). Lua podporuje volitelné parametry u funkcí – to znamená, že v případě zadání méně parametrů se za zbytek dosadí hodnota **nil**, při zadání více parametrů jsou přebytečné zahozeny. Kromě volitelných parametrů podporuje i proměnný počet parametrů (zápis se provádí pomocí výrazu ...). Práci s podprogramy v jazyce Lua ilustruje následující příklad:

```
--v jazyce Lua
--funkce se dvěma návratovými hodnotami
function vrat2funkce()
    local function prvni(par1, par2) print(par1, par2) end
    local function druha(par1, par2) print(par1, par2) end
    return prvni, druha
end

--funkce s proměnným počtem parametrů
function promennypocet(...)
    --převod na tabulku
    local arg = {...}
    --vytiskne počet zadaných proměnných parametrů
    print(#arg)
end

a, b = vrat2funkce()

print(a(1, 2))   --oba parametry jsou zadané, vytiskne se "1 2"
print(b(1))           --druhý parametr chybí, je za něj dosazena
                 --hodnota nil, vytiskne se "1 nil"

promennypocet(1, 2, 3)             --vytiskne se "3"
promennypocet(1, 2, 3, 4, 5)       --vytiskne se "5"
```

Z programátorského hlediska se jedná o nezanedbatelné výhody oproti Pascalu. Například díky podpoře proměnného počtu parametrů odpadá povinnost definovat více funkcí stejného jména kvůli různému počtu předávaných parametrů. Z výukového hlediska však lze spatřovat riziko v tom, že by studenti s oblibou využívali tuto možnost pro předávání velkého počtu parametrů spíše, než aby je ukládali do struktur a ty následně předávali.

3.4 Typová konverze

Při typové konverzi se setkáváme se dvěma možnostmi – trvalým převodem mezi dvěma datovými typy a přetypováním. Při převodu dochází ke změně v paměti – hodnota je převedena z jednoho tvaru do druhého. Při přetypování zůstává hodnota v paměti stejná, pouze je místo v paměti dočasně chápáno jako jiný datový typ, dochází ke kontrole shody alokovaných velikostí. Podle toho, zda se provádí automa-

ticky nebo je vynucená programátorem, se rozlišuje na implicitní a explicitní.

Implicitní typová konverze se vyskytuje především při vyhodnocování aritmetických výrazů, kterých se účastní operandy různých datových typů nebo při předávání parametrů, kde dochází k převodu skutečného parametru na formální. Kromě běžných implicitních konverzí mezi různými číselnými datovými typy poskytuje Lua automatickou konverzi mezi datovými typy **number** a **string**. Při jakékoli aritmetické operaci nebo volání funkce, která očekává jako parametr číslo, se pokouší převést **string** na **number**. Stejně tak v obráceném případě, kdy je očekáván **string** (například při zřetězení), se pokouší převést **number** na **string**. Implicitní převod mezi těmito dvěma datovými typy Pascal neprovádí a ohlásí chybu při kompilaci nebo za běhu programu (dojde k pádu programu).

Naproti tomu explicitní typovou konverzi má plně ve svých rukou programátor. V Pascalu může k tomu účelu využít např. procedury **Str** a **Val**, k dočasnému přetypování může použít identifikátor některého datového typu jako funkci (jako parametr předáme hodnotu, kterou chceme převést). Jazyk Lua nabízí pro explicitní konverzi funkce **tonumber** a **tostring**.

3.5 Soubory

Při řešení praktických problémů se setkáváme s velkým množstvím vstupních dat, se kterými chceme pracovat. Interaktivní zpracování takových dat (tj. v případě, kdy vstupní data poskytuje uživatel za běhu programu) by bylo velice neefektivní, a to kvůli dlouhé době potřebné k zadání a náchylnosti na chyby při zadávání. Proto se využívá dávkové zpracování, kdy jsou vstupní data předem připravena v souborech, které jsou poté načteny do programu. Obdobně i výstupní data lze zapsat automaticky do souboru (namísto standardního výstupu), se kterými pak můžeme dále pracovat v jiných aplikacích – sdílet tato data.

U strukturovaných datových typů, o kterých se až dosud hovořilo, platilo, že hodnota jejich mohutnosti (tj. jejich velikost) je konečná. Spousta složitějších datových struktur jako je soubor, lineární seznam, strom, je charakteristická nekonečnou mohutností. Velikost paměťového prostoru není známá v době překladu a může se měnit v průběhu zpracování programu. Při práci s datovým typem soubor je tedy nutné používat odlišný přístup než u předchozích strukturovaných typů kvůli výše zmíněným faktům. (Honzík, 1988)

Podstatou souborů je jejich sekvenční přístup – tj. v daném okamžiku je možný přístup pouze k jedné složce struktury. Důsledkem toho typu přístupu je skutečnost, že nelze ve stejný moment ze souboru číst a zároveň zapisovat.

V jazyce Pascal existuje pro otevření souboru procedura **reset** a **rewrite**.

V jazyce Lua se volá funkce **io.open**, kde druhým parametrem je řetězec obsahu-

jící zástupné znaky pro konkrétní režimy otevření. Studenta by tak mohlo napadnout zapsat tento parametr jako **"rw"**, tento pokus by skončil chybovým hlášením **invalid mode**.

Jazyk Pascal má definovány tři typy souborů, u každého z nich se liší způsob zacházení – textové, netextové s udaným typem a netextové bez udání typu. Při načítání ze souboru jsou pro Pascal klíčové funkce **eoln** a **eof** (vrací hodnotu **true** při nalezení konce řádku a konce souboru).

Jazyk Lua rozlišuje pouze soubory textové a netextové bez udání typu. V jazyce Lua se funkce **eoln** a **eof** nevyskytují přímo, ale jsou součástí zdrojového kódu jazyka napsaného v C – **eof** je přímo součástí jazyka C, **eoln** odpovídá testování rovnosti načteného znaku s řetězcovým literálem označujícím konec řádku. Pro explicitní testování konce souboru můžeme využít konstrukce **io.read(0)**, která vrací prázdný řetězec (nejsme na konci) nebo **nil** (v případě, že je dosaženo konce souboru). Konkrétně jsou součástí implementací funkcí **io.read** a **io.lines**.

3.5.1 Textové soubory

Následující příklad demonstruje práci s textovými soubory v obou jazycích.

Příklad: Ve vstupním textovém souboru se nachází řetězce oddělené koncem řádku. Načtěte tyto řetězce a vypište je do výstupního souboru vzestupně seřazené a bez duplicit.

```
--v jazyce Lua
function Nacti (soubor)
  local tabret = {}
  for ret in io.lines(soubor) do
    --odstranění duplicitních řetězců
    tabret[ret] = true
  end
  return tabret
end

function Vypis (soubor, tabret)
  local pole = {}
  local vystup = io.open(soubor, 'w')
  for klic in pairs(tabret) do
    pole[#pole + 1] = klic
  end
  table.sort(pole)
```

```
  for i = 1, #pole do
    vystup:write(pole[i] .. '\n')
  end
  io.close(vystup)
end

--hlavní program
tabret = Nacti("vstup.txt")
Vypis("unikat.txt",tabret)
```

Pro průchod souborem je použit iterátor **io.lines**, který automaticky zajišťuje otevření a zavření souboru. Každé zavolání vrátí načtený řádek ze souboru nebo **nil** v případě dosažení konce souboru (cyklus se ukončí). V jazyce Lua při načítání a ukládání řetězců můžeme zajistit unikátnost vložených hodnot za využití asociativního pole, kde platí, že hodnoty klíčů jsou vždy jedinečné – datový typ **table** je zde použit jako simulace datového typu množina. V tomto příkladu je tento princip užitečný, ale z didaktického hlediska může působit studentům problémy v případě, kdy si neuvědomí, že jimi zvolené klíče mohou nabývat duplicitních hodnot (např. u křestních jmen), a tak si přepíšou již vložená data.

Lua využívá řadící metodu Quick Sort, která má v ideálním případě časovou složitost lineárně logaritmickou, v nejhorším případě kvadratickou. Pro řazení slouží v jazyce Lua funkce **table.sort**, která porovnává hodnoty na indexech tabulky. Funkci můžeme zadat jako druhý parametr vlastní porovnávací funkci (nutné v případě, kdy nejsou ukládané hodnoty porovnatelné – např. záznam, musí platit, že funkce vrací **false**, pokud si jsou hodnoty rovny). Pro seřazení našich dat musíme vytvořit z asociativního pole indexované, abychom mohli použít řadící funkci. Klíče původní tabulky se stanou hodnotami tohoto pole.

```
//v jazyce Pascal
program unikat;

type ukprvek = ^prvek;
     prvek = record
       radek: pointer;
       levy, pravy: ukprvek;
     end;

//vkládání do binárního stromu
procedure Zarad (var S: ukprvek; co: pointer);
begin
  if S = nil then
```

```
  begin
    new(S);
    S^.radek := co;
    S^.levy := nil;
    S^.pravy := nil;
  end
  //následující podmínky zajistí unikátnost vložených dat
  else if String(co^) < String(S^.radek^) then
         Zarad(S^.levy, co)
       else if String(co^) > String(S^.radek^) then
              Zarad(S^.pravy, co);
end;

//výpis binárního stromu
procedure Vypis (var G: text; S: ukprvek);
begin
  if S <> nil then
  begin
    Vypis(G, S^.levy);
    writeln(G, String(S^.radek^));
    Vypis(G, S^.pravy);
  end;
end;

var F, G: text;
    radek: string;
    uk: pointer;
    S: ukprvek;
begin
  assign(F, 'vstup.txt');
  reset(F);

  assign(G, 'vystup.txt');
  rewrite(G);

  S := nil;
  while not eof(F) do
  begin
    readln(F, radek);
    //uložení řetězců na minimálním prostoru
    GetMem(uk, Length(radek) + 1);
    String(uk^) := radek;
```

```
    //vložení do stromu
    Zarad(S, uk);
  end;
  //výpis seřazených unikátních řetězců do souboru
  Vypis(G, S);

  close(F);
  close(G);
end.
```

Jak můžeme vidět, je délka zápisu příkladu v Pascalu dvojnásobná – museli jsme definovat typ binární strom a naprogramovat příslušné procedury, které s tímto typem pracují (zápis prodloužila především procedura **Zarad**). Výhodou použití stromu je jeho lineárně logaritmická časová složitost řazení. Kvůli snaze o efektivní využití paměti, jsou řetězce ukládány na minimálním prostoru pomocí obecných ukazatelů. Odstranění duplicitních řetězců (v našem případě neuložení duplicitních řetězců) je zajištěno podmínkami ve vkládací proceduře pomocí relačních operátorů (při rovnosti vkládaného řetězce s porovnávaným řetězcem končí rekurzivní volání).

3.5.2 Netextové soubory

Následující příklad ukazuje práci s netypovým souborem bez typu. Zadání příkladu: Je dán soubor bez udání typu, v němž jsou uloženy záznamy o zaměstnancích (id v podobě dvoubytového čísla, plat v podobě čtyřbytového čísla a jméno, uložené podle konvence jazyka C). Vložte tyto údaje do seznamu a vypište ho.

```
//v jazyce Pascal
program binar;
type
  UkPrvek = ^Prvek;
  TZam = record
    id: word;
    plat: longint;
    jmeno: string;
  end;
  Prvek = record
    zam: TZam;
    dalsi: UkPrvek;
  end;
```

```
  Zamestnanci = UkPrvek;

procedure init (var databaze: Zamestnanci);
begin
  databaze := nil;
end;

procedure vloz (var databaze: Zamestnanci; zam: TZam);
var pom: UkPrvek;
begin
  new(pom);
  pom^.zam := zam;
  pom^.dalsi := databaze;
  databaze := pom;
end;

procedure vypis (databaze: Zamestnanci);
var pom: UkPrvek;
begin
  pom := databaze;
  while pom <> nil do
  begin
    writeln(pom^.zam.id,' ', pom^.zam.plat,' ', pom^.zam.jmeno);
    pom := pom^.dalsi;
  end;
end;

var
  databaze: Zamestnanci;
  zam: TZam;
  f: file;
  znak: char;
  ret: string;

begin
  assign(f, 'vstup.dat');
  reset(f, 1);

  while not eof(f) do
  begin
    blockread(f, zam.id, 2);
    blockread(f, zam.plat, 4);
```

```
    ret := '';
    blockread(f, znak, 1);
    while znak <> #0 do
    begin
      ret := ret + znak;
      blockread(f, znak, 1);
    end;
    zam.jmeno := ret;

    vloz(databaze, zam);
  end;

  vypis(databaze);
  close(f);
end.
```

Jazyk Pascal přímo nabízí operace pro načítání a ukládání dat v binární podobě, práce s těmito soubory proto není náročná. Většinu programového kódu zabírá práce s lineárním seznamem a jeho definice.

```
--v jazyce Lua
local lib = require"struct"

Zamestnanec = function (zaznam)
  local tbl = {id = 0, plat = 0, jmeno = ""}
  --"H i4 s" = H pro dvoubytové neznaménkové číslo,
  --i4 pro znaménkové čtyřbytové číslo, s pro řetězec
  tbl.id, tbl.plat, tbl.jmeno = lib.unpack("H i4 s", zaznam)
  return tbl
end

function nacti (zamestnanci, zaznam)
  zamestnanci[#zamestnanci + 1] = Zamestnanec(zaznam)
end

function vypis (databaze)
  for _, zam in ipairs(databaze) do
    print(zam.id, zam.plat, zam.jmeno)
end
```

```
--hlavní program
databaze = {}
f = io.open("vstup.dat","r")

while f:read(0) do
  id = f:read(2)
  plat = f:read(4)
  local tbl = {}

  znak = f:read(1)
  while znak ~= "\0" do
    tbl[#tbl + 1] = znak
    znak = f:read(1)
  end
  ret = table.concat(tbl) .. "\0"
  nacti(databaze, id .. plat .. ret)
end

f:close()
vypis(databaze)
```

Pro práci s binárními soubory nenabízí Lua žádné standardní operace, **io.read** načítá znaky způsobem, který známe z textových souborů (načítá jednotlivé znaky). Tyto chybějící operace můžeme doprogramovat, ale například převod čísla typu **real** načteného jako řetězec na číslo je pro začínající programátory obtížný úkol. Proto je schůdnější variantou využít například knihovnu **struct** napsanou Robertem Ierusalimschy, který nabízí operace pro převod binárního toku dat na požadované datové typy. Součástí knihovny je tabulka se třemi funkcemi: **unpack** pro načítání binárních dat, **pack** pro tvorbu binárních dat a **size** pro zjištění velikosti těchto dat. (Ierusalimschy, 2012)

Délka kódu je menší než u jazyka Pascal, hlavní podíl má využití dynamické struktury **table**. Z hlediska náročnosti na vytvoření a pochopení je největším problémem volba správného formátu načítaných dat – jednotlivé volby prvního parametru funkce **unpack** odpovídají datovým typům definovaným v jazyce C. Pro zachování efektivity algoritmu je nutné pamatovat na správnou volbu algoritmu pro zřetězení znaků jména (viz podkapitola 3.3.4 na straně 27).

4 Abstraktní datové typy

Při tvorbě programů hledáme vhodný algoritmus a vhodné datové typy pro implementaci. V této kapitole se budeme zabývat návrhem a tvorbou vlastních abstraktních datových typů (zkratka ADT). Pro reprezentaci objektivní reality využíváme model (tj. abstrakce reality). Zřídkakdy nastává situace, že si tento model vystačí s datovými typy a jejich operacemi, které jsou předdefinovány v jazyce Pascal. Je tedy potřeba vytvořit nový datový typ, který je stejně jako každý jiný datový typ definován množinou hodnot a množinou povolených operací.

Při implementaci abstraktního datového typu procházíme těmito fázemi:

- Návrh reprezentace druhů ADT – v této fázi rozhodujeme, zda bude např. binární strom reprezentován jako dynamická datová struktura nebo jako pole.
- Návrh algoritmu operací – zde vybíráme co nejefektivnější algoritmus, bereme v potaz prostorovou a časovou složitost.
- Ověření implementace – kontrolujeme, zda implementace odpovídá specifikaci abstraktního datového typu, nejčastěji pomocí postupu, který nazýváme ladění programu (na příkladech ověřujeme, zda pro přípustná vstupní data obdržíme správné výsledky).
- Posouzení kvality implementace – srovnáváme naši implementaci s ostatními, rozhodující je časová a prostorová složitost – obecně platí, že můžeme tyto složitosti navzájem směnit (např. více paměti na úkor času). Mezi další kritéria patří jednoduchost zápisu, délka kódu apod. (Hudec, 2004)

4.1 Lineární seznam

Lineární seznam je jednou z nejobecnějších datových struktur, používaných pro práci s posloupností prvků. Jedná se o homogenní, lineární strukturu (tj. prvky seznamu jsou stejného typu, každému prvku lze určit prvek následující). Používá se jako základní prostředek pro práci s velkým objemem dat, umožňuje uchovávat a zpracovávat prakticky neomezené množství dat. (Wirth, 1988)

Pro srovnání obou jazyků je zvolena implementace ADT Fronta lineárním seznamem. Následující příklady ukazují implementaci pomocí pole a pomocí ukazatelů. Varianta pomocí pole:

```
//v jazyce Pascal
unit fronta;

interface
const MAXI = 10;
type
  TData = pointer;
  Fronta = record
    fr: array [1..MAXI] of TData;
    zac, kon: byte;
    n: byte;
  end;

  TypVypis = procedure(x: pointer);

  procedure init (var f: Fronta);
  procedure vloz (var f: Fronta; data: TData);
  procedure odeber (var f: Fronta);
  procedure vypis (f: Fronta; vypis: TypVypis);

implementation

procedure init (var f: Fronta);
begin
  f.zac := 0;
  f.kon := 0;
  f.n := 0;
end;

procedure vloz (var f: Fronta; data: TData);
begin
  if f.n < MAXI then
  begin
    if f.zac = 0 then f.zac := 1;

    if f.kon = MAXI then f.kon := 1
    else Inc(f.kon);

    f.fr[f.kon] := data;
    Inc(f.n);
  end;
end;
```

```
procedure odeber (var f: Fronta);
begin
  if f.n > 0 then
  begin
    if f.zac = MAXI then f.zac := 1
    else Inc(f.zac);

    Dec(f.n);
  end
  else begin
    f.zac:=0;
    f.kon:=0;
  end;
end;

procedure vypis (f: Fronta; vypis: TypVypis);
var i, poc: byte;
begin
  poc := f.n;
  i := f.zac;
  while poc > 0 do
  begin
    vypis(f.fr[i]);
    Inc(i);
    if i > MAXI then i := 1;
    Dec(poc);
  end;
end;
end.
```

V tomto modulu je zvolena varianta, kde odpadá potřeba posouvat prvky při odebrání prvního prvku. Namísto posouvání prvků pole se posouvají pouze indexy začátku a konce fronty. Počet prvků fronty je omezen velikostí pole. Oproti Lua je nutné pro zajištění univerzálnosti modulu (tj. aby pracoval s různými typy dat) pracovat s obecnými ukazateli a naprogramovat vně modulu podprogram pro výpis dat. Také musí programátor převést vkládaná data na obecný ukazatel, který vyžaduje vkládací funkce.

```
--v jazyce Lua
Fronta = {zac = 0, kon = -1}
```

```
function vloz (f, co)
  f.kon = f.kon + 1
  f[f.kon] = co
end

function odeber (f)
  if f.zac <= f.kon then
    f[f.zac] = nil
    f.zac = f.zac + 1
  end
end

function vypis (f)
  for i = f.zac, f.kon do
    print(f[i])
    end
end
```

V jazyce Lua použití stejné varianty implementace jednodušší jak z hlediska zápisové náročnosti, tak délky zápisu. Použitelné indexy nejsou omezeny jako v Pascalu, proto je při neprázdné frontě vždy zaručeno, že index konce fronty je větší než index jejího začátku (není nutné řešit možné přetečení indexu). Zároveň nemusíme na rozdíl od Pascalu pro zajištění univerzálnosti použití definovat nové podprogramy pro práci s obecnými ukazately. Počet prvků fronty lze vyjádřit výrazem:

p	$=$	$k - z + 1$, kde
p	...	počet prvků fronty
k	...	index začátku fronty
z	...	index konce fronty

Varianta pomocí ukazatelů:

```
//v jazyce Pascal
unit fronta;

interface
type
  TData = pointer;

  UkPrvek = ^Prvek;
  Prvek = record
    data: TData;
```

```
    dalsi: UkPrvek;
  end;

  Fronta = record
    zac, kon: UkPrvek;
    n: byte;
  end;

  TypVypis = procedure(x: pointer);

  procedure init (var f: Fronta);
  procedure vloz (var f: Fronta; data: TData);
  procedure odeber (var f: Fronta);
  procedure vypis (f: Fronta; vypis: TypVypis);

implementation
procedure init (var f: Fronta);
begin
  f.zac := nil;
  f.kon := nil;
  f.n := 0;
end;

procedure vloz (var f: Fronta; data: TData);
var prvek: UkPrvek;
begin
  new(prvek);
  prvek^.data := data;
  prvek^.dalsi := nil;

  if f.zac = nil then
  begin
    f.zac := prvek;
    f.kon := f.zac;
  end
  else begin
    f.kon^.dalsi := prvek;
    f.kon := f.kon^.dalsi;
  end;

  Inc(f.n);
end;
```

```
procedure odeber (var f: Fronta);
var pom: UkPrvek;
begin
  pom := f.zac;
  if f.zac <> nil then
  begin
    f.zac := f.zac^.dalsi;
    Dec(f.n);
  end;
  if f.zac = nil then f.kon := nil;
  dispose(pom);
end;

procedure vypis (f: Fronta; vypis: TypVypis);
var pom: UkPrvek;
begin
  pom := f.zac;
  while pom <> nil do
  begin
    vypis(pom^.data);
    pom := pom^.dalsi;
  end;
end;
end.
```

Z hlediska náročnosti na zápis programu je tento algoritmus obtížnější než varianta pomocí pole, především kvůli nutnosti práce s ukazateli, kdy ztráta nebo špatné použití ukazatelů vede ke ztrátě funkčnosti, vzniku neočekávaného chování nebo dokonce pádu programu. Datový typ Fronta (nebo Zásobník) slouží jako základní nástroj pro naučení a ovládnutí práce s ukazateli, proto je didakticky výhodné, že jazyk Pascal nutí programátora spravovat paměť manuálně. Naproti tomu Lua má automatickou správu paměti, včetně automatické reference a dereference ukazatelů, což vede obecně ke kratšímu zápisu algoritmu, ale snižuje pochopitelnost principu, díky kterému kód pracuje tak, jak má.

V jazyce Lua je použití ukazatelů pro reprezentaci lineárního seznamu zbytečné a neefektivní, protože se dá zapsat jako pole. Z didaktického hlediska ale může následující algoritmus názorně předvést fakt, že i přesto, že nemáme nástroje pro manuální referenci a dereferenci ukazatelů, můžeme s ukazateli pracovat, a to pomocí datového typu **table**, který je ukládán právě jako ukazatel. Tento poznatek je velice důležité připomínat, vzhledem k jeho důležitosti (ovlivňuje práci s **tabulkami**).

```
--v jazyce Lua
Fronta = {}

function vloz (Fronta, hodnota)
  local prvek = {hodnota = hodnota}

  if not Fronta.zac then
   Fronta.zac = prvek
   Fronta.kon = prvek
  else
   Fronta.kon.dalsi = prvek
   Fronta.kon = prvek
  end
end

function odeber (Fronta)
  if Fronta.zac then
    local pom = Fronta.zac
    Fronta.zac = Fronta.zac.dalsi
    pom = nil
  end
  if not Fronta.Zac then Fronta.kon = nil end
end

function vypis (Fronta)
  local pom = Fronta.zac
  while pom do
    print(pom.hodnota)
    pom = pom.dalsi
  end
end
```

Implementace v jazyce Lua se z hlediska náročnosti i přehlednosti zápisu neliší od té v Pascalu, jedinou změnou je odpadající nutnost inicializovat ukazatele před jejich použitím (všechny proměnné v jazyce Lua jsou automaticky inicializovány na **nil**).

4.2 Binární strom

Binární strom má pro předmět Programovací techniky velký význam. Je velice výhodné ho užívat všude tam, kde potřebuje zajistit rychlé vyhledávání, řazení atd. (dosahuje lineárně logaritmické časové složitosti). Strom je definován jako orientovaný, acyklický, souvislý graf.

Uzly binárního stromu se dělí do tří typů:

- kořen – nejvyšší uzel stromu (nemá žádné rodiče),
- koncový uzel – tj. uzel, který má právě jednoho rodiče a nemá žádného následovníka,
- ostatní uzly – uzel má právě jednoho rodiče a jednoho nebo dva následovníky.

Strom lze implementovat datovým typem pole, kde kořen stromu bude mít index 1, potomci i-tého uzlu pak budou ukládaní na indexy $2*i$ a $2*i+1$. V Pascalu bychom byli při výběru této varianty a použití standardního typu pole značně omezení jeho rozsahem (s každou další hladinou stromu se zdvojnásobuje hodnota indexu). I s využitím otevřeného pole bychom stále naráželi na rychle vzrůstající velikost pole. Proto se mnohem častěji dává přednost dynamické variantě, kde je prvek definován datovou položkou a dvěma ukazateli. (Rybička a Čačková, 2012)

Následuje implementace velmi často používané varianty tohoto stromu – binárního vyhledávacího stromu.

```
//v jazyce Pascal
program strombin;
type
  TData = byte;
  UkPrvek = ^Prvek;
  Prvek = record
    data: TData;
    l, p: UkPrvek;
  end;
  Strom = UkPrvek;

procedure init (var s: Strom);
begin
  s := nil;
end;

procedure vloz (var s: Strom; data: TData);
begin
  if s = nil then
```

```
  begin
    new(s);
    s^.data := data;
    s^.l := nil;
    s^.p := nil;
  end
  else if data < s^.data then vloz(s^.l, data)
       else vloz(s^.p, data);
end;

function najdi (s: Strom; data: TData): Strom;
begin
  while (s <> nil) AND (s^.data <> data) do
    if data < s^.data then s := s^.l
      else s := s^.p;
  najdi := s;
end;
```

Tento příklad implementace názorně demonstruje, jak jsme snížili spotřebu paměti za cenu složitějšího a pomalejšího vkládání a získávání hodnot ze stromu. Při vkládání jsme museli použít rekurzivní proceduru (u varianty pomocí pole bychom si vystačili s iteračním algoritmem).

```
--v jazyce Lua
Strom = {}

function vloz (Strom, data)
  local i = 1
  while Strom[i] do
    if data < Strom[i] then i = 2 * i
      else i = 2 * i + 1 end
  end
  Strom[i] = data
end

function najdi (Strom, data)
  local i = 1
  while Strom[i] and Strom[i] ~= data do
    if data < Strom[i] then i = 2 * i
      else i = 2 * i + 1 end
  end
  return Strom[i] end
```

V jazyce Lua nám datový typ **table** umožňuje na rozdíl od pole v Pascalu ukládat a vyhledávat hodnoty pomocí algoritmů využívajících iterace. Oproti Pascalu jsme dosáhli výraznému zvýšení zápisové přehlednosti (využíváme přímé indexace pole, nejsou potřebná rekurzivní volání, odpadá práce s ukazateli), kratšího zápisu zdrojového kódu a snížení zápisové složitosti.

4.3 Řídké pole

Představme si případ, kdy potřebujeme ukládat n-rozměrnou strukturu (např. pole, matici, …), ve které je obsazené velmi malé množství prvků, většina struktury se skládá z nulových prvků (např. 90 procent). Bylo by velmi neefektivní z pohledu spotřeby operační paměti uchovávat všechny tyto prvky. Namísto toho uložíme pouze obsazené prvky. V případě pokusu o získání hodnoty neobsazeného prvku vrátíme tzv. majoritní hodnotu (např. 0, **nil** apod.).

Příklad: Uložte informace o neorientovaném grafu pomocí incidenční matice. Graf obsahuje 100 uzlů. Předpokládejte, že mezi dvěma uzly může existovat maximálně jedna ohodnocená hrana. Napište operace pro vkládání a vyhledávání na základě čísel uzlů.

```
//v jazyce Pascal
program ridkep;
const MAXUZEL = 100;
type TData = byte;
  UkPrvek = ^Prvek;
  Prvek = record
    data: TData;
    uzel1, uzel2: byte;
    dalsi: UkPrvek;
  end;
  RidkePole = record
    majorita: TData;
    zac: UkPrvek;
  end;

procedure init (var rp: RidkePole; majorita: TData);
begin
  rp.majorita := majorita;
  rp.zac := nil;
end;
```

```
procedure vloz (var rp: RidkePole; uzel1, uzel2: byte;
data: TData);
var pom: UkPrvek;
begin
  if (uzel1 in [1..MAXUZEL]) AND (uzel2 in [1..MAXUZEL]) then
  begin
    new(pom);
    if uzel1 < uzel2 then
    begin
      pom^.uzel1 := uzel1;
      pom^.uzel2 := uzel2;
    end
    else begin
      pom^.uzel1 := uzel2;
      pom^.uzel2 := uzel1;
    end;
    pom^.data := data;
    pom^.dalsi := rp.zac;
    rp.zac := pom;
  end;
end;

function JeHrana (rp: RidkePole; uzel1, uzel2: byte): TData;
var pom: UkPrvek;
begin
  if (uzel1 in [1..MAXUZEL]) AND (uzel2 in [1..MAXUZEL]) then
  begin
    if uzel1 > uzel2 then Zamen(uzel1, uzel2);

    pom := rp.zac;
    while (pom <> nil) AND ((pom^.uzel1 <> uzel1)
    OR (pom^.uzel2 <> uzel2)) do
      pom := pom^.dalsi;

    if pom = nil then JeHrana := rp.majorita
    else JeHrana := pom^.data;
  end;
end;
```

Pro uchovávání informací je použit lineární seznam (konkrétně zásobník, protože na pořadí hodnot nezáleží). Vzhledem k faktu, že je matice sousednosti symetrická, jsou ukládány informace tak, že číslo prvního uzlu je menší nebo rovno číslu uzlu

druhého (tím ušetříme spotřebu paměti až o polovinu – ukládáme v podobě trojúhelníkové matice). Z didaktického hlediska je zápis algoritmu poněkud delší, studentům může činit problémy především správná formulace složené podmínky cyklu **while** kvůli její větší komplexnosti.

```
--v jazyce Lua
function RidkePole (maxindex)
  return {maxindex = maxindex, pole = {}}
end

function vloz (rp, uzel1, uzel2, data)
  if uzel1 > 0 and uzel1 <= rp.maxindex and uzel2 > 0
  and uzel2 <= rp.maxindex then
    if uzel1 < uzel2 then rp.pole[uzel1 .. "," .. uzel2] = data
      else rp.pole[uzel2 .. "," .. uzel1] = data
    end
  end
end

function najdi (rp, uzel1, uzel2)
  if uzel1 > 0 and uzel1 <= rp.maxindex and uzel2 > 0
  and uzel2 <= rp.maxindex then
    local u1 local u2
    if uzel1 < uzel2 then u1, u2 = uzel1, uzel2
      else u1, u2 = uzel2, uzel1
    end
    return rp.pole[u1 .. "," .. u2]
  end
end
```

Na tomto příkladu je výhoda jazyka Lua – datový typ **table** je implementován jako řídké pole, proto využití tohoto typu na těchto typech úloh výrazně zjednodušuje zápis zdrojového kódu. Výhodou oproti Pascalu je možnost přímé indexace jak při vkládání, tak hledání hodnot (např. na zjištění hrany mezi uzly 1 a 2 stačí zapsat výraz **rp.pole["1,2"]**) – algoritmus dosahuje konstantní časové složitosti. V případě neexistujícího prvku je automaticky vrácena hodnota **nil** (tj. majoritní hodnota).

5 Programové moduly

Při tvorbě programů s větším rozsahem je vhodné rozdělit práci mezi větší počet programátorů, kdy každý pracuje na své části. Poté tyto jednotlivé části (moduly) spojí do výsledného programu. Použití modulů usnadňuje jak tvorbu, tak i testování programu, umožňuje strukturovat program a znovupoužití jednou vytvořeného díla. Další výhodou je jeho obecnost – můžeme jej naprogramovat tak, aby umožňoval práci s více typy úloh (často se pro tento požadavek využívá vnějších podprogramů).

5.1 Moduly v jazyce Pascal

V jazyce Pascal je modul část programového systému, který má definované jméno a rozhraní a skládá se z množiny proměnných a množiny operací. Proměnná nebo operace modulu může být veřejná (tj. dostupná z vně modulu) nebo soukromá (tj. viditelná pouze uvnitř). Struktura modulu v Pascalu je následující:

- Hlavička modulu – začíná klíčovým slovem **unit**, za ním následuju název modulu, který musí být shodný se jménem souboru, v němž je uložen.
- Část rozhraní – začíná klíčovým slovem **interface**. Obsahuje veřejně dostupné prvky modulu – konstanty, datové typy, proměnné a hlavičky podprogramů.
- Implementační část – začíná klíčovým slovem **implementation**. Obsahuje soukromé prvky modulu – především těla podprogramů, jejichž hlavičky byly uvedeny v části **interface**.
- Inicializační část – nepovinná, obsahuje příkazy uzavřené v bloku.

Překladem modulu vznikne soubor s příponou **.ppu** (samostatně není spustitelný). Výhodou překladu je možnost skrýt implementaci modulu před jeho uživateli (ať už z důvodu utajení nebo ochrany proti editaci zdrojového kódu) – pro práci s modulem jim bude zpřístupněna pouze část **interface**. Pro použití modulu v programu pak slouží klauzule **uses**, za kterou následuje název požadovaného modulu. (Honzík, 1988)

5.2 Moduly v jazyce Lua

V jazyce Lua se jako modul chápe jako programový kód, který obvykle vytváří a vrací datový typ **table**. Vzhledem k faktu, že návratová hodnota je určena kódem v modulu, lze obecně vracet libovolný datový typ (pokud je to potřeba). Na rozdíl od Pascalu může být modul napsaný jak v jazyce Lua, tak v jazyce C.

Stejně jako vše ostatní, i moduly jsou načítány přímo za běhu programu. Pro použití modulu v programu pak slouží funkce **require"modul"**, která v tabulce **package.loaded** zkontroluje, zda již není modulu načtený. V kladném případě **require** vrátí příslušnou hodnotu (ve většině případů **table**, v případě absence návratové hodnoty modulu je uložena hodnota **true**). Výhodou tohoto přístupu je pouze jedno provedení kódu zapsaného v modulu.

Při prvním načítání modulu se vyhledává soubor s příponou **.lua** se zadaným názvem modulu. Při nalezení je načten pomocí funkce **loadfile**. V opačném případě hledá Lua knihovnu napsanou v jazyce C se zadaným názvem modulu. Knihovna je pak načtena pomocí funkce **package.loadlib**. (Jung a Brown, 2007)

Struktura modulu není v jazyce Lua striktně vymezená, na programátorovi leží zodpovědnost vybrat a použít vhodnou a přehlednou strukturu, která usnadní použití, upravitelnost a dokumentaci modulu. Příklady obvyklých struktur budou představeny v následující podkapitole.

5.3 Srovnání modulů v jazyce Pascal a Lua

Modul v jazyce Pascal může vypadat následovně:

```
unit nazev;

interface

//definice typů
type TData = byte;

//hlavičky procedur a funkcí
function spocti (a, b: TData): TData;
procedure vypis (a: TData);

implementation
```

```
function spocti (a, b: TData): TData;
begin
  //příkazy těla funkce
end;

procedure vypis (a: TData);
begin
  //příkazy těla procedury
end;

end.
```

Výhodou Pascalu oproti jazyku Lua je striktně vymezená struktura modulu, která zajišťuje snadnou orientaci, viditelně odděluje veřejné prvky (část **interface**) od soukromých prvků (část **implementation**), usnadňuje případnou dokumentaci, která by měla být součástí každého modulu. Nevýhodou je lehce delší zápis (hlavičky operací se vyskytují v modulu dvakrát), je nutné dbát na správný zápis syntaxe. Poslední výtkou může být riziko, že si při připojení modulu přepíšeme operace se shodnými jmény jako v modulu. Této hrozbě se ale můžeme vyhnout například pomocí zapouzdření do objektu nebo použitím tečkové notace: **nazevmodulu.nazevdatovehotypu**.

V Lua také můžeme načíst modul s globálními operacemi a typy, ale stejně jako v Pascalu bychom riskovali přepsání. Proto se velmi doporučuje uložit obsah modulu do lokální proměnné typu **table**, kterou pomocí příkazu return vrátíme jako návratovou hodnotu funkce **require**. Takový modul má pak nejčastěji tyto dva syntaktické zápisy:

```
--první varianta zápisu
local m = {}

function m.vypis()
  --tělo funkce
end

return m

--druhá varianta zápisu
local function vypis()
  --tělo funkce
end

return {vypis = vypis}
```

Pro použití v programu pak přiřadíme návratovou hodnotu naší vlastní proměnné – např. **modul = require"modul"**. Pro přístup k jednotlivým položkám tabulky pak používáme tečkovou notaci (např. **modul.vypis()**). Tímto způsobem se používají například knihovny pro práci s řetězci nebo pro matematické operace – **string.gsub**, **math.sin**, …

První varianta zápisu má výhodu, že stačí nastavit pouze tabulku **m** jako lokální. Veškeré další prvky tvoříme jako položky záznamu tabulky **m** přes tečkovou notaci. Výhodou je krátký a přehledný zápis modulu. Nevýhodou je, že každý název funkce musí mít prefix **m**.

V druhé variantě musíme nastavit jako lokální všechny prvky modulu a v příkazu **return** naplnit tyto prvky do tabulky. Výhodou této možnosti je odpadnutí nutnosti používat u všech názvů funkcí prefix **m**, na druhou stranu výčet funkcí, se kterými modul pracuje je až na konci modulu, což je méně výhodné především při tvorbě dokumentace. Také se zvětšuje délka zápisu, protože každý název funkce píšeme třikrát.

6 Objektové programování

Při tvorbě programových projektů si můžeme zvolit způsob, kterým na něj budeme pohlížet. Mezi možnosti patří metoda zdola nahoru u strukturovaného přístupu (jednotlivé části spojujeme do celků) a metoda shora dolů u objektového přístupu (celek rozdělujeme na dílčí části).

Základem objektového programování je objekt, který je chápán jako prostředek pro spojení datových položek a operací nad nimi do jednoho celku.

6.1 Základní vlastnosti objektů

Mezi tyto vlastnosti řadíme:
- Dědičnost – objekty mohou přebírat atributy a operace od jiných.
- Mnohotvarost – stejné rozhraní umožňuje ovládat podobné objekty.
- Zapouzdřenost – k datovým složkám se přistupuje výhradně prostřednictvím metod.

6.2 Tvorba objektu v jazyce Pascal

Jazyk Pascal nabízí pro práci s objekty datový typ object. Jeho tvar je velmi podobný typu **record** – obsahuje seznam datových položek (atributů) a podprogramů, které se nazývají metody objektu. Pro přístup do objektu se používá tečková konvence, stejně jako u typu **record**.

Zápis definice objektu může vypadat například následovně:

```
type obj = object
       //atributy
       atribut: byte;
       //metody
       procedure zvys (hodnota: byte);
     end;
```

K definici se potom v deklarační části doplňují těla metod, kde pro jasné určení, kterému objektu metoda náleží, předchází jejich názvům jméno objektu následované tečkou, např. **obj.nastav**. Mezi metodami se můžou vyskytovat dvě speciální – konstruktor (vytváří objekt, používá se pro inicializaci atributů a naplnění tabulky virtuálních metod) a destruktor (obsluhuje uvolnění paměti objektu). Pro jejich zápis se u metody namísto klíčového slova procedure uvede **constructor** nebo **destructor**.

Tvorba objektu v Pascalu je velice přehledná, především díky své podobnosti s datovým typem záznam. Co však může ztížit studentům pochopení objektu je automaticky předávaný parametr **self**, který představuje odkaz na daný objekt. Tento parametr umožňuje přístup objektu ke svým atributům a metodám, ale je předávaný skrytě – není obsažen v seznamu parametrů metod objektu. Jak uvidíme dále, v jazyce Lua můžeme (ale nemusíme) tento parametr předávat explicitně pro výukové účely.

6.3 Tvorba objektu v jazyce Lua

Jazyk Lua neobsahuje podporu pro objektové programování, ale nabízí prostředky, jak ho poměrně snadno simulovat. Jedním z možných návrhů je vytvořit objekt (tabulku), který bude prototypem pro ostatní objekty (tedy simulace vztahu třída – instance třídy). V metodě **new** (která se chová jako konstruktor) zajistíme vytvoření nového objektu, kterému za pomoci metatabulky a metametody pod klíčem **__index** přiřadíme atributy a metody prototypu.

Následující příklad demonstruje základní práci s objekty:

```
Obj = {atribut = 0}

--parametrický konstruktor
function Obj:new (tbl)
  local obj = tbl or {}
  setmetatable(obj, self)
  self.__index = self
  return obj
end

function Obj:zvys (hodnota)
  self.atribut = self.atribut + hodnota
end
```

```
--hlavní program
o = Obj:new()
o:zvys(10)
```

Tento kus programového kódu, ač velice krátký, je pro začátečníka (programátora) obtížný na pochopení.

Je důležité zdůraznit, že volání metody **Obj:new ()** není nic jiného než zkrácený syntaktický zápis volání **Obj.new (Obj)** – tedy, že parametrem metody je i samotný odkaz na objekt, který je nazýván **self**. Onen delší zápis může být užitečný pro usnadnění pochopení problematiky OOP v jazyce Lua.

V metodě **Obj:new** přiřazujeme do položky **__index** tabulku. Lua očekává pod tímto klíčem funkci se dvěma parametry (tabulkou, ve které nebyl klíč nalezen a chybějícím klíčem). Mohli bychom tedy zapsat:

```
Obj.__index = function (tbl, klic) return Obj[klic] end
```

Vzhledem k častému používání této metametody nabízí Lua výjimku (zkrácený zápis). Tento řádek tedy způsobí, že v případě, že není klíč nalezen v zadaném objektu, pokusí se ho program vyhledat v metatabulce a vrátí jeho hodnotu (tímto způsobem tvoříme v jazyce Lua dědičnost).

Další částí obtížnou na pochopení může být práce s atributem v metodě **zvys**. Student by mohl nabýt dojmu, že všechny objekty pracují s atributem v tabulce **Obj** – tedy, že při jeho zvýšení dojde ke změně této hodnoty v metatabulce. Při volání metody **zvys** je jako parametr **self** předáván námi vytvořený objekt **o**, nikoliv **Obj**. Vyhodnocení příkazu v těle této metody při jejím prvním volání je následující:

1. Lua se pokusí vyhledat v objektu **self** atribut – neúspěšně.
2. Vyvolá se metametoda pod klíčem **__index**.
3. Vyhledá se atribut v příslušné metatabulce a vrátí se jeho hodnota.
4. Přičte se zadaná hodnota a uloží se do **self.atribut**.

Při příštím volání už **self.atribut** existuje, proto skončí krok 1 úspěšným nalezením, kroky 2 a 3 se už neprovedou. Lua nabízí při tvorbě objektů mnohem větší flexibilitu než jazyk Pascal. Například pokud potřebujeme změnit nebo přidat atribut nebo metodu jediné instanci třídy, stačí přidat, přepsat příslušný klíč tabulky.

6.4 Dědičnost

V jazyce Pascal zdědí nový objekt atributy a metody tak, že v jeho definici za klíčové slovo **object** uvedeme do kulatých závorek jméno objektu, po kterém má dědit, například:

```
type Potomek = object (Predek)
  //přidaný atribut
  atribut2: byte;
  //překrytá procedura s odlišným tělem
  procedure zvys (hodnota: byte);
end
```

Zápis dědičnosti je v Pascalu velice přehledný a snadný – pro výukové účely ideální.

V jazyce Lua dosáhneme stejného efektu tímto syntaktickým zápisem (předkem je **Obj** z příkladu v podkapitole 6.3 na straně 59:

```
--potomek Obj s přidaným atributem a překrytou metodou zvys
Potomek = Obj:new{atribut2 = 5}

function Potomek:zvys(hodnota)
  self.atribut = self.atribut + 2*hodnota
end

--hlavní program
o = Potomek:new()
o:zvys(10)
```

Z hlediska pochopitelnosti je tento zápis opět náročnější. V prvním kroku zdědí **Potomek** všechny atributy a metody po prototypu **Obj**. K tomuto objektu je navíc přidán nový a atribut a překrytá metoda **zvys**. Následně budeme tuto instanci třídy **Obj** používat jako prototyp pro tvorbu dalších potomků: podtřídu. Objekt **Potomek** totiž zdědil i konstruktor **new** – po jeho zavolání, kde jako parametr předáme potomka, se stane metatabulkou nového objektu **Potomek**, který má jako metatabulku (předka) **Obj**.

6.5 Polymorfismus

Mnohotvarost umožňuje objektům, mezi kterými je vztah dědičnosti volat metodu se stejným jménem (v každém objektu může být implementována jinak) tak, že bude pro každý objekt výsledek volání odlišný.

V jazyce Pascal narážíme při tvorbě polymorfních objektů na fakt, že jeho implementace standardně používá statické metody a brzké vazby (tj. adresy tyto metody). Pokud potřebujeme, aby se vyvolala metoda podle dosazeného objektu, musíme po-

užít virtuální metody a pozdní vazby. To zajistíme tak, že všechny metody, které jsou u předka virtuální (za hlavičkami jeho metod se vyskytuje direktiva **virtual**), definujeme i u všech jeho potomků jako virtuální. Druhou podmínkou je přítomnost konstruktorů u všech těchto objektů (zajišťují naplnění tabulek virtuálních metod).

Definice objektu pak může vypadat takto:

```
type
  Predek = object
    atribut: byte;
    constructor zacni(hodnota: byte);
    procedure zvys (hodnota: byte); virtual;
  end;

  Potomek = object (Predek)
    constructor zacni(hodnota: byte);
    procedure zvys (hodnota: byte); virtual;
  end;
```

Na základě této definice se při práci s objekty budou volat metody nikoliv podle jejich typu uvedeného v deklaraci, ale podle skutečně dosazeného objektu.

V jazyce Lua nemají parametry operací přiřazeny datové typy, metody s parametrem **self** jsou vždy polymorfní – za **self** můžeme dosadit jakýkoliv objekt. Předpokládejme kromě definic objektů a jejich konstruktorů ještě existenci následujících metod:

```
--metody předka
function Obj:vypis ()
  print(self.atribut)
end

function Obj:zvys (hodnota)
  self.atribut = self.atribut + hodnota
  self:vypis()
end

--metody potomka
function Potomek:vypis ()
  print(self.atribut, self.atribut2)
end

--volání objektu třídy Obj
Obj1:zvys(10)
```

```
--volání objektu třídy Potomek
Obj2:zvys(10)
```

U prvního objektu parametrem **self** volané metody **zvys** instance třídy **Obj** a je zavolána metoda **Obj:vypis**. U druhého objektu vystupuje jako **self** instance třídy **Potomek**, proto se volá **Potomek:vypis**.

Ze srovnání vyplývá, že zajištění polymorfismu v Pascalu lehce prodlouží zápis kódu – musíme napsat konstruktor a používat direktivu **virtual**. Z výukového hlediska je ale v Pascalu tohle téma ideálním prostředkem pro vysvětlení problematiky brzké a pozdní vazby.

6.6 Viditelnost objektů

V objektovém programování je velice dobrou praxí znemožnění přímého přístupu k částem objektu (převážně atributům) – přístup je zajištěný příslušnými metodami, čímž chráníme objekt před neoprávněnou nebo chybnou manipulací.

V jazyce Pascal můžeme pro určení viditelnosti použít klíčová slova – konkrétně **private**, **protected** nebo **public** (zapisují se před název atributu, metody nebo před jejich skupinu). Při absenci klíčového slova se předpokládá **public**, kdy jsou všechny atributy a metody přístupné z vně objektu.

V jazyce Lua můžeme dosáhnout soukromých atributů nebo metod faktu, že si každý vytvořený objekt uchovává odkazy na proměnné, se kterými pracuje, bez ohledu na jejich lexikální kontext. S těmito proměnnými je možné dále pracovat (číst a ukládat do nich hodnoty), ale není možné k nim přistupovat přímo.

```
Obj = {}

function Obj.new (tbl)
  local self = tbl or {}
  self.atribut = 0

  local obj = {}

  function obj.zvys (hodnota)
    self.atribut = self.atribut + hodnota
  end
  return obj
end
```

Ve funkci **new** jsme vytvořili dvě proměnné typu **table** – jednu pro uchování soukromého atributu, druhou pro uchování veřejného rozhraní. Funkce vrací pouze druhou tabulku, lokální proměnná **self** je i přesto zachována, protože ji používá metoda **zvys**. Tento princip se hojně používá u funkcionálních jazyků, pro studenty zvyklé na jazyk Pascal je tento přístup náročnější na pochopení.

7 Práce s operačním systémem

Každý program, který vytvoříme, běží na nějakém operačním systému. Tato vazba je velmi důležitá proto, aby náš program mohl pracovat se vstupními a výstupními zařízeními, jejichž správu má na starosti právě operační systém, nebo službami, které poskytuje. Oba zmiňované jazyky nabízejí procedury pro spolupráci s OS (např. pro práci se soubory, pro získávání hodnot z proměnných prostředí, pro předávání parametrů z příkazového řádku, …).

7.1 Standardní vstup a výstup

Jazyk Pascal i Lua používají standardní vstup a výstup a chybový výstup, liší se pouze zvolené symbolické názvy (v jazyce Lua odpovídají názvům podle konvence jazyka C, jsou umístěny v knihovně **io**). V obou jazycích jsou chápány jako textové soubory.

Tabulka 7.1 Standardní soubory

Název souboru	V Pascalu	V Lua
standardní vstup	input	io.stdin
standardní výstup	output	io.stdout
standardní chybový výstup	StdErr	io.stderr

Všechny standardní operace pro práci se soubory implicitně používají standardní vstup a výstup.

```
                //explicitní zápis:
eof()           //eof(input)
read()          //read(input)
write()         //write(output)
```

Pascal nám toto výchozí nastavení neumožňuje změnit, pro práci s jiným souborem musíme uvést jeho identifikátor jako parametr operace.

V jazyce Lua můžeme toto nastavení změnit pomocí funkcí **io.input** a **io.output**, kterým předáme jako parametr identifikátor nového souboru.

```
io.input('vstup.txt')
io.output(io.stderr)

io.read()            --načítá ze souboru vstup.txt
io.write()           --vypisuje na chybový výstup

--nebo můžeme zavolat funkci konkrétního souboru
io.stdout:write()
io.stderr:write()
```

V obou jazycích je práce s těmito soubory téměř identická, liší se pouze syntaktický zápis volání funkcí.

7.2 Hodnoty získané z příkazového řádku

Při spuštění programu z příkazového řádku můžeme zadat různé přepínače a parametry, kterými ovlivníme jeho činnost. V obou jazycích je oddělovačem chápána pouze mezera, všechny hodnoty jsou načítány jako řetězce, které pak v případě potřeby musíme převést na čísla.

V jazyce Pascal máme v modulu **SysUtils** k dispozici funkci **ParamStr(n)**, kde **n** udává pořadí parametru. Funkce vrací datový typ **string**. Pokud očekáváme parametry delší jak 255 znaků, musíme použít stejnojmennou funkci z modulu **ObjPas**, která vrací datový typ **AnsiString**. Tento modul je automaticky načten použitím zápisu direktivy ve tvaru **{$MODE OBJFPC}**. Při volání **ParamStr(0)** získáme název běžícího programu. Pro zjištění počtu zadaných parametrů slouží funkce ParamCount. Načtený parametr lze převést na číslo pomocí procedury **Val**.

Příklad spuštění programu z příkazového řádku a použití výše zmíněných operací:

```
program.exe vstup.txt log.txt

//výsledky operací
ParamStr(0) = "program.exe"
ParamStr(1) = "vstup.txt"
ParamStr(2) = "log.txt"
ParamCount = 2
```

V jazyce Lua se při spuštění programu naplní pole **arg**, kde nezáporné indexy odpovídají parametrům tak, jak je chápe Pascal. Na rozdíl od Pascalu je Lua interpretovaný jazyk, při spuštění na příkazovém řádku je navíc název interpreta a jeho parametry. Stejně jako v druhém jazyce, parametr načtený jako řetězec lze převést na číslo – funkcí **tonumber**.

Příklad spuštění programu v jazyce Lua:

```
lua --e "a=5" program.lua vstup.txt log.txt

--obsah pole arg
arg[-3] = "lua"
arg[-2] = "-e"
arg[-1] = "a=5"
arg[0] = "program.lua"
arg[1] = "vstup.txt"
arg[2] = "log.txt"

--počet parametrů skriptu
#arg = 2
```

Ve většině případů nás však zajímají pouze nezáporné parametry – tj. takové, které se týkají našeho skriptu.

7.3 Hodnoty získané z proměnných prostředí

Pro získání hodnoty proměnné prostředí je ve Free Pascalu v modulu **SysUtils** k dispozici funkce **getEnvironmentVariable(název)** a v modulu **Dos** pak funkce **GetEnv(název)**, kde je parametrem název hledané proměnné. Výhodou první zmíněné je funkce je její návratová hodnota typu **ansistring**, která může obsahovat řetězce delší než 255 znaků. V případě, že nebyla proměnná nalezena, vrátí prázdný řetězec. (Free Pascal, 1993–2010)

V jazyce Lua poskytuje stejnou možnost funkce **getenv(název)** z knihovny **os**. Ani v jednom z jazyků neexistuje možnost do těchto proměnných nastavovat hodnoty.

8 Diskuse a závěr

Tato bakalářská práce popsala a srovnala programovací jazyky Pascal a Lua za pomocí didaktických hledisek uvedených v úvodu práce. Výsledky porovnání poskytují podklady pro rozhodování o případném začlenění jazyka Lua do výuky nejenom v předmětu Programovací techniky – definují výhody a nevýhody použití obou jazyků pro vysvětlení konkrétních problematik programování. Zároveň může text sloužit jako podklad při vytváření studijního plánu předmětu nebo jako usnadnění přechodu programátora z jednoho jazyka do druhého.

Při práci s datovými typy v jazyce Lua je potřeba apelovat na studenty, aby dbali zvýšené pozornosti při pojmenovávání proměnných a práci s nimi – absence definic datových typů u dynamicky typovaného jazyka zhoršuje přehlednost zápisu a umožňuje snadnější tvorbu chyb z nepozornosti. Obzvláště při přiřazování hodnoty do proměnné je potřeba dávat pozor, zda není nová hodnota jiného datového typu než původní, čímž bychom mohli způsobit neočekávané chování programu. To neplatí v případě, že si to uvědomujeme a chceme nadále používat tuto proměnnou k jinému účelu. Stejně tak s odlišným chováním logických operací v jazyce Lua, kdy výsledkem logických operací může být i jiná hodnota než datového typu **boolean**, je potřeba studenty pečlivě seznámit. Pro jazyk Lua má **nil** jako datový typ navíc další důležitý význam – přiřazením hodnoty **nil** do proměnné dojde k zavolání garbage collectoru a smazání proměnné z operační paměti. Neméně důležité je naučit studenty správné práci s ukazateli v jazyce Lua, kde kromě typu **boolean** a **number** jsou všechny datové typy ukazatele – je nutné mít na paměti, že např. při posílání struktury ve formě tabulky nějakému podprogramu jako parametr nedochází k vytvoření kopie jeho obsahu, ale pouze vytvoření kopie odkazu na tuto tabulku. Značnou část výuky je vhodné věnovat datovému typu **table**, který je základním prostředkem pro tvorbu většiny pokročilých struktur a programů, jeho nepochopení by téměř určitě vedlo k obrovským problémům studenta chápat navazující látku a tím ke značnému snížení jeho motivace k průběžné přípravě do předmětu Programovací techniky počas semestru a problémům s jeho úspěšným ukončením.

Při tvorbě abstraktních datových typů v jazyce Lua se stává alfou a omegou využívání tabulek a jejich metametod, které nabízí větší variabilitu než jazyk Pascal (odpadají problémy s omezenou velikostí, lze měnit chování operátorů, …), na druhou stranu se obtížněji navrhuje plán výuky tak, aby učivo působilo uceleně a netrpělo přílišnou roztříštěností. V jazyce Lua je tvorba a implementace ADT pomocí dynamických datových struktur snažší, protože už máme k dispozici dynamickou strukturu **table**, pomocí které můžeme navrhnout všechny ostatní. Veškerá správa paměti probíhá automaticky pomocí garbage collectoru. Tím se stává implementace v jazyce Lua snažší a přehlednější. Zápis stejných struktur v jazyce Pascal je sice

delší, nicméně umožňuje názorně demonstrovat práci s operační paměti a pochopit principy jejího přidělování a uvolňování, což jsou znalosti, které by měl mít každý programátor.

Jazyk Lua dosahuje lehce kratších zápisů modulů kvůli absencí klíčových slov jazyka Pascal a hlaviček podprogramů v Pascalu v části **interface**. Z didaktického hlediska je však tvorba modulů v jazyce Pascal výhodnější – struktura modulu je pevně daná, tím se zvyšuje přehlednost a jednotnost zápisů, bez ohledu na osobu programátora. Zároveň se zlehčuje i případná dokumentace. V jazyce Lua je jeho struktura téměř naprosto volná – vzniká prostor pro tvorbu velmi nepřehledných nebo chybných zápisů.

Objektově orientované programování není součástí implementace jazyka Lua na rozdíl od jazyka Pascal, kde je jeho definice přehledná a používání objektů relativně snadné. Implementace OOP v jazyce Lua důkladně prověří míru znalostí a pochopení problematik, předcházejících tomuto tématu. Tuto fázi výuky můžeme označit za kritickou – studenti nedostatečně seznámení s předchozími řešenými problematikami (především s prací s tabulkami) se můžou „ztratit“. Jedná se o látku náročnější na vysvětlení, která ale velmi dobře ilustruje využití možností tabulek a umožňuje procvičit důležité programátorské obraty v tomto jazyce.

Obecně se dá konstatovat, že jazyk Lua umožňuje dosahovat u implementací programů větší efektivity a kratšího zápisu kódu, ovšem za cenu zhoršení přehlednosti a pochopitelnosti zápisu. To je způsobeno už účelem vzniku jednotlivých jazyků. Jazyk Lua vznikl kvůli potřebě rychlosti, efektivnosti a přenositelnosti v programování. Jazyk Pascal vznikl jako jazyk primárně určený pro výuku programování – struktura kódu je pevně vázána, zápis přehledný a snadný na pochopení. Výuka studentů k ovládnutí jazyka Lua a jeho efektivnímu využívání by tedy byla náročnější oproti výuce Pascalu, s jehož základy se navíc studenti seznámili již v předcházejícím předmětu Algoritmizace.

Literatura

Free Pascal [on-line]. 1993–2010. [cit. 2015-04-01]. Dostupné na: `http://www.freepascal.org/`.

Free Pascal History [on-line]. 2015. [cit. 2015-04-01]. Dostupné na: `http://wiki.freepascal.org/History`.

Hudec B. *Programovací techniky.* Praha : Vydavatelství ČVUT, 2001. 234 s. ISBN 978-80-7375-671-0.

Ierusalimschy R. *Library for Converting Data to and from C Structs for Lua 5.1/5.2* [on-line]. 2012. [cit. 2015-04-01]. Dostupné na: `http://www.inf.puc-rio.br/~roberto/struct/`.

Jung K., Brown A. *Beginning Lua Programming.* Indianapolis : Wiley/Wrox, 2007. 644 s. ISBN 978-04-7006-917-2.

Lua 5.3 Reference Manual [on-line]. 2015. [cit. 2014-02-28]. Dostupné na: `http://www.lua.org/manual/5.3/manual.html`.

Metatable Events [on-line]. 2014-12-19. [cit. 2014-02-28]. Dostupné na: `http://lua-users.org/wiki/MetatableEvents`.

Object Pascal History [on-line]. 2014. [cit. 2015-04-01]. Dostupné na: `http://wiki.lazarus.freepascal.org/Object_Pascal_History`.

Polách E. *Programování v jazyku Turbo Pascal* [on-line]. 2006. [cit. 2014-02-28]. Dostupné na: `http://home.pf.jcu.cz/~edpo/program/program.html`.

Rybička J., Čačková P. *Programovací techniky.* Brno : ES Mendelovy univerzity v Brně, 2012. 258 s. ISBN 978-80-7375-671-0.

Tišnovský P. *Základní konstrukce v programovacím jazyku Lua* [on-line]. 2009-03-17. [cit. 2014-02-28]. Dostupné na: `http://www.root.cz/clanky/zakladni-konstrukce-v-programovacim-jazyku-lua/`.

Vyhledávání IV. – Tabulky s rozptýlenými položkami [on-line]. 2003-2014. [cit. 2014-05-18]. Dostupné na: `http://programujte.com/clanek/2006021604-vyhledavani-iv-tabulky-s-rozptylenymi-polozkami/`.

Wirth N. *Algoritmy a štruktúry údajov.* Bratislava : Alfa, 1988. 481 s.

Základy programování v jazyce Pascal [on-line]. 2004. [cit. 2015-04-01]. Dostupné na: `http://pascal.webz.cz/kurs/lekce/lekce0.htm`.