UNIVERZITA PALACKÉHO V OLOMOUCI
KATEDRA INFORMATIKY

Úvod do paradigmat programování

David Skoupil [1]

Září 1994 TR-CS-94-01

[1]Autor je asistentem na Katedře informatiky UP. Může být kontaktován elektronickou poštou na adrese `skoupil@risc.upol.cz`.

Abstrakt

Tento technický report je určen jako učební text pro potřeby studentů I. semestru kursu Paradigmata programování. Seznamuje s významem struktury počítačových programů a s programovacími jazyky, jakožto prostředky, které strukturu programů určují. Studenti jsou konfrontováni s historií vzniku programovacích jazyků, se vznikem a charakteristikou jednotlivých paradigmat a se způsoby popisu programovacích jazyků. Zvláštní důraz je kladen na dva podstatné aspekty programovacích jazyků: funkce, jakožto elementárními moduly programu, a typové systémy.

Obsah

1 Úloha strukturalizace v programování

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

Jistým překvapením pak bývá zjištění, kolik programovacích jazyků již bylo ve světě vyvinuto. Proč vlastně potřebujeme tolik programovacích jazyků, jak se od sebe liší? O co jde těm podivínům, kteří vytvářejí stále nové a nové jazyky, které se v 99% případů nikdy nedočkají komerční implementace? Nestačilo by několik programovacích jazyků, s jejichž pomocí by bylo možno efektivně psát počítačové programy?

1.1 Programovací jazyk a struktura programu

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

Důvodem ke všem těmto úvahám byla rostoucí komplexnost vytvářených programů, která s sebou nesla stále větší množství drobných, těžce odhalitelných programátorských chyb. V roce 1962 dokonce jedna "drobná" programátorská chyba způsobila zkázu americké kosmické sondy Mariner I a finanční ztrátu cca 20 milionů dolarů. Vznikl tak problém *spolehlivosti* programů.

Původní předpoklad přirovnával chyby programátorské k chybám pravopisným. Stačí tedy pořádně se naučit pravopis a máme vyhráno: chyby vymizí! Ukázalo se však, že tato analogie neplatí. Programátorské chyby musí být zcela jiného druhu než běžné pravopisné chyby. Nelze se jim vyvarovat a dělají je stejně tak programátoři zkušení i nezkušení. Programátorský folklór dokonce axiomaticky tvrdí, že každý program obsahuje alespoň jednu chybu. Za takovýchto okolností bylo nezbytné začít přemýšlet o tom, jak chyby z programů efektivně odstraňovat.

Jako první možnost, která se sama nabízela, bylo zdokonalení testování programů. Výše zmíněný program pro kosmickou misi k planetě Venuši však byl testován na 100 průchodů bez nejmenší zjištěné chyby a byl čtyřikrát úspěšně použit pro lunární expedice! Američtí kongresmani tehdy požadovali *úplné* testování všech použitých programů, aby se situace, která nastala u sondy Mariner I, již nikdy neopakovala. Není však tak těžké spočítat, že program, který obsahuje řádově miliony řádků a který se může nacházet ve

stovkách milionů stavů, není zrovna jednoduché úplně otestovat v relativně krátké době. Jak tedy můžeme do "rukou" nespolehlivého programu vložit obrovské finanční částky nebo dokonce lidské životy?

Nový vítr do celé problematiky vnáší E. W. Dijkstra v knize "Structured Programming" [3]: *Testováním lze dosáhnout korektnosti programu pouze tehdy, pokud se zaměříme na jeho vnitřní strukturu.* Do popředí pozornosti se tedy dostává vnitřní struktura programu. Je jasné, že vnitřní struktura programu je velmi úzce spojena s použitým programovacím jazykem.

Dijkstrovu myšlenku lze rozvést podle Meyera[2] následujícím způsobem. Rozlišujeme v zásadě dvojí faktory, určující kvalitu programu: *vnější* a *vnitřní*. Vnější faktory jsou viditelné uživateli programu, vnitřní faktory zůstávají před uživatelem skryty, jsou viditelné pouze počítačovým profesionálům.

Jako příklad vnějších faktorů kvality můžeme uvést:

Správnost, tj. schopnost programů přesně vykonávat svou úlohu tak, jak je definována v požadavcích zadavatele.

Robustnost, tj. schopnost programů fungovat i v abnormálních podmínkách. Program by měl být schopen ignorovat evidentně špatná data a nesmí na základě neočekávaných údajů způsobit katastrofu. Nemůže-li pokračovat ve vykonávání své funkce (například v případě chyby techniky), měl by čistě terminovat nebo přejít do stavu nazývaného "graceful degradation". Robustnost je jistě nadmnožinou správnosti programu. Lze ji však mnohem obtížněji definovat a testovat - nikde totiž nemáme přesně vymezeno, jaké situace mohou nastat. Vztah správnosti a robustnosti je schematicky znázorněn na obrázku 1.

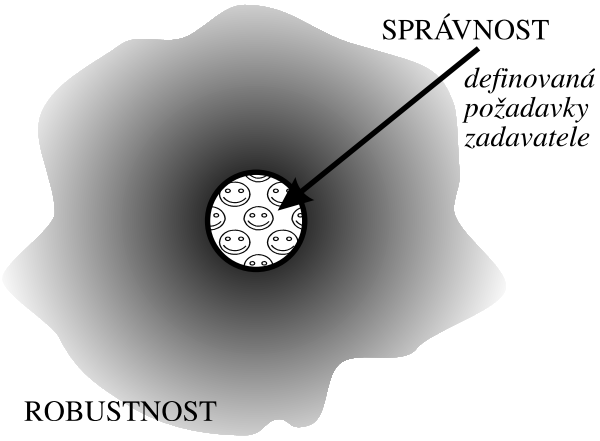
Rozšiřitelnost, tj. snadnost, s jakou mohou být programy přizpůsobeny novým podmínkám. Dobře známá je problematika zvaná "programming-in-large", kdy program postupně roste až do té obludné míry, že se v něm nevyzná ani sám programátor. Potom stačí i malá změna kódu programu a celý komplikovaný systém se zhroutí.

Rychlost, s jakou je program schopen plnit svoji úlohu.

Efektivnost, kterou program vykazuje při využívání strojového času, paměti, diskového prostoru počítačové sítě, tiskáren a dalších tzv. zdrojů.

Dále sem patří faktory jako kompatibilita, univerzálnost, cena programu a další.

Vnitřní faktory kvality programu nebudeme v tuto chvíli vypočítávat. Právě jim jsou totiž věnovány čtyři semestry předmětu Paradigmata programování.



Obrázek 1: Vztah robustnosti a správnosti programu

U komerčních aplikacích záleží jen a pouze na vnějších faktorech. Dijkstra však ukázal, že vnějších faktorů kvality je možno dosáhnou pouze na základě existence faktorů vnitřních, tedy tehdy, bude-li mít program svoji jasně definovanou vnitřní strukturu, podléhající určitému stylu.

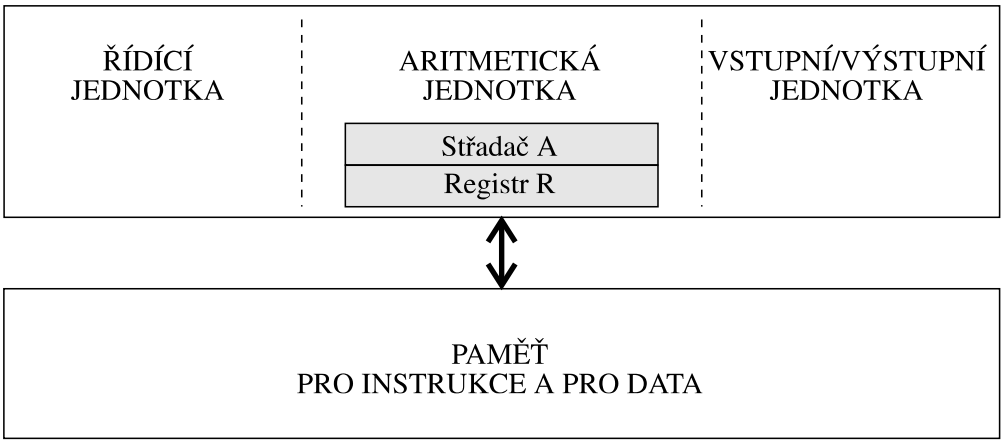
Tento styl vnitřní struktury programu se nazývá *programovací paradigma.*

Dostáváme se tak k odpovědi na položenou otázku opodstatněnosti existence takového množství programovacích jazyků. Jsou to právě programovací jazyky, které nám umožňují (nebo dokonce nutí či naopak zabraňují) vytvářet programy se strukturou, odpovídající některému z paradigmat. Každý z nově vytvářených programovacích jazyků se snaží lépe nebo alespoň "jinak" podporovat jedno nebo kombinaci několika programovacích paradigmat.

1.2 Von Neumannův stroj - jazyky nižší úrovně

Podoba programování, podoba programovacích jazyků a někdy i podoba programátorů je dána základním principem práce počítače. Pradědeček všech počítačů je dnes znám jako *von Neumannův stroj* a byl poprvé popsán v drobné práci Burkse, Goldstina a von Neumanna z roku 1947. Princip, na němž byl chod stroje postaven a který se vyznačoval zejména tím, že data a instrukce sdílely tutéž paměť, je nazýván *von Neumannův princip* a je považován za základní princip práce dnešních počítačů. John von Neumann, všemi svými spolupracovníky a studenty oslovovaný jako Johnny, asi netušil, že tímto návrhem na mnoho desetiletí ovlivní vývoj informatiky.

Stroj se skládal z řídící jednotky, aritmetické jednotky, jednotky vstupů a výstupů a z paměti. V paměti byly uloženy jednak instrukce programu, jed-



Obrázek 2: Organizace John von Neumannova stroje

nak data. Stroj pracoval v podstatě stejně jako soudobé počítače: jednotlivé instrukce byly postupně brány z paměti a vykonávány, výsledky byly ukládány zpět do paměti. Řízení výpočtu probíhalo pomocí instrukce GO-TO, kdy byla následující instrukce vzata z paměťové buňky zmíněné v instrukci GO-TO.

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

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

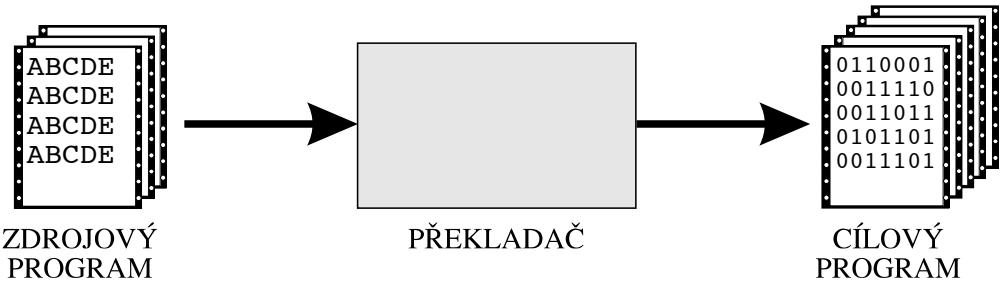
```
00000010101111001010  LOAD I
00000010111111001000  ADD J    k := i + j
00000011001110101000  STORE K
```

Obrázek 3: Segment kódu zapsaný v jazyku stroje, jazyku symbolických adres a ve vyšším programovacím jazyku

1.3 Jazyky vyšší úrovně

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

Prvním úspěchem, který přetrvává až dodnes, byl jazyk pro překlad matematických výrazů (FORmula TRANslation) zvaný FORTRAN. Program napsaný ve Fortranu nemohl být přímo interpretovaný strojem, musel být nejprve přeložen do jazyka stroje. To ostatně platí pro všechny ostatní vyšší programovací jazyky.[1]



Obrázek 4: Překlad programu ze zdrojového do cílového jazyka

S uvedením vyšších programovacích jazyků vznikla také nová disciplina v oblasti počítačové vědy: konstrukce *překladačů*, speciálních programů, jejichž účelem je transformovat program z jednoho jazyka do druhého. Ve většině případů je *zdrojovým* jazykem překladače nejaký vyšší programovací jazyk a *cílovým* jazykem je jazyk stroje.[2] Zároveň se vytvořily dva základní přístupy k překladu kódu: přístup kompilační a přístup interpretační. Podle toho dnes dělíme překladače na *kompilátory* a *interpretry*.

Kompilátory jsou překladače, které přeloží celý kód ze zdrojové formy do jazyku stroje najednou. Překlad celého kódu tedy předchází jeho spuštění.

Interpretry jsou překladače, obcházející přímý překlad do jazyka stroje. Zdrojový program je těmito překladači postupně interpretován a cílový

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

[2] O překladačích, jejichž cílovým jazykem je jazyk stroje, říkáme, že mají *native code generation*. Mnohé překladače experimentálních jazyků mají za cílový program nějaký jiný programovací jazyk (nejčastěji jazyk C).

program jako celek nevzniká. Překlad tak probíhá vlastně souběžně s během programu. Výhodou tohoto přístupu je, že kód je možno za běhu programu modifikovat. Na druhé straně jsou tyto překladače pomalejší, náročnější na paměť a program není schopen fungovat bez přítomnosti překladače.

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

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

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

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

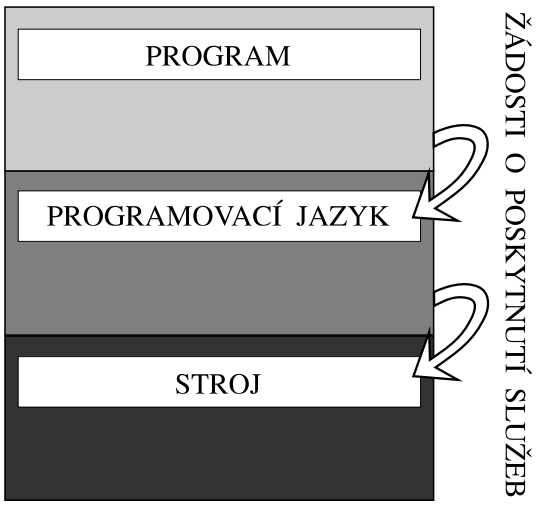
Triumfem vyšších programovacích jazyků bylo, když Dennis Ritchie použil jeden z nich na vytvoření operačního systému. Operační systémy, jakožto základní programy ovládající hardware počítače, byly totiž vždy výsadní doménou programátorů píšících v jazyku stroje. Zmíněným programovacím jazykem byl jazyk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

1.4 Vztah stroje, jazyka a programu

Vztah jazyků nižší a vyšší úrovně není však tak antagonistický, jak jsme to vylíčili v předchozí kapitole. Následující příklad poukazuje na existenci spolupráce mezi jazykem stroje a vyššími programovacími jazyky.

- Většina nižších jazyků poskytuje operaci <

- Většina vyšších jazyků má zabudovanou funkci `strcmp` (string compare) pro lexikální porovnávání řetězců (tj. slovo 'dokument' je před slovem 'dolar')
- Programy mohou poskytovat funkci `find` pro vyhledávání slova v utříděné kolekci slov, jakou je například slovník.



Obrázek 5: Vztah mezi programem, jazykem a strojem. Každá z vrstev využívá služeb vrstvy nižší.

Dá se proto říci, že stroj (resp. jazyk stroje), programovací jazyk a program představují tři na sobě závislé vrstvy, kde každá vrstva využívá služeb vrstvy podřízené. Programovací jazyky proto představují přirozené a žádoucí rozšíření možností stroje. Z kategorií služeb, které poskytují lze vyjmenovat:

- Výpočetní model - jazyky mohou nechat vyniknou stroji nebo naopak charakteristiku stroje potlačit.
- Datové typy a operace - stroj většinou podporuje datové typy `Char`, `Integer`, `Real`, ale neposkytuje typy jako jsou pole, záznamy atd. [3]
- Podpora abstrakce - je například možné zavést datový typ fronta, nad tímto datovým typem zavést operace a pak jej používat stejně jako strojový typ.
- Kontrola správnosti - mnoho programátorských chyb lze odhalit již za překladu.

[3] Datovým typům věnujeme kapitolu 5.

2 Přehled základních programovacích paradigmat

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

2.1 Naivní paradigma

Naivní programovací paradigma je typické pro počítačové laiky a začátečníky. Vyznačuje se nekoncepčností a chaotičností a proto bychom jej vlastně ani neměli nazývat paradigmatem. Programovací jazyky, podporující toto paradigma, jsou minimálně strukturované, neumožňují modularitu a poskytují minimální prostředky pro datovou abstrakci.
Programovací jazyky: Typickým naivním programovacím jazykem je BASIC.
Použití: Toto paradigma je radno nepoužívat.

2.2 Procedurální paradigma

Procedurální paradigma se též někdy nazývá *klasické* nebo *imperativní*. Z prvního synonyma lze vyčíst, že jde zřejmě nejstarší skutečné paradigma,[4] druhé synonymum napovídá, že základní úlohu v tomto paradigmatu hrají příkazy. Typické pro procedurální paradigma je, že průběh výpočtu je dán *sekvencí* po sobě jdoucích instrukcí (příkazů), přičemž rozhodující roli zde hraje přiřazovací příkaz.
Programovací jazyky: Klasické jazyky jako Fortran, C, Modula2 nebo Pascal. *Použití:* Toto paradigma má univerzální použití, rozšířené je zejména v komerční sféře.

2.3 Funkcionální paradigma

V případě funkcionálního paradigmatu je průběh výpočtu založen na postupném *aplikování funkcí*. Funkce zde bývají aplikovány na výsledky jiných funkcí. Pro toto paradigma je typické, že se zde nepoužívá přiřazovacího příkazu a silné místo zde zaujímají tzv. funkce vysoké úrovně a rekurze. Po procedurálním paradigmatu jde o druhé nejstarší paradigma.
Programovací jazyky: Typickým reprezentantem je Lisp (McCarthy, 1957),

[4] ... což ovšem není tak úplně pravda. Procedurální paradigma soupeří co do data vzniku s paradigmatem funkcionálním. Klasické se nazývá možná spíše proto, že se nejméně odlišuje od dobře známého jazyka symbolických adres.

Scheme (dialog Lispu, Sussman, Steele, 1975), z novějších experimentálních jazyků lze jmenovat ML, Mirandu či Haskell.
Použití: Původní použití vzešlo z oblasti umělé inteligence (dodnes jde nejpoužívanější paradigma pro umělou inteligenci v USA), dále je používáno v počítačová grafice, výuce a výzkumu.

2.4 Objektově orientované paradigma

Základními programovacími prvky v objektově orientovaném programování jsou útvary zvané *objekty*. Tyto útvary v zásadě modelují objekty reálného světa: osoby, předměty, události. Každý objekt je nositelem jistých informací o sobě samém (tzv. stavu) a má schopnost na požádání tento stav měnit. Průběh výpočtu je pak určen *posíláním zpráv* mezi jednotlivými objekty.
Programovací jazyky: K historickým jazykům patří např. Simula a Smalltalk, v dnešní době snad komerčně nejvyužívanějším jazykem je objektově orientované rozšíření jazyka C zvané C++. K moderním čistě objektově orientovaným jazykům patří třeba Eiffel.
Použití: Původní použití bylo soustředěno do oblasti simulace a umělé inteligence, v moderních jazycích má již univerzální použití.

2.5 Logické paradigma

V logickém programování je program je počítači předložen ve formě množiny faktů a pravidel - tzv. *klauzulí*. Průběh výpočtu založen na metodě unifikace a zpětného řetězení: programátor předloží systému nějaké tvrzení (cílovou hypotézu) a systém se na základě programu snaží dané tvrzení dokázat nebo vyvrátit.
Jazyky: Prolog (PROgramming in LOGic) a jeho dialekty.
Použití: Značného rozšíření se dočkal v oblasti umělé inteligence, a to zejména v Evropě. Komerční využití tohoto paradigmatu jsou velmi zřídká.

2.6 Paralelní paradigma

Zcela nové, vyvíjející se paradigma, zkoumající možnost dekompozice algoritmu na paralelní úlohy, které mohou být souběžně zpracovávány různými procesory. Klíčovým pojmem je tu komunikace a synchronizace mezi procesy.
Jazyky: Jádro systému UNIX, jazyk SR (Synchronized Resources).
Použití: Ačkoliv je toto paradigma velmi perspektivní, jeho použití zůstává zatím spíše v oblasti vědy a výzkumu.

2.7 Trendy jednotlivých programovacích paradigmat

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

Procedurální paradigma ve své čisté podobě zaniká a stále více se směšuje s ostatními paradigmaty, čehož nejlepším příkladem je jazyk C++. V současné době snad všechny moderní jazyky univerzálního určení nesou stopu procedurálního paradigmatu. Toto paradigma tak často plní úlohu skutečného "klasického" paradigmatu, ze kterého ostatní jazyky vycházejí a ke kterému přidávají další specifické rysy. Tento stav ovšem může silně negativně ovlivnit příchod masivně paralelních strojů.

Funkcionální programování prožívá jakési období renezance. Bylo znovu objeveno jakožto výborné prostředí pro výuku algoritmizace, prostředí pro počítačovou grafiku a pro výzkum v oblasti informatiky. Jako houby po dešti vznikají nové experimentální jazyky. Velké využití funkcionálních jazyků se vidí zejména v oblasti paralelních superpočítačů.

O objektově orientovaném programování platí něco podobného. I tam se jeví perspektivní spojení objektů s paradigmatem parallelismu. Navíc zde dochází ke spojování s imperativními jazyky a stále roste význam používání těchto jazyků v komerčních aplikacích. Toto paradigma se promítá i do jiných oblastí informatiky: vznikají objetově orientované databáze, objektově orientované návrhy systémů atd.

Logické programování představuje jedno z velkých zklamání posledních let. Po velmi úspěšném startu v oblasti umělé inteligence v Evropě a Japonsku a po japonském vyhlášení projektu počítače páté generace založeném právě na myšlenkách logického programování se každoročně snižuje počet publikací na toto téma. Zda je tato recese přechodná nebo trvalá ukáže čas.

Paralelní paradigma se stává postupem času stále více potřebné a vzhledem ke zvětšujícím se možnostem techniky i stále lépe uskutečnitelné. V nejbližších letech zřejmě uvidíme v této oblasti velký pokrok.

3 Syntaxe programovacích jazyků

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

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

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

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

$$DD/DD/DD$$

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

- Výraz $01/A4 - 90Z$ neodpovídá syntaktické definici
- Výraz 10/11/12 odpovídá syntaktické definici. Může označovat buď 10. listopad 1912 (Evropský standard) nebo 11. říjen 1912 (Americký standard) nebo 12. listopad 1910 atd.

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

$$DD/MM/YY$$

a specifikovat, že DD označuje den v měsíci, MM označuje měsíc v roce a YY označuje poslední dvojčíslí roku ve 20. století.

Výše uvedený výraz 10/11/12 lze takto jednoznačně interpretovat jako 10. listopad 1912. Naproti tomu výraz 29/02/09 je sice syntakticky správný, v rámci zvolené semantiky je však chybný neboť rok 1909 nebyl přechodným rokem.

3.1 Úvod do BNF

Syntaxe programovacích jazyků je možno popsat několika způsoby. Nejčastější z nich je *Backus-Naurova forma* (BNF). [5]

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

$$\begin{aligned}
&\langle \textit{real-number} \rangle \ ::= \ \langle \textit{digit-sequence} \rangle \ . \ \langle \textit{digit-sequence} \rangle \\
&\langle \textit{digit-sequence} \rangle \ ::= \ \langle \textit{digit} \rangle \ \mid \ \langle \textit{digit} \rangle \langle \textit{digit-sequence} \rangle \\
&\langle \textit{digit} \rangle \ ::= \ \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}
\end{aligned}$$

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

K tomuto příkladu je třeba poznamenat:

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

Třetí pravidlo tedy přečteme jako:

> Číslice je 0 nebo 1 nebo 2 nebo ... nebo 9. Nic jiného není číslice.

Druhé pravidlo přečteme jako:

> Sekvence čísel je číslice nebo číslice následovaná sekvencí číslic. Nic jiného není sekvence čísel.

První pravidlo přeložíme jako:

> Reálné číslo je sekvence číslic následovaná tečkou a další sekvencí číslic. Nic jiného není reálné číslo.

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

$$\begin{aligned} \langle \textit{digit-sequence} \rangle &::= \langle \textit{digit} \rangle \\ \langle \textit{digit-sequence} \rangle &::= \langle \textit{digit} \rangle \langle \textit{digit-sequence} \rangle \end{aligned}$$

Význam však zůstává v obou případech stejný.

Jako příklad můžeme popsat pomocí BNF syntaxi úplného jména souboru v operačním systému MS-DOS. Pro jednoduchost budeme předpokládat, že jména souborů v systému MS-DOS mají vždy délku 8 znaků a přípona má vždy délku 3 znaky. [6] Používaný symbol Λ (lambda) zde označuje redukci na prázdný řetězec - tedy možnost, že uvedený neterminál zcela chybí. Například druhé pravidlo je možno číst jako:

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

Drive je buď nic (tj. není uveden) nebo je to písmeno následované dvojtečkou.

$$
\begin{aligned}
&\langle File\rangle \;::=\; \langle Drive\rangle\langle Path\rangle\langle Filename\rangle\\
&\langle Drive\rangle \;::=\; \Lambda \mid \langle Letter\rangle :\\
&\langle Path\rangle \;::=\; \langle Absolute\rangle \mid \langle Relative\rangle\\
&\langle Absolute\rangle \;::=\; \Lambda \mid \backslash\, \langle Relative\rangle\\
&\langle Relative\rangle \;::=\; \Lambda \mid \langle Filename\rangle \backslash\, \langle Relative\rangle\\
&\langle Filename\rangle \;::=\; \langle Name\rangle \mid \langle Name\rangle\, .\, \langle Extension\rangle\\
&\langle Name\rangle \;::=\; \langle Ch\rangle\langle Ch\rangle\langle Ch\rangle\langle Ch\rangle\langle Ch\rangle\langle Ch\rangle\langle Ch\rangle\langle Ch\rangle\\
&\langle Extension\rangle \;::=\; \langle Ch\rangle\langle Ch\rangle\langle Ch\rangle\\
&\langle Ch\rangle \;::=\; \langle Letter\rangle \mid \langle Number\rangle \mid \langle Special\rangle\\
&\langle Letter\rangle \;::=\; \mathbf{A} \mid \ldots \mid \mathbf{Z}\\
&\langle Number\rangle \;::=\; \mathbf{0} \mid \ldots \mid \mathbf{9}\\
&\langle Special\rangle \;::=\; !\mid @ \mid \#\mid \$\mid \%\mid \&\mid \text{atd.}
\end{aligned}
$$

3.2 EBNF

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

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

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

$$
\begin{aligned}
&\textit{Real-number} \;::=\; \textit{Digit}\{\textit{Digit}\}\ \text{'.'}\ \textit{Digit}\{\textit{Digit}\}\\
&\textit{Digit} \;::=\; \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}
\end{aligned}
$$

Jako procvičení celé problematiky můžeme pomocí EBNF popsat reálné číslo, u něhož může chybět reálná část nebo desetinná část, ale nemohou

chybět obě současně, tj. výrazy 20.45, .17 a 20. jsou platné zápisy pro reálná čísla 20.45, 0.17 a 20.0, avšak tečka sama o sobě není platný zápis čísla.

$$\begin{aligned} &\textit{Real-number} ::= \textit{Digit}\{\textit{Digit}\}\ \text{`.'}\ \{\textit{Digit}\} \\ &\textit{Real-number} ::= \{\textit{Digit}\}\ \text{`.'}\ \textit{Digit}\{\textit{Digit}\} \\ &\textit{Digit} ::= \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9} \end{aligned}$$

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

Začneme třetím pravidlem, které stačí jen formálně přepsat:

$$\langle \textit{digit} \rangle ::= \mathbf{0} \mid \mathbf{1} \mid \mathbf{2} \mid \mathbf{3} \mid \mathbf{4} \mid \mathbf{5} \mid \mathbf{6} \mid \mathbf{7} \mid \mathbf{8} \mid \mathbf{9}$$

Pro převedení prvního a druhého pravidla zavedeme následující substituci

$$A ::= \{\textit{Digit}\}$$

Obě pravidla lze pak přepsat do tvaru:

$$\begin{aligned} &\textit{Real-number} ::= \textit{Digit}\ A\ \text{`.'}\ A \\ &\textit{Real-number} ::= A\ \text{`.'}\ \textit{Digit}\ A \end{aligned}$$

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

$$\begin{aligned} &\langle \textit{real-number} \rangle ::= \langle \textit{digit} \rangle \langle a \rangle\ .\ \langle a \rangle \\ &\langle \textit{real-number} \rangle ::= \langle a \rangle\ .\ \langle \textit{digit} \rangle \langle a \rangle \\ &\langle a \rangle ::= \Lambda \mid \langle \textit{digit} \rangle \langle a \rangle \end{aligned}$$

3.3 Syntaktické diagramy

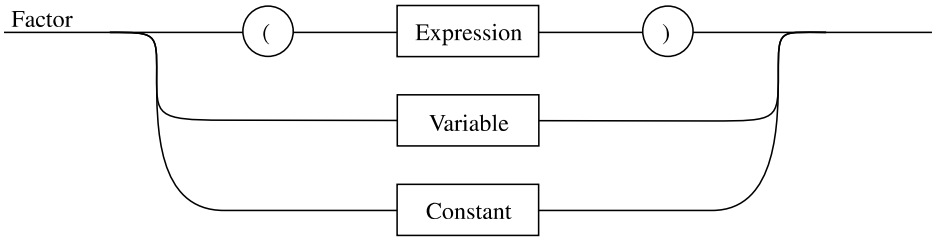
Syntaktické diagramy představují další způsob, jak popisovat syntaxi programovacího jazyka. Principy tvorby syntaktických diagramů lze shrnout do těchto bodů:

- Pro každý neterminál existuje jeden diagram, název neterminálu je uveden v levé části diagramu.

- Každé pravidlo BNF s daným neterminálem na levé straně generuje jednu cestu v diagramu
- Terminály jsou psány v kroužcích, neterminály ve čtverečcích
- Možnost opakování je v diagramu znázorněna smyčkou

Obrázek 6 představuje syntaktický diagram pro výraz, zapsaný pomocí EBNF jako:

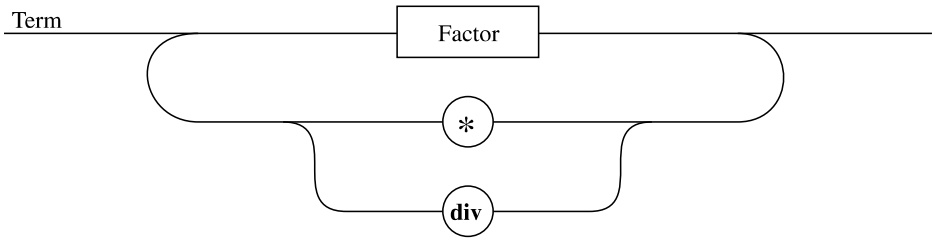
$$Factor \ ::= \ \text{`('} \ Expression \ \text{`)'} \ | \ Variable \ | \ Constant$$



Obrázek 6: Příklad syntaktického diagramu.

Obrázek číslo 7 představuje syntaktický diagram pro EBNF výraz:

$$Term \ ::= \ Faktor \ \{(\text{'*'} \ | \ \textbf{div}) \ Faktor\}$$



Obrázek 7: Příklad syntaktického diagramu.

3.4 Další způsoby popisu jazyka

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

Tutorial: základní průvodce daným programovacím jazykem. Tutorial nám dává základní představu o hlavních rysech a konstrukcích jazyka. Problematika je typicky vysvětlována na příkladech. Syntaxe a semantika jazyka je uváděna postupně, podle potřeby. Slouží k základnímu seznámení s daným programovacím jazykem.

Uživatelský manuál: manuál, popisující práci s příslušným překladačem z uživatelského hlediska. Důraz je tedy zejména kladen na technické otázky jako jsou instalace produktu, jeho spouštění, práce v menu atd.

Referenční manuál: manuál, popisující vyčerpávajícím syntaxi a semantiku programovacího jazyka. Je tradičně tříděn abecedně, podle syntaxe jazyka. Jednotlivé příkazy jazyka jsou popsány přirozeným jazykem (tj. většinou anglicky). Referenční manuál slouží zejména programátorům pro rychlé vyhledání podrobností o jednotlivých příkazech.

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

4 Modularita

Kapitole 1.1 jsme ukázali, že pro dosažení kvality je nutné, aby programy měly jistou přesně definovanou a známou *vnitřní strukturu*. Tato struktura je dána použitým programovacím paradigmatem. Ve většině případů je prvním cílem strukturalizace programu rozklad rozsáhlého problému na jisté malé, snadno pochopitelné a otestovatelné části. Tyto části obecně nazýváme *moduly*.

4.1 Základní rysy modulů

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

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

2. Modul by měl vykonávat jednu jasně definovanou úlohu, popřípadě několik jasně definovaných úloh. Nebylo by asi vhodné rozdělit jinak

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

3. Z definice modulu musí být všeobecně zřejmé, jakou úlohu modul vykonává a jaké předpoklady pro vykonávání své úlohy potřebuje. Tento princip se odborně nazývá principem explicitního rozhraní. Hovoříme-li například o modulu, který provádí řešení soustavy lineárních rovnic, musí být jasně stanoveno, v jaká data a v jakém formátu modul vyžaduje, aby mohl vykonávat svoji funkci.

4. Každý modul musí být relativně samostatný (někdy říkáme uzavřený), měl by komunikovat s co nejmenším množstvím ostatních modulů. Pokud s ostatními moduly komunikuje, množství informací takto vyměněných by mělo být co nejmenší.

5. Moduly musí mít navíc právo na jisté soukromí: je přípustné a žádoucí, aby veškeré informace, které nejsou potřebné pro klienty modulů, zůstaly skryté uvnitř modulu. To se týká zejména možnosti existence tzv. lokálních proměnných.

Modul lze z tohoto hlediska chápat jako malou soukromou firmu, specializovanou na určitou práci. Programátor musí být schopen zjistit, čím se firma zabývá a zajakých okolností je schopna provést inzerovanou práci. Udělá-li programátor s touto firmou kontrakt, znamená to, že jí dodá patřičná data (stejně tak jako operační paměť, diskovou paměť a strojový čas) a firma provede inzerovanou službu. Programátora-zadavatele přitom nemusí zajímat vnitřní organizace firmy (například počet a platy zaměstnanců firmy). Je taktéž možné, aby části zakázky firma zadadala jako zakázku subdodavatelům.

V různých programovacích paradigmatech jsou moduly tvořeny různými způsoby. Jedním z nejběžnějším a nejjednodušších modulů, vyskytujících se v různých obměnách téměř ve všech paradigmatech jsou *funkce*.[7]

4.2 Funkce

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

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

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

množiny B. Tento fakt zapisujeme:

$$f : A \to B$$

nebo

$$f : x \mapsto y \qquad x \in A,\ y \in B, \qquad \text{a píšeme} \quad f(x) = y$$

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

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

$$g(x) = \max\{n \in \mathrm{N},\ n^2 \leq x\}$$

Tato definice funkce splňuje všechna kritéria injektivního zobrazení (a tedy funkce v matematickém pojetí), nepředstavuje však funkci ve smyslu informatickém.

Informatika chápe funkci jako *algoritmus*, popisující výpočet funkční hodnoty v každém bodě definičního oboru. Navíc funkce v našem chápání splňuje požadavky kladené na modul, čímž se ještě více liší od matematického pojetí.

Část programu, která obsahuje zápis kódu funkce nazýváme *deklarace* funkce.[8] Deklarace má klasicky alespoň tři části: jméno funkce, seznam formálních parametrů funkce a tělo funkce. V některých jazycích obsahuje deklarace funkce i tzv. návratový typ. (Typům bude věnována následující kapitola.) Deklarace jednoduché funkce následovníka může v některém programovacím jazyku vypadat takto:

```
function succ(x)
begin
  return x + 1
end
```

`succ` představuje jméno funkce, `(x)` je seznam takzvaných formálních parametrů, `x` je tedy formálním parametrem funkce. Kód uvedený mezi klíčovými slovy `begin` a `end` je tak zvané tělo funkce. V mnoha programovacích jazycích je možné v těle funkce deklarovat lokální proměnné a dokonce i lokální funkce. Pro tuto chvíli však tyto další možnosti nejsou důležité, vystačíme s tímto jednoduchým pojetím.

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

Použití funkce v rámci výrazu nazveme *aplikace* funkce nebo též *volání* funkce. Parametry, uvedené v aplikaci funkce nazveme *skutečné* (též aktuální) parametry funkce.

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

Výsledkem tohoto volání funkce `succ` je hodnota `4`. Jak vlastně k této hodnotě dospějeme? Vyhodnocování funkcí lze shrnout do několika bodů:

- Vyhodnotí se výrazy, uvedené jako skutečné parametry.
- Výsledek vyhodnocení jednotlivých aktuálních parametrů přiřadíme do formálních parametrů.
- Tělo funkce vyhodnotíme, výsledek vyhodnocení vrátíme jako výsledek funkce. Mnohdy bývá výsledek funkce označen klíčovým slovem `return`.

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

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

V zájmu jednoduchosti jsme se v tomto příkladě dopustili několika závažných zjednodušení. K mnohým aspektům se ještě v podrobněji vrátíme v dalších kapitolách. Nyní nutno učinit pouze několik poznámek.

- Tomuto způsobu vyhodnocování aplikací funkcí říkáme *applicative-order evalution*. Existují i jiné způsoby, jejich popis však poněkud přesahuje rozsah kursu a proto se jim nebudeme věnovat.
- Akceptuje-li funkce více než jeden parametr, vyhodnotí se jednotlivé parametry v nějakém, blíže nespecifikovaném, pořadí (pořadí vyhodnocení je dáno implementací jazyka). Nelze v zásadě ani vyloučit, že u paralelních systémů bude vyhodnocování všech parametrů probíhat současně.
- V našem příkladě je skutečným parametrem výraz `2 + 1`. Je možné (a ve funkcionálním programování obvyklé), že skutečným parametrem je výsledek volání jiné funkce.

- Každá aktivace funkce "vlastní" svoji jedinečnou kopii formálních (v průběhu aktivace vlastně aktuálních) parametrů. Pokud by tedy funkce během své aktivace aktivovala další funkci, budou skutečné parametry obou aktivací existovat v paměti počítače nezávisle na sobě - a to i v případě, budou-li mít stejná jména!. Speciálně to platí i pro případ, kdy funkce ve svém těle aktivuje sama sebe. (K tzv. rekurzivním funkcím se vrátíme podrobněji v kapitole 4.4.) Více viz kapitola 4.3

- Zde uvedený způsob předávání parametrů se nazývá se *volání hodnotou*. Existují i další způsoby, které si popíšeme v kapitole 4.7 věnované předávání parametrů.

4.3 Zásobník programu a aktivační strom

V této podkapitole se budeme věnovat velice častým případům, kdy jedna funkce vyvolává (aktivuje) ve svém těle jinou funkci.

Předávání řízení mezi funkcemi (*control flow*) probíhá tak, že pokud funkce `P` volá funkci `Q` a funkce `Q` volá `R` (obrázek 8), tak řízení se vrací po ukončení práce funkce `R` zpět do `Q` a z `Q` zpět do `P`. Návrat do volající funkce pochopitelně probíhá těsně za to místo programu, odkud byla funkce vyvolána. Jednotlivé aktivace můžeme chápat jako kostky dětské stavebnice, které, jak dochází k aktivacím, stavíme na sebe a jak aktivace končí, postupně odebíráme.

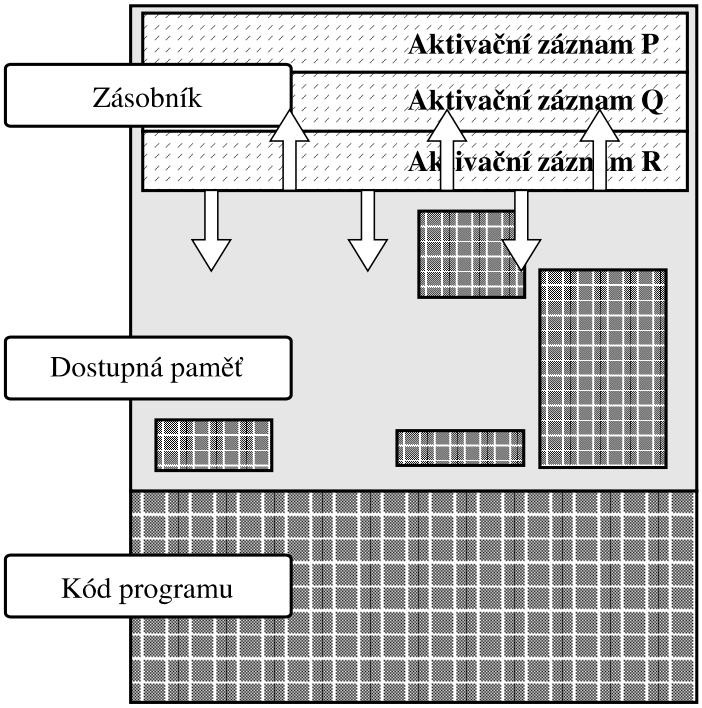
```
function P(x)          function Q(x)
begin                  begin
   return 2*Q(x+2)        return 3*R(x+3)
end                    end
            function R(x)
            begin
               return 4*x
            end
```

Obrázek 8: Předávání řízení mezi funkcemi.

V těchto úvahách je třeba důsledně rozlišit mezi funkcemi (tj. kousíčky programu, moduly) a jejich aktivacemi. Zatímco funkce existuje v programu trvale, její aktivace je iniciována, proběhne a skončí mnohokrát v rámci daného běhu programu.

Aktivace funkce velmi úzce souvisí s organizací paměti programu. Tradičně je paměť programu (procesu) rozdělena na tři části. První částí je paměť vyhrazená *kódu* a globálním proměnným. Programátor s touto částí paměti nemá příliš starostí: v průběhu vykonávání programu se tato část

paměti ani nezvětšuje ani nezmenšuje, modifikovat je možno pouze hodnoty globálních proměnných. Druhou částí je datová oblast zvaná *hromada* (*heap*). To je místo, kde se může programátor plně realizovat. V této oblasti může požádat systém o přidělení libovolně velkého místa a v rámci přidělené paměti dělat, co uzná za vhodné. Poslední částí paměti je *zásobník* (*stack*). Jde o oblast paměti, která se zvětšuje a zmenšuje podle toho, jak probíhá vykonávání programu. Z jakýchsi důvodů ve většině programovacích jazyků zásobník roste směrem dolů (na úkor velikosti hromady). Příklad konfigurace paměti procesu můžeme vidět na obrázku 9.



Obrázek 9: Schematické znázornění paměti procesu.

Zásobník je složen z datových položek, se kterými se pracuje systémem LIFO (Last In First Out). Jde o stejný princip, kterým se řídí nastupování a vystupování do výtahu: ti, kteří do výtahu nastoupili jako poslední a jsou tak nejblíže dveřím, vystoupí z něj jako první. Stejně tak na zásobníku programu záznam, který byl vytvořen jako poslední, je jako první odstraněn. Některé záznamy tak na zásobníku pobudou jen relativně krátkou dobu, zatímco záznamy v hloubce zásobníku tam vydrží i velmi dlouho.

Každé vyvolání funkce přidá jeden záznam na vrchol zásobníku. Po ukončení dané aktivace funkce se záznam z vrcholu zásobníku opět smaže. Vyvolá-li funkce ve svém těle jinou funkci, dojde k tomu, že se záznamy kladou na sebe. Jak aktivace funkcí končí a řízení programu se vrací na místa,

odkud byly funkce volány, záznamy ze zásobníku se umazávají. Tyto zásobníkové záznamy se také někdy nazývají aktivační záznamy. Na obrázku 9 je znázorněna situace, kdy je prováděna funkce `R`. Vidíme, že funkce `R` byla aktivována uvnitř funkce `Q` a funkce `Q` během aktivace funkce `P`.

Co tyto záznamy obsahují? Nejpodstatnější částí záznamu jsou paměťová místa pro uložení aktuálních parametrů funkce. Vzhledem k tomu, že každá aktivace funkce má vlastní záznam na zásobníku, má každá funkce i svoji jedinečnou kopii formálních (v průběhu aktivace vlastně skutečných) parametrů - nehledě na to, že v danou chvíli může být aktivováno více funkcí jejichž formální parametry mohou mít stejné jméno. Obsahuje-li funkce deklaraci lokálních proměnných, jsou taktéž uloženy v záznamu dané aktivace. K dalším položkám se dostaneme v kapitole číslo 4.8 věnované rozsahům platnosti proměnných.

Uvážíme-li deklarace z obrázku 8, co bude výsledkem aktivace `P(Q(R(1)))`? Jak bude probíhat výpočet?

Předávání řízení mezi funkcemi lze přehledně zobrazit pomocí tzv. *aktivačního stromu*.[9] Aktivační strom je strom, jehož uzly představují jednotlivé aktivace funkcí. Pokud se v rámci aktivace funkce `P` inicializuje aktivace funkce funkce `Q`, potom v aktivačním stromu je `P` rodičem `Q`. Průběh výpočtu zjistíme tak, že prohledáme celý strom do hloubky.

Ukažme si komplexnější příklad, ve kterém budeme demonstrovat program na analýzu řetězce závorek. Budeme rozeznávat hranaté a špičaté závorky. Program bude probíhat úspěšně, pokud nalezne odpovídající si dvojice závorek a ohlásí chybu, pokud si závorky nebudou odpovídat.[10] Například následující sekvence závorek označí takto:

`[<>]`	správně
`[[]<[]>]`	správně
`[<]>`	chybně

Program bude předpokládat, že znaky ze vstupu najde v systémové proměnné `input`. Voláním systémové funkce `getnextchar` se do proměnné `input` nastaví další znak ze vstupu. Předpokládáme přitom, že na vstupu bude vždy nějaký znak připravený a tudíž volání funkce `getnextchar` bude vždy úspěšné. Dále předpokládáme existenci systémové funkce `error`, která zastaví vykonávání programu a na chybový výstup vypíše patřičné hlášení.

Budeme deklarovat funkci `M`, která zkontroluje, zda znak na vstupu je stejný jako znak, který jí byl předán jako parametr. Pokud ano, zavolá systé-

[9] Strom je pojem z teorie grafů s širokým uplatněním v celé informatice. Bližší informace lze nalést například v Nečas[4].

[10] Celý program je z důvodu jednoduchosti psán v procedurálním duchu. Funkce zde nevracejí hodnoty - pouze aktivují další funkce, které mění stav systému nebo zastaví vykonávání programu. Ačkoliv by nebylo obtížné, přepsat program do čistě funkcionálního stylu, pro demonstraci aktivačního stromu vystačíme s touto verzí.

movou funkci `getnetxchar`, pokud ne, zastaví vykonávání programu s chybou.

```
function M(t)
begin
   if (input = t) getnextchar
   else error("syntax error")
end
```

Dále budeme deklarovat funkci `parens`, která bude kontrolovat pozice závorek.[11]

```
function parens()
begin
   loop
      if input = '[' begin
                        M('[');
                        parens;
                        M(']');
                     end
      else if input = '<' begin
                             M('<');
                             parens;
                             M('>');
                          end
   endloop
end
```

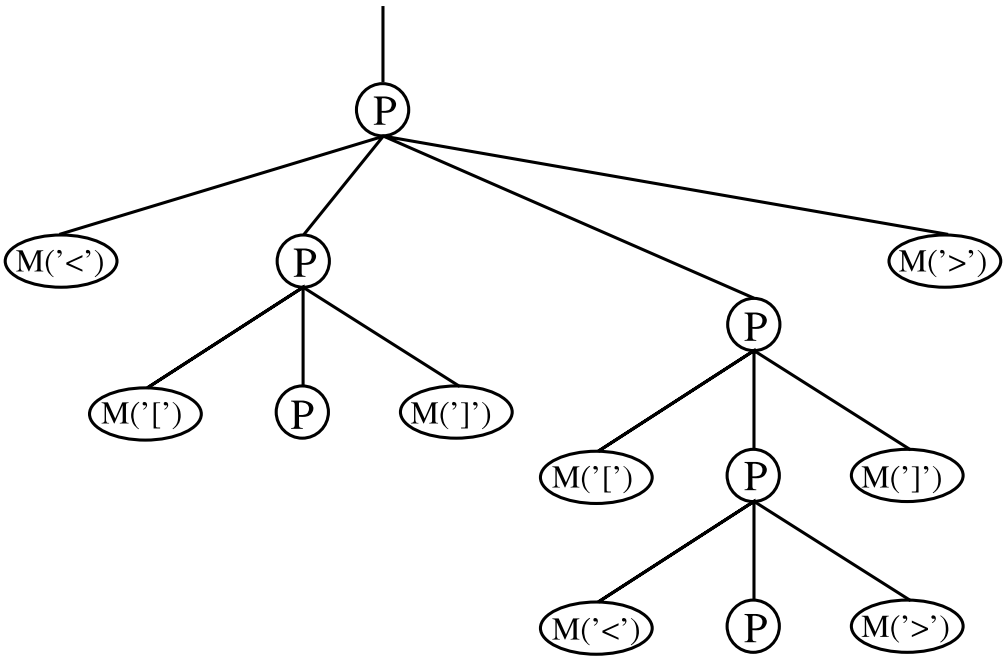
Celý program spustíme voláním funkce `parens`. Pak již můžeme pozorovat, jak probíhá analýza řetězce na vstupu. Například pro řetězec `<[][<>]>` je aktivační strom zobrazen na obrázku 10. P zde označuje aktivaci funkce `parens`, M aktivaci funkce `M`.

4.4 Rekurzivní funkce

Funkci nazýváme *rekurzivní* pokud může ve svém těle aktivovat sama sebe, a to třeba i nepřímo prostřednictvím jiných funkcí. Funkce `parens` z předchozí kapitoly byla rekurzivní. Jako typický příklad rekurzivní funkce nám může sloužit funkce na výpočet n-tého prvku Fibonacciho posloupnosti čísel. Tato funkce je v matematické notaci definována formulí:

$$Fib(x) = \begin{cases} 1 & x = 0 \vee x = 1 \\ Fib(x-1) + Fib(x-2) & \forall x \geq 2 \end{cases}$$

[11] Předpokládáme přitom, že význam řídících struktur označených klíčovými slovy `loop`, `if` a `else` je intuitivně jasný.



Obrázek 10: Aktivační strom funkce `parens` pro vstupní řetězec `<[][<>]>`.

Prvních 6 členů fibonacciho posloupnosti je tedy 1, 1, 2, 3, 5, 8. Kód takovéto funkce může mít v nějakém programovacím jazyce tvar:

```
function Fib(n)
begin
   if n = 0 or n = 1 return 1
   else return Fib(n - 1) + Fib(n - 2)
end
```

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

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

```
   if n = 0 or n = 1 return 1
```

Tento kód tedy řeší problém pro dvě nejmenší přípustné hodnoty čísla `n` - hodnoty `0` a `1`. V těchto případech funkce `Fib` vrátí přímo hodnotu `1`.

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

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

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

Funkci `f` nazýváme *lineárně rekurzivní*, pokud aktivace `f` vyvolá nejvýše jednu novou aktivaci `f`. Příkladem takovéto lineárně rekurzivní funkce může být funkce faktoriál. Naopak funkce pro výpočet Fibonacciho čísel není lineárně rekurzivní, neboť v těle funkce `Fib` jsou vyvolány dvě aktivace funkce `Fib`.

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

$$Fact(x) = \begin{cases} 1 & x = 0, \\ xFact(x-1) & \forall x \geq 1 \end{cases}$$

Kód pro výpočet faktoriálu může vypadat takto:

```
function Fact(n)
begin
   if n = 0 return 1
   else return n * Fact(n - 1)
end
```

Jak vypadá vyhodnocování lineárně rekurzivní funkce? Lze je rozdělit do dvou po sobě následujících fází:

Fáze navíjení (winding phase) - v rámci fáze navíjení jsou aktivovány nové aktivace rekurzivní funkce

Fáze odvíjení (unwinding phase) - řízení se postupně vrací jednotlivým aktivacím

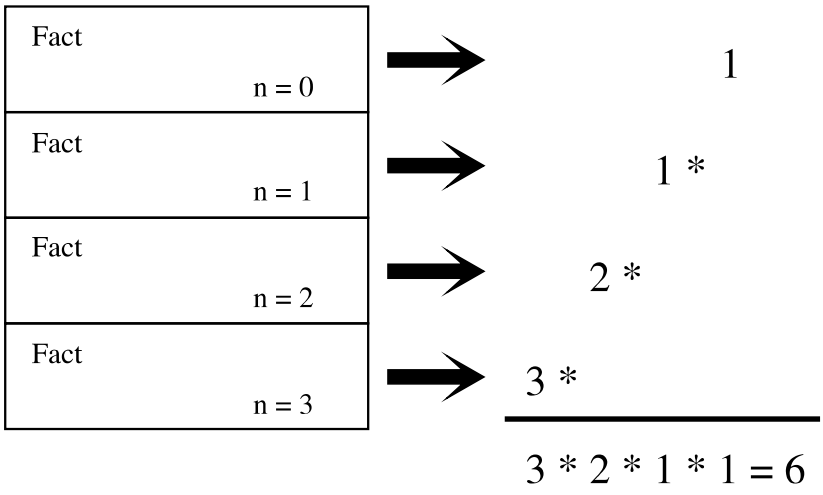
Ukažme si na příkladu, jak probíhá výpočet pro faktoriál čísla 3:

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

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

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

Obrázek 11 znázorňuje stav zásobníku ve chvíli, kdy končí fáze navíjení.



Obrázek 11: Stav zásobníku programu při provádění funkce `Fact` pro výpočet faktoriálu. Předpokládáme, že zásobník roste směrem k hornímu okraji strany.

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

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

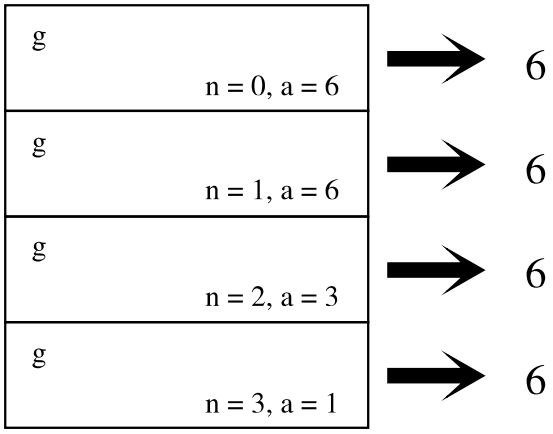
```
function g(n, a)
begin
   if n = 0 return a
```

```
   else return g(n - 1, n * a)
end
```

Vyhodnocení této funkce se skutečnými parametry `3` a `1` vypadá takto:

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

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



Obrázek 12: Stav zásobníku programu při provádění funkce `g` pro koncově rekurzivní výpočet faktoriálu. Předpokládáme, že zásobník roste směrem k hornímu okraji strany.

Koncově rekurzivní funkce je možno podstatně efektivněji implementovat tak, aby jejich výpočet probíhal v konstatním zásobníkovém prostoru. Jinými slovy řečeno, každou koncově rekurzivní funkci je možné převést na cyklus, který nevyžaduje ukládání a pozdější rušení zásobníkových záznamů. Tím lze ušetřit mnoho paměti a strojového času stráveného údržbou zásobníku. Ve skutečnosti koncově rekurzivní funkce *je* cyklus - jedná se jen o jiný zápis iterativního algoritmu!

Předchozí lineárně rekurzivní algoritmus je možno převést na cyklus následujícím způsobem. Formální parametry funkce jsou zde použity jako proměnné cyklu.[12]

[12]Tento příklad jako první používá symbol `=` jak ve smyslu *přiřazovacího příkazu*, tak

```
function g(n, a)
begin
   loop
      if n = 0 return a
      else begin
             a = n * a;
             n = n - 1
           end
   end
end
```

Některé programovací jazyky jsou schopny automaticky detekovat koncově rekurzivní funkce a automaticky převést rekurzivní volání na iterativní proces. Takovýmto jazykům pak také říkáme koncově rekurzivní.

4.5 Vedlejší efekt funkcí

Elementární funkční moduly jsou v jednotlivých paradigmatech nazývány různými jmény: hovoříme buď jen o funkcích, o funkcích a procedurách, o metodách, routinách, podprogramech atd. V našem kursu se soustředíme především na to, co mají všechny tyto druhy modulů společné. Tato podkapitola jen stručně objasňuje problematiku vedlejšího efektu funkcí.

Klasická funkce, typická pro funkcionální programování, je co do chování obdobou funkce matematické. Je-li aktivována, spočte ze seznamu skutečných parametrů a dalších proměnných a konstant prostředí výslednou hodnotu a tu vrátí volající funkci. Pro matematickou funkci není přípustné modifikovat jakékoli hodnoty vně funkce. Takovouto funkci můžeme volat libovolně-krát po sobě a dostaneme vždy stejný výsledek.

V jiných paradigmatech je pro funkce přípustné, aby v průběhu svého vykonávání modifikovaly své vnější prostředí. Vypočtená návratová hodnota tedy není to jediné, co po zbude po jedné aktivaci funkce. Vyvoláme-li takovouto funkci vícekrát po sobě, můžeme dostat různé výsledky. Tento trend může jít tak daleko, že funkce nemá žádnou návratovou hodnotu a její celá činnost je zaměřena na vykonávání vedlejšího efektu. Takováto funkce se většinou nazývá procedura.

O legitimitu existence vedlejšího efektu funkcí v některých paradigmatech se dosud vedou spory. V zásadě lze říci, že vedlejší efekt je na závadu

ve smyslu predikátu ekvivalence. V rámci jednoduchosti jsme nezaváděli zvláštní symbol pro přiřazovací příkaz a doufáme, že význam symbolu = je zřejmý z kontextu. Přiřazovací příkaz *nastavuje* hodnotu proměnné uvedené na levé straně výrazu na hodnotu výrazu uvedeného na pravé straně výrazu. Jde o typický rys procedurálního programování, v ostatních paradigmatech hraje spíše doplňkovou úlohu, pokud není zcela potlačen.

teoretické čistotě jazyka, zhoršuje odladitelnost, komplikuje paralelní zpracování úloh. Na druhé straně zjednodušuje problematiku vstupu a výstupu, řešení chybových situací a v některých případech značně zmenšuje velikost kódu.

4.6 Funkce vysokého řádu

Na začátku této kapitoly jsme uvedli, že funkce představují základní osvědčený způsob strukturace programu. Chápali jsme tedy funkci, jakožto speciální případ modulů. Některé jazyky však umožňují mnohem pružnější manipulaci s funkcemi - chovají se k funkcím stejně jako ke všem ostatním výrazům. V takovýchto jazycích je například možné uvést funkci jako aktuální parametr jiné funkce, vrátit funkci jako výsledek jiné funkce, nebo dokonce psát funkce, které ve svém těle konstruují na základě daných parametrů jiné funkce! O takovémto jazyku řekneme, že pracuje s funkcemi jakožto s *hodnotami vysokého řádu* (*high order values*). Funkce takovéhoto jazyka, které pracují s ostatními funkcemi, nazveme *funkce vysokého řádu* (*high order function*).

Posluchači budou mít mnoho příležitostí důkladně se seznámit s funkcemi vysokého řádu v druhém semestru kursu Paradigmata programování. V tuto chvíli uvedeme pro ilustraci pouze jednoduchý příklad na výpočet kořene libovolné funkce f metodou bisekce (půlení intervalu). Tato metoda předpokládá, že funkce f bude mít na intervalu $\langle a, b\rangle$ alespoň jeden kořen a že funkční hodnoty v bodech $f(a)$ a $f(b)$ mají opačná znaménka. Budeme ji realizovat funkcí vysokého řádu `bisect`, která jako své argumenty akceptuje funkci `f`, mezníky intervalu `a` a `b` a požadovanou přesnost výpočtu `prec`.

```
function bisect(f, a, b, prec)
begin
   if (b - a) < prec return (a + b) / 2
   else
      if f((a + b) / 2) = 0 return (a + b) / 2
      else
         if f(a) * f((a + b) / 2) > 0
            return bisect(f, (a + b) / 2, b, prec)
         else
            return bisect(f, a, (a + b) / 2, prec)
end
```

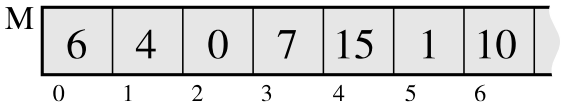
Všimněme si, že funkce `bisect` je koncově rekurzivní.

4.7 Předávání paramerů

V této kapitole jsme si ukázali, jak se funkce deklarují a jak je možno předávat do funkcí hodnoty mechanismem předávání parametrů. Především v procedurálním paradigmatu se však tento způsob předávání parametrů ukázal v praxi nedostatečným a do jazyků byly zaváděny další způsoby předávání parametrů. Z procedurálních jazyků tyto další možnosti v různých modifikacích přecházely do ostatních paradigmat.

V popisu možností předávání parametrů funkci budeme používat pojmy L-value a R-value. Dříve než přistoupíme k samotnému popisu, vyvsvětlíme si význam těchto pojmů. Termíny L-value a R-value vznikly původně při práci s přiřazovacím příkazem, kdy termín L-value označuje hodnotu, která musí být přítomna na levé straně přiřazovacího příkazu a R-value hodnotu stojící vpravo od přiřazovacího příkazu. Máme-li tedy dva výrazy A a B, potom, chceme-li provést přiřazení $A = B$, musí pro výraz A existovat L-value a pro B R-value.

Obecně pojmem *L-value* označujeme místo v paměti (paměťovou buňku) do které lze zapisovat. Jakožto *R-value* chápeme hodnotu, která je uložena na nějakém místě paměti. Mnoho výrazů má jak L-value, tak R-value. Demonstrujme si celou situaci na jednoduchém příkladě. Označme paměť počítače písmenem `M`, jednotlivé buňky paměti jako `M[i]`. Předpokládejme, že paměť je naplněna hodnotami tak, jak je to znázorněno na obrázku 13.



Obrázek 13: Úsek paměti.

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

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

Pro demonstrativní účely si nadefinujeme funkci `swap` takto:

```
function swap(x, y)
var temp;
begin
   temp = x;
   x = y;
   y = temp;
end
```

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

1. Volání hodnotou (Call by value)

   Příklad jazyka: C, Lisp

   Postupně jsou vyhodnoceny všechny aktuální parametry. Výsledné hodnoty jsou přiřazeny dočasným lokálním proměnných, odpovídajícím formálním parametrům funkce. S těmito proměnnými se pracuje jako s libovolnými jinými lokálními proměnými.

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

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

   a provede se kód funkce ve tvaru:

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

   Po vykonání funkce hodnoty `T1` a `T2` zaniknou. Důležitým poznatkem je, že vzhledem k tomu, že veškerá manipulace probíhá s dočasnými proměnnými, hodnoty výrazů `I` a `A[I]` se po provedení funkce nezmění.

2. Volání odkazem (Call by reference)

   Příklad jazyka: PL1, C++

   Nejprve se vyhodnotí aktuální parametry. Předpokládá se přitom, že všechny aktuální parametry mají L-value. Pokud by některý skutečný parametr neměl L-value, je možné jeho R-value přenést do dočasné proměnné a vzít L-value této proměnné. Mnoho jazyků však v této

situaci jednoduše vyhlásí chybu. Do funkce pak pošleme adresy (tedy L-values) aktuálních parametrů.

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

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

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

Dále se provede se kód funkce ve tvaru:

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

Po vykonání funkce hodnoty `Addr1` a `Addr2` zaniknou. Všimněme si však, že manipulace neprobíhala s hodnotami `Addr1` a `Addr2`, ale obsahy paměťových buněk, které leží na těchto adresách. Vedlejším efektem této funkce tedy je, že do paměťové buňky `I` se přiřadí obsah buňky `A[I]` a do původní paměťové buňky `A[I]` se přiřadí obsah buňky `I`.[13]

Používá-li jazyk předávání parametrů odkazem, je důležité si vždy uvědomit, že ve funkcích pracujeme ve skutečnosti s adresami proměnných exitujících mimo funkci. Tento způsob předávání parametrů je velmi výhodný, chceme-li aby funkce měla vedlejší efekt na své parametry. Často se také používá tam, kde jednotlivé výrazy představují velké úseky paměti (například při zpracování obrazu, kdy jedna proměnná může představovat velký blok paměti naplněný obrazovými daty) a bylo by pomalé, nevýhodné nebo dokonce nemožné ukládat do zásobníku lokální kopie hodnot těchto výrazů.

3. Volání hodnotou-výsledkem (Call by value-result, copy-restore linkage)

   Příklad jazyka: Fortran, SR

   Nejprve se u všech skutečných parametrů zjistí L-value i R-value. R-value jsou předány do funkce stejně jako u volání hodnotou. L-value všech parametrů jsou uschovány do tabulky. Po vykonání těla funkce,

[13] Demonstrujte tento efekt na konkrétních hodnotách!

dříve než jsou dočasné proměnné odstraněny ze zásobníku, jsou hodnoty dočasných proměnných nakopírovány zpět do L-value aktuálních parametrů.

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

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

Do tabulky se uloží L-values jednotlivých skutečných parametrů:

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

a provede se kód funkce ve tvaru:

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

Po vykonání funkce se provede přiřazení:

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

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

Tento způsob předávání parametrů dává v mnoha případech podobné výsledky, jako předávání parametru odkazem. Tak tomu je i v našem příkladě. V obecném případě však takovéto tvrzení jistě neplatí. Do programovacích jazyků byl tento způsob předávání parametrů zaveden jak z důvodu jednoduché implementace (Fortran), tak pro výhody, které toto předávání parametrů má oproti volání funkcí odkazem. K typickým případům patří situace tzv. co-rutin, kdy připouštíme paralelní běh volající a volané funkce ve stejné virtuální paměti. V těchto případech může dojít i k tomu, že volající funkce terminuje dříve, než stačí terminovat funkce volaná. V tom případě bychom při volání odkazem dostali ve volané funkci referenci na oblast paměti, jejíž obsah není definovaný a který může být využíván jiným procesem (tzv. dangling pointers). Používáme-li předávání parametrů hodnotou-výsledkem, tento problém nenastane: stačí, když systém zajistí, aby neproběhla poslední fáze.

4. Volání jménem (Call by name)

   Příklad jazyka: Algol68

   U volání funkcí jménem se aktuální parametry považují za jména a jsou opakovaně vyhodnocovány kdykoli, kdy jsou uvedeny. Tento způsob předávání parametrů je dosti komplikovaný, přináší zvláštní vedlejší efekty a v moderních programovacích jazycích se běžně nepoužívá. Předávání parametru jménem je však co do funkčnosti identické s používáním makra.[14]

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

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

   Efekt tohoto kódu si posluchači jistě zjistí sami.

4.8 Rozsah platnosti proměnných

Uvažujme následující kousek kódu. Jde o funkci s tajemným názvem *mystery*. Tato funkce deklaruje kromě lokální proměnné `x` také lokální funkce `P` a `Q`.

```
function mystery

   var x;

   function P
   begin
      print x;
   end

   function Q(x)
   begin
      print x
      P;
   end

begin
   x = 1;
```

[14] V případě makra je před provedením překladu každý výskyt identifikátoru makra nahrazem definovaným segmentem kódu.

```
   Q(2);
end
```

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

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

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

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

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

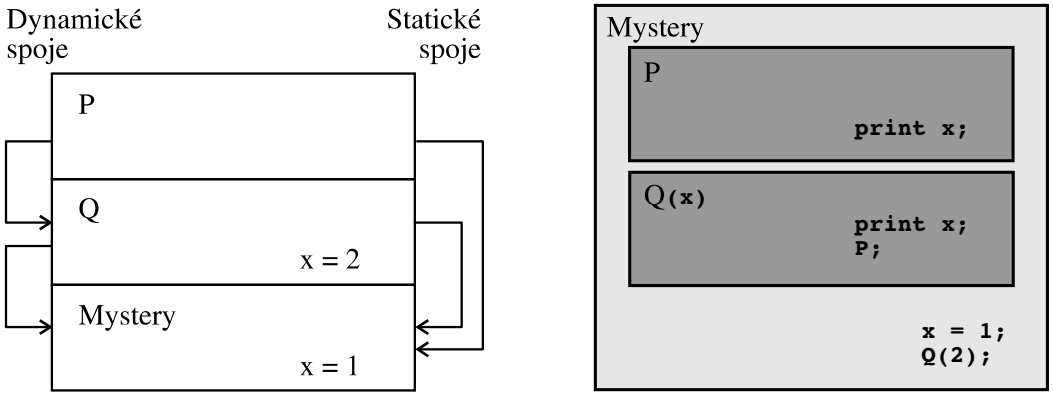
V prvním případě hovoříme o jazyku, podporujícím *statický* (*lexikální*) rozsah platnosti proměnných, ve druhém případě o jazyku, podporujícím *dynamický* rozsah platnosti proměnných.

V kapitole 4.3 jsme se seznámili s pojmem zásobník programu. Kromě návratové adresy, lokálních proměnných a skutečných parametrů obsahuje záznam zásobníku tyto dvě položky, využívané k zjišťování rozsahu platnosti proměnných:

dynamický spoj (dynamic link): ukazatel na předchozí aktivační záznam uložený v zásobníku, tedy ukazatel na aktivační záznam volající funkce.

statický spoj (static link): ukazatel na aktivační záznam nejbližšího lexikálně nadřazeného bloku.

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



Obrázek 14: Dynamické a statické spoje.

Při zjišťování hodnoty dané proměnné programovací jazyk nejprve zjistí, nejde-li o proměnnou lokální danému modulu. Není-li proměnná deklarovaná v daném modulu, hledá modul, ve kterém je proměnná nadeklarovaná a kde je tedy možno zjistit její hodnotu. Toto hledání představuje prohledávání zásobníku za využití spojů. Používá-li jazyk pro tento účel dynamické spoje, uplatňuje *dynamický rozsah platnosti proměnných*, používá-li statické spoje, jde o *lexikální rozsah platnosti proměnných*.

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

Moderní programovací jazyky preferují statický rozsah platnosti proměnných. Dynamický rozsah platnosti proměnných má například Lisp.

5 Typy v programovacích jazycích

V programech se kromě klíčových slov, určených k řízení a strukturaci programu, vyskytují proměnné a konstanty. Proměnné a konstanty můžeme kombinovat pomocí operátorů a funkcí a vytvářet tak složitější výrazy. Každá proměnná, konstanta nebo složený výraz může nabývat jen jistých hodnot a lze s nimi provádět pouze jisté operace. V této souvislosti hovoříme o typu výrazu: *typ* nám označuje, jakých hodnot může výraz nabývat a jaké operace na něj mohou být aplikovány.

Většina programovacích jazyků například rozeznává typ zvaný `Integer`, který reprezentuje celá čísla a operace, které je nad celými čísly možno

provádět. Stejně tak bývá běžný typ `String`, reprezentující textové řetězce spolu s operacemi, prováděnými nad řetězci. Má-li nějaký výraz typ `Integer`, znamená to, že můžeme očekávat jako jeho hodnoty výrazy `25`, `78`, `-2567` atd., ale nemůžeme očekávat, že tento výraz nabude hodnoty např. `"Bobr je hlodavec"`. Dále víme, že na tento výraz můžeme aplikovat operace jako sčítání, násobení, umocňování atd., ale nemůžeme výrazy tohoto typu spojovat nebo převádět na velká písmena. Operace jako spojování nebo konverze na velká písmena jsou definovány pro typ `String` a pro typ `Integer` jsou neznámé.

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

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

5.1 Druhy datových typů

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

Podle prvního kriteria rozeznáváme typy vznikající na:

- úrovni stroje - tyto typy jsou přímo podporované strojem. Bývají to zejména typy `Char` (`Byte`), `Integer` a `Real`.
- úrovni jazyka - tyto typy nejsou definované strojem, ale jsou simulovány programovacím jazykem. Algoritmy operací nad těmito typy nejsou integrovány v elektronických obvodech stroje, ale jsou součástí programovacího jazyka. Patří sem často typy jako `Boolean`, `Array` nebo `Record`.
- úrovni programátora - tyto typy si v daném programovacím jazyku definuje sám programátor. Můžeme sem řadit například typ `List` nebo `Tree`, ale i jednoduchý typ jako je `Enum`.

Podle druhého kriteria lze typy roztřídit na dvě základní skupiny:

- Jednoduché datové typy - datové typy, které jsou atomární (tj. dále již nedělitelné). Ze zástupců těchto datových typů vybereme (bez jakýchkoli nároků na úplnost) například:
  - `Char` - typ zabírající tradičně jednu slabiku paměťového místa počítače, reprezentující písmena nebo čísla v daném paměťovém rozsahu. Často též nazývaný `Byte`.
  - `Integer` - typ zabírající jednu nebo více slabik paměti, reprezentující kladná nebo záporná celá čísla. Též nazývaný `Int`.
  - `Real` - typ reprezentující čísla v plovoucí desetinné čárce, uložené v paměti počítače většinou ve formě mantisy a exponentu.
  - `Boolean` - již zmíněný typ, nabývající hodnot `True` a `False`.
  - `Enum` - datový typ, u něhož je vyjmenováno, jakých hodnot může nabývat. Programátor si může například nadefinovat datový typ `Season` (roční období), jehož jedinými možnými hodnotami budou `Spring`, `Summer`, `Autumn` a `Winter`.
- Strukturované datové typy - datové typy, které vznikly spojením několika jednoduchých datových typů. Každý strukturovaný datový typ musí definovat operaci selekce (tzv. selektor) pomocí něhož je možno přistupovat k jednotlivým položkám datového typu. Mezi nejběžnější typy tohoto druhu patří:
  - `Record` (Záznam) - odpovídá kartézskému součinu několika typů. Je-li například jedním typem `String`, druhým typem `Integer` a třetím typem `Boolean`, můžeme vytvořit datový typ `String` × `Integer` × `Boolean`. Symbolický zápis bývá podobný zápisu:

    ```
    Person = Record
                Name: String;
                Age:  Integer;
                Sex:  Boolean
             end
    ```

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

    Na základě typu `Record` lze budovat další důležité datové typy, jako je například `List` (seznam).

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

– `Array` (Pole) - odpovídá zobrazení z podmnožiny přirozených čísel do daného typu. Tento typ je též nazýván agregovaným typem. Příkladem může být pole 5-ti reálných čísel, formálně zapisovaných třeba jako:

  ```
  Data = array [0 .. 4] of Real
  ```

  Pole `Data` lze pak chápat jako zobrazení

  $$\texttt{Data} : \{0, 1, 2, 3, 4\} \rightarrow \texttt{Real}$$

  Do každé položky pole můžeme zapsat jedno reálné číslo. Selektorem typu pole bývá symbol `[i]`, kde `i` je *index* dané položky pole. Je-li `X` proměnná typu `Data`, potom `X[0]` označuje první položku pole, `X[1]` druhou položku atd.

Zvláštní poznámku si v tomto kontextu zaslouží funkce. V kapitole 4.6 jsme se seznámili s funkcemi, které se chovají stejně jako libovolné jiné výrazy jazyka. Je-li tomu tak, ukazuje se jako nezbytné chápat funkci jako další datový typ - zcela rovnoprávný s jinými datovými typy. Další dimenzi toto pojetí získává u jazyků s manifestovanými datovými typy.

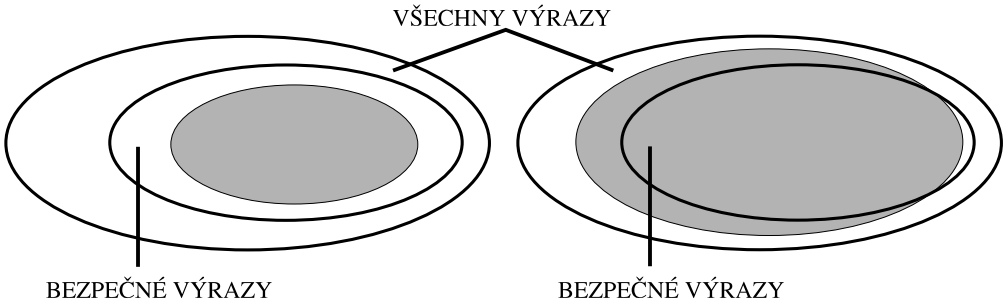
5.2 Typový systém jazyka

Typovým systémem jazyka rozumíme soubor pravidel, která přiřazují výrazům v daném jazyce typ. Máme-li například výraz `2 + 3`, typový systém přidělí tomuto výrazu nejspíše typ `Integer`. Výrazu `2 * (3.14 - 2)` přidělí nejspíše typ `Real`. Pro výraz `7 + "ABCD"` však typový systém typ nenajde. Říkáme proto, že typový systém akceptuje výraz, pokud pro něj nalezne typ a zamítne výraz, pokud k němu typ nenalezne. Výrazy zamítnuté typovým systémem představují chybu a nemohou být provedeny. Samotný proces přiřazování typu výrazům nazýváme též typovou kontrolou výrazů. Typová kontrola má značný význam z hlediska praxe programátora, neboť umožňuje odhalit značné množství programátorských chyb. Ve zkratce lze říci, že typovou kontrolou zjišťujeme, zda typy jednotlivých operandů jsou slučitelné s typy jednotlivých operátorů.

Rozeznáváme dva druhy typových systémů: typový systém silný a slabý. Typový systém nazveme *silným*, pokud akceptuje pouze bezpečné výrazy. Bezpečné výrazy jsou přitom takové, které nemohou za běhu programu způsobit chybu. Typový systém, který není silný se nazývá *slabý*.

Oba systémy mají své výhody a nevýhody. Slabý typový systém může dovolit provedení výrazů, které nejsou bezpečné a hrozí tak nouzové zastavení programu chybou za běhu výpočtu. Jedná-li se program řídící provoz jaderné elektrárny, může mít takováto chyba dosti vážné následky. Silný typový systém tento problém nemá, na druhé straně zase zakazuje provedení

některých výrazů, které jsou bezpečné. Extrémním případem silného typového systému by byl systém, zamítající všechny výrazy. Takovýto systém by však nebyl příliš užitečný.



Obrázek 15: Typový systém silný (vlevo) a slabý (vpravo). Oblast označená rastrem představuje akceptované výrazy.

Důležitou úlohu v programovacím jazyce hraje fakt, kdy se typová kontrola provádí. Existují v zásadě dva přístupy: provádět typovou kontrolu hned při překladu výrazu nebo ji provádět až těsně před vykonáním výrazu strojem. Podle toho hovoříme o statické a dynamické typové kontrole a o manifestačních a implicitních typech.

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

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

Uvědomíme-li si, že typ je také možno chápat jako množinu všech hodnot, jichž může proměnná nabývat, můžeme funkci chápat jako algoritmus, který každé n-tici z kartézkého součinu typů definičního oboru přiřadí prvek z oboru hodnot. Typ funkce, diskutovaný v minulé kapitole, je potom

specifikován definičním oborem a oborem hodnot funkce.

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

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

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

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

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

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

- Zjistí, zda existuje v typovém systému pravidlo pro přiřazení typu výrazu `A + B`
- jedná-li se o statickou typovou kontrolu, zjistí dále, zda `X` může pojmout zjištěný typ výrazu `A + B`

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

- zjištění, zda `f` je typu funkce
- zjištění, zda `f` akceptuje dva parametry
- při statické typové kontrole je navíc třeba provést:
  - zjistit, zda `A` a `B` jsou výrazy, kterým lze přiřadit typ shodný s typy formálních parametrů funkce `f`
  - zjistit, zda `X` může pojmout návratovou adresu funkce `f`

5.3 Přetížení

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

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

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

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

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

Příkladem přetížení funkce může být funkce na výpis hodnot na obrazovku počítače.

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

Zatímco v prvním případě je vykonána funkce, která znak po znaku zobrazí obsah řetězce, v druhém a třetím případě je obsah jistého úseku paměti nejprve převeden nějakým způsobem na číslo v decimální podobě a teprve

potom vytištěn na obrazovku. Funkce `print` je tedy přetížená - označuje celou řadu funkcí pro výpis proměnných různých typů na obrazovku.

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

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

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

```
function "/" (x: Real, y: Real): Real
function "/" (m: Integer, n: Integer): Integer
function "/" (m: Integer, n: Integer): Real
```

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

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

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

může za použití výše uvedených funkcí vést jednak k výsledku `1.4`, a jednak k výsledku `1.5`. Cesty, jak se k těmto výsledkům dostat necháme na posluchačích.

5.4 Polymorfismus

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

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

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

Je důležité si uvědomit, že problém řešený polymorfními funkcemi není principiálně řešitelný přetížením. Polymorfní funkce pracuje uniformně s libovolným typem - a tedy potenciálně s nekonečně mnoha typy. Prakticky je však možné přetížit identifikátor pouze konečného množství funkcí. Navíc je zde i zásadní koncepční rozdíl: v případě přetížení máme mnoho funkcí označeno tímtéž identifikátorem, přičemž každá funkce vykonává jiný úkol. V případě polymorfní funkce máme jedinou funkci, která je schopna plnit tentýž úkol pro různé typy.

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

5.5 Genericita

Programovací jazyk podporuje genericitu, pokud umožňuje parametrizovat datové typy. Parametrizované datové typy se nazývají generické datové typy. Typickým příkladem zabudovaného generického typu je typ pole, zavedený v kapitole 5.1. Deklaraci:

```
Data = array [0 .. 4] of Real
```

lze chápat jako parametrizaci typu `array` typem `Real` a intervalem `1 .. 4`. O skutečné genericitě však většinou hovoříme až tehdy, umožňuje-li jazyk parametrizovat programátorem vytvořené datové typy. To může být příklad typu `Pair`, popisující dvojici prvků stejného typu:

```
Pair = Record[T]
            Item1: T;
            Item2: T
        end
```

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

O generických datových typech lze uvažovat pouze v systémech s manifestovanými typy. Systémy s implicitními dynamickými typy jsou generické

již svou povahou. Význam genericity zpočívá v tom, že umožňuje vyšší formu abstrakce datových struktur. Programátor, píšící například třídící algoritmus, se nemusí zajímat o to, bude-li třídit reálná čísla, celá čísla, řetězce nebo záznamy podle zvoleného klíče. Může se soustředit výhradně na samotný třídící algoritmus. Tentýž kód je potom použitelný na práci s různými datovými typy - podle toho, jaký zvolíme parametr. [15]

5.6 Přetypování a koerce

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

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

```
Real(3) = 3.0
Integer(3.0) = 3
Integer(3.72) = 4
```

Přetypování je pochopitelně možné jen tehdy, jedná-li se o "podobné" datové typy. Asi těžko bychom v obecném případě smysluplně převáděli datový typ `String` na `Integer`. (Převod obráceným směrem je však myslitelný.)

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

```
Pair = Record
          LoByte: Char;
          HiByte: Char
       end
```

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

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

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

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

```
3.14 + 7
```

na výraz

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

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

6 Závěr

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

Autor s radostí přijme jakékoli výtky, náměty na zlepšení a upozornění na chyby, vyskytující se v textu.

Reference

[1] Sethi, R. [1989] *Programming languages.* Addison-Wesley, Reading, Massachusetts. ISBN 0-201-10365-6.

[2] Meyer, B. [1988] *Object-oriented Sotware Construction.* Prentice-Hall Int., London. ISBN 0-13-629049-3 (paperback 0-13-629031-0)

[3] Dahl, O. J., Dijkstra, E. W. a Hoare, C. A. R. [1972] *Structured Programming.* Academic Press, London.

[4] Nečas, J. [1978] *Grafy a jejich použití.* Polytechnická knižnice, Praha.