1 Programovacie Jazyky

1.1 Základné pojmy

V kontexte programovacích jazykov uvažujeme dva základné pojmy:

- Výraz (expression) - kombinácia operátorov, konštánt, premenných, funkcií a iných prvkov jazyka ktorá sa vyhodnotí na nejakú výslednú hodnotu (môže ísť o prázdnu hodnotu). Výraz môže ale nemusí mať vedľajšie efekty. (Príklady: 2+1.5, A == 3, null, pop())
- Príkaz (statement) - Najmenšia vykonateľná samostatná jednotka imperatívneho jazyka. Príkaz nevracia hodnotu a podstatné sú teda len jeho vedľajšie efekty. (Príklady: A = 4, return 5, if A == 3 then X)

Program sa tradične skladá z postupnosti príkazov ktoré pri svojom vykonaní môžu vyhodnocovať rôzne výrazy potrebné k ich dokončeniu. Vyhodnocovanie výrazov môže prebiehať podľa rôznych pravidiel, tieto pravidlá potom súhrnne nazývame *vyhodnocovacia stratégia* (napr. lenivá alebo striktná).

Príkazy sa bežne združujú do blokov (v niektorých jazykoch je pojem blok zameniteľný s pojmom funkcia) , pričom bloky môžu byť pomenované aj nepomenované a tradične majú jeden vstupný bod a jeden alebo viac výstupných bodov (return).

Na označenie výrazov ktoré sa vyhodnotia na true alebo false (tzv. booleovské výrazy) sa často používa pojem *podmienka*.

Pri programovacích jazykoch môžeme hovoriť jednak o *syntaxi* - teda o spôsobe, akým sú jednotlivé príkazy a výrazy zapísané - a o *sémantike* - teda o tom, akým spôsobom sa jednotlivé príkazy a výrazy vykonávajú.

1.2 Dátové typy

Typ predstavuje vlastnosti a spôsob reprezentácie nejakých dát počítačom. Typy delíme na:

1.2.1 Prázdny dátový typ (void)

Reprezentuje absenciu hodnoty.

1.2.2 Primitívne dátové typy

Tieto typy sú typicky implementované na úrovni jazyka a väčšinou sa priamo mapujú na reprezentáciu údajov v procesore. Typicky sem patrí integer, Float, Boolean, Char, Fixed Point(fixný počet desatinných miest), Bignum(ľubovoľne veľké číslo) a ich varianty s rôznymi bitovými šírkami v závislosti na architektúre.

1.2.3 Zložené dátové typy

Zložené typy vznikajú spojením niekoľkých hodnôt primitívneho typu. Výsledkom je potom dátová štruktúra. Rôzne jazyky implementujú rôzne základné dátové štruktúry. (Dátová štruktúra = spôsob efektívneho uloženia informácie v pamäti) Nízkoúrovňové jazyky často žiadne dátové štruktúry neimplementujú a poskytujú iba priamy prístup do pamäte. Vyššie programovacie jazyky potom implementujú napr:

- Štruktúra - fixný počet prvkov s jasnou adresou (nemusí byť priamo za sebou, ale je pevne dané ako sa k jednotlivým prvkom dostanem)
- Union/Tagged Union - Taktiež má fixnú veľkosť, avšak môže obsahovať hodnoty rôznych typov. Prípadne môže zahŕňať príznak indikujúci ktorý typ je aktuálne uložený.
- Pole - prvky sú uložené v pamäti postupne za sebou. Typicky ide o prvky jedného typu.
- Reťazec - typicky pole znakov, avšak môže byť zložitejší, keďže nie všetky kódovania majú fixnú dĺžku znaku
- Zoznam - Prvky sú nejakým spôsobom zoradené, avšak nemusia byť nutne v pamäti za sebou (spojovaný zoznam). Nie vždy musí ísť o prvky jedného typu.
- Množina - Prvky nie sú zoradené a každý prvok je v množine maximálne raz.
- Asociatívne pole/mapa - Umožňuje prístup k prvkom na báze kľúč/hodnota.

1.2.4 Výpočtové typy (enum)

Výpočtový typ je typ, ktorý môže nadobúdať hodnoty z pevnej danej množiny prvkov. Tieto hodnoty môžu, ale nemusia byť usporiadateľné.

1.2.5 Odkazy a referencie

Ide o hodnoty ktoré obsahujú odkaz na nejaký kus pamäte. Rozdiel medzi odkazom a referenciou nie je pevne daný, ale všeobecne sa predpokladá že referencia je ïnteligentnejší"pointer (napr. taký pre ktorý sa priebežne kontroluje či je pamäť kam odkazuje platná).

Typicky existuje špeciálna hodnota pre odkaz, ktorá predstavuje neplatnú pamäť (null).

Odkaz môže potenciálne ukazovať aj na spustiteľný kód.

1.2.6 Abstraktné dátové typy

Abstraktný dátový typ popisuje isté vlastnosti pamäte ktorú reprezentuje, ale necháva priestor na rôzne implementačné detaily. Typicky napr. rozhranie v objektovom programovaní.

1.3 Riadiace štruktúry *(control structures)*

Riadiaca štruktúra je príkaz ktorý na základe poskytnutých údajov (precondition) mení poradie/smer vykonávania programu (control flow).

1.3.1 If-Then-Else

Riadiaca štruktúra If-Then-Else umožňuje podmienené vykonávanie blokov kódu. V prípade splnenia podmienky danej klauzulou IF sa vykoná blok kódu daný klauzulou Then, v opačnom prípade blok kódu daný klauzulou Else. V extrémne jednoduchých jazykoch nemusí byť klauzula Else prítomná, avšak dá sa nahradiť testom na negáciu pôvodnej podmienky.

If-Then-Else sa dá v prípade potreby násobne vnárať (do Else vetvy sa vloží ďalšie IF). Pokiaľ jazyk takúto konštrukciu podporuje priamo na syntaktickej úrovni, hovoríme o riadiacej štruktúre If-Then-Elseif-Else.

1.3.2 Switch/Case a Pattern Matching

V prípade že treba zvážiť veľké množstvo prípadov, stáva sa často kombinácia if-else veľmi neprehľadnou. Z toho dôvodu sa zaviedla riadiaca štruktúra Switch (V mnohých jazykoch nazývaná aj Case). Príkaz switch sa tradične vyhodnocuje pre nejakú premennú, pričom v tele príkazu sú uvedené možné hodnoty danej premennej a k nim sú priradené bloky kódy. Pri vykonávaní príkazu sa vyhodnotí ten blok kódu, ktorý je priradený hodnote ktorú daná premenná aktuálne nadobúda. Často je taktiež možné uviesť ešte dodatočný blok, ktorý sa vykoná v prípade že žiadna z hodnôt nevyhovuje aktuálnemu obsahu premennej.

Pattern matching (Hľadanie vzorov?) je potom rozšírenie tejto techniky, ktoré dovoľuje okrem klasickej rovnosti testovať na širšiu skupinu vlastností (napr. regulárne výrazy v prípade textových reťazcov) prípadne sa pýtať na samotnú štruktúru/typ premennej (je to dvojica? Je to zoznam s troma prvkami?). Pattern matching je oveľa silnejší ako samotný switch, avšak narozdiel od switchu, daná premenná nemusí vždy vyhovovať len jednému vzoru (problém či sú dva vzory disjunktné/ekvivalentné nemusí byť ani rozhodnuteľný). V takom prípade sa štandardne vykoná kód asociovaný s prvým nájdeným vyhovujúcim vzorom. Konkrétna špecifikácia podporovaných vzorov je potom závislá priamo na jazyku.

1.3.3 While-Do/Do-While

Riadiaca štruktúra While-Do (while cyklus) umožňuje opakovane vykonávať daný blok kódu v prípade že je splnená potrebná podmienka. Pri vykonávaní sa najskôr otestuje podmienka daná klauzulou While. V prípade že je podmienka splnená, vykoná sa blok kódu daný klauzulou Do (telo cyklu). Následne sa proces opakuje až do okamihu kedy je podmienka nesplnená. Ide o najjednoduchší používaný spôsob iterácie s premenlivým počtom opakovaní.

Alternatívou k While-Do je štruktúra Do-While ktorá sa od While-Do líši iba v tom, že vyhodnotenie prebieha v opačnom poradí. Teda najskôr sa vykoná kód daný klauzulou Do a až následne sa testuje podmienka. V prípade že je podmienka splnená sa opäť pokračuje ďalším vykonaním tela cyklu. Praktický význam Do-While spočíva v tom, že garantuje aspoň jedno vykonanie tela cyklu.

1.3.4 For-Do/Foreach

Riadiaca štruktúra For-Do (for cyklus) je rozšírenie while cyklu ktoré umožňuje prirodzenejšiu kontrolu nad počtom iterácií. For cyklus sa skladá z dvoch príkazov, jednej podmienky a tela cyklu. Prvý príkaz slúži na inicializáciu iterácie a vykoná sa len raz, hneď pri vstupe do for cyklu. Následne sa vyhodnotí podmienka. V prípade že je podmienka splnená, vykoná sa telo cyklu. Ak podmienka splnená nie je, cyklus končí. Po úspešnom vykonaní tela cyklu sa vykoná posledný tretí príkaz, ktorý má za úlohu posunúť stav na ďalšiu iteráciu. Následne sa pokračuje opäť testovaním podmienky.

Druhou, striktnejšou variantov for cyklu je tzv. Foreach cyklus. Foreach cyklus sa používa na zlepšenie prehľadnosti kódu v miestach, kde je počet iterácií cyklu dopredu známy (napr. iterácia cez všetky prvky poľa). Foreach cyklus vykoná telo cyklu raz pre každý prvok nejakej iterovateľnej premennej (Čo je to iterovateľná premenná závisí na implementácií daného jazyka, štandardne ide o nejaký zoznam, množinu, prípadne ohraničený interval).

> Poznámka: Alternatívny spôsob delenia cyklov je na Podmienkami riadené (while), Počítadlom riadené (for) a Riadené kolekciou (foreach).

1.3.5 Break/Continue

Riadiace štruktúry Break a Continue umožňujú predčasné ukončenie iterácie cyklu alebo celého cyklu.

Príkaz break vykoná okamžitý skok hneď za koniec cyklu, bez toho aby kontroloval podmienky alebo vykonával nejaké dodatočné akcie. Break je taktiež jediný spôsob ako zastaviť nekonečný cyklus. Alternatívou príkazu break môžu byť cykly s tzv. stredovou podmienkou, čo je v podstate podmienený break.

Príkaz continue funguje obdobne, avšak skočí iba na koniec tela cyklu, čím efektívne preskočí zbytok práve vykonávanej iterácie. Ďalej sa pokračuje štandardne, teda sa vyhodnotí podmienka cyklu prípadne sa vykonajú nejaké ďalšie nutné akcie.

Okrem break a continue sa používajú ešte príkazy Retry a Redy ktoré sú prakticky opakom break a continue. Retry spôsobí skok pred začiatok cyklu, takže sa celý cyklus začne vykonávať znovu (ale nevracia premenné do pôvodného stavu). Retry zase spôsobí skok na začiatok iterácie. Tieto príkazy sa často nepoužívajú, lebo majú pomerne neintuitívnu sémantiku, avšak sú implementované napr. v Ruby alebo Perle.

1.3.6 Goto a Značky (labels)

Label je označené/pomenované miesto v programe. Príkaz goto label (prípadne podmienené goto) je potom skok na danú značku. Všeobecne sa neodporúča používať kvôli zlej čitateľnosti výsledného kódu (program stráca "lineratiu").

Príkaz lebel sa niekedy používa v súvislosti s vnorenými cyklami a príkazmi break/continue na indikáciu toho, na ktorý cyklus sa daný break/continue vzťahuje.

1.3.7 Throw, Try-Catch, Finally (Výnimky)

Throw-Try-Catch-Finally je riadiaca štruktúra súvisiaca s obsluhou výnimiek (výnimočných/nečakaných udalostí v programe). Výnimky slúžia na relatívne pohodlnú obsluhu možných zlyhaní a problémov v programe bez nutnosti zavádzať niektoré explicitné kontroly.

Príkaz Try-Catch(E) indikuje, že sa má vykonať kód v bloku danom klauzulou Try. Pokiaľ je pri vykonávaní tohto

kódu vyvolaná výnimka typu E, pôvodné vyhodnocovanie sa zastaví a vykoná sa blok daný klauzulou Catch.

V súvislosti s Try-Catch sa používa aj príkaz Finally, ktorý je spojený s dvomi blokmi kodu. Prvý blok sa vyhodnotí normálne (teda v prípade vyvolania výnimky skončí) pričom pre druhý blok platí, že sa vyhodnotí po skončení prvého bloku vždy, bez ohľadu na to či sa v prvom bloku vyvolá výnimka alebo nie.

Príkaz Throw vyvolá výnimku. Vyvolanie výnimky v podstate znamená skok do najbližšieho Catch bloku zodpovedného za obsluhu daného typu výnimky. Pokiaľ taký blok neexistuje, vykonávanie programu je zastavené s chybou. (Výnimka môže byť okrem throw vyvolaná aj inými príkazmi jazyka, napr. pri delení nulou)

1.4 Typy programovacích jazykov

Existuje množstvo kritérií podľa ktorých sa dajú deliť programovacie jazyky. Väčšina jazykov sa ale nedá vždy úplne striktne zaradiť do jedného typu.

Jedným kritériom je paradigma ktoré implementujú:

- Imperatívne - štruktúrované/objektové paradigma (C, C++, Java, .NET)
- Deklaratívne - funkcionálne/logické paradigma (Haskell, Prolog, Lisp, F#)

Iným je typový systém ktorý používajú:

- Staticky typované - Typy sa kontrolujú iba podľa obsahu zdrojového kódu (napr. v čase kompilácie) (C)
- Dynamicky typované - Typy sa kontrolujú priamo za behu programu (Python)

Iné možné kategorizácie sú potom podľa účelu, miery abstrakcie a podobných "neobjektívnych"metrík :)

Posledným dôležitým kritériom je spôsob vykonávania kódu. V tomto prípade sú tri základné možnosti: Interpretácia, Kompilácia a Just-In-Time kompilácia.

1.4.1 Kompilácia

Kompilované programovacie jazyky sú pred spustením prevedené z textového zápisu priamo do strojového kódu ktorý je vykonávaný procesorom. Kompilované jazyky sú väčšinou rýchle, pretože kompilátor môže stráviť netriviálny čas optimalizáciou jednotlivých častí kódu. Zároveň ale vyžadujú rôzny kompilátor pre rôzne architektúry procesorov a ťažko sa prispôsobujú dynamickým vlastnostiam zariadenia (napr. počtu dostupných procesorov alebo množstvu pamäte). Kompilácia veľkých programov je taktiež často zdĺhavá, teda spomaľuje samotný vývoj.

Kompilácia prebieha v niekoľkých krokoch, ktoré sa môžu prípadne opakovať. Celý proces sa štandardne rozdeľuje na tzv. Front-End a Back-End.

Front-End je tradične zodpovedný za:

- Preprocessing - Rozvinutie makier, importovanie súborov...
- Lexikálna analýza - Rozdelenie súvislého textu do tzv. tokenov. Token je jedna atomická jednotka jazyka (číslo, kľúčové slovo, názov premennej). Na špecifikovanie a hľadanie tokenov sa štandardne používajú regulárne výrazy a automaty.
- Syntaktická analýza - Sekvencia tokenov je usporiadaná do stromu(parse tree) daného gramatikou spracúvaného jazyka. Rozpoznávajú sa jednotlivé príkazy a výrazy.
- Sémantická analýza - Vyhodnocuje sa korektnosť zadaného programu z pohľadu sémantiky daného jazyka. Objekty v spracúvanom strome sa doplnia o sémantické informácie na základe ktorých sa vyhodnotia typy, prítomnosť nedefinovaných symbolov a pod.
- Výstup do medzijazyka - Pre zjednodušenie sa výsledný strom často ukladá do medzijazyka ktorému bude rozumieť back-end fáza spracovania. Napr. assemblér alebo LLVM bitkód.

Back-End následne pracuje s výstupom front-endu, teda buď so samotným sémantickým stromom alebo s kódom v medzijazyku. Tradične vykonáva tieto kroky:

- Analýza - analyzuje sa control flow a data flow programu, rôzne závislosti premenných a pod.
- Optimalizácia - Na základe analýzy sa program transformuje tak, aby pracoval efektívnejšie prípadne aby bol menší.
- Generovanie kódu - Optimalizovaný program sa uloží vo finálnej podobe spustiteľnej na cieľovom procesore.

1.4.2 Interpretácia

Interpretovaný program je vykonávaný interpretom, čo je program (sám môže byť kompilovaný alebo interpretovaný) ktorý priebežne parsuje a vykonáva zdrojový kód daného programu. Interpretované programy sú typicky pomalšie, pretože kód sa prakticky neoptimalizuje a navyše je nutné ho dynamicky spracúvať počas vykonávania. Avšak odpadá nutnosť kompilácie, čím sa zrýchľuje vývoj a zlepšuje prenositeľnosť jazyka.

Alternatívne niektoré interpretery môžu zdrojový kod pred spustením preložiť do istej formy medzijazyka ktorý sa potom jednoduchšie interpretuje.

1.4.3 Just-In-Time Kompilácia

Just in time (JIT) kompilácia sa snaží spojiť výhody interpretácie a kompilácie. Kód takéhoto jazyka sa štandardne kompiluje až v okamihu keď je spustený (tradične sa taktiež nekompiluje celý naraz, ale kompilujú sa len tie kúsky ktoré budú skutočne spustené). Prípadne je možné časť programu iba interpretovať a kompilovať iba často vykonávané miesta. Tým sa výrazne zvýši výkon a zároveň sa zachová väčšina dobrých vlastností interpretovaných jazykov.

JIT sa hodí hlavne pre dlhodobo bežiace aplikácie, keďže čas potrebný na prvotnú kompiláciu je v takom prípade zanedbateľný. Výhodou je taktiež to, že je potenciálne možné za behu meniť implementáciu a znovu kompilovať jednotlivé časti kódu, prípadne uplatňovať rôzne optimalizácie až v okamihu keď je potvrdené že budú efektívne.