evropský sociální fond v ČR
EVROPSKÁ UNIE
MINISTERSTVO ŠKOLSTVÍ, MLÁDEŽE A TĚLOVÝCHOVY
OP Vzdělávání pro konkurenceschopnost
INVESTICE DO ROZVOJE VZDĚLÁVÁNÍ

VYSOKÁ ŠKOLA BÁŇSKÁ – TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA STROJNÍ

APLIKOVANÁ INFORMATIKA

2. LEKCE – ALGORITMUS A JEHO POPIS

prof. Ing. Radim Farana, CSc.

Ostrava 2013

ISBN 978-80-248-3017-9

Tento studijní materiál vznikl za finanční podpory Evropského sociálního fondu (ESF) a rozpočtu České republiky v rámci řešení projektu: CZ.1.07/2.2.00/15.0463, MODERNIZACE VÝUKOVÝCH MATERIÁLŮ A DIDAKTICKÝCH METOD

OBSAH

2 ALGORITMUS A JEHO POPIS

OBSAH KAPITOLY:

MOTIVACE:

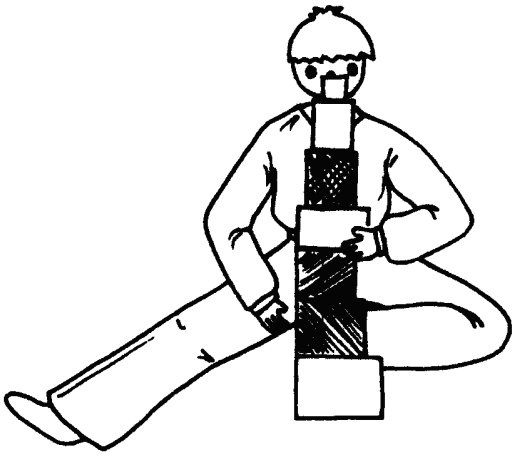
Základem dobré činnosti programů, informačních a řídicích systémů je kvalitně zpracovaný algoritmus jejich činnosti. Aby popis algoritmu snadno pochopili všichni členové realizačního týmu, je vhodné dokumentoval algoritmy v grafické podobě. To je důležité také pro možnost jejich dalšího rozvoje a úprav činnosti programu.

CÍL:

Pochopit jaké vlastnosti musí mít algoritmus a seznámit se s obvyklými metodami dokumentace algoritmů od různých verzí slovního popisu ke grafickým metodám jako jsou vývojové diagramy, diagramy aktivit, strukturogramy, nebo méně používané nástroje – kopenogramy a NS-diagramy. Seznámit se se základními strukturamy a členěním algoritmů. Umět používat správné druhy proměnných a předávání parametrů mezi jednotlivými programovými rutinami. Naučit se hodnotit algoritmy pro možnost jejich vzájemného porovnání a vyhodnocení jejich efektivity. Pochopit rozdělení složitosti algoritmů podle časové složitosti na polynomiální – P-problémy, nepolynomiální NP-problémy a NP-úplné problémy.

K tomu, aby mohl počítač provádět jisté manipulace s informací (daty) je potřeba určit jaké operace, v jakém pořadí a za jakých okolností mají být provedeny. Počítači je tedy třeba určit **postup práce** s daty, který samozřejmě může být závislý na tom, jaká data jsou zpracovávána. Tento postup práce je většinou obsažen v programu, který realizuje daný postup práce s využitím konkrétního **programovacího jazyka**, tedy jeho příkazů. Protože existuje řada různých programovacích jazyků, znamená to, že může existovat také řada různých programů, které realizují stejný postup práce. Převod programu do prostředí jiného programovacího jazyka je srovnatelný s překladem knihy do jiného jazyka. Duševní bohatství (chcete-li tedy informační obsah) díla se nemění. To je u programu uloženo právě v postupu práce, který program realizuje. V oblasti zpracování informace tento postup práce obvykle nazýváme **algoritmu**s. Definice algoritmu se různí, přijměme definici A. A. Markova [61]:

"Algoritmus je přesný předpis definující výpočtový proces vedoucí od měnitelných výchozích údajů až k žádaným (vždy správným) výsledkům. Tento předpis se skládá z jednotlivých výpočtových kroků, které jsou zapsány v určitém pořadí. Počet výpočtových kroků musí být konečný."



Obrázek 2.1 Algoritmus = postup práce

Abychom odlišili často zaměňované pojmy **algoritmus** a **program**, můžeme zjednodušeně uvést:

program = posloupnost příkazů,

algoritmus = postup práce.

Poznámka:

Slovo algoritmus pochází od uzbeckého matematika z IX. století Mohameda ibn Musa Al-Chórezmiho, který definoval základní pravidla pro provádění početních úkonů v desítkové soustavě.

Algoritmus musí splňovat několik základních požadavků:

1. **determinovanost** - shrnuje v sobě požadavky jako přesnost, srozumitelnost a jednoznačnost. To znamená, že v každém okamžiku řešení musí být jasné, jakou operaci má algoritmus provádět. (I když algoritmus čeká na příchod informace, která rozhodne o další činnosti, je jasné co má dělat.)
2. **hromadnost (masovost)** - algoritmus nesmí popisovat jen zpracování jedné sady konkrétních vstupních dat, ale celé skupiny příbuzných hodnot.
3. **rezultativnost** - algoritmus musí vždy dospět ke správnému výsledku, a to pomocí konečného počtu kroků.

4. **opakovatelnost** - při stejných hodnotách vstupních dat musí algoritmus vždy dospět ke stejnému výsledku.

2.1 POPIS ALGORITMU

Zatímco **popisem programu** je jeho **výpis**, s **popisem algoritmu** je to složitější. Postupně se vyvíjel a nabýval různých podob, které byly více či méně poplatné některému konkrétnímu programovacímu jazyku. Podle toho se také liší množina základních struktur, které je možno použít. V dalším textu nejprve na jednoduchém příkladu představíme několik typů popisu algoritmů a na závěr uvedeme přehled významných struktur, používaných v popisu algoritmů.

2.1.1 Slovní popis

Slovní popis vyjadřuje algoritmus prostředky přirozeného jazyka.

Příklad 2.1:

Z klávesnice čteme celá čísla a vypisujeme je na obrazovku doplněná o informaci, zda je číslo sudé nebo liché. Práce končí po vstupu čísla 0, které se nezpracovává.

U složitějších algoritmů je slovní popis komplikovaný, proto se často dělí do jednotlivých částí (bodů) nejčastěji očíslovaných vzestupnou řadou.

Příklad 2.2:

1 přečti číslo ze vstupu
2 když je číslo 0 jdi k bodu 9
3 vyšli číslo na výstup
4 když je číslo sudé, jdi k bodu 7
5 vyšli na výstup "liché"
6 jdi k bodu 1
7 vyšli na výstup "sudé"
8 jdi k bodu 1
9 konec

Slovní popis se tak velmi přiblížil k popisu **programu** pomocí **příkazů** [41][46][55][57]aj. Jednotlivé body se zpracovávají v přirozeném pořadí, pokud není proveden skok na jiný bod. U rozsáhlých popisů se poněkud ztrácí souvislosti při neustálých přeskocích. Při tvorbě algoritmu se navíc špatně vkládají další body do souvislé řady čísel. Aby se předešlo nutnosti opravovat čísla bodů a odskoků na ně, používá se číslování nesouvislou vzestupnou řadou (typicky s krokem 10). Tím se zápis algoritmu dále přibližuje zápisu jeho realizace, zejména v programovacím jazyku **BASIC** (Beginners All-purpose Symbolic Instruction Code). Tento jazyk vznikl v USA v roce 1964. Jeho snahou je co největší přiblížení příkazů přirozenému jazyku, samozřejmě anglickému. Postupně ovládl především mikropočítače, zejména díky své jednoduchosti. Později byl jeho vliv přehlušen vyššími programovacími jazyky (PASCAL, C, aj.) neboť základní množina jeho příkazů neumožňuje jednoduše realizovat některé běžné postupy (opakování s testem na začátku apod.). V současné době (pět let do konce tisíciletí) se BASIC opět vrací, samozřejmě výrazně rozšířený.

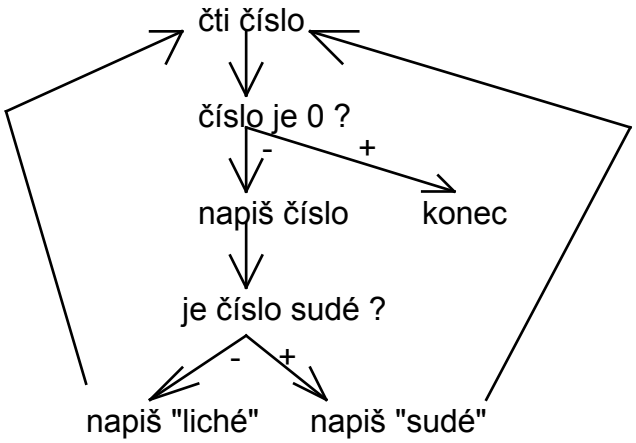
Zápis algoritmů přímo pomocí příkazů jazyka BASIC je špatně přenositelná do jiných programovacích jazyků a je rovněž špatně čitelný.

Příklad 2.3:

```
10 Read Cislo
20 If Cislo=0 Then Goto 90
30 Write Cislo
40 If Int(Cislo/2)*2=Cislo Then Goto 70
50 Write "liché"
60 Goto 10
70 Write "sudé"
80 Goto 10
90 End
```

Protože návaznosti jednotlivých částí algoritmu jsou při slovním popisu špatně patrné, používají někteří autoři **grafický zápis** [47]. V něm vhodně rozmístí jednotlivé činnosti a jejich návaznost označí šipkami. Zápis je pak podobný orientovanému grafu, kde uzly představují činnosti a hrany návaznost činností.

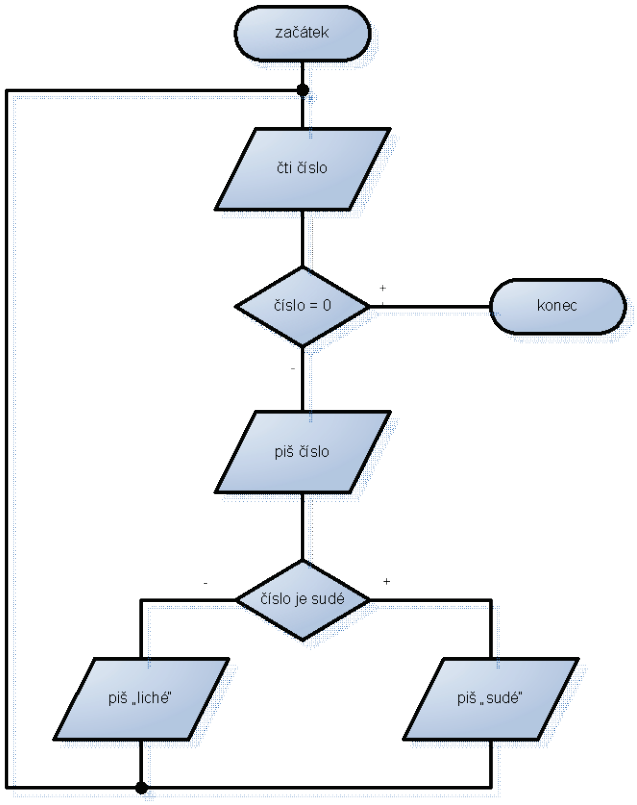
Příklad 2.4:



2.1.2 Vývojový diagram

Vývojový diagram můžeme chápat jako více formalizovaný grafický zápis algoritmu [7][14][27][61]. Pro jednotlivé druhy činností (výkonná činnost, rozhodování, vstup dat apod.) jsou zavedeny speciální grafické symboly, do kterých se zapisuje konkrétní prováděná činnost. Vývojové diagramy vznikly především pro popis algoritmů zpracovávaných v jazyce **FORTRAN** a nese s sebou také řadu omezení s tím spojených (programovací jazyk FORTRAN byl vyvinut v r. 1954 vývojovou skupinou IBM, kterou vedl John Backus, název je převzat ze slov FORmula TRANslator) Používání vývojových diagramů se později natolik rozšířilo, že tvar jednotlivých značek byl formalizován ČSN 36 9030, která rozeznává přes 30 značek. Část značek je obecná, (zpracování, rozhodování aj.), jiné jsou vysoce specializované (magnetický buben, disketa aj.).

Příklad 2.5:



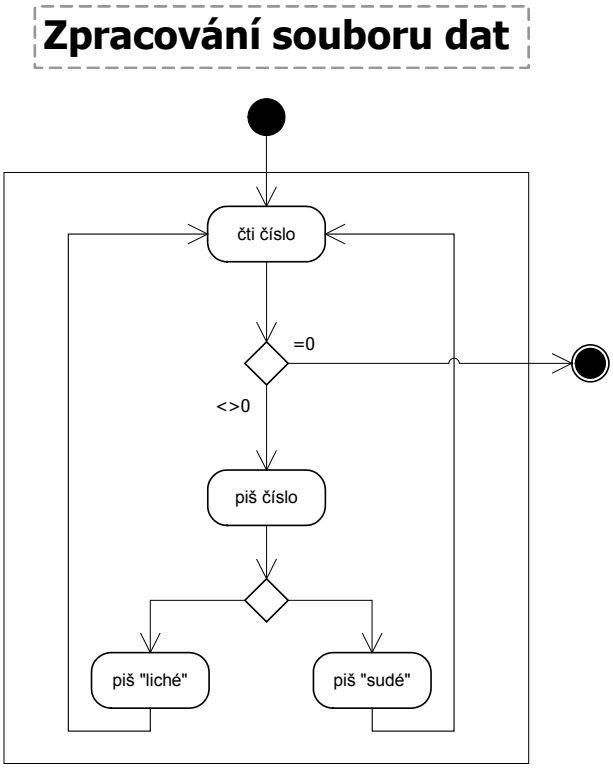
Mezi výhody blokových diagramů patří zejména možnost volby různé úrovně podrobnosti (hrubý vývojový diagram algoritmu se slovním popisem činnosti nebo podrobný popis odpovídající použitým příkazům). Vývojový diagram umožňuje poměrně snadno realizovat popsaný algoritmus prostředky různých programovacích jazyků.

Jako nevýhody jsou chápány skutečnosti, že vývojový diagram nevede uživatele k dodržování zásad **strukturovaného programování**, jak bude popsáno dále. Navíc se některé běžné struktury (např. opakování s udaným počtem cyklů) v blokovém diagramu realizují poněkud těžkopádně.

2.1.3 Diagram aktivit UML

Diagramy aktivit se podobají vývojovým diagramům, ale jsou nástrojem komplexního CASE (Computer-Aided Software Engineering) systému s názvem Unified Modeling Language, ve zkratce UML. UML představuje vizuální modelovací jazyk, který je v současné době standardem v oblasti analýzy a návrhu softwarových systémů, zejména objektově orientovaných. UML umožňuje vytvořit komplexní model celého systému, na který pohlížíme jednotlivými pohledy, z nichž právě diagram aktivit nejlépe popisuje činnost systému. Je určen k modelování řídicích a datových toků, tedy k posloupnosti aktivit, podmínek jejich vykonávání a toku dat mezi jednotlivými aktivitami. Jako jediný z popsaných grafických systémů zápisu algoritmu podporuje masivní paralelismus

Příklad 2.6:

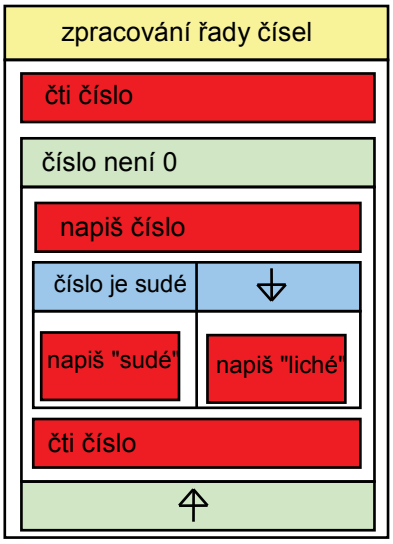


2.1.4 Kopenogram

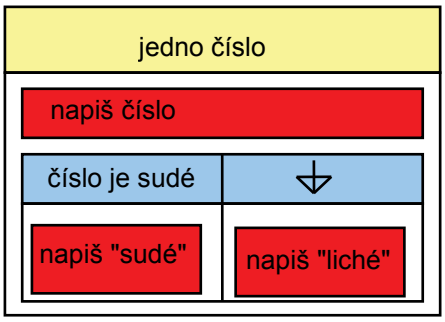
Název metody popisu algoritmů nazvané kopenogram je odvozen od jmen jejích tvůrců, kterými jsou Kofránek, Pecinovský a Novák. Vznikly především pro zápis programů dodržujících zásady **strukturovaného programování**, ve kterém je striktně předepsáno, že každá ucelená struktura algoritmu (opakování, rozhodování apod.) je vždy buď celá vnořena do jiné struktury, je s ní na stejné úrovni, nebo jinou strukturu obsahuje jako vnořenou [26][51].

Kopenogramy se používaly zejména při tvorbě algoritmů ve výukovém programovacím jazyku **KAREL**. Tento jazyk striktně vyžaduje správné vnořování struktur. Např. násilné opuštění cyklu opakování (oblíbený příkaz nepodmíněného skoku známý z jazyka BASIC) zde vůbec neexistuje. Čitelnost složitějších algoritmů popsaných kopenogramy není nejlepší, což vede k nutnosti rozkládat algoritmus na jednotlivé ucelené činnosti realizované samostatnými **procedurami**. Tato skutečnost může být chápána jako výhoda, či jako nevýhoda. Do značné míry to závisí na osobních názorech hodnotícího. Pro zvýraznění jednotlivých typů struktur (opakování, větvení apod.) je doporučováno je vybarvovat vždy stejnou barvou. To však v našich ukázkách nemůžeme využít.

Příklad 2.7



Příklad 2.8:



Jedním ze silných nástrojů strukturovaného programování je **rekurzivní volání** (**rekurze**), při kterém procedura volá sama sebe. Její využití ukazuje příklad 2.7. Rozlišujeme dva druhy rekurze:

nepravá rekurze - za rekurzivním voláním již nenásleduje žádná struktura algoritmu, ale jen konce struktur, do kterých je rekurzivní volání vnořeno.

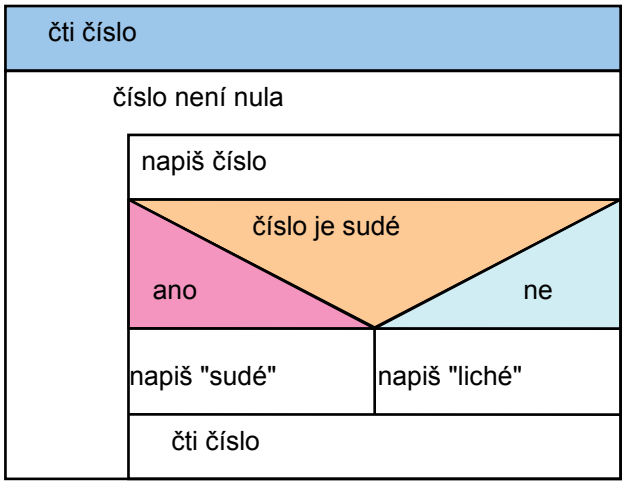
pravá rekurze - za rekurzivním voláním následuje alespoň jedna struktura na stejné nebo nižší úrovni vnoření.

S využitím rekurze souvisí některé nepříjemné problémy, zejména nebezpečí **přetečení zásobníku**. Budeme se jim podrobněji věnovat v kapitole věnované **dynamickým datovým strukturám**.

2.1.5 N-S diagram

Název **N-S diagram** je opět odvozen od jmen autorů, kterými jsou Isaac „Ike“ Nassi a Ben Schneiderman. Jsou velmi podobné kopenogramům a mají v zásadě shodné vlastnosti. Liší se pouze grafickým ztvárněním jednotlivých typů struktur [26].

Příklad 2.9:



Výsledný zápis N-S diagramu působí oproti kopenogramu kompaktním dojmem. Navíc zatímco kopenogramy jsou českou specialitou, byť je považujeme za přehlednější (při využití barevného zvýraznění), výhodou N-S diagramů je skutečnost, že byly publikovány v předním světovém časopise.

2.1.6 Strukturogram

Konstrukce **strukturogramů** vychází z postupu definovaného Michaelem Anthony Jacksonem v roce 1975. Vychází z poznatků strukturovaného programování a zjištění, že v algoritmu se nachází jen dva základní typy struktur [17][36].

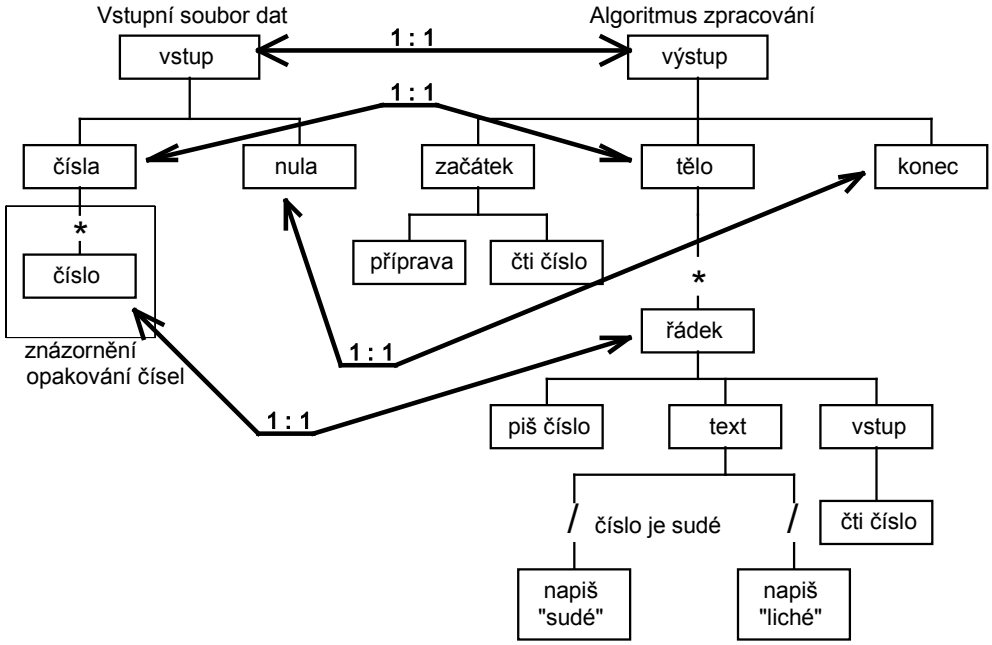
> **sekvence** (posloupnost operací) - tedy postupné provedení operací na stejné úrovni vnoření. Přitom je nevýznamné, zda jsou operace složeny z vnořené struktury dílčích operací či nikoliv.
>
> **selekce** (větvení) - provede se vybraná operace podle vyhodnocené podmínky.

Pokud vám v tomto výčtu schází **opakování** (cyklus, iterace), pak věžte, že je chápáno pouze jako zvláštní případ sekvence.

Kromě výrazně jednoduššího pohledu na základní struktury operací se Jacksonova technologie snaží pohlížet stejným způsobem na popis vstupního souboru dat i algoritmus pro jejich zpracování. Mezi těmito popisy jsou navíc jednoznačné vztahy, které usnadňují kontrolu správnosti algoritmu. Strukturogram autora navíc přímo vede k postupnému upřesňování algoritmu od hrubého návrhu až po velmi podrobný popis, odpovídající požadavkům použitého programovacího jazyka. Operace na stejné úrovni vnoření jsou umístěny na společné vodorovné linii. Upřesnění (rozvinutí) operace je znázorněno posloupností operací, které ji tvoří, zobrazenou níže. Tím se strukturogram liší od dříve uvedených popisů algoritmů, které čteme shora dolů. Strukturogram je nutno číst v zásadě zleva doprava, směr shora dolů udává postupné zpřesňování popisu činností.

Zápis strukturogramu, který budeme v dalším textu používat, vychází z Jacksonova zápisu, ale poněkud jej upravuje.

Příklad 2.10:



Jak vidíme z ukázky, algoritmus zpracování je chápán pouze jako popis převodu vstupního souboru dat na výstupní soubor.

Jacksonova technologie je aplikována v programu SGP [50], který umožňuje zápis algoritmu až do podrobností nutných pro vytvoření zdrojového textu pro překlad ve zvoleném programovacím jazyku. Jeho výhodou je skutečnost, že vůbec nepřipouští předčasné ukončení cyklu a skok do jiného místa algoritmu. Pokud je to potřeba, je možno jedině násilně opustit celý algoritmus (EXIT PROCEDURE). Tím je systém snadno aplikovatelný zejména pro jazyky podporující strukturovaný přístup, jako **PASCAL** (jazyk PASCAL vytvořil Niklaus Wirth původně pro výuku programování, velmi rychle si však získal širokou popularitu, jeho název je zkratkou z Programme Apliqué á la Sélection et la Compilation Automatique de la Littérature, která dává jméno francouzského filozofa Blaise Pascala 1623 * 1662) nebo C (univerzální programovací jazyk původně navržený Denisem Ritchiem v roce 1975 pro operační systém UNIX, dnes je známa spíše jeho nadmnožina C++ navržená Bjarnem Stroustrupem v Bellových laboratořích r. 1983).

Jako výhody strukturogramů jsou uváděny především snadná údržba algoritmů, a to i jiných autorů a přehledná dokumentace algoritmu. Zatím nenacházejí velké rozšíření, nejspíš z důvodů konzervativního přístupu programátorů, kteří si již zvykli využívat jiný způsob popisu algoritmu.

2.1.7 Pseudojazyk

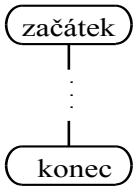
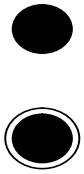
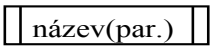
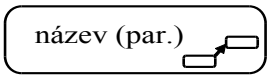
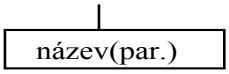
Použití **pseudojazyka** je vlastně návratem k textovému popisu algoritmu, ale na vyšší úrovni. Principy strukturovaného programování jsou zajištěny tím, že vnořenou strukturu algoritmu znázorňujeme odsazením jejího zápisu doprava. Každá dílčí struktura vždy začíná a končí speciálními příkazy. Jejich znění je zvoleno tak, aby se co nejvíce blížilo co největšímu počtu programovacích jazyků. Hlavní inspirací přitom byl **Strukturovaný BASIC**, který rozšiřuje známý programovací jazyk BASIC o programové struktury, které odpovídají zásadám **strukturovaného programování** a přitom zachovávají jednoduchost přístupu. V současné době (pět let do konce tisíciletí) získává velkou popularitu, zejména u produktů v prostředí OS-Windows. Pseudojazyk budeme v dalším textu využívat i my.

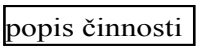
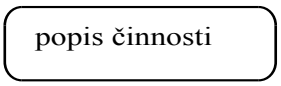
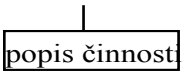
Příklad 2.11:

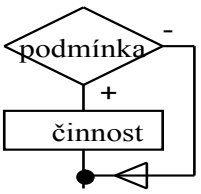
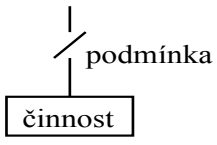
```
čti číslo
while číslo <> 0
  tiskni číslo
  if číslo je sudé
        tiskni "sudé"
  else
        tiskni "liché"
  end if
  čti číslo
end while
```

2.2 ZÁKLADNÍ STRUKTURY ALGORITMU

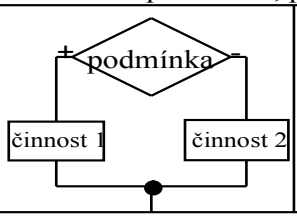
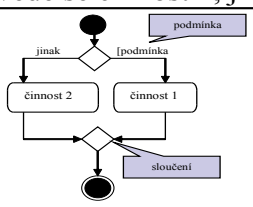
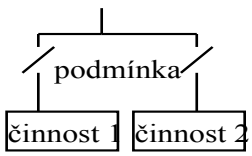
V této kapitole uvádíme srovnání základních struktur a jejich vyjádření vybranými způsoby zápisu.

Pseudojazyk	Vývojový diagram	Diagram aktivit	Strukturogram
Programový celek (rutina, podrogram, procedura a pod.) - definice			
název (parametry) . . . end název			jeden celý strukturogram
programový celek - použití (volání)			
název (parametry)			

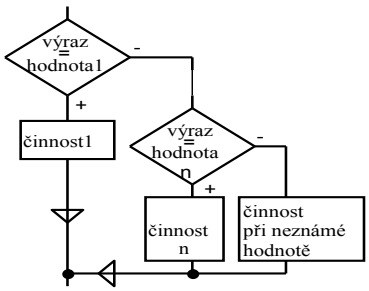
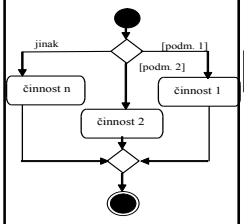
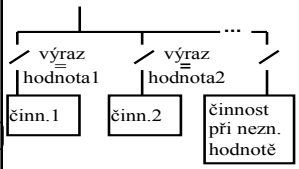
Pseudojazyk	Vývojový diagram	Diagram aktivit	Strukturogram
Provedení konkrétní činnosti			
popis činosti			
Podmíněná činnost (provádí se pouze pokud je splněna určitá podmínka)			

if podmínka
  podmíněná
  činnost
end if
		smysl má jen rozhodování	

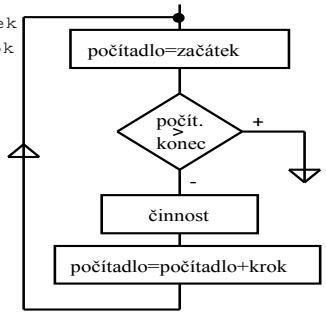
Rozhodování (pokud platí určená podmínka, provede se činnost 1, jinak činnost 2)			

if podmínka
  činnost 1
else
  činnost 2
end if
			

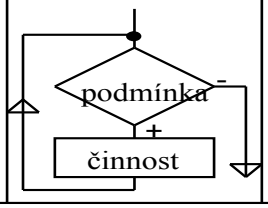
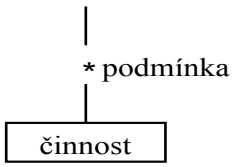
Pseudojazyk	Vývojový diagram	Diagram aktivit	Strukturogram
Větvení (podle hodnoty výrazu se provádí určená činnost)			

`case výraz=hodnota 1`
`  činnost 1`
`case výraz=hodnota 2`
`  činnost 2`
`.....`
`case else`
`  činnost při`
`  neznámé hodnotě`
`end case`
			

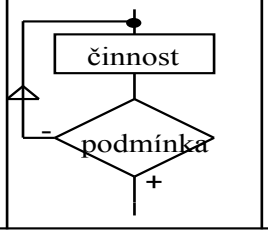
Opakování s pevným počtem opakování			

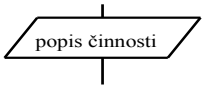
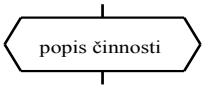
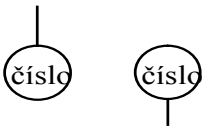
`for počítadlo=začátek`
`  to konec step krok`
`  činnost`
`end for`
		je nutno sestavit z ostatních značek podobně jako u vývojového diagramu	

Pseudojazyk	Vývojový diagram	Diagram aktivit	Strukturogram
Opakování s testem na začátku (dokud platí podmínka, činnost se opakuje - pokud na začátku opakování není podmínka splněna, činnost se vůbec neprovede)			

`while podmínka`
`  činnost`
`end while`
		je nutno sestavit z ostatních značek podobně jako u vývojového diagramu	

Opakování s testem na konci (opakování končí, pokud je splněna podmínka - i když podmínka platí již na začátku opakování, činnost se jednou provede)			

`repeat`
`  činnost`
`until podmínka`
 	
(v některých jazycích
`while`
`  činnost`
`end while podm.)`
		je nutno sestavit z ostatních značek podobně jako u vývojového diagramu	neexistuje

Pseudojazyk	Vývojový diagram	Diagram aktivit	Strukturogram
Vstupní nebo výstupní operace			
Jako každá jiná činnost		jako každá jiná aktivita	Jako každá jiná činnost
Přípravná činnost			
Jako každá jiná činnost		jako každá jiná aktivita	Jako každá jiná činnost
Spojka (činnost končí v jedné části algoritmu a pokračuje v jiné části)			
Neexistuje		Neexistuje	Neexistuje

2.3 ČLENĚNÍ ALGORITMU

Po uvedení výčtu základních struktur algoritmů se ještě chvíli zabývejme tvorbou dílčích celků algoritmů. Pro lepší přehlednost algoritmu je vhodné zpracovat algoritmus pro ucelenou část činnosti samostatně. Výsledkem je pak obvykle **rutina** realizující tuto konkrétní činnost (procedura, podprogram aj.).

Jako **rutinu** označujeme programový celek, který je po svém spuštění proveden celý až do konce, při opětovném spuštění jeho činnost začíná opět od začátku. Kromě toho existují programové celky, které provedou určenou činnost a v určitém místě ji ukončí, při dalším spuštění pokračují tam, kde přestaly. Označujeme je jako korutiny, v dalším textu se jimi nebudeme zabývat.

Většina programovacích jazyků, které podporují strukturované programování, odlišuje dva typy rutin:

funkce, které svým názvem vracejí specifikovanou vypočtenou hodnotu,

procedury, které žádnou hodnotu nevracejí.

2.4 PROMĚNNÉ

V algoritmech, které dále uvádíme, budeme pracovat s daty (informací) různých **datových typů**, jak budou popsány v následující kapitole včetně vysvětlení pojmů data a datový typ. Data ukládáme do proměnných, které mají určený datový typ a jejich obsahem je pak konkrétní informace. Základní operací s proměnnou je přiřazení hodnoty proměnné **přiřazovacím příkazem**. Tento příkaz můžeme také chápat jako binární operátor (budeme ho označovat "="). Jeho levý operand je název proměnné, pravý operand operace (výraz) jejíž výsledek se uloží jako obsah této proměnné, např.:

```
p = 1 + 1
Karel = Jméno + Příjmení + Titul
```

V programu a také v proceduře se můžeme setkat s několika typy proměnných:

globální proměnné - existují nezávisle na činnosti procedury, tedy již před jejím spuštěním a také po jejím ukončení. Představují jednu z možností, jak může informace vstoupit do procedury.

lokální proměnné - existují jen po dobu činnosti procedury, vytváří se při spuštění procedury, po jejím ukončení neexistují. Používají se jako pomocné proměnné pro realizaci činnosti procedury.

Pro předání informace proceduře je tedy možno použít jen globální proměnné. Nepříjemným důsledkem jejich použití je skutečnost, že každá změna obsahu globální proměnné zůstává v platnosti po skončení procedury. Tento problém je řešen zavedením pojmu **parametr** procedury. Je to lokální proměnná, která je uvedena v seznamu parametrů za názvem procedury. Při volání procedury současně určíme hodnoty všech jejích parametrů, takže procedura může při každém volání zpracovávat jinou informaci. Rozlišujeme přitom dva způsoby **předávání hodnot parametrů**:

předání **hodnotou** - proměnná má charakter **lokální proměnné**. Po opuštění procedury parametr zaniká. Touto cestou je tedy možno předat informaci do procedury, ale nikdy ne ven. Výhodou je skutečnost, že předáváním obsahu (hodnoty) proměnné je zaručeno, že obsah této proměnné se nezmění. Např.: hledej("Karel"), hledej(Jmeno+Příjmení).

předání **odkazem** - Tento způsob některé programovací jazyky vůbec nepřipouštějí a je nutno ho obcházet předáním hodnoty ukazatele na proměnnou. Předávaná proměnná je ztotožněna s parametrem, takže všechny operace s parametrem se projevují přímo na obsahu předávané proměnné. Výsledek činnosti tedy zůstává i po skončení činnosti procedury. Předání proměnné odkazem budeme zdůrazňovat slovem REF (reference) před jejím názvem v seznamu parametrů. Je zřejmé, že při volání procedury musí být parametru přidělena vždy proměnná, nikdy ne konstanta ani výsledek výrazu. Např.: hledej(Jméno).

Poznámka:

Aby nedošlo k nedorozumění, upozorňujeme, že pojmy globální a lokální proměnná jsou chápány z hlediska celého programu. Odkazem do procedury může být předána proměnná, která je v nadřazené proceduře lokální. Např.:

```
najdi (jméno)
  deklarace lokální proměnné počet
  hledej (jméno, počet)
  napiš jméno a počet výskytů.
end najdi

hledej (jm,REF poč)
  pro určené jméno (parametr jm) najde počet výskytů a
uloží do proměnné poč
end hledej.
```

Z hlediska uložení v paměti má rozlišení globálních a lokálních proměnných také velký význam. Blíže to bude uvedeno na začátku kapitoly věnované dynamickým datovým strukturám.

2.5 HODNOCENÍ ALGORITMU

Ke srovnání dvou algoritmů řešení stejného problému potřebujeme nějakým způsobem vyjádřit jejich výkonnost. Nejčastěji se k tomuto účelu využívá vyjádření **složitosti algoritmu**. [43][62]. Velmi zjednodušeně můžeme říci, že složitost algoritmu (**časová složitost algoritmu**) udává, jak roste počet operací algoritmu v závislosti na růstu množství

zpracovaných dat. Podobně se můžeme setkat také s **paměťovou složitostí algoritmu**, kterou se dále zabývat nebudeme.

Při vyjádření složitosti algoritmu můžeme vyjít od binární (bitové) složitosti operací. Za bitovou operaci přitom považujeme sčítání dvou jednobitových čísel (ať již s přenosem, nebo bez přenosu).

Sčítání dvou k-bitových čísel má binární složitost úměrnou délce k. Například sečtení následujících dvou čísel má binární složitost 8 :

```
  10101100          172
+ 11101010        + 234
----------        -----
 110010110          406
```

Podobně je tomu u jiných operací. Násobení dvou k a j bitových čísel představuje v nejhorším případě sčítání j sčítanců, po doplnění nulami dlouhých nejvýše $(k + j - 1)$ bitů. Pro dosažení výsledku musíme sčítat první s druhým, výsledek se třetím sčítancem atd. Tedy celkem nejvýše **(j - 1)** součtů **(k + j – 1)**-místných čísel. Pro odhad binární složitosti pak zjednodušíme :

$(j - 1)*(k + j - 1) < j*(k + 1) < 2*k*j$.

Binární složitost násobení je tedy úměrná **2*k*j**.

```
    101101        45 * 13 = 585
      1110
----------
    101101
    101101
    101101
----------
1001001001
```

Pro desítkovou soustavu platí, že n-místné dekadické číslo bude mít nejvýše k binárních znaků, kde je $\mathbf{k \leq \log_2(n) + 1}$. Sčítání dvou takových čísel má tedy binární složitost úměrnou hodnotě $\mathbf{\log_2(n) + 1}$.

Násobení n a m místného dekadického čísla má binární složitost úměrnou:

$$2*k*j < 2*[\log_2(n) + 1]*[\log_2(m) + 1] =$$

$$= 2*[\log_2(n)*\log_2(m) + \log_2(n*m) + 1] < 4*\log_2(n)*\log_2(m)$$

Binární složitost budeme zapisovat zkráceně: **f(n) = O(g(n))**, jestliže pro funkce **f** a **g** definované na množině přirozených čísel existuje konečná limita:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \qquad (2.1)$$

Můžeme tedy zapsat:

1. **n**-místné dekadické číslo má **O(log₂(n))** binárních číslic,
2. součet dvou **n**-místných dekadických čísel reprezentuje **nejvýše O(log₂(n))** bitových operací,
3. součin (podíl) **n** a **m** místného dekadického čísla vyžaduje **O(log₂(n)*log₂(m))** bitových operací.

U algoritmů jsou významné některé typy složitostí. Především je to polynomiální složitost. Pojem **polynomiální složitost** budeme používat u operací, u kterých výpočet funkce **f(n)** dekadického čísla o **n** místech, které má k dvojkových cifer vyžaduje:

$O(k^d)$ bitových operací, (2.2)

kde je d přirozené číslo.

Operace + - * /, tedy mají polynomiální složitost: pro + a - je $d = 1$,

pro * a / je $d = 2$.

Problémy, které jsou řešitelné pomocí algoritmů s polynomiální složitostí označujeme jako problémy třídy **složitosti P** nebo také **P-problémy** (případně PTIME) a považujeme je za efektivně řešitelné, nebo také řešitelné v polynomiálním čase.

Výpočet čísla **n! = 1*2*....*n** (faktoriál) má při použití postupného násobení binární složitost

$$O(n^2 * \log_2^2(n)) = O(2^{2k} * k^2) \quad (2.3)$$

Tento algoritmus tedy má **nepolynomiální složitost**. To mimo jiné znamená, že jeho složitost s rostoucím n roste velmi rychle. Takové algoritmy jsou v praxi velmi nepříjemné, vyžadují pro svoji realizaci velký výpočetní výkon. U problémů, které vyžadují k řešení algoritmy s nepolynomiální složitostí, hovoříme o **třídě složitosti NP** nebo také o NP-problémech (případně také exponenciálních problémech), jestliže správnost nalezeného řešení je možno ověřit v polynomiálním čase:

Speciálním případem jsou situace, kdy není možno ani ověřit správnost nalezeného řešení pomocí algoritmu s polynomiální složitostí. **Pak hovoříme o NP-úplných problémech**.

Známý **Eukleidův algoritmus** pro nalezení největšího společného dělitele celých čísel **(a, b)** má bitovou složitost **$O(\log_2^3(a))$ pro a > b**. Můžeme ho zapsat např.:

```
nejvetsi_delitel(a, b, REF d)
  a = ABS(a)
  b = ABS(b)
  while a > 0
        x = a
        a = b mod a
        b = x
  end while
  d := b;
end nejvetsi_delitel
```

Stejnou bitovou složitost má nalezení čísel u, v splňujících rovnost: **u*a + v*b = (a, b)**, nebo algoritmus na zjištění nesoudělnosti čísel, když je **(a, b) = 1**.

POUŽITÁ LITERATURA

[1] Arlow, J. & Neustadt, I. *UML a unifikovaný proces vývoje aplikací*. 1. Vyd. Brno, Computer Press, 2003, 388 s. ISBN 80-7226-947-X.

[2] Barton, D. P. & Pears, A. N. Application of Evolutionary Computation. In *Proceedings of First International Conference on Genetic Algorithms "Mendel '95"*. Red. Ošměra, P. Brno, VUT 1995, s. 15 - 21.

[3] Bayer, R. & McCreight, E. M. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1, 1972.

[4] Březina, T. *Informatika pro strojní inženýry I*. 1. vyd. Praha ČVUT 1991, 187 s.

[5] Brodský, J. & Skočovský, L. *Operační systém UNIX a jazyk C*. 1. vyd. Praha, SNTL 1989, 368 s.

[6] Cockburn, A. *Use Case – Jak efektivně modelovat aplikace*. 1. vyd. Brno, CP Books a.s., 2005, 262 s. ISBN 80-251-0721-3.

[7] Častová, N. & Šarmanová, J. *Počítače a algoritmizace*. 3. vyd. Ostrava, skriptum VŠB 1983, 190 s.

[8] Donghui Zhang. *B Trees*. Northeastern University, 22 pp. Dostupný z webu: http://zgking.com:8080/home/donghui/publications/books/dshandbook_BTree.pdf

[9] Drózd, J. & Kryl, R. *Začínáme s programováním*. Praha, Grada 1992, 312 s.

[10] Drozdová, V. & Záda, V. *Umělá inteligence a expertní systémy*. 1. vyd. Liberec, skriptum VŠST 1991, 212 s.

[11] Farana, R. *Zaokrouhlovací chyby a my*. Bajt 1994, č. 9, s 243 – 244.

[12] Flaming, B. *Practical data structures in C++*. New York, USA, Wiley, 1993.

[13] Hodinár, K. *Štandardné aplikačné programy osobných počítačov*. 1. vyd. Bratislava, Alfa 1989, 272 s.

[14] Holeček, J. & Kuba, M. *Počítače z hlediska uživatele*. Praha, SPN 1988, 240 s.

[15] Honzík, J. M., Hruška, T. & Máčel, M. *Vybrané kapitoly z programovacích technik*. 3. vyd. Brno, skriptum VUT 1991, 218 s.

[16] Hudec, B. *Programovací techniky*. Praha, ČVUT 1990.

[17] Jackson, M. A. *Principles of Program Design*. New York (USA), Academic Press 1975.

[18] Jandoš, J. *Programování v jazyku GW BASIC*. Praha, NOTO - Kancelářské stroje 1988, 164 s.

[19] Kačmář, D. *Programování v jazyce C++. Objektová a neobjektová rozšíření jazyka*. Ostrava, ES VŠB-TU 1995, 92 s.

[20] Kačmář, D. & Farana, R. *Vybrané algoritmy zpracování informací*. 1. vyd. Ostrava: VŠB-TU Ostrava, 1996. 136 s. ISBN 80-7078-398-2.

[21] Kaluža, J., Kalužová, L., Maňasová, Š. *Základy informatiky v ekonomice*. 1. vyd. Ostrava, skriptum VŠB 1992, 193 s.

[22] Kanisová, H. & Müller, M. *UML srozumitelně*. 1. vyd. Brno, Computer Press, 2004. 158 s. ISBN 80-251-0231-9.

[23] Kapoun, K. & Šmajstrla, V. *Základní fyzikální problémy - programy v jazyce BASIC a FORTRAN*. 1. vyd. Ostrava, skriptum VŠB 1987, 312 s.

[24] Kelemen, J. aj. *Základy umelej inteligencie*. 1. vyd. Bratislava, Alfa 1992, 400 s.

[25] Knuth, D. E. *The Art of Computer Programming*. Volumes 1-4A, 3rd ed. Reading, Massachusetts, Addison-Wesley, 2011, 3168 pp. ISBN 0-321-75104-3.

[26] Kopeček, I. & Kučera, J. *Programátorské poklesky*. Praha, Mladá fronta 1989, s. 150-155.

[27] Krček, B. & Kreml, P. *Praktická cvičení z programování. FORTRAN*. 1. vyd. Ostrava, skripta VŠB 1986, 199 s.

[28] Kučera, L. *Kombinatorické algoritmy*. 2. vyd. Praha, SNTL 1989, 288 s.

[29] Kukal, J. *Myšlením k algoritmům*. 1. vyd. Praha, Grada 1992, 136 s.

[30] Marko, Š. - Štěpánek, M. *Operačné systémy mikropočítačov SMEP*. 2. vyd. Bratislava/Praha, Alfa/SNTL 1988, 264 s.

[31] Medek, V. & Zámožík, J. *Osobný počítač a geometria*. 1. vyd. Bratislava, Alfa 1991, 256 s.

[32] Michalewicz, Z. *Genetic Algorithms + Data Structures = Evolution Programs*. 1. vyd. Heidelberg, Springer-Verlag Berlin 1992, 250 s.

[33] *Microsoft Developer Network*. Development Library January 1995. Microsoft Corporation 1995, CD - ROM.

[34] Molnár, Ľ. & Návrat, P. *Programovanie v jazyku LISP*. 1. vyd. Bratislava, Alfa 1988, 264 s.

[35] Molnár, Ľ. *Programovanie v jazyku Pascal*. Bratislava/Praha, Alfa/SNTL 1987, 160 s.

[36] Molnár, Z. *Moderní metody řízení informačních systémů*. Praha, Grada 1992, s. 211 - 221.

[37] Moos, P. *Informační technologie*, 1. vyd. Praha, ČVUT 1993, 200 s.

[38] Nešvera, Š., Richta, K. & Zemánek, P. *Úvod do operačního systému UNIX*. 1. vyd. Praha, ČVUT 1991, 185 s.

[39] Olehla, J. & Olehla, M. aj. *BASIC u mikropočítačů*. 1. vyd. Praha, NADAS 1988, 386 s.

[40] Ošmera, P. Použití genetických algoritmů v neuronových modelech. In *Sborník konference "EPVE 93"*. Brno VUT 1993, s. 88 - 95.

[41] Paleta, P. *Co programátory ve škole neučí aneb Softwarové inženýrství v reálné praxi*. 1. vyd. Brno, Computer Press, 2003, 337 s. ISBN 80-251-0073-1.

[42] Petroš, L. *Turbo Pascal 5.5* – Uživatelská příručka. 1. vydání. Zlín, MTZ 1990, s. 20 – 21.

[43] Plávka, J. *Algoritmy a zložitosť*. Košice, TU Košice, 1998. ISBN 80-7166-026-4.

[44] Podlubný, I. *Počítať na počítači nie je jednoduché*. PC World, 1994, č. 2, s. 112 – 115.

[45] Rawlins, G. J. E. *Compared to what – an introduction to the analysis of algorithms*. Computer Science Press, New York, 1992.

[46] Reverchon, A. & Ducamp, M. *Mathematical Software Tools in C++*. West Sussex (England) John Wiley & Sons Ltd. 1993, 507 s.

[47] Rychlík, J. *Programovací techniky*. České Budějovice, KOPP 1992, 188 s.

[48] Sedgewick, R. *Algorithms*. 1st ed. Addison-Wesley. ISBN 0-201-06672-6.

[49] Sirotová, V. *Programovacie jazyky*. 1. vyd. Bratislava, skriptum SVTŠ 1985, 138 s.

[50] Soukup, B. SGP verze 2.30. *Referenční a uživatelská příručka systému*. Uherské hradiště, SGP Systems 1991, s. 25-55.

[51] Synovcová, M. *Martina si hraje s počítačem*. 1. vyd. Praha, Albatros 1989, 144 s.

[52] Šarmanová, J. *Teorie zpracování dat*. Ostrava, FEI VŠB-TU Ostrava, 2003, 160 s.

[53] Šmiřák, R. *Unified Modeling Language*. Softwarové noviny, 2004, č. 12, s. 76 – 77.

[54] Tichý, V. *Algoritmy I.* Praha, FIS VŠE v Praze, 2006, 190 s. ISBN 80-245-1113-4.

[55] Vejmola, S. *Hry s počítačem*. 1. vyd. Praha, SPN 1988, 256 s.

[56] Virius, M. *Základy algoritmizace*. Praha, ČVUT 2008, 265 s. ISBN 978-80-01-04003-4.

[57] Víteček, A. aj. *Využití osobních počítačů ve výuce*. 1. vyd. Ostrava, ČSVTS FS VŠB Ostrava 1986, 202 s.

[58] Vítečková, M., Smutný, L., Farana, R. & Němec, M. *Příkazy jazyka BASIC*. Ostrava, katedra ASŘ VŠB Ostrava 1989, 44 s.

[59] Vlček, J. *Inženýrská informatika*. 1. vyd. Praha, ČVUT 1994, 281 s.

[60] Wirth, N. *Algoritmy a štruktúry udajov*. 2. vydání. Bratislava, Alfa 1989, s. 19 – 89.

[61] *Základy algoritmizace a programové vybavení*. 2. vyd. Praha, Tesla Eltos 1986, 168 s.

[62] Zelinka, I., Oplatková, Z., Šeda, M., Ošmera, P. & Včelař, F. *Evoluční výpočetní techniky. Principy a aplikace*. 1. vyd. Praha, BEN – Technická literatura, 2009. ISBN 978-80-7300-218-3.