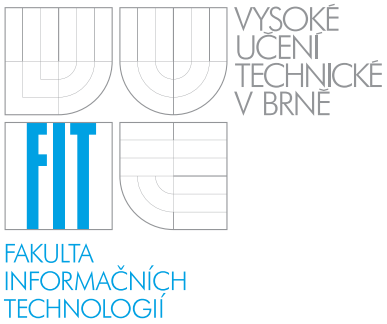


Principy programovacích jazyků a objektově orientovaného programování IPP – I

Dušan Kolář
Ústav informačních systémů
Fakulta informačních technologií
VUT v Brně

Květen '03 – Říjen '06
Verze 1.1

Tento učební text vznikl za podpory projektu „Zvýšení konkurenceschopnosti IT odborníků – absolventů pro Evropský trh práce“, reg.č. CZ.04.1.03/3.2.15.1/0003. Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky.

Abstrakt

Programovací jazyky stojí mezi programátorem a počítačem, tedy mezi člověkem a strojem. Tak, jako lidé mezi sebou komunikují prostřednictvím jazyků, které mají svá pravidla, taje i různé zkratky a slangové výrazy, tak i pro komunikaci s počítačem, pokud chceme docílit toho, aby vykonával svou činnost dle našich představ, musíme použít vhodný programovací jazyk a naučit se jeho pravidlům, tajům, trikům apod.

Tato publikace shrnuje základní principy různých tříd programovacích jazyků a to nejen z pohledu programátora, ale i z pohledu toho, kdo vytváří překladač, či interpret takového jazyka. Koncentruje se však především na jazyky imperativní: nestrukturované, blokově strukturované a modulární.

Věnováno Mirkovi

Vřelé díky všem, kdo mě podporovali a povzbuzovali při práci na této publikaci.

Obsah

Notace a konvence použité v publikaci

Každá kapitola a celá tato publikace je uvozena informací o čase, který je potřebný k zvládnutí dané oblasti. Čas uvedený v takové informaci je založen na zkušenostech více odborníků z oblasti a uvažuje čas strávený k pochopení prezentovaného tématu. Tento čas nezahrnuje dobu nutnou pro opakované memorování paměťově náročných statí, neboť tato schopnost je u lidí silně individuální. Příklad takového časového údaje následuje.

Čas potřebný ke studiu: 2 hodiny 15 minut

Podobně jako dobu strávenou studiem můžeme na začátku každé kapitoly či celé publikace nalézt cíle, které si daná pasáž klade za cíl vysvětlit, kam by mělo studium směřovat a čeho by měl na konci studia dané pasáže studující dosáhnout, jak znalostně, tak dovednostně. Cíle budou v kapitole vypadat takto:

Cíle kapitoly

Cíle kapitoly budou poměrně krátké a stručné, v podstatě shrnující obsah kapitoly do několika málo vět, či odrážek.

Poslední, nicméně stejně důležitý, údaj, který najdeme na začátku kapitoly, je průvodce studiem. Jeho posláním je poskytnout jakýsi návod, jak postupovat při studiu dané kapitoly, jak pracovat s dalšími zdroji, v jakém sledu budou jednotlivé cíle kapitoly vysvětleny apod. Notace průvodce je taktéž standardní:

Průvodce studiem

Průvodce je často delší než cíle, je více návodný a jde jak do šířky, tak do hloubky, přitom ho nelze považovat za rozšíření cílů, či jakýsi abstrakt dané stati.

Za průvodcem bude vždy uveden obsah kapitoly.

Následující typy zvýrazněných informací se nacházejí uvnitř kapitol, či podkapitol a i když se zpravidla budou vyskytovat v každé kapitole, tak jejich výskyt a pořadí není nijak pevně definováno. Uvedení logické oblasti, kterou by bylo vhodné studovat naráz je označeno slovem „Výklad“ takto:

Výklad

Důležité nebo nové pojmy budou definovány a tyto definice budou číslovány. Důvodem je možnost odkazovat již jednou definované pojmy a tak významně zeštíhlet a zpřehlednit text v této publikaci. Příklad definice je uveden vzápětí:

Definice!

Definice 0.0.1 *Každá definice bude využívat* poznámku na okraji *k tomu, aby upozornila na svou existenci. Jinak je možné zvětšený okraj použít pro vpisování poznámek vlastních. První číslo v číselné identifikaci definice (či algoritmu, viz níže) je číslo kapitoly, kde se nacházela, druhé je číslo podkapitoly a třetí je pořadí samotné entity v rámci podkapitoly.*

Pokud se bude někde vyskytovat určitý postup, či konkrétní algoritmus, tak bude také označen, podobně jako definice. I číslování bude mít stejný charakter a logiku.

Algoritmus!

Algoritmus 0.0.1 *Pokud je čtenář zdatný v oblasti, kterou kapitola, či úsek výkladu prezentuje, potom je možné skočit na další oddíl stejné úrovně.*

Přeskoky v rámci jednoho oddílu však nedoporučujeme.

V průběhu výkladu se navíc budou vyskytovat tzv. řešené příklady. Jejich zadání bude jako jakékoliv jiné, ale kromě toho budou obsahovat i řešení s nástinem postupu, jak takové řešení je možné získat. V případě, že řešení by vyžadovalo neúměrnou část prostoru, bude vhodným způsobem zkráceno tak, aby podstata řešení zůstala zachována.

Řešený příklad

Zadání: Vyjmenujte typy rozlišovaných textů, které byly doposud v textu zmíněny.
Řešení: Doposud byly zmíněny tyto rozlišené texty:

- Čas potřebný ke studiu
- Cíle kapitoly
- Průvodce studiem
- Definice
- Algoritmus
- Právě zmiňovaný je potom Řešený příklad

Některé informace mohou být vypíchnuty, či doplněny takto bokem.

V závěru každého výkladové oddílu se potom bude možné setkávat s opětovným zvýrazněním důležitých pojmů které se v dané části vyskytly a případně s úlohou, která slouží pro samostatné prověření schopností a dovedností, které daná část vysvětlovala.

Pojmy k zapamatování

- Rozlišené texty
- Mezi rozlišené texty patří: čas potřebný ke studiu, cíle kapitoly, průvodce studiem, definice, algoritmus, řešený příklad.

Úlohy k procvičení:

Který typ rozlišeného textu se vyskytuje typicky v úvodu kapitoly. Který typ rozlišeného textu se vyskytuje v závěru výkladové části?

Na konci každé kapitoly potom bude určité shrnutí obsahu a krátké resumé.

Závěr

V této úvodní stati publikace byly uvedeny konvence pro zvýraznění rozlišených textů. Zvýraznění textů a pochopení vazeb a umístění zvyšuje rychlost a efektivnost orientace v textu.

Pokud úlohy určené k samostatnému řešení budou vyžadovat nějaký zvláštní postup, který nemusí být okamžitě zřejmý, což lze odhalit tím, že si řešení úlohy vyžaduje enormní množství času, tak je možné nahlédnout k nápovědě, která říká jak, případně kde nalézt podobné řešení, nebo další informace vedoucí k jeho řešení.

Klíč k řešení úloh

Rozlišený text se odlišuje od textu běžného změnou podbarvení, či ohraničením.

Možnosti dalšího studia, či možnosti jak dále rozvíjet danou tématiku jsou shrnuty v poslední nepovinné části kapitoly, která odkazuje, ať přesně, či obecně, na další možné zdroje zabývající se danou problematikou.

Další zdroje

Oblasti, které studují formát textu určeného pro distanční vzdělávání a samostudium, se pojí se samotným termínem distančního či kombinovaného studia (distant learning) či tzv. e-learningu.

Kapitola 1

Úvod

Čas potřebný ke studiu: 20 hodin
Tento čas reprezentuje dobu pro studium celého modulu.

Údaj je pochopitelně silně individuální záležitostí a závisí na současných znalostech a schopnostech studujícího. Proto je vhodné jej brát jen orientačně a po nastudování prvních kapitol si provést vlastní revizi.

Cíle modulu

Cílem modulu je seznámit studujícího s širokou oblastí programovacích jazyků. Modul se zaměřuje na klasifikaci jazyků z nichž si vybírá jazyky imperativní. Ty studuje z hlediska možností, typických vlastností, možnosti užití apod., přičemž dává nahlédnout na klíčové prvky, které se pojí s implementací překladačů či interpretů takových jazyků. *(Z tohoto pohledu je možné modul považovat za do jisté míry encyklopedický.)*
Po ukončení studia modulu:

- budete schopni klasifikovat programovací jazyky z různých hledisek;
- budete schopni správně volit jazyk pro daný problém;
- budete schopni (v případě dalších znalostí z jiných oborů) implementovat překladače či interprety takových jazyků;
- bude mít znalosti o formálních aparátech spojených s programovacími jazyky, které tak můžete samostatně dále rozvíjet.

Průvodce studiem

Modul začíná celkovou klasifikací programovacích jazyků z různých hledisek. Dále potom postupuje podle jednoho zvoleného kritéria a předkládá vlastnosti jazyků takových typů z hlediska uživatelského i implementačního. Pozastavuje se nad možnými formalismy spojenými s takovými jazyky.
Postup je od jazyků s menšími výrazovými schopnostmi ke složitějším a bohatším. Proto je možné, pokud je vám nějaká oblast blízká, přejít vpřed na další oblast.
Pro studium modulu je důležité být aktivním programátorem a nově nabyté znalosti prakticky verifikovat. Aktivní užití některého z typických představitelů dané kategorie se tak stává jasnou výhodou. Nestačí pochopit jen to, jak jazyk vypadá zvenčí, ale i co se skrývá za tím, že se chová tak, jak se chová.

Tento modul úzce souvisí s moduly, které předkládají problematiku teorie formálních jazyků a automatů a dále problematiku zpracování formálních jazyků (překladačů/interpretů). Tuto oblast dále rozvíjí řadou praktických pohledů na vlastnosti jazyků i na možnosti a způsoby, jak lze vybrané vlastnosti implementovat.

Návaznost na předchozí znalosti

Pro studium tohoto modulu je tedy nezbytné , aby studující měl základní znalosti ze zpracování formálních jazyků, architektury počítačů, assembleru (jazyk symbolických instrukcí) a, jak již bylo zmíněno, byl aktivním programátorem alespoň v jednom vyšším programovacím jazyce (znalost ladění na úrovni strojového kódu apod.).

1.1 Koncepce modulu

Následující kapitola provádí základní klasifikaci programovacích jazyků a vymezuje význam jednotlivých pojmů, které nemusejí být nutně nové.

Další kapitoly se potom postupně zabývají jazyky nestrukturovanými, jazyky strukturovanými a modulárními. Výklad se přitom orientuje zejména na jazyky imperativní. Tak, jak se postupně mění typ jazyka, tak se stává jazyk bohatším na výrazové schopnosti, což klade vyšší nárok na analýzu i syntézu a model programu za běhu. Přitom řada, zejména pozitivních vlastností zůstává.

Koncepty zmíněny dříve se tak dále strukturují, dodávají se nové charakteristiky, pravidla se celkově zpřísňují. Studium modulu tedy vyžaduje přísně sekvenční přístup. Informace obsažené jsou založeny na řadě zkušeností a lze je získat i osobní aktivitou, která je však, v porovnání se studiem tohoto modulu, časově mnohem náročnější. Dochází totiž ke kombinaci znalostí z několika vymezených oborů.

1.2 Potřebné vybavení

Pro studium a úspěšné zvládnutí tohoto modulu není třeba žádné speciální vybavení. Je však *vhodné* doplnit text osobní zkušeností s nějakým reprezentantem dané skupiny jazyků. Potom je ovšem nutné mít přístup k odpovídající výpočetní technice a k zdrojům nabízející překladače, či interprety daných jazyků, což je v současnosti typicky Internet.

Další zdroje

Programovací jazyky jsou jednotlivě představovány v řadě publikací. Podobně i formalismy a otázky spojené s během programu, či jeho analýzou můžeme najít v řadě publikací. Zatímco u jazyků jako takových najdeme řadu i českých titulů (např. Herout: Učebnice jazyka C, či Herout: Učebnice jazyka Java, nebo Bloch: Java efektivně), tak u ostatních jsou tituly typicky anglické (např. Aho, Sethi, Ullman: Compilers—Principles, Techniques, and Tools; Sebesta: Concepts of Programming Languages; Reynolds: Theories of Programming Languages). Proto jsou v textu u klíčových termínů uváděny i odpovídající anglické termíny, které by měly umožnit rychlé vyhledání relevantních odkazů na Internetu, který je v tomto směru bohatou studnicí znalostí, jenž mohou vhodně doplnit a rozšířit studovanou problematiku.

Kapitola 2

Klasifikace jazyků

Tato kapitola seznamuje s různými pohledy a výklady klasifikace programovacích jazyků. Definuje různé typy názvosloví a výklady k nim.

Čas potřebný ke studiu: 4 hodiny 30 minut.

Cíle kapitoly

Cílem kapitoly je zvládnout orientaci v základních pojmech, které se váží ke klasifikaci programovacích jazyků a mít schopnost na základě těchto pojmů daný programovací jazyk zařadit, klasifikovat.

Průvodce studiem

Před tím, než se budeme zabývat jednotlivými typy programovacích jazyků podrobně, tak je třeba si vybudovat jistou taxonomii a názvosloví, aby potom bylo jasné, jaký termín se v daných souvislostech jak chápe.
Ke studiu této kapitoly není třeba žádného speciálního vybavení. Jako u většiny ostatních se jedná o to si zapamatovat danou terminologii a důležité a podstatné věci v kapitole. Pro hlubší studium, nebo bližší objasnění termínů je vhodné využít informací na Internetu. Potom stačí počítač s prohlížečem WWW stránek a připojení k Internetu. Pro vyhledávání je vhodné využít anglickou terminologii, která je při zavádění termínů uváděna v závorkách.

Obsah

Výklad

2.1 Co to je programovací jazyk

S programovacími jazyky se pojí značné množství pojmů, než se však dostaneme přímo k programovacím jazykům, musíme se zabývat tím, kde je programovacích jazyků potřeba.

Definice 2.1.1 *Pojmem* programování *(proces tvorby programu) budeme rozumět takovou činnost, která jistý postup, algoritmus, typicky myšlenkový, převádí na posloupnost elementárních úkonů nějakého stroje, typicky počítače. Přitom dochází k uložení tohoto postupu tak, aby jej stroj mohl opakovat periodicky.* **Definice!**

Definice 2.1.2 Programátor *je ten, kdo proces tvorby programu realizuje.* **Definice!**

Tento postup zdaleka není jednoduchý, jak by se na první pohled mohlo zdát. Úzce souvisí s algoritmizací jako takovou, která je bez pochyby náročným procesem. Mezi typické vlastnosti procesu tvorby programu, programování patří:

- vysoká náročnost na čas a zdroje — tvorba programu vyžaduje přítomnost lidí, materiální zabezpečení (jak technické, tak programové) nejen po dobu vytváření programu, ale i po dobu jeho života, přitom všechny typy zdrojů musejí splňovat určité kvalitativní a kvantitativní požadavky dané fází zpracování a rozsahem i náročností projektu;

- znalost programovacího jazyka jako takového není dostačující — znalost syntaxe a sémantiky nezaručuje to, že nějaký postup správně nakóduji, souvislost je pouze okrajová;

- specifikace požadavků na cílový produkt není bezchybná — specifikace často neobsahuje požadavky na chování tak detailně, aby bylo možné podle nich řešit všechna sporná místa, kromě toho dochází k jejich změně často i poté, co programování bylo zahájeno, navíc u některých požadavků je těžko měřitelné, jestli jsou splněny (např. systém je uživatelsky přátelský, má rychlé odezvy apod.), atd.;

- přeceňování/podceňování schopností/možností zdrojů — dochází k podceňování složitosti projektů a naopak se přeceňuje schopnost programátorů (syndrom 90-10, či 80-20, kdy máme pocit, že je hotovo 80% veškeré práce, ale je ve skutečnosti hotovo jen 20% všech prací a přitom do konce vymezeného času projektu zbývá už jen 20% času), u některých činností (analýza apod.) dochází k podcenění jejich významu, přitom se přeceňuje význam programových nástrojů užitých při realizaci apod.;
- ostré termíny — velice často jsou požadavky na realizaci vázány ostrými termíny, které jsou umocněny sklonem programátorů přeceňovat své schopnosti;
- atd.

Pojem programování se dále váže jen na počítače!

I když jsme naznačili, že programování se nemusí týkat jen výpočetní techniky, tak od tohoto místa dále to tak budeme chápat.

Co to tedy je programovací jazyk, který se při programování používá?

Definice!

Definice 2.1.3 Programovací jazyk *(definice I) je prostředník mezi běžnou řečí a posloupností typicky binárních číslic.*

Definice!

Definice 2.1.4 Programovací jazyk *(definice II) je konečná množina příkazů (převáděných do binární formy), která má specifickou syntaktickou strukturu a pevně a přesně vymezenou sémantiku.*

Proč se vůbec programovací jazyky používají? Proč vznikly? Z definice 2.1.3 vyplývá, že je to prostředník mezi přirozenou řečí a binárním kódem.

- Nevýhodou přirozeného jazyka je:
  - analýza přirozeného jazyka tak, aby byla jednoznačně převoditelná do binární formy, je příliš náročná (doposud neexistuje) — sémantika i syntaxe přirozeného jazyka je velmi komplexní, o analýze přirozené mluvy ani nemluvě;
  - nejednoznačnost, slovní popisy příliš zdlouhavé — všimněte si, že i ve specifikaci požadavků na program se často používají různé formalismy, aby se nejednoznačnost a složitost vyjadřování přirozeným jazykem odstranila;
  - a další.

- Nevýhodou binárního jazyka je:
  - příliš náročný na zapamatování — uvážíme-li i poměrně jednoduchý procesor pracující s šířkou instrukčního slova 16 bitů, tak máme 65536 kombinací, u kterých si musíme pamatovat význam, i když by tato šíře v praxi asi nebyla využita cele, tak i přesto by takové číslo bylo vysoké;
  - nevhodný pro složité problémy — pomineme-li předchozí bod, tak kódování složitých problémů by bylo extrémně náročné (jednoduchá možnost chyby, rozsah, apod.);
  - nevhodný pro rychlou a efektivní tvorbu programů — částečně to souvisí s předchozím bodem, znovu využití kódu, udržovatelnost, rychlá orientace, nebo jen rychlé psaní, to jsou vše pojmy na hony vzdálené binárnímu jazyku;
  - a další.

Při programování v programovacím jazyce tak vzniká jistý specifický text, který proces tvorby programu zefektivňuje a zrychluje. Vzniká tak program:

Definice 2.1.5 Počítačový program *(dále často jen program) je zápis v programovacím jazyce, který je abstrakcí reality.* **Definice!**

- *Implementuje abstraktní model — myšlenkový postup aplikovatelný v realitě je abstrahován a přizpůsoben možnostem počítače.*
- *Abstrahuje model počítače/procesoru — program zastiňuje konkrétní podobu výpočtu a jeho realizaci na cílové platformě.*

Historie programovacích jazyků

počátek 50. let

Programovací jazyky se objevují v 50. letech 20. století. Zprvu se jedná jen o symbolické pojmenování binárních kódů operací cílové architektury či paměťových míst počítače. Pojem podprogram tak, jak ho známe dnes, v podstatě neexistuje. Jedná se pojmenované skupiny operací, které se do místa užití vkládají na textové úrovni (makra, textová záměna). Později se vytváří možnosti užití podprogramu tak, jak jej známe dnes, a odděluje se *implementace* podprogramu, to jest popis toho, co podprogram dělá, a *volání* (invocation) podprogramu, to jest jeho rozhraní a jeho projev navenek (výsledek operace).

počátek 60. let

Od počátku 60. let 20. století dochází postupně k růstu abstrakce, která se používá v programovacích jazycích (Fortran, Algol, Cobol, LISP). Roste programátorský komfort. V tomto období se objevují

první abstrakce na datové úrovni i v oblasti řízení běhu programu. Je tak možné definovat *datový model* daného programu a podobně *výpočetní model* programu. Zatímco první definuje, jak jsou které prvky reality modelovány na úrovni dat, jaká je jejich vzájemná korespondence a význam, tak druhý model definuje to, jak se jednotlivé příkazy jazyka vzájemně provazují, jak to dochází k větvení výpočtu, jak dochází k předávání řízení mezi jednotlivými částmi apod.

počátek 70. let

V 70. letech 20. století je patrný trend, který vede k zjednodušení programovacích jazyků a jejich větší univerzálnosti — snižuje se aplikační závislost. Objevují se jazyky jako Pascal, jazyk C, Algol-W, Euclid a další. Abstrakce datové i řídicí se opět posouvají na vyšší úroveň — jazyky blokově strukturované (datově i řízením), moduly. Využití jazyků i nasazení souvisí s pronikáním výpočetní techniky od vojenského a silně specializovaného využití do širšího výzkumu a dalších odvětví.

počátek 80. let

Během 80. let 20. století se potom dále rozvíjejí již dříve nastolené trendy. K tomu se objevují nové prvky a v rámci kategorií jazyky s novými vlastnostmi do té doby nevídanými — objektové jazyky, automatické odvození typu proměnných i funkcí, nové strategie vyhodnocení, vylepšení modularity a jiné. V té době se objevují na scéně nebo se výrazně uplatňují jazyky jako Ada, Modula-2, Smalltalk, C++, ML, Miranda a další.

počátek 90. let

V 90. letech 20. století dochází k precizaci již vytvořených konceptů a nové se již dále neobjevují. Dochází k tomu, že jazyky s ač obecnými vlastnostmi z hlediska možnosti tvorby libovolných algoritmů se profilují díky jiným vlastnostem — knihovny, překladač/interpret, atd. Zejména rozvoj Internetu a vestavěných systémů profiluje jazyky a stojí za dalším rozvojem. Vznikají, či do povědomí se dostávají jazyky jako Java, Haskell, AWK, Perl, Python, Javascript, VisualBasic a další.

současnost a budoucnost

Trendy konce 20. století trvají dodnes. Budoucnost bude určitě ovlivněna rozvojem stávajících a nástupem dalších odvětví kde se počítače uplatní. Roli bude hrát výkonnost počítačů, velikost, spotřeba elektrické energie, fyzikální limity a jiné. Předpovídat se dá jen těžko, neboť oblast výpočetní techniky a informatiky je stále velmi dynamická a stále poskytuje dostatečný prostor pro nové myšlenky.

jazyk a program

Od tohoto místa dále budeme *program* výhradně pod pojmem (nebude-li uvedeno jinak) myslet počítačový program a pod pojmem *jazyk* výhradně programovací jazyk, nebo jiný typ počítačově zpracovávaného jazyka, který slouží pro zpracování a transformaci vstupních dat na výstupní.

Pojmy k zapamatování

- Programování, programátor
- Programovací jazyk, jazyk
- Počítačový program, program
- Implementace/volání podprogramu
- Abstrakce datová, řízení toku běhu programu
- Datový a výpočetní model

Výklad

2.2 Dělení jazyků dle abstrakce

Správná volba programovacího jazyka je jedním z důležitých rozhodnutí při řešení daného projektu. I když program definuje *výpočet* realizovaný cílovou architekturou, tak svým charakterem nemusí žádný výpočet popisovat — např. program opisující text na obrazovku. V současnosti tak existují jazyky několika typů:

- univerzální programovací jazyky (general purpose languages) — jazyk C, Pascal, Java, C#, atd., jedná se o jazyky obecného použití, bez daného zacílení;
- specializované programovací jazyky — APL (Array/A Programming Language), jazyky řídicích automatů, atd.;
- jazyky pro popis integrovaných obvodů — VHDL, System C a jiné;
- jazyky pro sazbu textu — např. rodina jazyků TEX;
- a další typy.

V textu se dále soustředíme na programovací jazyky a zejména na univerzální programovací jazyky, i když řada jejich vlastností je aplikovatelná i do specializovaných programovacích jazyků, či do jiných kategorií.

Jedním z kritérií, jak lze jazyky dělit, je úroveň abstrakce a způsob zpracování dat. Další možností je potom abstrakce na úrovni řízení.

abstrakce **dat** a **řízení**

Abstrakce dat

Z hlediska přístupu k dělení a manipulaci s daty lze rozlišit tyto kategorie jazyků:

- jazyky strojové/assemblery;
- jazyky vyšší úrovně (než strojové) (Fortran, Cobol, Algol 60);
- tzv. univerzální jazyky — PL/I;
  Pozn: *V tomto případě se jedná o terminus technicus* univerzální jazyk*, je spojován právě s jazykem PL/I, nezaměňovat s jazyky pro obecné užití.*
- blokově strukturované jazyky;
- modulární blokově strukturované jazyky;
  Pozn: *Jedná se o jazyky s vlastnostmi blokově strukturovaných jazyků, které umožňují skrývat implementaci v modulech.*
- objektově orientované jazyky;
- jazyky rozšiřující datové paradigma.

Strojově orientované jazyky

Veškerá data u strojově orientovaných jazyků jsou skupiny bitů či bajtů, které jsou shlukovány dle vlastností cílové architektury (16bitové procesory, 12bitové procesory apod.). Pokud taková data nereprezentují přímo typy, které podporuje cílová architektura (celá čísla, desetinná čísla, bitová pole, atd.), tak je jejich vyšší sémantika podchycena na úrovni zcela mimo daný jazyk — v hlavě programátora, v dokumentaci, v poznámce v textu programu.

Typické operace, se kterými se můžeme setkat jsou aritmetické (sčítání/odčítání, násobení/dělení) a bitové operace (posuny, logické součty, negace apod.). Podpora vyšší abstrakci dat není žádná, podpora základních typů je specifická dle cílové architektury. Programy jsou tak specifické pro konkrétní počítač.

První jazyky vyšší úrovně

První jazyky vyšší úrovně stále nabízejí jen jednoduché datové typy, ale na rozdíl od předchozí kategorie zde dochází k abstrakci od cílového systému — je skryta implementace, jsou známy jen obecné vlastnosti, množina dostupných operací je typicky vyšší než ta, co je přímo dostupná na cílové architektuře. Nicméně orientace na cílovou

architekturu je stále vysoká (velikost a rozsah typů kopíruje cílovou platformu apod.).

Kromě toho jsou tyto jazyky velmi poplatné tomu, pro jaký druh aplikací jsou použity — vojenské, vědecko-technické, zpracování dat. Žádný z nich není určen pro obecné použití. Často mají definovanou pevnou strukturu zdrojového textu (např. Fortran).

ukázka práce s daty

Složitější datové vztahy jsou obcházeny, např. seznam studentů je implementován jako několik polí, z nichž jedno nese jméno, druhé příjmení, další věk, apod.

Tento stav vyústil v definici následující, prakticky jednoprvkové kategorie.

Jazyk PL/I

Jazyk PL/I definuje velké množství různých datových typů různých vlastností — od jednoduchých po velmi komplexní. Snaží se být prvním jazykem obecného použití (proto se mu říká univerzální jazyk).

ukázka práce s daty

Obrovskou nevýhodou tohoto jazyka, z dnešního pohledu, je to, že není možné definovat další, uživatelské datové typy. Jediná možnost je použít ty existující. Pokud mám datovou entitu pro popis osoby a v té mi chybí např. rodné číslo, tak tuto datovou entitu nemohu o novou složku rozšířit. Další nevýhodou je neexistence klíčových slov, která i z dnešního pohledu značně zesložiťuje analýzu programů zapsaných v tomto jazyce.

Tento typ jazyka nemá v podstatě pokračovatele a byl nahrazen další, úspěšnější skupinou, která řadu vlastností jazyka PL/I dovedla získat jiným, efektivnějším způsobem a jejíž vlastnosti přetrvávají v programovacích jazycích dodnes.

Blokově strukturované jazyky

Typickými představiteli může být jazyk Pascal (jak jej definoval prof. Wirth) a určitou podmnožinou vlastností i jazyk C (pokud pomineme modularitu, existenci knihoven).

Tato skupina jazyků umožňuje definovat složitější datové (i řídicí) struktury pomocí jednoduchých konstrukcí, které lze spojovat i vnořovat.

první jazyky pro softwarové inženýrství

Tato skupina jazyků jako první umožňuje použít a využít návrhové metodologie a prvky softwarového inženýrství. Je to díky tomu, že je možné prvně definovat vhodné datové abstrakce a potom teprve podle nich implementovat program. Doposud se totiž implementace odehrávala naopak — hledala se nejvhodnější datová abstrakce dostupná v daném jazyce, aby co nevhodněji kopírovala požadavky.

Programy v těchto jazycích se stávají přehlednější, čitelnější. Je to dáno tím, že více odrážejí zpracovávanou realitu na datové úrovni, sémantika dat není nijak skryta. Seznam studentů je opravdu realizován jako seznam datových struktur, z nichž každá nese informace o studentovi.

ukázka práce s daty

Modulární blokově strukturované jazyky

Rozdíl oproti předchozí kategorii je v tom, že jazyky umožňují oddělit definici typu od operací, které ho manipulují. Dokonce i popis typu pro jeho uživatele je prezentován jen takovou formou, aby jej mohl použít, zatímco např. vnitřní struktura je skryta. Tento princip umožňuje skrývat implementaci a lepší modifikovatelnost programů.

ukázka práce s daty

Datový typ pro osobu tak reprezentuje jen jeho jméno a modul, který jej zpracovává - vytváří nové položky tohoto typu, mění jednotlivé složky tohoto typu apod.

Objektově orientované jazyky

Tato skupina jazyků tu předchozí rozšiřuje o možnost spojit konkrétní data i s operacemi, které je manipulují. Data stojí v centru pozornosti jak návrhářů jazyka, tak potom těch, kteří implementují program.

ukázka práce s daty

Potom, například, datový typ osoby není manipulován v modulu, který tyto operace skrývá, ale nese si je s sebou. Podobně, jako datová struktura může zpřístupňovat své složky, tak potom objekt, reprezentující osobu, může předem daným způsobem zpřístupnit operaci, která umožňuje změnu jména, adresy, apod.

Jazyky jiných paradigmat

Mezi výrazné představitele jiných paradigmat patří jazyky:

- logické,
- funkcionální,
- pro sazbu textu,
- pro definici a manipulaci dat.

Jazyky logické (slouží k automatizovanému dokazování) pracují na úrovni dat také s predikáty a termy (viz predikátová logika). Jazyky funkcionální řadí mezi data funkce. U jazyků pro sazbu textu je potom datovou položkou typ písma, umístění textu, stránka (má různé atributy, jako šířka, výška apod.), nadpis, atd. Jazyky pro definici a

manipulaci dat (DDL — Data Definition Language, DML — Data Manipulation Language) užívané např. v relačních databázových systémech mají jako datový typ tabulku.

Abstrakce řízení

Na nejvyšší úrovni můžeme rozlišit dva typy programovacích jazyků, jazyky procedurální, cizím slovem imperativní (imperative) a deklarativní (declarative).

Definice 2.2.1 Imperativní *je takový jazyk, kde programátor musí řešit tyto otázky řízení běhu programu:* **Definice!**

- **co** *za operace má být provedeno,*
- **v jakém pořadí** *to má být provedeno.*

Definice 2.2.2 Deklarativní *je takový jazyk, kde programátor musí řešit tuto otázku řízení běhu programu:* **Definice!**

- **co** *za operace má být provedeno.*

Procedurální jazyky

Procedurální jazyky sestavují program jako posloupnost příkazů, z nichž některé a u některých jazyků lze také vnořovat. Zatímco pro strojově orientované jazyky, nebo velmi primitivní jazyky platí, že jediné větvení v jazyce je instrukce/příkaz skoku, tak vyšší jazyky nabízejí v podstatě shodnou sadu příkazů pro vyjádření opakování (smyčky, cykly) a variantního výběru/větvení.

Existence pouze skokové instrukce, či příkazu snižuje čitelnost kódu, rychlost tvorby programu a zvyšuje chybovost celku. Strukturované příkazy pro větvení a opakování výpočtu naopak vedou na čitelný kód, ve kterém se navíc ustalují určité vzory (patterns).

Na vyšší úrovni programového kódu (bloky, moduly, jednotky) potom u současných vyšších programovacích jazyků rozpoznáváme tyto typy abstrakcí pro řízení běhu programu: podprogramy, bloky, koprogramy, paralelní programy, odložené zpracování.

Podprogramy umožňují vnořené zpracování určitých logických funkcí/operací. Podprogram je volán skrze své rozhraní. Rozhraní definuje předávané parametry a výsledek. Implementace je skryta v definici podprogramu, kde může docházet k volání jiných podprogramů, podprogramy

nebo opakovaně sama sebe (rekurze). Proto má každý podprogram svou jednoznačnou identifikaci, která jej zpřístupňuje.

bloky

Bloky, na rozdíl od podprogramů, nemají jméno, takže je nelze volat z různých míst a jejich kód se uplatňuje pouze v místě, kde je vložen (jazyk C, Algol 60). Může mít však své lokální definice, nicméně díky absenci identifikace nemůže volat sebe sama.

ko-programy

Ko-programy jsou pojmenované bloky kódu, které mají vzhledem k sobě symetrický vztah. Jejich kód je zpracováván současně, případně prokládaně (dle cílové architektury). Jejich rozhraní a identifikace je shodné jako u podprogramů.

paralelní programy a procesy

Paralelní programy/procesy jsou oproti ko-programům mnohem větší logické celky. Jejich vztahy nejsou nutně symetrické, může mezi nimi docházet k synchronizaci (vzájemná výluka, serializace, čekání, apod.). V současnosti na nejvyšší úrovně abstrakce hovoříme o procesech, v rámci procesů potom o vláknech (threads). Ko-programy mohou být implementovány jako vlákna, ale nedisponují synchronizačními mechanismy.

odložené zpracování

Pro různé úrovně abstrakce můžeme u některých programovacích jazyků narazit na odložené zpracování. Logika je poměrně jednoduchá, pokud výsledek některé operace v daném místě nemusí být nutně dále použit, tak vyhodnocení operace je pozdrženo. Provedeno je později a jen v případě, že výsledek je potřeba, jinak není vyhodnocení provedeno vůbec.

Deklarativní jazyky

Deklarativní jazyky jsou často jazyky vysoké abstrakce a vyjadřovací síly. Proto se zde setkáváme s příkazy pro větvení toku řízení a opakování, které lze samozřejmě vnořovat.

Podle typu jazyka vstupují do role různé strategie vyhodnocení a různé strategie volání podprogramů. U volání podprogramů rozeznáváme tyto strategie (možno aplikovat i na procedurální jazyky):

- volání hodnotou (call-by-value) — u tohoto typu volání se parametry vyhodnocují před voláním podprogramu, v případě, kdy se eliminuje i opakované vyhodnocení jednoho a téhož podvýrazu (užitého na více místech), tak hovoříme o vyhodnocení striktním (strict);

- volání jménem (call-by-name) — parametry se do volaného podprogramu předávají nevyhodnoceny a jsou zde reprezentovány zástupným jménem. K jejich vyhodnocení dojde, až to vyžaduje volaný podprogram. Ve výsledku potom jeden a tentýž výraz

nemusí být vyhodnocen vůbec, ale také vícekrát, obecně je tato strategie vyhodnocení označována jako nestriktní;

- volání v případě potřeby (call-by-need) — podprogram je prvně zavolán, teprve když jeho vyhodnocení potřebuje hodnotu nějakého parametru, tak dojde k jeho vyhodnocení, hodnota se však uschová pro další použití, takže nedochází k jeho opakovanému vyhodnocení, takové vyhodnocení se také označuje anglickým termínem lazy (přímo nepřekládáme) a je také nestriktní.

Pojmy k zapamatování

- Abstrakce dat a toku řízení
- Jazyky strojově orientované, strukturované, modulární, objektově orientované
- Imperativní a deklarativní jazyky
- Blok, podprogram, ko-program
- Paralelní programy a procesy
- Odložené zpracování
- Volání hodnotou, jménem, v případě potřeby

Výklad

2.3 Specifikace jazyků, názvosloví

Programovací jazyky jsou definovány minimálně dvěma významnými oblastmi: syntaxí a sémantikou.

Definice 2.3.1 Syntaxe *jazyka definuje strukturu programu, tj. to, jakým způsobem je dovoleno jednotlivé konstrukce řadit za sebe.* **Definice!**

Pomocí syntaxe se často vysvětluje i lexikální stavba jazyka. Formálně lze syntaxi jazyka popsat formálními gramatikami. Z tohoto pohledu jsou v současnosti využívány regulární, či bezkontextové gramatiky. Někdy však existují kontextová omezení, která jsou potom definována slovním doprovodem.

Pro definici syntaxe pro účely popisu jazyka pro programátory se v současnosti používá buďto slovní popis (zřídkavě), syntaktické grafy, BNF, či EBNF, nebo gramatiky.

Definice!

Definice 2.3.2 Sémantika *je popis/definice významu jednotlivých syntaktických konstrukcí, způsobu jejich vyhodnocení, zpracování, atd.*

Pro popis sémantiky se na uživatelské úrovni nepoužívají formalismy, obvykle to je pouze slovní vysvětlení, nebo ukázka na příkladech. Jako formalismu lze použít např.

- axiomatickou sémantiku — pro každou syntaktickou konstrukci definuje množinu axiomů, které musí být splněny, aby byla konstrukce platná;
- operační sémantiku — definuje sémantiku chování programu jako posloupnost přechodů mezi danými stavy;
- denotační sémantiku — program je definován jako matematická funkce, která zobrazuje vstupy na výstupy.

Svou povahou může sémantika definovat vlastnosti z dvou pohledů:

- **statická sémantika** — popisuje vlastnosti, které mohou být studovány a ověřovány v době analýzy/překladu programu, např. typová kompatibilita, existence proměnných, apod.;
- **dynamická sémantika** — popisuje vlastnosti, jejichž splnění lze ověřit až v době běhu programu, např. velikost indexu pole daného výrazem, velikost výsledku, apod.

Významnými a často zaměňovanými, nebo nesprávně zrovnoceňovanými pojmy, které se pojí se sémantikou jazyka, jsou definice a deklarace. Je však mezi nimi důležitý rozdíl:

- **Deklarace** úplně vymezuje atributy dané entity. Může být explicitní i implicitní.
- **Definice** úplně vymezuje atributy dané entity a dále u proměnných způsob alokace paměti a u funkcí/procedur navíc tělo funkce.

Příklad deklarací v jazyce C:

```
extern int variable;
int function(int, double, char *);
```

Naproti tomu příklad definice těch stejných entit může být tento:

```
int variable = 54;
int function(int i, double d, char *s) {
  return i+(int)(d/(double)s[0]);
}
```

Tím, jak dochází k definicím, či deklaracím v rámci vytvářeného programu, dochází také k definici vazeb mezi vytvořenou entitou a celou skupinou vlastností či atributů. Tato vazba může vznikat v různých časech:

- během definice samotného jazyka — např. množina operací nad daným typem;
- během implementace programu — přiřazení typu proměnné;
- během překladu programu — inicializační hodnota proměnné;
- během spojování přeložených modulů (linking) — určení adresy objektu cizího modulu pro přístup k němu;
- během spouštění programu — např. navázání na standardní vstup;
- v době běhu programu — např. přiřazení adresy lokální proměnné.

Tyto vazby mohou být taktéž statické i dynamické. Statické jsou po vytvoření neměnné, zatímco dynamické jsou za běhu programu měněny.

Řešený příklad

Zadání: Vyjmenujte alespoň 5 druhů vazeb, které vznikají mezi entitami a vlastnostmi/atributy v tomto příkladě:

```
int count;
...
count = count + 5;
```

Řešení: V tomto jednoduchém příkladě je možné pozorovat například tyto vazby:

- množina typů, jichž může proměnná `count` nabývat;
- přiřazení typu proměnné `count`;
- množina hodnot, jichž může proměnná `count` nabývat;
- hodnota, kterou proměnná `count` nabývá;
- množina významů operátoru `+`;
- význam operátoru `+` v tomto případě;
- interní reprezentace literálu `5`.

V případě každé entity programu můžeme studovat různé vlastnosti. v další části textu se zaměříme na proměnnou, jejíž role je poměrně jednoduchá a lehce srozumitelná. Nejdříve vyjmenujme studované vlastnosti proměnné:

- jméno;
- adresa a umístění/lokace v paměti;
- hodnoty, jichž může nabývat;
- typ;
- doba života;
- rozsah platnosti (scope).

jméno

Jméno proměnné je možné charakterizovat několika vlastnostmi. Typická a často opomíjená vlastnost je délka jména. Maximální délka identifikátoru udává *maximální* počet znaků, které pro danou implementaci jazyka je možné využít. Tento údaj však není až tak důležitý z hlediska programátora jako *efektivní* délka, která udává počet znaků, který je skutečně zpracováván a rozlišován.

Znaky, které tvoří jméno proměnné jsou také pevně vymezeny, často jsou to písmenné znaky anglické abecedy, číslice a některé zvláštní znaky. Jsou však i jazyky, které povolují na místě identifikátorů používat i znaky národní abecedy. Často jsou povolena jak velká, tak malá písmena, ale jazyky se liší tím, jestli rozlišují velikost (case sensitive), např. jazyk C, nebo nikoliv (case insensitive), např. Pascal.

U současných moderních jazyků se setkáváme také s množinou zakázaných identifikátorů, které jsou použity pro označení příkazů, či konstrukcí jazyka, označujeme je jako *klíčová slova*. U některých jazyků se setkáváme s pojmem *rezervovaná slova* — ne všechna vyjmenovaná slova se objevují jako příkazy či konstrukty jazyka. Jsou však i jazyky bez omezení na jméno identifikátoru, což znesnadňuje jejich analýzu, např. Fortran.

umístění a adresa, hodnota

Máme-li jednoduchý přiřazovací příkaz:

```
L = R
```

tak potom proměnná na levé straně určuje umístění (typ paměti, ve které leží) a adresu v rámci tohoto umístění, krátce se označuje jako *L-hodnota*. Proměnná na pravé straně reprezentuje hodnotu, kterou obsahuje, zkráceně se označuje jako *R-hodnota*.

Jedno jméno proměnné může být spojováno s více umístěními a adresami, např. lokální proměnné v podprogramech se mohou jmenovat stejně a přitom leží, či přesněji mohou ležet, na různých místech.

Nebo lokální proměnná v rekurzivním podprogramu. I když se jedná o tentýž podprogram, tak každé volání umísťuje proměnnou stejného jména na jinou adresu. Globální proměnná může být umístěna do jiné paměti než globální konstantní proměnná — ta může být v paměti, která je nezávislá na napájení a určená jen pro čtení.

Naproti tomu jedna adresa může být spojována s více různými jmény. Například ukazatel na proměnnou a proměnná, nebo sdílení v důsledku optimalizací, apod.

Vazba mezi umístěním + adresou a jménem proměnné, případně její hodnotou může být taktéž *statická* či *dynamická*. Statická vazba je u statických proměnných či globálních proměnných, nebo u proměnných s pevně danou adresou explicitně. Staticky je vázána hodnota ke konstantě. Dynamická vazba k adrese je u proměnných lokálních programů umísťovaných na zásobník, nebo proměnných vytvářených dynamicky na haldě (heap) — rušení je potom implicitní, nebo explicitní, nebo automatické.

typ

Z typu proměnné se odvozuje řada dalších informací a skutečností. Určuje množinu možných hodnot, jichž může proměnná nabývat a dále množinu operací, které na ni lze aplikovat. Vazba mezi typem a proměnnou je často statická a vyžaduje explicitní deklaraci či definici. Implicitní definice se statickou vazbou je typická např. pro jazyk Fortran. Dynamická vazba tohoto druhu je u skriptovacích jazyků a interpretů, kde dochází také k dynamické typové kontrole (zatížení v době běhu programu). Příkladem dynamické vazby může být tento kousek kódu jazyka Perl:

příklad

```
if ( $p ) $x = 10; else $x = "abc";
echo $x+1;
```

Podle hodnoty proměnné `p` je potom výstupem buďto číselná hodnota `11`, nebo řetězec `abc1`.

Jazyky z hlediska typovosti rozdělujeme na *beztypové* (type-less), *netypované* (non-typed) a *typované*. U typovaných může být přiřazení typů explicitní, nebo automaticky odvozené (type inference).

Definice 2.3.3 Rozsah platnosti *proměnné určuje tu část programu, kdy je možné s proměnnou pracovat.*

Definice!

Termín rozsahu platnosti je spojen s *viditelností* proměnné. I když proměnná je platná, může být skryta jinou proměnnou stejného jména (například lokální proměnná zakryje globální).

Velká většina jazyků u rozsahu platnosti uplatňuje statickou vazbu, která je dána strukturou samotného programu. Dynamická vazba —

do další deklarace/definice se uplatnila jen u některých jazyků (APL, SNOBOL4, částečně LISP). Dynamická vazba je jednoduše realizovatelná v interpretech, ale nevhodná z pohledu softwarového inženýrství.

Definice!

Definice 2.3.4 Doba života *proměnné je časový interval, po který je pro danou proměnnou alokována paměť.*

Alokace opět může probíhat staticky, tj. před během programu, nebo dynamicky. Dynamická alokace může být automatická (lokální proměnné), nebo explicitně nějakým příkazem (dynamické proměnné na haldě).

Pojmy k zapamatování

- Syntaxe a sémantika jazyka
- Axiomatická, operační, denotační sémantika
- Statická a dynamická sémantika, vazba
- Deklarace, definice
- L-hodnota, R-hodnota
- Vliv typu na proměnnou
- Umístění a adresa
- Rozsah platnosti
- Doba života

Závěr

Úspěšným zvládnutím této kapitoly zvládnete základní klasifikaci jazyků, orientaci v pojmech, které se s definicí či popisem jazyků pojí. Sami byste měli být schopni jazyk klasifikovat podle jeho vlastností.

Úlohy k procvičení:

Pro každý typ jazyka, či vlastnost s jazykem spojovanou zkuste najít na Internetu nějakého představitele. Porovnejte potom jazyky z více různých hledisek. Který byste rádi používali a proč?

Klíč k řešení úloh

Vyjděte z vám známého jazyka a přiřaďte mu vlastnosti, pro nepřiřazené vytypujte jazyk na základě základních charakteristik a ostatní doplňte po jeho důkladnějším prozkoumání. Takto postupujte až do úplného vyčerpání všech uvedených vlastností a pojmů.

Kapitola 3

Nestrukturované jazyky

Tato kapitola seznamuje se základními charakteristikami a problematikou nestrukturovaných imperativních jazyků. Dále rozvíjí a doplňuje pojmy definované v Kapitole 2.

Čas potřebný ke studiu: 4 hodiny.

Cíle kapitoly

Cílem kapitoly je blíže se seznámit s obecnými vlastnostmi nestrukturovaných programovacích jazyků — co od nich můžeme očekávat a jak by se daly některé vlastnosti implementovat.

Průvodce studiem

Tato kapitola předpokládá dobrou orientaci v základních pojmech z klasifikace a popisu programovacích jazyků. Na tyto pojmy navazuje a dále je rozvíjí na skupině nestrukturovaných imperativních jazyků. Přitom se zabývá těmito jazyky jak z pohledu uživatele: možnosti a vhodnosti nasazení, očekávané vlastnosti, apod.; tak z pohledu implementace a typických problémů s ní spojenými.
Ke studiu této kapitoly není třeba žádného speciálního vybavení. Jako u většiny ostatních se jedná o to si zapamatovat danou terminologii a pochopit klíčové principy předkládaných konceptů. Pro hlubší studium, nebo bližší objasnění prezentovaných skutečností je vhodné využít informací na Internetu a vlastní praktické ověření vlastností na konkrétním jazyce.

Obsah

Výklad

3.1 Základní charakteristika

Nestrukturované programovací jazyky se objevily jako první a jejich vlastnosti se objevují dodnes, i když už v jazycích určených pro jiné aplikace. Celé to dokumentují 3 jazyky uvedené jako příklad. Jazyk Fortran, který byl určen a je dodnes používán pro vědecké a technické výpočty, se objevil už na konci 50. let 20. století. Dalším případem může být jazyk Basic, který v 80. letech byl využíván jako výukový. V současnosti se nestrukturované jazyky využívají ve skriptovacích jazycích, které však jsou často rozšiřovány tak, že disponují i některými dalšími, pokročilými vlastnostmi.

Definice 3.1.1 Formální báze *je takový formální prostředek (kalkul, algebra, apod.), který umožňuje exaktně popsat všechny konstrukce daného jazyka.* **Definice!**

Formální báze slouží k tomu, aby bylo možné jisté vlastnosti jazyka formálně dokázat, či ověřit. Často se jedná o sémantiku, bezespornost jazyka, atd. Pokud je formální báze k dispozici dopředu, tak je navíc ideálním vodítkem pro implementaci.

formální báze

Nestrukturované jazyky však formální bázi nemají. Zpětné vytvoření takové báze by navíc ani asi nebylo možné (u jazyků navržených v minulosti se s tím ani nepočítalo, takže jejich řídicí struktury jsou velmi komplikované). Nicméně u těchto typů jazyků to není potřeba, neboť mají takové určení, kde podobné úkony nejsou vyžadovány.

syntaxe

Tak jako u každého jazyka, je i každý nestrukturovaný jazyk doprovázen dokumentací, která vysvětluje jeho syntaxi a sémantiku. Syntaxe těchto jazyků je typicky podána přirozeným jazykem s příklady ukazujícími správný zápis. Formálnější zápisy, jako např. (E)BNF, jsou zřídkavé a spíše u současně vytvářených implementací. Důvodem je i to, že syntaxe je poměrně jednoduchá a je často doplňována sémantikou. Syntaxe u těchto jazyků může mít i další omezení (volná či pevná syntaxe). Např. jazyk Fortran (starší verze) nespecifikuje pořadí předávaných parametrů do určitých funkcí/procedur, na druhou stranu na jednom řádku umožňuje definovat jen jedno záhlaví cyklu, podobně jako třeba některé mutace jazyka Basic:

- Fortran:

```
OPEN(UNIT = 2, FILE="file", ACCESS="SEQUENTIAL")
OPEN(7, FILE="myfile", STATUS="NEW")
```

Poznámka: parametr UNIT je povinný, ostatní jsou nepovinné.

- Basic:

```
FOR I=1 TO 20
PRINT I; I*I
NEXT I
```

sémantika

Sémantika je podávána u této kategorie jazyků často neformálně. Typicky jsou vytvořeny příklady doplněné popisem, který vysvětluje dané chování. Toto je možné jednak proto, že někdy sémantika není složitá, nebo naopak proto, že je schovaná (viz další příklad). Popis sémantiky je často inkrementální (jednoduché věci jako první). Důvodem je to, že vestavěné funkce/procedury mohou mít sémantiku různou podle toho, jaké mají předány parametry. Prohlédněte si následující příklad v jazyce Basic (Sinclair ZX Spectrum).

```
10   PRINT USER; 23760
50   PRINT USR 23760
```

Zatímco na řádku `10` dochází k tisku hodnoty proměnné `USER` následované číslem `23760`, tak na řádku `50` jde o spuštění programu ve strojovém kódu, který leží v paměti na adrese `23760`, a tisku jeho návratové hodnoty.

3.2 Formalismy a užití

I když jazyky často nedisponují formální bází, tak přesto můžeme chtít ověřit, že nějaká vlastnost algoritmu je splněna. Mluvíme tedy o formální verifikaci a validaci.

formální V&V

U nestrukturovaných jazyků se s tímto nesetkáváme. Z části proto, že to není u těchto jazyků používáno (nikdo to nikdy nechtěl), zčásti proto, že to je prakticky nerealizovatelné. Floyd-Hoare logika (viz Kapitola 4) je uplatnitelná jen na některé konstrukce a navíc v současnosti tento typ jazyků neleží v centru pozornosti programátorů. Pochopitelně až na výjimky historické (Fortran), či skriptovací jazyky.

softwarové inženýrství

Možnost aplikovat na takové jazyky nějaké zásady softwarového inženýrství jsou poměrně omezené. Vlastnosti, které takové jazyky mají (viz níže), nejsou totiž vhodné pro to, aby byly využívány v týmové práci pro rozsáhlé projekty. Programátor totiž může lehce porušit zásady „dobrého programování“, které jinak musí následovat. Navíc to, že je porušil, je jen obtížně kontrolovatelné. Jedna z možností, jak

dosáhnout určité čistoty kódu, je použít pro návrh formálních prostředků pro modelování a návrh aplikace a kód pro daný jazyk vygenerovat (v horším případě ručně naimplementovat). Tento přístup je však často vykoupen tím, že řada vlastností jazyka zůstává nevyužita, což však nemusí být nutně na škodu.

Shrnutí charakteristik

Nestrukturované jazyky nejsou příliš vhodné (z dnešního pohledu) pro rozsáhlé projekty. Pokud jde o využití skriptovacích jazyků pro psaní jednoduchých skriptů, tak jde o vhodné využití, protože program svou podstatou je malý.

Pro výuku takové jazyky nejsou vhodné rovněž. Důvodů je řada, za všechny jmenujme aspoň to, že takové jazyky vytvářejí u programátorů nesprávné návyky, které se potom těžko odstraňují. Kromě toho, ty správné návyky se potom o to hůře získávají.

V současnosti jsou z této kategorie nejvíce vidět skriptovací jazyky, které se využívají pro psaní pomocných aplikací (např. bioinformatika), nebo pro podporu dynamických WWW stránek, či filtrování textových informací (log soubory apod.). Další jazyk, který je stále živý je Fortran. Aplikace v něm, pokud nemají návaznost, už nevznikají ve velkém. Spíše se udržují stávající, jichž je stále velké množství.

Pojmy k zapamatování

- Formální báze, typ/existence
- Popisy syntaxe, záludnosti nestrukturovaných jazyků v tomto bodě
- Sémantika
- Formální V&V nestrukturovaných jazyků
- Nasazení softwarového inženýrství u nestrukturovaných jazyků
- Rozšíření a využití nestrukturovaných jazyků

Výklad

3.3 Datové a řídicí abstrakce

Nestrukturované jazyky buď neposkytují uživateli vůbec žádné datové ani řídicí abstrakce, nebo jen ty nejjednodušší. V podstatě se u těchto jazyků setkáváme se základními numerickými typy, znakovými řetězci a poli (nezaměňovat s různými rozšířeními). Na řídicí

úrovni jsou to typicky jen smyčka s pevným krokem a řídicí proměnnou (cyklus for), příkaz pro větvení (if) a skok.

Otevřené podprogramy

Definice nových typů, či jen synonym zcela chybí. Pro manipulaci s daty je možné použít pouze vestavěné operace. Pokud lze kombinací více operací obdržet nějakou rozšiřující operaci, tak její opakované využití na více místech není možné jinak, než vytvořením kopie textu a vložením na dané místo, neboť řada těchto jazyků postrádá možnost definovat podprogram (funkci/proceduru). Často je však možné tento nedostatek alespoň částečně eliminovat pomocí textových maker, či *otevřených podprogramů*.

Definice!

Definice 3.3.1 Otevřený podprogram *je uložen v rámci hlavního (často jediného) zdrojového textu. Nemá definované pevné rozhraní, tzn. vstupní a výstupní bod, parametry, výsledek apod. Vstup se děje skokem na příkaz, jímž má výpočet podprogramu začít, ukončení podprogramu je dáno vyvoláním příslušného příkazu (nikoliv doběhnutím výpočtu do/za určité místo).*

---

podprogramy

Otevřené podprogramy tak slouží vlastně k uložení kódu, který je možné spustit vícekrát a uspořit tak místo v paměti a čas programování — například rutina, která vypíše aktuální hodnoty nějakých proměnných na obrazovku. Velmi často je nemožné vnořovat volání otevřených podprogramů, uplatnění rekurze je zcela nemožné. Parametry i výsledky jsou předávány prostřednictvím globálních proměnných. Pro určité mutace jazyku Basic je typickým příkazem pro vstup do otevřeného podprogramu příkaz `GOSUB` a pro výstup příkaz `RETURN`. Zvyklostí je umísťovat takové podprogramy na začátek textu programu s tím, že první příkaz je příkaz skoku, který přeskočí až do hlavního těla programu. Možnost chyby z nepozornosti je u takových programů pochopitelně velmi vysoká.

pseudo-moduly

Aby bylo možné vytvářet jeden program jako jeden celek složený z více souborů, tak v některých případech se otevřené podprogramy nahradily přechodem do jiného souboru, který podprogram reprezentuje (např. soubor, kde byl implementován tisk, či výpočet nějaké funkce apod.). U takovýchto jazyků se setkáváme s dynamickou definicí proměnných — vznikají (jakožto položka v paměti a přístupná entita v programu) prvním použitím (přiřazením hodnoty), nebo deklarací (málokdy), nicméně existuje příkaz, který od daného místa dále ruší platnost takového identifikátoru (odstraňuje z paměti). Nebezpečí

u takových jazyků tkví v tom, že by se mohly odkazovat určité soubory cyklicky vzájemně na sebe. Toto je typicky zakázáno a buďto to jazyk přímo kontroluje a hlásí jako chybu, nebo potom takový celek nepracuje dobře. Od otevřených podprogramů se liší tím, že vstupní bod je pevný, stejně tak je dán konec — posledním platným řádkem souboru. Přesto se však nejedná o uzavřený podprogram ani o modul v pravém slova smyslu.

Shrnutí – podprogramy

- Parametry i výsledky jsou předávány jen jako globální proměnné — neexistují lokální proměnné, tudíž je jednoduché udělat chybu, kdy dojde k přepsání jiných dat.
- Implicitní podpora rekurze chybí — v určitých případech může řešit programátor ve vlastní režii.
- Složitá struktura programu — větší a složitější programy mohou být vytvořeny tak, že jejich analýza/úprava může být velmi složitá (složitost tvorby je na textové úrovni kódu — rozsah, vazby, tok programu apod. — vlastní složitost zpracovávaného problému je často spíše menší).

Návrh programu

Návrh programu založený na postupném zpracování a manipulaci s daty je poměrně těžký, protože díky absenci uzavřených podprogramů nelze návrh strukturovat. Data jsou tak viditelná od místa jejich vzniku (deklarace/definice) až do konce běhu programu, pokud nejsou explicitně vymazána, což ale není typická vlastnost takových jazyků (viz odstavec pseudo-moduly výše). I další metodiky jsou jen těžko aplikovatelné. Jazyky jsou náchylné k tvorbě chyb při programování a ztěžují týmovou práci.

Silnou stránkou jsou však vestavěné operace a *ad hoc* řešení, která často mohou výrazně urychlit práci a tvorbu aplikace. Ještě více však činí program poplatný jazyku, nicméně v případně nutnosti jsou jediným efektivním řešením.

Typy a jejich zpracování

Jazyky nestrukturované jsou většinou netypované, tj. explicitně se neuvádí typ proměnné, ale v rámci kontextu je dobře znám, lze jej detekovat (systémem) za běhu, a typ se sám přizpůsobuje okolnostem

v rámci definovaných omezení. Stačí tak pouze přiřadit hodnotu jiného typu a proměnná typ sama změní od daného místa výpočtu dále.

Změnám typů u takových jazyků brání implicitní typ, který je zde často rozpoznatelný dle jména proměnné. Např. v jazyce Basic jméno proměnné končící znakem `$` označovalo textový řetězec, bez něj obecnou číselnou proměnnou, či proměnnou typu pole. V jazyce Fortran zase proměnné začínající své jméno znakem `N` či `I` jsou implicitně celočíselné.

Kromě polí se nesetkáváme u takovýchto jazyků s jinými než číselnými typy a znakovými řetězci. Díky tomu, že typ se může měnit za běhu, v každém kroku výpočtu tak dochází k zjišťování, jakého typu jsou parametry operandu, případně jsou provedeny implicitní konverze a na závěr provedena samotná operace. Výsledek je přiřazen na správné místo. Důležité je, aby bylo jednoduché rozpoznat, jaký typ hodnoty se v proměnné nachází a to efektivně.

Řešený příklad

Zadání: Navrhněte pro jazyk Basic způsob uložení celého znaménkového čísla a desetinného čísla tak, aby byl umožněn efektivní přístup a rozpoznání typu. Také uvažte efektivní využití paměti. Cílová architektura je 16 bitová.
Řešení: Ze zadání plyne, že je nutné do reprezentace zavést nějakou značku, kterou poznáme o jaký typ se jedná. Čím rychlejší bude detekce typu, případně nějaké typické významné hodnoty (např. 0), tím lepší. Řešení je možné rozdělit na dva základní proudy:

1. cílový systém disponuje vestavěnou podporou pro čísla s plovoucí desetinnou čárkou;
2. podpora čísel s plovoucí desetinnou čárkou je buďto na softwarové úrovni, nebo žádná.

Ať tak, či tak, počet přístupů do paměti budeme chtít minimalizovat, ale asi ideální řešení nenajdeme. Více se proto zaměříme na paměťové uspořádání. Nejmenší uspořádání je uspořádání přes sebe. V případě podpory čísel s plovoucí desetinnou čárkou však v případě standardu IEEE 754 (dá se předpokládat) pro 32bitová čísla (vyšší u 16bitové architektury nebudeme uvažovat) můžeme narazit na problém, jak poznáme, o jaký typ se jedná. Standard IEEE 754 totiž využívá všechny bitové kombinace pro reprezentaci okrajových hodnot (nekonečno, nenormalizované číslo, výsledek není číslo apod.). V takovém případě bychom proto uložení přes sebe volit nemohli.

U vlastní podpory jsou možnosti otevřené — od vlastních formátů po zakázání některých bitových kombinací (např. nenormalizované číslo a záporná nula). Potom lze např. zarovnáním doprava překrýt obě hodnoty. Prvních 16 bitů nulových říká, že se jedná o celé číslo uložené v dalších 16 bitech. Opak říká, že číslo je desetinné.
Pokud by se jednalo o systém s podporou čísel s plovoucí desetinnou čárkou, tak volba závisí od toho, jestli systém podporuje čtení i z lichých bytů, nebo je úplně 16 bitový. V prvém případě vyhradíme jeden byte na uložení informace o typu čísla — podle jeho hodnoty přistoupíme k dalším 32 bitům jako k 16 bitovému celému číslu nebo jako k 32 bitovému desetinnému číslu. V případě zarovnání na sudé adresy (16 bitů) se potom nabízí uložení vedle sebe (opět problém s rozpoznáním typu), nebo využití 16 bitů jen jako značky.
Jak je vidět, tak ideální a univerzální řešení neexistuje. Nicméně právě zvážení a uvědomění si všech eventualit hraje velmi důležitou roli při implementaci nejen programovacích jazyků.

Tok řízení programu

Nestrukturované jazyky nenabízejí žádnou možnost, jak blokově strukturovat skupiny příkazů, které jsou řazeny sekvenčně, nebo logicky vnořené. Jedná se tak o sekvenci jednoduchých operací (neskrývají v sobě další vnořené operace), která je interně provázána systémem příkazů skoku (ať podmíněných, nebo jednoduchých). Podprogramy jsou podpořeny maximálně na úrovni otevřených podprogramů. Vstup do otevřených podprogramů, podobně jako cíle skoků jsou často určeny číslem řádku, na kterém se nachází požadovaný příkaz (v současnosti se setkáváme i s textovými návěštími, nebo s tím, že není třeba číslovat všechny řádky programu).

Kromě podmíněného příkazu a případného skoku se můžeme setkat s příkazy pro nekonečné smyčky, či smyčky typu „for“. Výjimečně (zejména později) i s dalšími typy smyček, případně jiným druhem větvení. Typicky se však obvyklé programové struktury obcházejí právě pomocí podmíněného příkazu a skoku. Vznikají tak však kromě typických struktur (cyklus s podmínkou na začátku, či na konci) i struktury méně obvyklé, jako cyklus s podmínkou uprostřed.

důsledky

Tok řízení a samotného vyhodnocení programu je potom těžko zpětně čitelný. Tato složitost není jen na úrovni programátorů, ale ve svém důsledku i optimalizující překladač by ztrácel účinnost. Náročnost jak tvorby programu, tak jeho kontrol v době překladu roste. Přitom převod takto zachycených algoritmů do strukturované podoby je velmi složitý, nebo získaný automatizovaným způsobem nečistý (povšimněte si například překladače F2C, který překládá jazyk Fortran

přes jazyk C).

Pojmy k zapamatování

- Otevřený podprogram
- Typy v nestrukturovaných jazycích — implicitní, změna typu za běhu, kódování hodnot
- Možnosti řízení toku vyhodnocení programu — typické obraty

Výklad

3.4 Zpracování — analýza, vyhodnocení, překlad

K programovacím jazykům obecně lze sestavit a využít tři základní typy programů:

- analyzátor;
- vyhodnocovací program, či interpret;
- překladač (compiler).

analyzátor

Analyzátor je program, který analyzuje vstupní text v nějakém programovacím jazyce a provádí jeho důkladnou kontrolu pouze na základě jeho textu. Výstupem je potom potenciální seznam chyb, varování či doporučení k danému programu. Takový program je vhodný tehdy, pokud prověřuje vstupní text z hlediska náležitosti k nějaké normě, nebo jeho analýza je tak důkladná, že odhalí chyby, které i sám programátor přehlíží (např. lint pro jazyk C, nebo verifikátory jazyka C dle standardu MISRA).

interpret

Vyhodnocovací program/interpret je takový program, který jakmile rozpozná nějaký příkaz ve vstupním programu, který má na vstupu, tak jej ihned provede. Převádí (překládá) tak vstupní program na posloupnost okamžitě prováděných akcí (např. interprety pro jazyk Basic).

překladač

Překladač, někdy slangově označován jako kompilátor, je program, který vstupní text programu převádí na posloupnost příkazů jiného jazyka, či stroje. Cílem takového překladu může být např. binární soubor, který je přímo spustitelný na dané architektuře (cílovým jazykem překladu jsou instrukce procesoru dané platformy).

Překladače

lexikální analýza

Prvním stupněm analýzy programovacích jazyků je obecně *lexikální analýza*. Pro nestrukturované jazyky v ní nenacházíme nic zvláštního, snad jen to, že u starších jazyků se objevuje kontextová závislost lexikálních symbolů na umístění v programu, což v současnosti není typické.

syntaktické analýza

Na úrovni *syntaktické analýzy* se jedná o typické bezkontextové vlastnosti a je možné využít příslušné formální aparáty. U prvních implementací však nedocházelo ještě k využití podobných teorií, což může být patrné zejména při pozdějších implementacích analyzátorů takových jazyků. Mezi typické bezkontextové vlastnosti patří závorkové struktury, které se u těchto jazyků projevovaly zejména v cyklech typu „for“, příkazech větvení „if-then-else“, nebo ve výrazech (pro jejich analýzu se však dříve používalo jiných mechanismů).

sémantická analýza

Sémantická analýza je v době překladu u typických představitelů nestrukturovaných jazyků poměrně v pozadí. V době překladu lze bezpečně kontrolovat pouze existenci cíle skoků. Částečně lze potom rozhodnout, zda-li je čtená proměnná již definována, nebo použití indexů pro přístup k elementům polí. Ostatní operace sémantické analýzy nelze v době překladu provádět vůbec. Kontroly, které nelze uskutečnit v době překladu, je tak nutné provést v době běhu programu.

Pro kontextovou analýzu prováděnou v rámci analýzy sémantické je typické užití tabulek symbolů. U nestrukturovaných jazyků je typické užití jednoúrovňové tabulky symbolů. Na konci překladu/analýzy je znám počet užitých symbolů a minimální velikost paměti, která bude potřeba pro uložení hodnot všech proměnných. Jiný typ informací není možné bezpečně ze statické analýzy zjistit.

generování kódu

Pro generování cílového kódu (míněno instrukcí procesoru) lze uplatnit poměrně přímočaré algoritmy. V takovém případě se každý příkaz překládá odděleně (volání knihovní funkce, nebo vložení jednoduchého kódu). Na druhou stranu je možné program analyzovat jako celek, takže lze globálně aplikovat celou řadu optimalizací, které jsou jinak limitovány na jednu funkci, maximálně modul. Efektivní nasazení takových optimalizací je však často komplikováno tím, že zapsaný program díky vlastnostem jazyka neumožní aplikovat pokročilé optimalizační algoritmy, neboť nejsou dostupné potřebné informace, nebo nejsou na zdrojové úrovni k dispozici takové konstrukce, kde by bylo možné optimalizační postupy aplikovat.

Typickým výsledkem je tak poměrně „čitelný“ kód, který není nijak zvláště optimalizován. Součástí výsledku jsou často rozsáhlé knihovny pro podporu programu za běhu — ty jsou však často vysoce optimalizovány. Počet operací, které jsou spojeny s tím, co programá-

tor napsal, v poměru k ostatním operacím (knihovny, kód pro provádění dodatečných kontrol za běhu programu, apod.) je tak poměrně nízký a rychlost provádění se tak může jevit jako nižší (ve srovnání s přeloženým programem z jiného typu jazyka).

Interprety

lexikální analýza

Lexikální analýza probíhá u interpretů co nejdříve, avšak jen v omezeném rozsahu. U systémů, kde se pracuje v rámci jistého programovacího rozhraní (např. legendární Sinclair ZX Spectrum) se s vložením programového řádku provede převod klíčových prvků jazyka (klíčová slova, konstanty, identifikátory apod.) do interní reprezentace, aby při samotném běhu programu nedocházelo k přílišnému prodlení díky časově náročné analýze. Samotná analýza probíhá podobně, jako dnes, nebo jak je tomu u překladačů těchto jazyků.

syntaktická analýza

Příkazy interpretovaných jazyků jsou často řádkově orientovány a proto stejným způsobem probíhá i syntaktická analýza s využitím stejným prvků jako u překladačů. Často lze uplatnit intuitivní postupy a určité operace se provádějí již při vkládání textu, jako u lexikální analýzy — jsou to takové operace, kdy je možné bezpečně rekonstruovat původní text (až na opakované mezery apod.).

U jazyků, které mají některé typické rysy nestrukturovaných programovacích jazyků, které je tak cele prorůstají i přes jiné, pokročilé vlastnosti, je možné pozorovat, že závislost příkazu na řádku je eliminována (využívají se pro analýzu moderní metody založené a teorii formálních jazyků a automatů). Také se již nesetkáváme s programovacími rozhraními, takže text nebývá předzpracováván, ale je buďto neustále znovu plně analyzován, nebo je pomocí metody překladu v době vyhodnocení/běhu (just-in-time compilation) přeložen do instrukcí cílové, či virtuální architektury, kde při opakovaném průběhu toku řízení programu přes dané místo se využije již analyzovaný a přeložený kód, který je tak vyhodnocen mnohem rychleji, než při čisté interpretaci.

sémantická analýza

Sémantickou analýzu je možné buďto provádět v jisté posloupnosti, nebo ji zcela odložit až na samotné vyhodnocení. V případě posloupném je možné v době analýzy pochopitelně provést jen část kontrol. Typicky však po základní analýze probíhá okamžitě vyhodnocení spojené se všemi nutnými kontrolami. Jistým prvkem nestability mohou být dopředné skoky, kdy je nutné přeskočit část textu programu, nikoliv však zcela bez analýzy. Často zde probíhá pouze základní lexikální a syntaktická bez jakýchkoliv dalších kontrol — cílem je pouze vyhledání cíle skoku. U jistých konstrukcí (podmíněné příkazy, či smyčky, apod.) může však definice jazyka limitovat délku textu mezi dvěma oddělovači (počátek a ukončení takové konstrukce), aby se zefektivnila

analýza a vyhodnocení (typické pro experimentální, úzce specializované a výzkumné jazyky).

Pro analýzu kontextu se opět používají jednoúrovňové tabulky symbolů. U čistě interpretovaných systémů se jako jistý problém může jevit cíl skoku u příkazů, kde se tento cíl určuje k implicitně daným destinacím (smyčky, příkazy větvení apod.). Skoky samotné také snižují efektivitu vyhodnocení. V průběhu vyhodnocení se tak často buduje separátní tabulka cílů skoků, která je indexována návěštími (u explicitních skoků), nebo pozicí příkazů (u implicitních skokových operací). Snahy o vyšší efektivitu vyhodnocení takových konstrukcí také mohou vést k omezení jazyka — např. jeden příkaz (počátek) na řádek, takže vnořené musejí začínat na dalším řádku, případně nelze vnořovat (zřídkavé).

za běhu

Pro vlastní vyhodnocení/interpretaci programu se využívají rozsáhlé knihovny funkcí a operací (podobně jako u překladačů). Je snaha však co nejvíce operací provádět vloženě, s využitím procesoru (viz předchozí řešený příklad). Vyhodnocení se provádí vždy, jakmile je úspěšně analyzován jistý úsek vstupu, takže jej lze převést na jednu operaci. Výraz tak může být vyhodnocován po jednotlivých operátorech, ale tisk hodnoty se provede atomicky.

Pojmy k zapamatování

- Analyzátor, interpret, překladač
- Kontextová lexikální analýza
- Možnosti generování kódu
- Výjimečné vlastnosti interpretů
- Možnosti, doba a způsoby sémantických kontrol

Závěr

Úspěšným zvládnutím této kapitoly proniknete do zásad a typických vlastností nestrukturovaných programovacích jazyků. Měli byste vědět, jak tyto jazyky pracují s typy, jaké nabízejí typické konstrukce pro řízení toku vyhodnocení/běhu programu. Také byste měli vědět, jak tyto jazyky zpracovávat, s čím je třeba počítat, co naopak může chybět.

Úlohy k procvičení:

Zkuste najít nějakou starší definici jazyka Basic, či definici jazyka Fortran 77. Potom si vyhledejte definici moderního skriptovacího jazyka (perl, apod.). Zkuste tyto jazyky porovnat:

- Co mají společného?
- Co mají naopak zcela odlišného?
- Pro jaký typ aplikací jsou především určeny?
- Jaký typ aplikací naopak s nimi není vhodné řešit
- Jedná se o interprety, překladače, interprety s překladem za běhu?
- Jak a kdy se která chyba na úrovni zdrojového textu projeví/ohlásí?
- Je možné zjistit, jak jsou uloženy proměnné, text programu (u interpretu), apod.? Jestli ano, tak potom jak tomu je?

Vždy se snažte si uvést a odzkoušet krátký příklad, srovnejte některé obraty s možnostmi pokročilejších jazyků (C/C++, Java, apod.).

Klíč k řešení úloh

Snažte se najít nějaký vám blízký nebo známý jazyk. Jděte zejména po tipech a tricích, které jsou s danými jazyky spojeny, tam se dozvíte o způsobu implementace nejvíce.

Kapitola 4

Strukturované jazyky

Tato kapitola seznamuje se základními charakteristikami a problematikou (blokově) strukturovaných imperativních jazyků. Dále rozvíjí a doplňuje pojmy definované v Kapitole 2.

Čas potřebný ke studiu: 6 hodin.

Cíle kapitoly

Cílem kapitoly je blíže se seznámit s obecnými vlastnostmi blokově strukturovaných programovacích jazyků — co od nich můžeme očekávat a jak by se daly některé vlastnosti implementovat. Koncentrace bude kolem datových struktur, ukazatelů. Seznámíme se také s jednou z možností formální verifikace vlastností programů.

Průvodce studiem

Tato kapitola předpokládá dobrou orientaci v základních pojmech z klasifikace a popisu programovacích jazyků. Na tyto pojmy navazuje a dále je rozvíjí na skupině blokově strukturovaných imperativních jazyků. Přitom se zabývá těmito jazyky jak z pohledu uživatele: možnosti a vhodnosti nasazení, očekávané vlastnosti, apod.; tak z pohledu implementace a typických problémů s ní spojenými. Přitom provádí některá zpětná srovnání s jazyky nestrukturovanými.
Ke studiu této kapitoly není třeba žádného speciálního vybavení. Jako u většiny ostatních se jedná o to si zapamatovat danou terminologii a pochopit klíčové principy předkládaných konceptů. Pro hlubší studium, nebo bližší objasnění prezentovaných skutečností je vhodné využít informací na Internetu a vlastní praktické ověření vlastností na konkrétním jazyce.

Obsah

Výklad

4.1 Základní charakteristika

Jazyky blokově strukturované se objevují jako logický důsledek jazyků nestrukturovaných a univerzálních. Jejich hlavní snahou je dát do rukou programátora flexibilitu a dostatečné výrazové prostředky pro tvorbu programů ze všech oborů efektivnějším způsobem. Patrná je snaha zpřehlednit program, přímo „nutit“ programátorovi strukturu textu programu tak, aby byl čitelný (tato snaha díky řadě vlivů však není úplná a naplno se projevu až se zásadami softwarového inženýrství o mnoho let později).

Mezi typické představitele lze zařadit Algol a Pascal. U jazyka Pascal je míněn původní návrh prof. Wirtha, který sledoval opravdu řadu vznešených cílů a jako výukový jazyk se na dlouhou dobu začlenil do řady míst. Tento původní návrh však doznal řadu modifikací, které v současnosti vyústili až v jazyk ObjectPascal známý z prostředí Delphi.

formální báze

Strukturované jazyky formální bázi primárně nemají. Jazyk byl navržen bez užití formalismů. Nicméně pro řadu z nich, nebo alespoň jejich velkou podmnožinu, by bylo možné takový formalismus dodatečně definovat (viz literatura na konci této kapitoly). I když formální báze chybí, tak tyto jazyky se snaží využít předchozí dobré vlastnosti a eliminovat nedobré charakteristiky jazyků předchozích. Např. jazyk Algol se tak dočkal několika revizí.

syntaxe

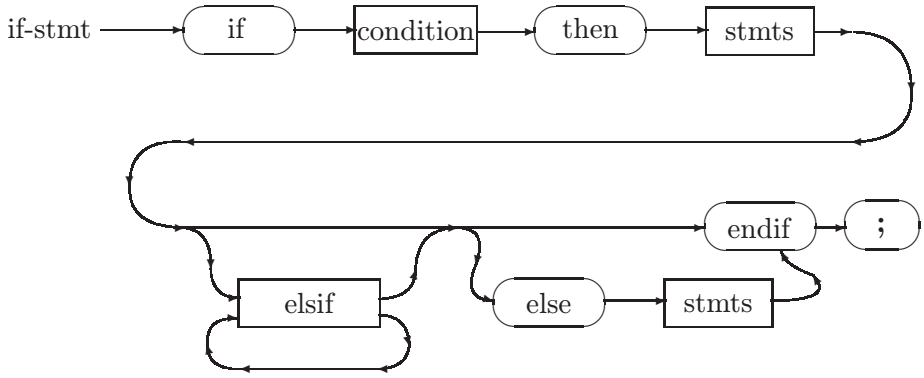
U této kategorie jazyků se setkáváme s tím, že syntaxe je podávána formální, či semiformální cestou. Jedním z důvodů je právě ukázat logiku a hierarchii jazykových konstrukcí, druhým je potom přehlednost a převzetí těchto základních formalismů i do širší obce uživatelů programovacích jazyků. Bezkontextové gramatiky potom stojí v základu takových popisů. Mezi typicky užívané popisy pro syntaxi patří zejména:

- (E)BNF — (rozšířená/extended) Backus-Naurova forma
- syntaktické grafy

Zápis části definice syntaxe pro podmínění příkaz v EBNF následuje:

```
<if-stmt>
  ::= if <condition> then <stmts> <else-if>
<else-if>
  ::= endif ;
   |  else <stmts> endif ;
   |  <elsif> <else-if>
```

Tentýž zápis lze pomocí syntaktického grafu potom vyjádřit například takto:



sémantika

Popis sémantiky je i v případě strukturovaných jazyků často veden neformální cestou. Popis je však často velmi propracovaný, jelikož představitelé této kategorie se dočkali i standardizace. Přístup, kdy vlastnost je vysvětlena na příkladě mizí, vysvětlení je obecné. Příklady jsou uvedeny pouze na dokreslení. V místech, kde to je vhodné, se, zejména v pozdější době, objevuje semiformální, nebo dokonce formální doplňky. Typickým rysem se od tohoto okamžiku je dlouhý a velmi přesný popis sémantiky.

4.2 Formalismy a užití

Jak bylo zmíněno, formální báze u těchto jazyků chybí a i když ji z velké části je možné zpětně vyrobit, tak pro ověření vlastností algoritmů zapsaných v daném jazyce se neužívá.

formální V&V

Pro blokově strukturované jazyky lze však pro formální ověření vlastností algoritmů použít Floyd-Hoare logiku (viz odkazy na konci kapitoly). Tato logika specifikuje pravidla pro práci se základními konstrukcemi jazyka (lehce aplikovatelná na řadu případů). Kromě toho určuje i pravidla pro práci s čísly a jejich ekvivalenty.

Floyd-Hoare logika

Pro každý programový prvek (příkaz, posloupnost příkazů, blok příkazů, apod.) je definována podmínka splněná před a po provedení operace popsané daným prvkem. Díky odvozovacím pravidlům je možné takové prvky dále slučovat (z oddělených příkazů vytvořit posloupnost, z posloupnosti blok, apod.).

Definice!

Definice 4.2.1 *Nechť C označuje příkaz, P označuje podmínku platící před provedením příkazu a Q podmínku platící po provedení příkazu, potom zápis:*

$$\{P\}\ C\ \{Q\}$$

vyjadřuje parciální správnost (korektnost) vztahu podmínek a příkazu a zápis:

$$[P]\ C\ [Q]$$

úplnou (totální) správnost.

Jelikož úplná správnost bere v úvahu úplnou množinu vstupů a často garantuje i zastavení výpočtu, tak se běžně užívá pouze parciální správnost, se kterou budeme pracovat i dále.

Pro ilustraci je dále uveden seznam základních příkazů, které je možné přímo zpracovávat a uvažovat v rámci Floyd-Hoare logiky. V seznamu označuje C (případně s indexem) příkazy, V (taktéž může být indexováno) proměnné, E reprezentuje výraz a S logický výraz:

- Přiřazovací příkaz:
$$V\ :=\ E$$
- Posloupnost příkazů:
$$C_1; C_2; \ldots C_n;$$
- Blok:
$$\text{BEGIN VAR } V_1; V_2; \ldots V_n;\ C \text{ END}$$
- Neúplný podmíněný příkaz:
$$\text{IF } S \text{ THEN } C$$
- Úplný podmíněný příkaz:
$$\text{IF } S \text{ THEN } C_1 \text{ ELSE } C_2$$
- Příkazy cyklu:
$$\text{WHILE } S \text{ DO } C$$
$$\text{FOR } V\ := E_1 \text{ UNTIL } E_2 \text{ DO } C$$
$$\text{REPEAT } C \text{ UNTIL } S$$
- A další.

K tomu, aby bylo možné ověřit, že námi zapsaný algoritmus je správný, nebo má určité vlastnosti, je třeba provést odvození takového důkazu. Odvození se opírá o tři základní prvky:

- Predikátová logika — využívá se logický důsledek, důkaz, atd.

- Axiomy — uplatňují se axiomy jak predikátové logiky, tak axiomy z Floyd-Hoare logiky.
- Odvozovací pravidla — odvozovací pravidla predikátové logiky stojí v základu, jsou však rozšířena o pravidla z Floyd-Hoare logiky.

Pouze jako příklad (jinak viz literatura na konci kapitoly) jsou uvedena odvozovací pravidla pro podmíněné příkazy:

$$\frac{\vdash \{P \wedge S\}\ C\ \{Q\},\ \ \vdash P \wedge \neg S\ \Rightarrow Q}{\vdash \{P\}\ \text{IF}\ S\ \text{THEN}\ C\ \{Q\}}$$

$$\frac{\vdash \{P \wedge S\}\ C_1\ \{Q\},\ \ \vdash \{P \wedge \neg S\}\ C_2\ \{Q\}}{\vdash \{P\}\ \text{IF}\ S\ \text{THEN}\ C_1\ \text{ELSE}\ C_2\ \{Q\}}$$

aplikace pravidel

Při odvozování samotném se odvozovací pravidla predikátové i Floyd-Hoare logiky uplatňují typicky odzadu směrem k začátku programu. Tento postup je nejefektivnější. Logika umožňuje i práci s cykly, pro

invariant cyklu

které je nezbytné správně a vhodně definovat tzv. *invariant cyklu*. To je taková podmínka, která se s jednotlivými průchody cyklem nemění. Právě správná volba invariantu umožňuje efektivní zvládnutí celé důkazové činnosti. Dále uvádíme, opět pro účely demonstrace, jeden krok důkazu ve Floyd-Hoare logice.

demonstrace

Premisy (T označuje vždy pravdivý výrok — true):

$$\vdash \{T \wedge (X \geq Y)\}\ \text{MAX}\ :=\ X\ \{\text{MAX} = \max(X, Y)\}$$

$$\vdash \{T \wedge \neg(X \geq Y)\}\ \text{MAX}\ :=\ Y\ \{\text{MAX} = \max(X, Y)\}$$

Odvozovací krok (pravidlo pro úplný podmíněný příkaz):

$$\vdash \{T\}$$
$$\text{IF}\ X \geq Y\ \text{THEN MAX} := X\ \text{ELSE MAX} := Y$$
$$\{\text{MAX} = \max(X, Y)\}$$

softwarové inženýrství

Z hlediska uplatnění metodologií známých z oblasti softwarového inženýrství se oproti jazykům nestrukturovaným situace zlepšuje. Například lze zcela uplatnit všechny zásady dekompozice problému jak v datové, tak v řídicí rovině. Nicméně, při pohled na další ukazatele zjišťujeme, že strukturované jazyky jsou stále omezovány např.:

- v rovině týmové spolupráce — program je tvořen jediným souborem, i když jej lze teoreticky sloučit z několika, tak tento postup určitě není ideální;

- v možnosti nechtěných změn při opravě — jediný soubor může být velmi nepřehledný, nechtěná změna nemusí znamenat chybu při překladu;

- v možnosti plné jak datové tak současně i kódové dekompozice — chybí moduly, takže stále je celý kód vidět a není možné skrývat manipulaci s daty apod.

Nicméně, celková rychlost tvorby programu (včetně jeho testování a případných oprav) a vzhled a charakter kódu jsou na mnohem vyšší kvalitativní úrovni. Také bezpečnost a odolnost vůči nechtěným chybám vzrůstá.

Shrnutí charakteristik

Tento typ jazyků je možné používat i pro větší projekty, i když velikost je stále shora pevně ohraničena — opravdu rozsáhlé projekty jsou bez modularity naprosto nemyslitelné.

Jazyky strukturované jsou typicky překládány do binárního kódu — vstupem je stále jen jeden soubor, takže je možné aplikovat různé optimalizace, kterou jsou navíc podpořeny strukturovaností návrhu.

Svou podstatou jsou tak velmi výhodné pro výuku — některé jazyky byly upraveny v praxi tak, aby byly použitelné i pro běžné účely a vhodné pro komerční nasazení (modularita, objektovost, apod.). Příkladem může být jazyk Pascal a např. prostředí Delphi. Kromě tohoto výjimečného případu však strukturované jazyky zůstávají na úrovni jazyků výukových, či výzkumných. Pokud nejsou tedy využity jen pro výukové/výzkumné účely, měly by být nahrazeny profesionálními jazyky, které již nabízejí plnou šíři vlastností a přitom nijak neomezují strukturované programování.

Pojmy k zapamatování

- (E)BNF, syntaktický graf
- Floyd-Hoare logika — užití, základní vlastnosti
- Možnost vzniku formálních bází
- Nasazení softwarového inženýrství u strukturovaných jazyků
- Využití strukturovaných jazyků

Výklad

4.3 Datové a řídicí abstrakce

Strukturované jazyky většinou nabízejí sadu základních, někdy zvaných atomických typů (i když formálně vzato se nejedná o totéž). Tyto zahrnují potom různé číselné, znakové a případně pravdivostní reprezentace.

odvozené typy

Kromě sady základních typů však umožňují definovat typy odvozené. Ty vznikají na bázi již existujících typů a to jak základních tak již existujících odvozených. Mechanismus vzniku nového typu má přitom programátor plně pod kontrolou.

ukazatele

Nově se také objevují ukazatele, které umožňují vznik rekurzivních datových struktur tím, že umožňují odkazovat určitému typu dat na sebe sama. Je tak možné definovat abstraktní datové typy jako seznam, strom apod. *Pozn.: mechanismus ukazatele nemusí být vždy explicitně vystaven ke zpracování a manipulaci programátorovi — viz např. deklarativní jazyky, jazyk Java, atd.*

Pro manipulaci s nově vytvářenými daty musí programátor definovat svoje vlastní operace. Přímo v jazyce je k dispozici často jen zpřístupnění nějaké složky datové struktury pro čtení/zápis. Pokud jsou v jazyce přítomny již předdefinované typy, tak pro práci s nimi jsou také definovány příslušné operace či funkce. Jejich přístupnost na volbu zatím není možná, neboť pro vznik knihoven zatím chybí modularita. Jedním z řešení, které se proto objevuje, je existence tzv. per-

pervasivní funkce

vasivních funkcí — tyto funkce jsou přítomny pro manipulaci s daty, které jsou v jazyce přímo definovány, nicméně uživatel může vytvořit funkci stejného jména (čímž původní od místa definice pozbývá).

definice/deklarace

Strukturované jazyky také často umožňují plně rozlišit pojem deklarace a definice. Jedna z nich přitom musí předcházet prvnímu užití symbolu. Deklarace spojené s typem přitom mají pouze omezený charakter užití (např. ukazatele na strukturu v jazyce Pascal). Umístění deklarací/definic přitom může být na různých místech — někdy je striktně vyžadováno na začátku programu/funkce, jindy je možné je umístit na začátku bloku. Význam deklarací je v úrovni řízení pro vznik vzájemně rekurzivních funkcí, pro data znamená vznik rekurzivních datových typů.

Návrh programu

Jak bylo zmíněno, tak strukturované programovací jazyky umožňují použít alespoň některé principy dobrého programovacího stylu. Za důležité v tomto směru lze považovat tyto vlastnosti:

- **vznik uzavřených podprogramů** — dramatický milník, který zjednodušuje:
  - **rekurzi** — vůbec první přímočará možnost implementace rekurze;
  - **ukrytí implementace** před ostatním programovým kódem — nelze náhodně do kódu přistoupit, jediný vstupní bod, jasně definované rozhraní vstupu/výstupu (parametry, výsledek);
  - **odluka od hlavního toku programu** — jednodušší a bezpečnější modifikace, explicitní vyvolání, atd.;
- lokální proměnné v zanořených blocích — proměnné stejného jména, různého významu, různých možností přístupu k nim.

Tyto nové vlastnosti také vedou k minimalizaci datových konfliktů, nicméně díky tomu, že se začíná rozvíjet týmová spolupráce, ač limitovaně, konflikty mezi různými autory mohou stále vznikat.

týmová spolupráce

Z hlediska týmové spolupráce sice strukturované jazyky nenabízejí takové možnosti, jako jazyky modulární, ale jisté možnosti tu jsou:

- uzavřené podprogramy mohou být vytvářeny nezávisle: datové manipulátory, čistě výpočetní (matematické) funkce, atd.;
- program je možné rozdělit na logicky nezávislé celky: tyto lze dále využít v jiných projektech.

Nevýhodou stále ale zůstává to, že program je ve finální podobě tvořen na textové úrovni jediným souborem. Přesto lze však vytvářet něco jako knihovny často užívaných funkcí, nebo typů, jakési šablony, které stojí na začátku každého dalšího projektu.

I když tyto jazyky jsou mnohem bohatší v konstrukcích nabízených programátorovi, tak na něj zároveň kladou větší nároky ve zvládnutí všech prvků jazyka a zejména potom v oblasti zpracování jednotlivých konstrukcí. Za to však poskytují posléze větší efektivitu při tvorbě a návrhu programu a v limitované míře poskytují i prvky týmové spolupráce.

Typy a jejich zpracování

Jazyky strukturované jsou typicky jazyky typované s tím, že programátor musí typ každé entity uvádět explicitně. Typ proměnné za běhu programu často zůstává od místa definice/deklarace tentýž a nedochází již k jeho automatické změně podle typu výrazu, jehož výsledek je do dané proměnné přiřazován. Jelikož však změna typu (převod mezi celými a desetinnými čísly, převod na textovou reprezentaci

přísné typování

a zpět, apod.) je typická vlastnost programů, tak se objevují vestavěné funkce a operátory, případně konvence v rámci pravidel jazyka, které tento převod (typovou konverzi) umožňují. Podle přísnosti jazyka potom rozlišujeme, zda určité konverze probíhají automaticky dle daných pravidel (např. převod celého čísla na desetinné), nebo je vždy nutná explicitní typová konverze (přísně typované jazyky). Pokud je převod proveden automaticky na základě pravidel jazyka, tak o vložení příslušné konverzní operace se rozhoduje v době překladu/analýzy. Existence možnosti vytvářet uživatelské typy klade však na překlad/analýzu vyšší nároky, což se může navenek projevit pomalejším během překladače.

Řešený příklad

Zadání: V níže specifikovaných případech rozhodněte, zda při překladu dojde k chybě, nebo implicitní typové konverzi. Pokud se bude jednat o implicitní konverzi, tak kdy se vykoná samotný převod typu — v době překladu, nebo až za běhu programu?

```
var x, y   : real;
    i      : integer;
    c      : character;
```

1. `x := y + 1024;`
2. `x := y * i;`
3. `i := x;`
4. `c := i;`

Řešení: Každý podbod zadání příkladu budeme řešit zvlášť.

ad 1) V tomto případě dojde k automatické typové konverzi, a co více, tato konverze je provedena již v době překladu, neboť 1024 je číselný literál. Ten je převeden na hodnotu odpovídajícího typu a do výsledného kódu tak uložen, takže v době běhu programu již k dalším konverzím nedochází.

ad 2) V případě násobení těchto dvou proměnných také dojde k automatické typové konverzi. Proměnná i, respektive její hodnota, je konvertována na desetinné číslo. Jelikož se jedná o proměnnou, tak konverzní operace je vložena do výsledného kódu a děje se tak za běhu programu.

ad 3) Při přiřazení desetinného čísla do proměnné celočíselného typu překladač bude hlásit chybu. Zde je nutné explicitně použít nějakou konverzní funkci, např.

```
i := round(x);
```

Takové funkce často nejsou triviální, ale mohou být obstarány prostřednictvím operací matematického ko-procesoru, je-li jím cílový systém vybaven.

ad 4) I v tomto případě ohlásí překladač chybu. Celé číslo a znak nejsou shodné typy, i když by se tak mohlo zdát. Opět je nutné použít konverzní funkci:

```
c := chr(i);
```

V tomto případě se však jedná o triviální operaci, která je řešena jednoduchou instrukcí cílového systému.

ukazatele

Novým typem, který se začíná výrazně prosazovat je ukazatel. Tento typ nese adresu teoreticky libovolné paměťové buňky v cílovém systému. V praxi to tak, díky optimalizacím, nebo druhu architektury být nemusí. Ideální je, pokud cílový systém má tzv. plochý adresový prostor — potom totiž pro ukazatel stačí vybrat nejmenší celočíselný typ, který pojme všechny adresy. Často však taková ideální situace není. V úvahu je třeba brát segmentaci paměti, stránkování (to však často bývá transparentní, takže není třeba programově ošetřovat), oddělené datové a kódové paměťové prostory (harvardská architektura). Také je nutno vzít v úvahu zarovnání paměti a velikost slova. Např. u paměti organizované po 16 bitech se typicky přistupuje na sudé adresy, bráno po bytech. V případě, že je nutné přistoupit na lichou adresu, je potom nutné často vložit speciální obslužný kód, který vymaskuje příslušnou část slova.

segmentace

Dále se blíže ji podíváme na segmentaci a harvardskou architekturu. Segmentace byla běžná pro počítače třídy IBM PC a operační systém MS-DOS. Procesory v těchto počítačích byly implicitně 16 bitové, ale paměť (v počátku) teoreticky 1MB (později i více). I tak ale pomocí 16 bitů jsme schopni adresovat jen 64kB, takže ať byla velikost jakákoliv, bylo to nad sílu 16 bitů. Pro adresaci se tak použilo dvou 16 bitových čísel. Jedno udávalo segment a druhé posunutí (offset). Při vytvoření vlastní adresy se pak obě čísla sečetla s posunutím 4 bitů:

```
offset  (16 bitů)          XXXXXXXXXXXXXXXX
segment (16 bitů)      XXXXXXXXXXXXXXXX0000
-----------------------------------------
adresa 1MB (20 bitů)   YYYYYYYYYYYYYYYYYYYY
```

V programu jsme potom mohli pracovat s různými paměťovými modely, které umožňovaly pracovat jen v rámci segmentu (ukazatel tvořen

jen offsetem, 16 bitů), nebo i ve více souvisle navazujících segmentech (ukazatel složen z části segment a offset, 32 bitů), s různě velkými daty.

harvardská architektura

Harvardská architektura logicky odděluje prostor pro data a pro uložení kódu programu. Jedna hodnota adresy tak znamená něco jiného, podle toho, do jaké paměti přistupujeme. Pokud jsme schopni detekovat, že ukazatele pracují jen s daty, tak výsledný kód není nijak zvláštní, pokud však přistupujeme do obou, tak už dochází k tomu, že musíme být schopni za běhu programu rozpoznat, zda ukazatel ukazuje do té, či oné paměti. Hlavní důvod je ten, že instrukce pro přístup do každého typu paměti se liší! Řešení je poměrně jednoduché, i když ne příliš efektivní. Ukazatel se skládá ze dvou částí: samotné adresy (v úvahu se bere větší paměť z obou) a ze značky. Značka ukazuje, do které paměti směřuje adresa v druhé části ukazatele. Kód pro práci s ukazateli je však potom poměrně pomalý, neboť se větví podle typu ukazatele. Navíc přístup do kódové paměti je často možný jen velmi omezeným způsobem. Na úrovni programu je proto vhodné se konstrukcím, které by přistupovaly do obou pamětí, vyhnout.

Úlohy k procvičení:

Uvažte jednočipový počítač harvardské architektury (na napájení nezávislá je pouze paměť pro kód). Jak by se realizovalo inicializované pole? Jak by se realizovalo inicializované pole konstant? Vysvětlete rozdíl a hlavní úskalí řešení.

datové struktury

Datové struktury, jejichž tvorba je plně pod kontrolou uživatele, jsou revolucí strukturovaných programovacích jazyků na úrovni tvorby a definice dat. Z formálního pohledu je typ popisující datovou strukturu n-tice, která je jednoznačně identifikována svým jménem. Podobně, každá složka této n-tice je pojmenována a pouze skrze toto jméno je dále zpřístupněna. Každá složka má také typ, přičemž se opět může jednat o datovou strukturu. Součástí, nebo variantou datové struktury je **variantní datová struktura**, která je sjednocením jednotlivých složek struktury. Každá složka má také své jméno a typ (opět libovolný) a je možné k ní přistupovat jak pro čtení, tak pro zápis. Jednotlivé složky se však překrývají, takže aktivní je vždy právě jedna.

uložení v paměti

Různé chování n-tic a variantních struktur se promítá i do uložení těchto struktur v paměti. N-tice ukládají jednotlivé složky v paměti za sebou, zpravidla bez mezer tak, aby struktura zabrala co nejméně paměti. I zde jsou však výjimky, které jsou dány cílovou architekturou a požadavky na optimalizace (rychlost/velikost obsazené paměti): způsob zarovnání paměti (na byte, slovo, ...), přístup do paměti (někdy jiný přístup než ke slovu není možný), využití paměti, bitová pole,

atd. Variantní datová struktura ukládá jednotlivé své složky v paměti přes sebe. Velikost obsazené paměti je potom určena největší složkou struktury. Často se tak variantní datové struktury využívají pro implementační triky (uložím jako jednu složku/typ, přečtu jako jinou složku/typ). Tato struktura se přitom v paměti ukládá tak, aby byla jako celek co nejvýhodněji zarovnána, což může být (díky obsaženým strukturám) někdy problematické, nebo může vést k pomalému přístupovému kódu pro určité složky.

manipulace se strukturami

Manipulace se strukturami potom obnáší generování přístupového kódu, který je často vkládán přímo do míst, kde se přístup k dané složce děje. V určitých případech je však možno využít přístupových funkcí, které se volají v místě potřeby (často neefektivní a ovlivněno charakterem cílové architektury). Hlavní zásadou však vždy je nediskriminovat nějakou složku struktury, pokud k tomu není důvod (optimalizace apod.).

adresa složky

Adresa složky struktury se skládá ze dvou údajů: z adresy datové struktury jako takové a z posunutí (offset) v rámci ní. Dle umístění proměnné se strukturou je potom možné tyto údaje vyčíslit staticky (v době překladu), nebo je třeba dynamické vyhodnocení (až za běhu).

bitová pole

Bitová pole svoji velikost zarovnávají často na velikost nejbližšího vyššího byte-u, či slova. Může tak vzniknout prostor několika nevyužívaných bitů. Pro přístup k složce bitového pole je však nutné znát kromě adresy struktury a posunutí bitového pole v ní ještě posunutí bitového pole v rámci úseku bitových polí (pokud by bylo jen jedno, tak je toto druhé posunutí nulové, ale zpravidla se počet bitových polí za sebou různí od 1). Pro vyčtení/zápis hodnoty však nestačí kód pro přístup do paměti, ale je nutné použít instrukce/operace pro posunutí/rotace v rámci slova/byte a operace pro maskování; u znaménkových číselných typů je pak nutné číslo převést na plný formát/nebo zmenšit, aby jeho další zpracování bylo korektní.

pole

Datový typ pole specifikuje homogenní datovou strukturu, kdy každá složka je pevně indexována určitou hodnotou a všechny složky jsou stejného typu. Složky jsou umístěny v paměti za sebou, dle indexu, ale nikoliv nutně bez mezer (určeno jazykem, architekturou) — problémem je pole struktur, kde z definice jazyka musí být těsně za sebou, ale pro přístup ke složkám pole to může znamenat problém díky zarovnání paměti. V případě nutnosti je potom přístup k takové složce pole velmi neefektivní díky náročnému přístupovému algoritmu, který čte větší část paměti, než je nutné, a tuto část musí „přerovnat“ tak, aby odpovídala zarovnání, a teprve potom je možné aplikovat „klasický“ kód. Určení adresy složky pole i s definicí obecného příkladu jednorozměrného pole vypadá takto:

```
a : array [x..y] of <typ>
a[i] ~ <báze> + (i-x) * sizeof(<typ>)
```

Z uvedeného lze vyvodit, že výraz obsahuje jisté konstantní podvýrazy: bázová adresa pole, dolní mez pole, velikost typu jedné složky pole. Pokud tedy výrazu upravíme, dostaneme:

```
a[i] ~ <báze> - x * sizeof(<typ>) + i * sizeof(<typ>)
```

mapovací funkce

kde podvýraz `<báze> - x * sizeof(<typ>)` je konstantní, dá se vypočítat v době překladu a nazývá se **mapovací funkce** (mapping function). Podobné úpravy lze vytvořit pro pole o libovolném počtu dimenzí.

kompatibilita typů

Kompatibilita typů u polí datových struktur se posuzuje dvěma způsoby:

- shoda typu na jméno,
- shoda typu dle shody vnitřní struktury.

Kompatibilita se posuzuje u přiřazení, předávání parametrů a podobně. U polí však můžeme někdy pozorovat odlišné definice chování od datových struktur, protože některé jazyky neumožňují přiřazení pole do pole (je nutno obcházet programově: program pro kopii, vložením pole do datové struktury).

Způsob předávání struktur do podprogramů a vracení jako výsledek z funkcí je prvkem, který významně ovlivňuje možnosti programovacího jazyka i to, jak je jazyk překládán. U předávání struktur jako parametrů předávaných hodnotou existují tyto přístupy:

struktura parametrem

- **Nelze**
  V takovém případě jazyk nepodporuje předávání datových struktur jako parametry předávané hodnotou.
- **Předání odkazem**
  Jazyk manipuluje datové struktury pouze prostřednictvím odkazů. Textově se zápis neliší od zápisu předáváním jednoduchých parametrů hodnotou, implementačně se však jedná o předávání odkazem — např. Java.
- **Předání hodnotou**
  Implementace je standardním způsobem, nicméně může být limitována velikost takto předávané struktury. Pro velké struktury je také možná implementace, kdy se nevytváří při volání podprogramu kopie dat struktury na standardní místo (typicky programový zásobník), ale vytváří se na místo na haldě (heap) a potom se k ní přistupuje odkazem. Takováto implementace je však pochopitelně pomalejší.
  Pozn.: I z hlediska tvorby a návrhu programu je proto dobré se předávání velkých datových struktur jako parametrů předávaných hodnotou vyhnout. Podobně i u výsledků funkcí.

U předávání struktur jakožto výsledků funkce je situace obdobná, některé jazyky to nedovolují (Pascal), jiné struktury manipulují odkazem (Java) a některé umožňují strukturu jako výsledek operace (jazyk C). Posledně jmenovaný přístup je opět náročný v případě velkých struktur, kdy předání přes registry, či zásobník není možné, nebo je nevhodné. I v tomto případě se využívá paměti na haldě k tomu, aby se data struktury dočasně uložila.

struktura výsledkem

Tok řízení programu

Blokově strukturované jazyky definují příkazy pro vytvoření bloku příkazů. V určitých případech může takový blok obsahovat i lokální definice proměnných. Tyto konstrukce lze vnořovat a jelikož nahrazují příkaz, tak je lze prakticky libovolně začleňovat do ostatních programových konstrukcí.

I jiné programové konstrukce lze vzájemně vnořovat a kombinovat, jedná se o smyčky/cykly (for, while–do, repeat–until), podmíněné příkazy (if–then–else), příkaz násobného větvení (switch/case). Vliv příkazu skoku (goto) je umenšen a je snaha ho co nejvíce eliminovat — pro ukončení či vynucení dalšího průchodu cyklů/smyček se zavádějí nová klíčová slova (break, continue). Nicméně příkaz skoku stále zůstává v definici takových jazyků.

Vnořování, kromě výše zmíněných příkazů, je možné i pro definice funkcí (Pascal). Často je však omezen počet takovýchto vnoření (např. 7). V případě vnoření na úrovni textu programu potom hovoříme o **statickém zanoření** a jeho **úrovni**. V době běhu programu, když se jednotlivé funkce navzájem volají, případně rekurzivně sami sebe potom hovoříme o **dynamické úrovni zanoření**. Staticky vnořené definice funkcí díky tomu, že vnořená funkce má přístupné všechny nadřazené definice a to nejen funkcí, ale zejména proměnných, vyžaduje na implementační úrovni další prvky pro přístup k proměnným z nadřazených definic. Užívaná řešení jsou dvě:

statické zanoření

dynamické zanoření

- Display
  Paralelně k programovému zásobníku existuje další, pomocný zásobník, který obsahuje ukazatele na aktuální pozici proměnných z dané statické úrovně.

- Přístupové ukazatele
  Každá funkce z vyšší statické úrovně zanoření dostává jako skryté parametry ukazatele na aktuální proměnné z vyšších úrovní.

Tyto vlastnosti přinášejí nové možnosti i do návrhu a tvorby pro-

přínos vlastností strukturovaných jazyků

gramů. Z dnešního pohledu jsou to zejména pozitivní vlastnosti umožňující např. práci s abstraktními datovými typy, využití návrhových a implementačních technik apod. Nicméně staré zvyky z nestrukturovaných jazyků je stále možné, zejména díky příkazu skoku, aplikovat — toto nese nevýhodu v tom, že strukturované nemodulární jazyky je možné vysoce optimalizovat, ale toto je možné zejména, pokud jsou využity pouze blokově strukturované konstrukce. Avšak jsou-li dodržena pravidla pro správnou tvorbu strukturovaných programů, je možné dokonce již v době překladu jisté chyby v implementaci či návrhu detekovat.

Pojmy k zapamatování

- Základní, odvozené typy, ukazatele — vlastnosti, způsoby a problematika implementace
- Pervasivní funkce
- Možnosti řízení toku vyhodnocení programu — typické obraty

4.4 Zpracování — analýza, vyhodnocení, překlad

Strukturované jazyky (bez dalších vlastností, jako např. modularita) se i v minulosti vyskytovaly poměrně řídce (nejznámější je čistý Pascal). Proto lze těžko hledat typickou formu pro zpracování, či vyhodnocení takových jazyků.

Překladače

U strukturovaných programovacích jazyků lze lexikální, syntaktickou i sémantickou analýzu zpravidla provést najednou v rámci jednoho průchodu zdrojovým textem. Jazyky lze běžně popsat z hlediska syntaxe bezkontextovými gramatikami, na základě kterých je možné analýzu postavit. Kontextové vazby, které jsou v jazycích pochopitelně zastoupeny, se řeší pomocí kontextových tabulek. Díky tomu, že bloky lze staticky vnořovat, jsou i tabulky symbolů víceúrovňové.

lexikální analýza

V době lexikální analýzy nicméně je třeba rozeznávat (v porovnání s předchozí kategorií jazyků) více kategorií lexikálních symbolů. Jelikož je celý program obsažen v jednom souboru a navíc je možné snadněji vytvářet rozsáhlejší programy, tak se lexikální analýza stává nejzatíženější částí překladače. Některé jazyky díky přílišnému vlivu

nestrukturovaných jazyků mají kontextově závislou množinu lexikálních symbolů — někdy je změna řízena ze syntaktické či sémantické analýzy, jindy si změnu řídí sám lexikální analyzátor na základě předchozího/následujícího symbolu.

syntaktická analýza

V syntaktické analýze se využívají techniky pro analýzu bezkontextových jazyků, jako je prediktivní analyzátor jazyků třídy $LL(k)$ — k je typicky 1, pracuje metodou shora dolů — či analyzátory pracující mechanismem zdola nahoru — $(LA)LR(1)$ jazyky. Využívají se také konstruktory syntaktických analyzátorů (y.a.c.c, bison).

sémantická analýza

Sémantická analýza potřebuje víceúrovňové tabulky symbolů, kdy implementačně je možné potkat se s několika přístupy (viz druhou část podkladů). Je možné se také setkat s oddělením jmen prostorů pro různé entity a dle toho také různými pravidly pro jejich užití (např. návěští pro cíle skoků versus jména lokálních proměnných). V době kompilace lze velmi často (jazyky strukturované jsou často typované) provádět typovou kontrolu, ověření, zda existuje entita potřebného typu a jména, detekci cíle skoků. Částečně lze detekovat to, zda entita byla před prvním čtením inicializována (to už se dále nezlepší). Nelze však definovat chybné užití proměnných (překrytí definic) a chyby způsobené chybnou typovou konverzí.

Interprety

Interprety strukturovaných jazyků jsou často tzv. *školní* interprety, nebo výzkumné. Obecně jsou zřídkavé. Je to dáno tím, že dosah definic v rámci bloku má poměrně dlouhý dosah, takže by bylo nutné si pamatovat, kde se nacházíme. Ve spojení s příkazy skoku se situace dále komplikuje — skok mimo blok, skok dopředný obecně, atd.

Řešení situace do značné míry kopíruje způsob implementace překladače. Analýza je prováděna vždy na úrovni celého bloku (někdy dokonce bloku na nejvyšší úrovni). Také je typický překlad do jisté vnitřní reprezentace, nad kterou se posléze provede vyhodnocené/interpretace — typickou reprezentací je graf, nebo určitý abstraktní kód (P-kód, apod.). V důsledku tak vlastně dochází k překladu, neboť vyhodnocení programu není tak okamžité, jak by teoreticky mohlo být, a dochází k němu vlastně až v druhém průchodu zpracování. To přináší jak výhody, tak nevýhody:

- detekce (některých) chyb je možná mnohem dříve než u čisté interpretace;
- spotřeba paměti roste;
- vykonání programu je rychlejší (odpadá opakovaná analýza zdrojového textu);

- použití v režimu inkrementálních změn programu a jejich okamžitého promítnutí do vyhodnocení, kdy program je zastaven, modifikován a spuštěn od místa zastavení,je však velmi obtížné.

Pojmy k zapamatování

- Konstruktor analyzátoru
- Kontextově závislá lexikální analýza
- Víceúrovňová tabulka symbolů
- Překlad do interní reprezentace/mezikódu
- Možnosti sémantických kontrol v době překladu

Závěr

Úspěšným zvládnutím této kapitoly proniknete do zásad a typických vlastností strukturovaných programovacích jazyků. Měli byste vědět, jak tyto jazyky pracují s typy, jaké nabízejí typické konstrukce pro řízení toku vyhodnocení/běhu programu a konstrukce pro práci s daty. Také byste měli vědět, jak tyto jazyky zpracovávat, s čím je třeba počítat, co naopak může chybět. Neměl by vám zůstat utajen pojem logiky Floyd-Hoare, její možnosti a uplatnění.

Úlohy k procvičení:

Uvažte definici jazyka Pascal (od prof. Wirtha). Definujte v něm abstraktní datový typ (ADT) polymorfní seznam (strom). Definujte/navrhněte implementaci i operace k danému typu (užijte zejména rekurzi).

- Je něco podobného možného i v Basicu (uvažte úlohy předchozí kapitoly, neuvažujte Visual Basic)?
- Pokud ano, jak?
- Které části implementace/návrhu jsou ve strukturovaném jazyku složitější?
- Bylo by možné některé operace definovat bez rekurze?
- Jaké datové typy jsou zejména zastoupeny v definici ADT?
- Bylo by možné nějak využít datového typu pole v implementaci? Jak?

V jazyku Pascal (od prof. Wirtha) definujte algoritmus pro výpočet faktoriálu s použitím cyklu. Dokažte, pomocí logiky Floyd-Hoare, že je správný.

Klíč k řešení úloh

Zaměřte se na polymorfii ADT. Uvažte možnosti, jak lze programově obcházet nedostatky jazyků. Uvažte rychlost vyhodnocení a vlastnosti operací nad danými ADT.

Najděte invariant cyklu.

Další zdroje

Schmidt, D.A.: *The Structure of Typed Programming Languages*, MIT Press, 1994.
Gordon, J. C.: *Programming language theory and its implementation*, New Jersey : Prentice-Hall, 1988.
Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading MA, 1986.
Pittman, Peters: *The Art of Compiler Design*.

Kapitola 5

Modulární jazyky

Tato kapitola seznamuje se základními charakteristikami a problematikou modulárních imperativních jazyků, které navíc už mají i vlastnosti jazyků strukturovaných. Dále rozvíjí a doplňuje pojmy definované v Kapitole 2.

Čas potřebný ke studiu: 5 hodin.

Cíle kapitoly

Cílem kapitoly je blíže se seznámit s obecnými vlastnostmi modulárních programovacích jazyků — co od nich můžeme očekávat a jak by se daly některé vlastnosti implementovat. Koncentrovat se budeme na vazby mezi moduly, jak se vytvářejí, co to znamená při zpracování jazyka a na spojování modulů do jednoho celku.

Průvodce studiem

Tato kapitola předpokládá dobrou orientaci v základních pojmech z klasifikace a popisu programovacích jazyků. Na tyto pojmy navazuje a dále je rozvíjí na skupině modulárních imperativních jazyků s tím, že předpokládá u nich už i vlastnosti blokově strukturovaných jazyků (zejména na úrovni datových a řídicích struktur), proto je nutná dobrá znalost i této problematiky. Přitom se zabývá těmito jazyky jak z pohledu uživatele: možnosti a vhodnosti nasazení, očekávané vlastnosti, apod.; tak z pohledu implementace a typických problémů s ní spojenými — zejména vliv modularizace na generování cílového kódu a problematika spojení modulů.
Ke studiu této kapitoly není třeba žádného speciálního vybavení. Jako u většiny ostatních se jedná o to si zapamatovat danou terminologii a pochopit klíčové principy předkládaných konceptů. Pro hlubší studium, nebo bližší objasnění prezentovaných skutečností je vhodné využít informací na Internetu a vlastní praktické ověření vlastností na konkrétním jazyce.

Obsah

Výklad

5.1 Základní charakteristika

Jazyky modulární se objevují jako další vývojový stupeň, který umožňuje další rozvoj týmové spolupráce a to nejen v rámci jednoho pracoviště, ale i mezi několika subjekty, kdy uživatel vlastností určitého modulu už nedostává do rukou zdrojový kód modulu, ale pouze kód binární a popis, jak využít funkčnost modulu — *rozhraní modulu*. Tyto vlastnosti můžeme pozorovat i u jazyků nestrukturovaných, ale plný rozvoj nastává až s jazyky strukturovanými a širším užitím výpočetní techniky vůbec.

Mezi typické představitele lze zařadit jazyky Modula-2, Ada, C, jazyky typu ML (mají i vlastnosti jazyků funkcionálních) a další. U jazyků odvozených od jazyka Pascal, jako je Turbo-Pascal, Free-Pascal, Object-Pascal (má i objektové vlastnosti) a jiné byla implementována rozšíření, která umožňují také tvorbu modulů.

formální báze

Modulární jazyky formální bázi primárně nemají kromě jazyků vznikajících v současnosti, či nedávné minulosti — příkladem může být jazyk ML, který má však funkcionální rysy a čistý imperativní implementační styl nelze zcela bezezbytku využít, aniž by tím utrpěla čitelnost programu apod. Nicméně pro řadu z nich, nebo alespoň jejich velkou podmnožinu, by bylo možné takový formalismus dodatečně definovat (viz literatura na konci této kapitoly). Objevilo se dokonce několik projektů, které pro některé jazyky dokonce chtěly doplnit chybějící formalismus, ale díky určitým vlastnostem to nebylo možné, nebo byl výsledek v praxi málo využitelný. I když formální báze tedy chybí, tak tyto jazyky se snaží využít předchozí dobré vlastnosti a eliminovat nedobré charakteristiky jazyků předchozích.

syntaxe

U této kategorie jazyků je syntaxe podávána formální, či semi-formální cestou jako u jazyků strukturovaných. Mezi typicky užívané popisy pro syntaxi tedy patří zejména:

- (E)BNF — (rozšířená/extended) Backus-Naurova forma,
- syntaktické grafy,
- formální gramatiky — zejména bezkontextové.

sémantika

Sémantika modulárních jazyků, kromě těch, co mají i formální bázi, je podávána neformálně. Nicméně, díky vzniku modulů je umožněn vznik i knihoven funkcí, což vede k jejich standardizaci, aby byla možná jejich širší využitelnost a nasazení v praxi. Vznikají standardy knihoven k jednotlivým jazykům, které jsou velmi rozsáhlé a jejichž

popis je velmi precizní, ač neformální — standard ANSI-C je na několika stovkách stran. Často je popis doplněn příklady pro rychlejší pochopení náročného popisu. Kromě toho se příklady stávají nedílnou součástí popisů takových jazyků, neboť programování a výpočetní technika se dostává do stále většího počtu oborů a je potřeba po stále větším počtu programátorů.

5.2 Formalismy a užití

Jak bylo zmíněno, formální báze u těchto jazyků velmi často chybí. Tam, kde byla zpětně vytvořena se v praxi neužívá. Tam, kde byla vytvořena dopředu, je možné ji využít pro jednodušší důkazy vlastností některých algoritmů, nebo jejich částí. Typicky se však tato technika užívá pro funkcionální jazyky, nebo části programů, které tomuto paradigmatu vyhovují.

formální V&V

U jazyků, kde formální báze chybí, nebo je její užití náročné, lze využít logiky Floyd-Hoare. Pokud má být důkaz rozsáhlejší, tak je třeba sáhnout po modifikacích vhodných pro modulární jazyky (jen drobná rozšíření), nicméně díky vlastnostem, které v modulárních jazycích často přetrvávají z dřívějška, je aplikace postupů logiky Floyd-Hoare omezená.

softwarové inženýrství

Z pohledu softwarového inženýrství se modulární jazyky dostávají na vrchol a koncept modularizace je úspěšně využíván dodnes! I když modularita umožňuje a přivádí na vrchol možnost týmové spolupráce a dekompozici problémů, tak vyžaduje pochopení a aplikaci řady postupů už v době návrhu programového vybavení. Mezi nejzákladnější postupy pro dekompozici problému na nezávislé části patří metoda shora dolů, či zdola nahoru.

Každý modul se tak může stát nezávislým projektem, který má definováno své vlastní zadání, tým, který jej realizuje, návrh, implementaci, testovací postupy, nasazení a údržbu. To vše přitom může být zcela nezávislé od ostatních modulů, avšak, pro celek složený z modulů potom existují vyjmenované aktivity odděleně a život modulu patřičně ovlivňují.

Modularizace však přináší i nový typ chyb, které se v programech objevují a které je nutné detekovat. Mezi ty nejtypičtější patří:

- Nemožnost spojit moduly do výsledného celku díky špatné specifikaci modulů, či nevhodné dekompozici (program třeba i lze sestavit, ale ten nepracuje spolehlivě/správně).
- Program lze sestavit, ale díky špatné logice řízení toku programu celek nepracuje správně, či „zamrzává“.

- Nevhodné užití návrhových metodologií pro modularizaci, či vazby mezi moduly jak na úrovni dat, tak na úrovni toku řízení — grafy toku řízení, konečné automaty, události apod.

Shrnutí charakteristik

Pokud není přímo vyžadováno, nebo není z pohledu daného problému výhodnější objektově orientované programování, potom je modulární (blokově strukturované) programování současně nejvýhodnějším přístupem k návrhu a tvorbě programového vybavení. Jeden z oborů, kde takovýto typ jazyků v současnosti nachází široké uplatnění jsou vestavěné systémy (embedded systems), kde požadavky kladené na cílový systém často nevyžadují, nebo neumožňují nasazení objektově orientovaného programování.

Modulární programování a tím tedy i modulární programovací jazyky v jakékoliv podobě jsou vhodné pro týmovou práci při návrhu a tvorbě programového vybavení (rozhraní modulu a funkčnost poskytovaná skrze toto rozhraní jsou často dostatečným zadáním).

Modulární programovací jazyky jsou velmi často ve formě překladače a k němu náležejícímu spojovacímu programu (linker) — viz níže. Mezi zástupci modulárních jazyků často nalezneme i příklady jazyků vhodných pro programování operačních systémů, nebo na úrovni HW architektury. Údržba modularizovaného kódu je poměrně snadná, což je další jeho výhoda. Zde je však nutné poznamenat, že tato vlastnost je silně ovlivněna už návrhem daného programového vybavení a dále programovacím stylem jednotlivých implementátorů.

Na druhou stranu, pro extrémně velké programy modularita spolu se strukturovanými vlastnostmi je často zastíněna výhodami objektově orientovaného přístupu a proto by měl být uplatněn ten (pokud není nějaký důvod, proč jej nepoužít).

Pojmy k zapamatování

- Existence formální báze před implementací jazyka
- Modul, jeho rozhraní
- Knihovny, standardizace
- Vliv modularizace na tvorbu programového vybavení
- Nasazení modulárních (blokově strukturovaných) jazyků

Výklad

5.3 Datové a řídicí abstrakce

Modulární jazyky v oblasti typů nezavádějí nic nového a využívají vlastností z jiných paradigmat — nejčastěji tedy blokově strukturovaných jazyků. Jako nový typ lze brát modul. Z uživatelského hlediska se jistě o typ nejedná, ale formálně (viz reference na konci kapitoly) se na modul můžeme jako na jakýsi typ dívat.

role knihoven

Moduly umožňují ale vznik **knihoven**, které poskytují kompletní sadu funkcí pro manipulaci s určitým abstraktním typem, případně poskytují jakési zázemí pro danou výpočetní tématiku. Typická je však kombinace datových typů a jejich manipulátorů na modul. Potom moduly jistě nabývají na významu, neboť umožňují daleko širší nasazení abstraktních datových typů s tím, že implementace je před uživatelem skryta v binární části modulu. Uživatel pak nemůže třeba nechtěně implementaci změnit na chybnou. V případě jistého know-how tak lze dokonce zakrýt klíčové prvky užitých algoritmů (minimálně lépe, než poskytnutím zdrojových kódů).

skrývání dat

Pro data definovaná a nabízená modulem, stejně tak jako pro data skrytá v modulu, platí, že modul se o ně stará v plném rozsahu. Pokud je zpracovávaná problematika na jeden modul příliš široká, tak místo jednoho modulu vzniká knihovna, která ale plní obdobnou funkci. Může se stát, že nějaký vnitřní modul je potom v rámci knihovny využit několika jinými moduly. Pro výpočetní modul to typicky nebývá problém, ale pro datový manipulátor to může být problém, zvláště pokud na to nebyl stavěn. V případě, že jsme autory modulu, nebo máme jeho zdrojové kódy, je možné situaci elegantně zvládnout, v případě, kdy kód chybí, tak je třeba věnovat podobnému případu pozornost a předejít případným chybám.

rozhraní modulu

O tom, jaká data, datové typy a funkce/procedury jsou vyváženy z modulu rozhoduje jeho rozhraní (interface). To obsahuje deklarace (zřídkavě i definice) pro jednotlivé konstrukce, které je možné využít vně modulu. V implementační části potom skrývá kromě vyvážených entit i definice dalších typů, dat a procedur/funkcí, které zajišťují správnou činnost modulu. O tom, že situace se i tak může komplikovat svědčí malá ukázka na jazyku C. V jazyku C je přípona souboru implementující modul „.c“, rozhraní modulu je uloženo v souboru stejného jména až na příponu, tou je „.h“ — tzv. hlavičkový soubor (header file). Jazyk C nezná pojem deklarace typu, jen proměnných a funkcí. Oba soubory, tedy jak implementační, tak soubor rozhraní jsou vytvářeny explicitně (jeden není automatizovaným způsobem z druhého

příklad

odvozen/generován). I kdyby tomu tak bylo, tak zabránit manipulaci s textovým hlavičkovým souborem nelze. Může tak dojít k tomu, že definice typu, či deklarace funkce, či proměnné se může lišit v hlavičkovém a implementačním souboru, což je pochopitelně chyba, kterou lze někdy odhalit v době spojování modulů, někdy až za běhu programu (jeho chybnou činností) — viz níže, spojovací program.

Návrh programu

Programová dekompozice se velmi často odehrává na úrovni dat, neboť modulární jazyky často disponují i vlastnostmi blokově strukturovaných jazyků, přičemž obě se výhodně doplňují. Jednotlivé moduly kombinují datové typy a k nim příslušné obslužné funkce. Moduly lze skládat do hierarchií, přičemž mezi typické struktury patří stromová hierarchie, někdy se jedná o acyklický orientovaný graf (DAG). V případě cyklických závislostí mezi moduly je výsledná struktura po kolapsu silně souvislých grafových podstruktur opět stromová či DAG — některé jazyky neumožňují cyklickou závislost mezi moduly a potom je nutné umístit potřebná data i funkce do modulu jediného i za cenu zhoršení kvality implementace. U nestromových struktur závislosti je nutné vzít v úvahu riziko násobného užití modulu, který podobné zacházení ale nepodporuje.

závislost mezi moduly

Většina programovacích jazyků nepodporuje skrývání definic typů (podpora deklarací typů) na vhodné úrovni, často vůbec. Potom jsou typy vyváženy kompletně za cenu nebezpečí jejich externí manipulace, nebo implementace vyváží pouze ukazatel na typ, aby skryla implementační detail. Tímto ale dochází k záměně jednoho problému jiným a takovýto návrh nese další rizika, navíc těžko odhalitelná v době překladu.

deklarace typů

I přesto však tvorba (uživatelských) knihoven je velkým milníkem v oblasti tvorby programového vybavení, který byl položen díky modulárním jazykům. Další výhodou je umožnění skrývání implementace v implementační části modulu (nutné pro širší nasazení a tvorbu modulů/knihoven na zakázku, třetí stranou). Přitom doba učení není o moc delší než u jazyků se stejným základem datové abstrakce. Nevýhodou může být nemožnost sdílení jednoho modulu více dalšími, pokud na to není přípraven a zdrojové texty jsou nedostupné.

knihovny

skrývání implementace

Úlohy k procvičení:

Vyberte si vhodný strukturovaný modulární jazyk. Pro hypotetický případ navrhněte systém modulů, který by mohl danému případu implementačně vyhovět. Jakou grafovou strukturu tyto moduly tvoří?

Zvolte jiný případ tak, abyste v návrhu obsáhli všechny zmíněné grafy pro závislost modulů. U případu s cyklickou závislostí se zamyslete nad možnými způsoby odstranění cyklu.

Typy a jejich zpracování

Modulární jazyky nepřinášejí samy o sobě žádné typové konstrukce, či abstrakce. Proto je jejich zpracování v zásadě podobné tomu, na jakém paradigmatu (datovém) je jazyk založen. Uvažme jazyky blokově strukturované. Při překladu hraje velkou roli umístění jednotlivých datových struktur/proměnných v paměti. U modulárních jazyků ale není možné v době překladu modulu určit adresu entit, ta je známa až v době sestavení celého cílového programu — viz níže, spojovací program (program pro sestavení cílového kódu, linker).

Tok řízení programu

Základní struktury pro řízení toku programu opět nejsou přímo dány modularitou, ale jsou poplatné jiné vlastnosti jazyka. V případě blokově strukturovaných jazyků dostáváme celou řadu typických konstrukcí. Jejich překlad/zpracování je však prakticky shodné jako u jazyků nemodulárních. Oproti nemodulárním jazykům však přibývá jiný fenomén — je možné předávat (díky volání funkcí/procedur) řízení z jednoho modulu do druhého.

V době překladu je však obecně znám pouze jeden modul, případně jejich omezené množství. Proto vyvstává problém, jak:

- předat parametry;
- vrátit/přijmout výsledek funkce;
- využít registry pro optimalizaci.

Důsledky jsou potom tyto:

- **Přiřazování registrů** se děje pouze v rámci funkce — pokud nejsou v nějakém modulu využity specializované registry, je možné na závěr tyto přiřadit globálně (děje se jen ve specializovaných překladových systémech).
- **Standardizace předávání parametrů** — jak volající, tak volaný dodržují určité konvence.
- **Standardizace předávání výsledku** — jak volající, tak volaný dodržují určité konvence.

Z hlediska způsobu předávání parametrů je možné rozpoznat předávání parametrů:

- **hodnotou, výsledkem, hodnotou a výsledkem** — vytváří se kopie, které hodnota se případně kopíruje zpět;
- **odkazem** — předává se ukazatel na danou proměnnou;
- **jménem** — předává se dvojice přístupových metod k dané entitě: pro zápis a čtení hodnoty.

Řešený příklad

Zadání: Uvažte architekturu stolních počítačů třídy PC s operačním systémem Windows 2000, či Linux. Navrhněte způsob předávání parametrů a výsledku tak, aby pracovalo mezi moduly a přitom bylo maximálně optimální z hlediska rychlosti i užité paměti. Jazyk je blokově strukturovaný modulární.
Řešení: Parametry budou předávání přes programový zásobník, či v registrech (dle velikosti). Výsledky přes registry například takto:

- 8 bitů: registr AL
- 16 bitů: registr AX
- 32 bitů: registr EAX
- 64 bitů: registry EDX:EAX

Navrhované řešení trpí nedostatkem — ne všechny architektury podporují tak veliké registry, mají dostatek registrů, mají programový zásobník, vejdou se do 64 bitů (např. datové struktury v jazyku C). Nové řešení pro parametry by potom mohlo vypadat takto. Parametry se pomocí registrů předávají zřídkavě (na pracovních stanicích) — jedná se o specializované architektury, či extrémní optimalizace, které jsou často do značné míry řízené uživatelem. Běžně je použit zásobník a i ten jen do určité velikosti dat, jinak se užívá halda (heap). Ta se užívá i v případě, že architektura neobsahuje programový zásobník pro jeho emulaci. Přesto jistý horní limit pro velikost předávaných dat může existovat. K tomu je dodržen jistý společný rámec, jakým se organizují údaje na programovém zásobníku tak, aby volající i volaný správně údaje zpracovali — viz konec příkladu. Pro výsledky funkcí platí obdobná pravidla — zpravidla jsou tedy předávány přes programový zásobník. I zde je však velikost limitována a po překročení limitu se užívá halda, či je přenos tak velikého výsledku zcela odmítnut.
Příklad uložení položek na zásobník při volání funkce může být například takový (začátek je na vrcholu zásobníku před voláním):

- *volající* — Oblast vyhrazená pro výsledek (není-li možné předat registrem).

- *volající* — Parametry
- *volající* — Hodnoty zdrojů s rozpracovanými údaji (registry apod., jsou-li)
- *volající* — Návratová adresa
- *volaný* — Úschova zdrojů, které tvoří rámec a nebylo možné je odložit dříve
- *volaný* — Lokální proměnné

Pro některé jazyky je však možné použít i jinou organizaci předávání dat na zásobníku.

I když příklad demonstroval možné řešení, které je v praxi užívané, tak pro případy, kdy je nutná vysoká optimalizace, jsou užity další strategie, které však na spojovací program (viz níže) kladou nemalé nároky. Dochází typicky k hojnému využívání registrů, omezení velikosti dat, která mohou být předána do/z funkce, apod. Zachování jednotných pravidel je nutností zejména i proto, že moduly mohou pocházet nejen z různých zdrojů, ale i z různých překladačů, či dokonce z různých programovacích jazyků.

Pojmy k zapamatování

- Skrývání implementace, rozhraní modulu
- Problematika závislosti mezi moduly
- Problematika způsobů předávání parametrů/výsledků mezi moduly

Výklad

5.4 Zpracování — analýza, vyhodnocení, překlad

Modularita nepřináší do překladu samotného modulu žádné zvláštnosti. Výjimkou je výše zmíněné generování kódu, kdy adresy entit nejsou známy a to dokonce ani u těch, které jsou definovány v rámci modulu.

Překladače

Zejména analytická část překladače kopíruje vlastnosti, které jsou dány charakterem jazyka z hlediska typů a řídicích struktur. Díky modularitě však některé operace zejména v zadní části překladače nejsou možné (důvod viz výše):

- některé druhy globálních (mezi funkcemi) optimalizací nelze provádět díky neznalosti kompletního programu;
- volání a návrat z volání funkcí musí probíhat přes standardní rámec;
- objevuje se spojovací program (linker), který **přebírá** některé role překladače.

lexikální analýza

U modulárních jazyků se objevuje vlastní textový preprocesor, který je zprvu implementován jako separátní program spouštěný ještě před vlastním překladačem. V pozdějších implementacích se však již stává součástí lexikálního analyzátoru, což na něj klade vysoké nároky, zejména v oblasti rychlosti zpracování. Díky rozšíření jazyků na více různých platforem a přenosu programů na zdrojové úrovni mezi nimi se objevila v jazyku C vlastnost, která umožňuje kódovat znaky, které nebyly ve znakové sadě některých počítačů, přes trojici znaků jiných. Překódování zpět byla opět práce lexikálního analyzátoru, která vyžadovala maximální rychlost.

syntaktická analýza

Jazyky modulární jsou založeny na bezkontextových gramatikách, takže i zpracování probíhá standardním způsobem. Novým prvkem z hlediska konstrukcí, je označování symbolů vyvážených z modulu a dovážených do modulu. U některých jazyků je rozhraní zpracováváno v textové formě (např. jazyk C) a z definic či deklarací se odvozuje, jeli symbol pro modul lokální, či nikoliv, nebo jestli je dokonce dovážený. V zásadě to není ale nic nadstandardního a je ovlivněn pouze charakter výstupního kódu. Rozdíl v rozhraní a implementaci je potom možné, alespoň částečně, detekovat v době spojování modulů do jednoho programu. Druhá možnost jak přistoupit na symboly vyvážené z jiných modulů je, že rozhraní se vytahuje přímo z binární reprezentace modulu. Výhodou je, že nemůže dojít ke změně rozhraní, nevýhodou ale to, že překladač musí obsahovat speciální část pro načtení informací z binárního či semi-binárního souboru.

sémantická analýza

V porovnání s blokově strukturovanými jazyky nepřinášejí modulární jazyky do sémantické analýzy nic zvláštního. Generování cílového kódu probíhá do jeho relativní formy s tím, že je nutný ještě průchod spojovacího programu.

spojovací program

Spojovací program (linker) spojuje všechny moduly a části knihoven do jediného bloku, proveditelného souboru — sestavuje cílový kód.

Jelikož doplňuje část zadní části překladače, tak je to jedno z míst, kde se objevují chybová hlášení:

- **Chybějící symbol** — některý z modulů požaduje symbol (proměnnou, funkci, typ), který ale žádný jiný modul nedefinuje, nebo tuto definici nevyváží.
- **Vícenásobná definice** — některý ze symbolů je definován a exportován více moduly.
- **Rozdílné typy symbolu** — ne všechny linkery tuto chybu detekují, jde o konflikt mezi typem symbolu, který je vyvážen jedním a dovážen jiným modulem.

vstup linkeru

Relativní kód, který je výstupem překladu, má vazby na proměnné uskutečněny přes tabulku, která určuje o jaký symbol se jedná, jestli je definován modulem, nebo vyvážen apod. Podobně i volání funkcí zatím neobsahuje skutečné adresy, ale jen odkazy to tabulky, přes kterou se skutečné adresy doplňují. Často je ještě obsažena i tabulka pro symboly lokální v modulu, protože ani ty neznají své umístění v paměti. *(Pozn.: rozčlenění na 3 tabulky je logické, implementačně může být vše rozlišeno jinak.)*

úloha linkeru

Spojovací program potom spojuje a vzájemně provazuje symboly, které někdo požaduje, se symboly, které jiný modul vyváží. Jak vyplývá z chybových hlášení, tak při spojování se provádí kontrola výskytu, aby byly odhaleny násobné, či chybějící definice. Výsledkem práce spojovacího programu je potom buďto relokatibilní kód (např. soubory EXE v MS-DOS), nebo kód absolutní (typické pro vestavěné systémy, nebo pro operační systémy s virtualizovanou pamětí).

absolutní kód

Pokud je generovaný kód absolutní, tak kromě propojení symbolů vzájemně musí linker určit adresu všech entit v programu a zmodifikovat kód, který se na ně odkazuje. Pro určité architektury to nese problémy dané segmentací paměti, přístupovými metodami do různých typů paměti, velikostí paměti, proměnnou délkou instrukcí (např. skok na bližší adresu může mít kratší instrukci, než na vzdálenější), apod.

vliv modulů na kvalitu kódu

Díky modularizaci však přicházíme o některé optimalizace. Uvažme tento příklad z jazyku Pascal:

```
type tArray:array [1..10] of integer;
var a:tArray;
function f;
begin
  a[4] := a[1] + 4;
end;
```

```
begin
  ...
  f;
  ...
end;
```

Jelikož je jazyk Pascal nemodulární, tak je v době generování kódu známa velikost typu `integer`, tím pádem velikost typu `tArray` a, zejména, **umístění** proměnné `a`. Tím pádem je možné operace ve funkci `f` optimalizovat a předvyhodnotit. Již víme, že adresa elementu prvku pole `Q[x]` se vypočte takto:

$$\text{addr}(Q) - \text{low}(Q) * \text{sizeof}(\text{integer}) + x * \text{sizeof}(\text{integer})$$

V našem případě proměnná `x` nabývá konstantních hodnot 4 a 1. Tím pádem je celý výraz konstantní (ne jen mapovací funkce). Výstupem může být tento kód (virtuální assembler):

```
Load  Reg, [_addr(a[1])]
Add   Reg, 4
Store [_addr(a[4])], Reg
```

Uvažme nyní stejný případ v jazyku C, kde navíc byl celý program modularizován, nechť:

```
ops.c
  extern int a[10];
  void f(void) {
    a[3]=a[0]+4;
  }

main.c
  int a[10];
  ...
  f();
  ...
```

V době generování kódu je známa velikost typu `int` a také velikost pole `a`. To je ale vše! Takže přístup k elementu pole `a` nelze vyhodnotit jako konstantu a musí se generovat kód, který výpočet provede za běhu programu — je delší a pomalejší:

```
Load  Reg1,  [_addr(a)]
Add   Reg1,  #offset_0   // lze optimalizací eliminovat
Add   Reg1,  4
Load  Reg2,  [_addr(a)]
Add   Reg2,  #offset_3
Store [Reg2],Reg1
```

Pokud tedy shrneme vliv modularizace na optimalizaci kódu v době jeho generování, tak vidíme, že některé optimalizace nelze uskutečnit. Běžně se překladače (a příslušné spojovací programy) tuto situaci nesnaží řešit. Vysoce pokročilé překladače, které nabízejí vysoký stupeň optimalizace řeší i tuto situaci, nicméně jak překladač, tak spojovací program vyžadují mnohem propracovanější a náročnější implementace.

Interprety

Interprety u modulárních jazyků nejsou typické, objevují se jen jako tzv. *školní* interprety, nebo výzkumné projekty. Častěji je možné se setkat s překladači do abstraktního kódu, který je nadále interpretován — jedná se o nízkoúrovňový virtuální kód, který je interpretován virtuálním strojem, nikoliv přímo pomocí konkrétního hardware.

interprety virtuálního kódu

Virtuální kód, respektive jeho interpretace, umožňuje vylepšit vlastnosti interpretů, neboť eliminuje nutnost opakované analýzy zdrojového textu, přitom umožňuje rychlou modifikaci kódu jednotlivých funkcí, možnosti ladění programu jsou teoreticky také na vysoké úrovni, navíc je možnost inkrementální kompilace modulů, což dále zvyšuje rychlost odezvy takových systémů. Při vlastní interpretaci jsou přitom rychlejší, než klasické interprety, nicméně za cenu vyšších nároků na paměť. Jazyky modulární se vyskytují v takovéto formě zejména jako objektově orientované nebo funkcionální, či logické.

Pojmy k zapamatování

- Omezení modulárních jazyků pro generování kódu
- Preprocesor
- Linker — kompletní problematika
- Zpracování rozhraní modulů
- Interprety virtuálního kódu

Závěr

Úspěšným zvládnutím této kapitoly proniknete do zásad a typických vlastností modulárních programovacích jazyků. Měli byste vědět co se pojí s problematikou zpracování modulů — co to přináší do generování kódu, jaká omezení vznikají, jaká a kde vznikají a uplatňují se různá pravidla, že existuje spojovací program. Přes drobná omezení však je jejich význam vysoký, neboť umožňuje vznik knihoven a masivní týmovou spolupráci.

Úlohy k procvičení:

Uvažte jazyk C, např. dle normy ANSI. Definujte v něm abstraktní datový typ (ADT) polymorfní seznam (strom, atd.). Definujte/navrhněte implementaci i operace k danému typu a implementuje modul, který daný ADT takto podporuje. Po té si modul vyměňte s kolegou.

- Je snadné jeho modul použít?
- Je tento modul dostatečně bezpečný?
- Je modul možné použít vícekráte z různých míst?
- Změnili byste, pod vlivem zkušeností s jiným modulem, svoji vlastní implementaci? Jak?
- Je vaše implementace opravdu správně napsaná, aby skryla vše co skrýt má?
- Jaké problémy přináší možnost vícenásobného užití modulu z hlediska bezpečnosti jeho užití?

Nainstalujte si nástroje pro ladění, jak vypadá rámec pro předávání argumentů u vašeho překladače?

Klíč k řešení úloh

Rámec začněte detekovat už u volajícího, ne až u volaného.

Další zdroje

Schmidt, D.A.: *The Structure of Typed Programming Languages*, MIT Press, 1994.
Gordon, J. C.: *Programming language theory and its implementation*, New Jersey : Prentice-Hall, 1988.
Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading MA, 1986.
Pittman, Peters: *The Art of Compiler Design*.

Kapitola 6

Závěr

Publikace představuje v několika kapitolách typické rysy několika základních tříd imperativních jazyků. Soustřeďuje se na zvládnutí těchto kategorií jednak z hlediska vlastního užití těchto jazyků, jednak z hlediska zpracování těchto jazyků.

Po absolvování by student měl být schopen v rámci prezentovaných kategorií rozpoznat a začlenit různé programovací jazyky. Měl by zvládnout pochopení jejich vyhodnocení a tím i vhodnost pro nasazení v různých situacích. Výklad popisující zpracování by měl umožnit absolventovi lépe se orientovat při vývoji programového vybavení v jazyku s danými vlastnostmi, případně i při tvorbě překladače jazyka dané kategorie.

Modul je základním modulem pro navazující moduly, které pojednávají o objektově orientovaných jazycích a o deklarativních jazycích. Pro doplnění znalostí jsou vhodné moduly pojící se s výkladem nějakého konkrétního programovacího jazyka.

Obsah modulu obsahuje pouze nejnutnější informace a nástiny problematiky. Pro úplné zvládnutí problematiky je vhodné rozšířit si vědomosti nějakou vhodnou doporučenou literaturou jak z oblasti překladačů, teorie programovacích jazyků, tak třeba z okrajových témat.

Literatura

[1] Abadi, M., Cardelli, L.: *A Theory of Objects*, Springer, New York, 1996, ISBN 0-387-94775-2.

[2] Aho, A.V., Hopcroft, J.E., Ullman, J.D.: *The Design and Analysis of Computer Algorithms*, Addison Wesley, 1974.

[3] Aho, A.V., Sethi, R., Ullman, J.D.: *Compilers: Principles, Techniques, and Tools*, Addison Wesley, Reading MA, 1986.

[4] Aho, A.V., Ullman, J.D.: *The Theory of Parsing, Translation, and Compiling, Volume I: Parsing*, Prentice-Hall, Inc., 1972, ISBN 0-13-914556-7.

[5] Aho, A.V., Ullman, J.D.: *The Theory of Parsing, Translation, and Compiling, Volume II: Compiling*, Prentice-Hall, Inc., 1972, ISBN 0-13-914564-8.

[6] Appel, A.W.: Garbage Collection Can Be Faster Than Stack Allocation, *Information Processing Letters 25 (1987)*, North Holland, pages 275–279.

[7] Appel, A.W.: *Compiling with Continuations*, Cambridge University Press, 1992.

[8] Augustsson, L., Johnsson, T.: *Parallel Graph Reduction with the* $< \nu, G >$*-Machine*, Functional Programming Languages and Computer Architecture 1989, pages 202–213.

[9] Barendregt, H.P., Kennaway, R., Klop, J.W., Sleep, M.R.: *Needed Reduction and Spine Strategies for the Lambda Calculus*, Information and Computation 75(3): 191-231 (1987).

[10] Beneš, M.: *Object-Oriented Model of a Programming Language*, Proceedings of MOSIS'96 Conference, 1996, Krnov, Czech Republic, pp. 33–38, MARQ Ostrava, VSB - TU Ostrava.

[11] Beneš, M.: *Type Systems in Object-Oriented Model*, Proceedings of MOSIS'97 Conference Volume 1, April 28–30, 1997, Hradec nad Moravicí, Czech Republic, pp. 104–109, ISBN 80-85988-16-X.

[12] Beneš, M., Češka, M., Hruška, T.: *Překladače*, Technical University of Brno, 1992.

[13] Beneš, M., Hruška, T.: *Modelling Objects with Changing Roles*, Proceedings of 23rd Conference of ASU, 1997, Stara Lesna, High Tatras, Slovakia, pp. 188–195, MARQ Ostrava, VSZ Informatika s r.o., Kosice.

[14] Beneš, M., Hruška, T.: *Layout of Object in Object-Oriented Database System*, Proceedings of 17th Conference DATASEM 97, 1997, Brno, Czech Republic, pp. 89–96, CS-COMPEX, a.s., ISBN 80-238-1176-2.

[15] Brodský, J., Staudek, J., Pokorný, J.: *Operační a databázové systémy*, Technical University of Brno, 1992.

[16] Bruce, K.B.: A Paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics, *J. of Functional Programming*, January 1993, Cambridge University Press.

[17] Cattell, G.G.: *The Object Database Standard ODMG-93*, Release 1.1, Morgan Kaufmann Publishers, 1994.

[18] Češka, M., Hruška, T., Motyčková, L.: *Vyčíslitelnost a složitost*, Technical University of Brno, 1992.

[19] Češka, M., Rábová, Z.: *Gramatiky a jazyky*, Technical University of Brno, 1988.

[20] Damas, L., Milner, R.: *Principal Type Schemes for Functional Programs*, Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, USA, January 1982, pages 207–212.

[21] Dassow, J., Paun, G.: *Regulated Rewriting in Formal Language Theory*, Springer, New York, 1989.

[22] Douence, R., Fradet, P.: *A taxonomy of functional language implementations. Part II: Call-by-Name, Call-by-Need and Graph Reduction*, INRIA, technical report No 3050, Nov. 1996.

[23] Ellis, M.A., Stroustrup, B.: *The Annotated C++ Reference Manual*, AT&T Bell Laboratories, 1990, ISBN 0-201-51459-1.

[24] Finne, S., Burn, G.: *Assessing the Evaluation Transformer Model of Reduction on the Spineless G-Machine*, Functional Programming Languages and Computer Architecture 1993, pages 331-339.

[25] Fradet, P.: *Compilation of Head and Strong Reduction*, In Porc. of the 5th European Symposium on Programming, LNCS, vol. 788, pp. 211-224. Springer-Verlag, Edinburg, UK, April 1994.

[26] Georgeff M.: *Transformations and Reduction Strategies for Typed Lambda Expressions*, ACM Transactions on Programming Languages and Systems, Vol. 6, No. 4, October 1984, pages 603–631.

[27] Gordon, M.J.C.: *Programming Language Theory and its Implementation*, Prentice Hall, 1988, ISBN 0-13-730417-X, ISBN 0-13-730409-9 Pbk.

[28] Gray, P.M.D., Kulkarni, K.G., Paton, N.W.: *Object-Oriented Databases*, Prentice Hall, 1992.

[29] Greibach, S., Hopcroft, J.: Scattered Context Grammars, *Journal of Computer and System Sciences*, Vol: 3, pp. 233–247, Academia Press, Inc., 1969.

[30] Harrison, M.: *Introduction to Formal Language Theory*, Addison Wesley, Reading, 1978.

[31] Hruška, T., Beneš, M.: *Jazyk pro popis údajů objektově orientovaného databázového modelu*, In: Sborník konference Některé nové přístupy při tvorbě informačních systémů, ÚIVT FEI VUT Brno 1995, pp. 28–32.

[32] Hruška, T., Beneš, M., Máčel, M.: *Database System G2*, In: Proceeding of COFAX Conference of Database Systems, House of Technology Bratislava 1995, pp. 13–19.

[33] Issarny, V.: *Configuration-Based Programming Systems*, In: Proc. of SOFSEM'97: Theory and Practise of Informatics, Milovy, Czech Republic, Novemeber 22-29, 1997, ISBN 0302-9743, pp. 183-200.

[34] Jensen, K.: *Coloured Petri Nets*, Springer-Verlag Berlin Heidelberg, 1992.

[35] Jeuring, J., Meijer, E.: *Advanced Functional Programming*, Springer-Verlag, 1995.

[36] Jones, M.P.: *A system of constructor classes: overloading and implicit higher-order polymorphism*, In FPCA '93: Conference on Functional Programming Languages and Computer Architecture, Copenhagen, Denmark, June 1993.

[37] Jones, M.P.: *Dictionary-free Overloading by Partial Evaluation*, ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, Orlando, Florida, June 1994.

[38] Jones, M.P.: *Functional Programming with Overloading and Higher-Order Polymorphism*, First International Spring School on Advanced Functional Programming Techniques, Båstad, Sweden, Springer-Verlag Lecture Notes in Computer Science 925, May 1995.

[39] Jones, M.P.: *GOFER, Functional programming environment, Version 2.20*, mpj@prg.ox.ac.uk, 1991.

[40] Jones, M.P.: *ML typing, explicit polymorphism and qualified types*, In TACS '94: Conference on theoretical aspects of computer software, Sendai, Japan, Springer-Verlag Lecture Notes in Computer Science, 789, April, 1994.

[41] Jones, M.P.: *A theory of qualified types*, In proc. of ESOP'92, 4th European Symposium on Programming, Rennes, France, February 1992, Springer-Verlag, pp. 287-306.

[42] Jones, S.L.P.: *The Implementation of Functional Programming Languages*, Prentice-Hall, 1987.

[43] Jones, S.L.P., Lester, D.: *Implementing Functional Languages.*, Prentice-Hall, 1992.

[44] Kleijn, H.C.M., Rozenberg, G.: On the Generative Power of Regular Pattern Grammars, *Acta Informatica*, Vol. 20, pp. 391–411, 1983.

[45] Khoshafian, S., Abnous, R.: *Object Orientation. Concepts, Languages, Databases, User Interfaces*, John Wiley & Sons, 1990, ISBN 0-471-51802-6.

[46] Kolář, D.: *Compilation of Functional Languages To Efficient Sequential Code*, Diploma Thesis, TU Brno, 1994.

[47] Kolář, D.: *Overloading in Object-Oriented Data Models*, Proceedings of MOSIS'97 Conference Volume 1, April 28–30, 1997, Hradec nad Moravicí, Czech Republic, pp. 86–91, ISBN 80-85988-16-X.

[48] Kolář, D.: *Functional Technology for Object-Oriented Modeling and Databases*, PhD Thesis, TU Brno, 1998.

[49] Kolář, D.: *Simulation of LL_k Parsers with Wide Context by Automaton with One-Symbol Reading Head*, Proceedings of 38th International Conference MOSIS '04—Modelling and Simulation of Systems, April 19–21, 2004, Rožnov pod Radhoštěm, Czech Republic, pp. 347–354, ISBN 80-85988-98-4.

[50] Latteux, M., Leguy, B., Ratoandromanana, B.: The family of one-counter languages is closed under quotient, *Acta Informatica*, 22 (1985), 579–588.

[51] Leroy, X.: *The Objective Caml system, documentation and user's guide*, 1997, Institut National de Recherche en Informatique et Automatique, Francie, Release 1.05, http://pauillac.inria.fr/ocaml/htmlman/.

[52] Martin, J.C.: *Introduction To Languages and The Theory of Computation*, McGraw-Hill, Inc., USA, 1991, ISBN 0-07-040659-6.

[53] Meduna, A.: *Automata and Languages: Theory and Applications*. Springer, London, 2000.

[54] Meduna, A., Kolář, D.: Regulated Pushdown Automata, *Acta Cybernetica*, Vol. 14, pp. 653–664, 2000.

[55] Meduna, A., Kolář, D.: One-Turn Regulated Pushdown Automata and Their Reduction, In: *Fundamenta Informaticae*, 2002, Vol. 16, Amsterdam, NL, pp. 399–405, ISSN 0169-2968.

[56] Mens, T., Mens, K., Steyaert, P.: *OPUS: a Formal Approach to Object-Orientation*, Published in FME '94 Proceedingss, LNCS 873, Springer-Verlag, 1994, pp. 326-345.

[57] Mens, T., Mens, K., Steyaert, P.: *OPUS: a Calculus for Modelling Object-Oriented Concepts*, Published in OOIS '94 Proceedings, Springer-Verlag, 1994, pp. 152-165.

[58] Milner, R.: A Theory of Type Polymorphism In Programming, *Journal of Computer and System Sciences*, 17, 3, 1978.

[59] Mycroft, A.: *Abstract Interpretation and Optimising Transformations for Applicative Programs*, PhD Thesis, Department of computer Science, University of Edinburgh, Scotland, 1981. 180 pages. Also report CST-15-81.

[60] Okawa, S., Hirose, S.: Homomorphic characterizations of recursively enumerable languages with very small language classes, *Theor. Computer Sci.*, 250, 1 (2001), 55–69.

[61] Păun, Gh., Rozenberg, G., Salomaa, A.: *DNA Computing*, Springer-Verlag, Berlin, 1998.

[62] Rozenberg, G., Salomaa, A. — eds.: *Handbook of Formal Languages; Volumes 1 through 3*, Springer, Berlin/Heidelberg, 1997.

[63] Salomaa, A.: *Formal Languages*, Academic Press, New York, 1973.

[64] Schmidt, D.A.: *The Structure of Typed Programming Languages*, MIT Press, 1994.

[65] Odersky, M., Wadler, P.: *Pizza into Java: Translating theory into practice*, Proc. 24th ACM Symposium on Principles of Programming Languages, January 1997.

[66] Odersky, M., Wadler, P., Wehr, M.: *A Second Look at Overloading*, Proc. of FPCA'95 Conf. on Functional Programming Languages and Computer Architecture, 1995.

[67] Reisig, W.: *A Primer in Petri Net Design*, Springer-Verlag Berlin Heidelberg, 1992.

[68] Tofte, M., Talpin, J.-P.: *Implementation of the Typed Call-by-Value λ-calculus using a Stack of Regions*, POPL '94: 21st ACM Symposium on Principles of Programming Languages, January 17–21, 1994, Portland, OR USA, pages 188–201.

[69] Traub, K.R.: *Implementation of Non-Strict Functional Programming Languages*, Pitman, 1991.

[70] Volpano, D.M., Smith, G.S.: *On the Complexity of ML Typability with Overloading*, Proc. of FPCA'91 Conf. on Functional Programming Languages and Computer Architecture, 1991.

[71] Wikström, Å.: *Functional Programming Using Standard ML*, Prentice Hall, 1987.

[72] Williams, M.H., Paton, N.W.: *From OO Through Deduction to Active Databases - ROCK, ROLL & RAP*, In: Proc. of SOFSEM'97: Theory and Practise of Informatics, Milovy, Czech Republic, Novemeber 22-29, 1997, ISBN 0302-9743, pp. 313-330.

[73] Lindholm, T., Yellin, F.: *The Java Virtual Machine Specification*, Addison-Wesley, 1996, ISBN 0-201-63452-X.