Zevrubná mluvnice jazyka počítačového

PLIN048 – Základy programování pro humanitní obory

Richard Holaj

Obsah

Úvod

Lidský jazyk jako takový se na první pohled jeví jako obrovský komplexní systém, který nám poskytuje takřka nekonečné možnosti. Schopnost popsat v úplnosti jen slovní zásobu jednoho jazyka je dnes už úkol daleko nad síly jednotlivce, protože se neustále rozšiřuje, a snaha ji kompendiálně popsat by se v současnosti dala přirovnat takřka k boji s větrnými mlýny. Přesto, schopnost jazyk používat a rozumět mu je uložena v několika malých částech orgánu, který jako celek váží méně než jeden a půl kila.

Jak je tedy možné uložit v podstatě nekonečné množství kombinací slov a jejich významů při takovém omezení velikosti úložiště? Na to, abych vám dal správnou a exaktní odpověď, nejsem tou povolanou osobou, a nejsem si jist, zda taková osoba v současnosti vůbec existuje, ale asi stejně jako já tušíte alespoň odpověď přibližnou.

Tou odpovědí je systém. Neukládáme všechny kombinace, ale systém, kterým je kombinujeme. To, jak tento systém ve skutečnosti funguje, je již velice dlouho předmětem celé řady bádání, ale tato skripta nejsou o tom, jak funguje lidský jazyk, ale kladou si za cíl studentům, kteří nejsou technicky orientovaní, osvětlit principy jazyka počítačového.

Proč tedy tento zdánlivě nesouvisející úvod? U jazyka počítačů se nacházíme ve velice podobné situaci, potřebujeme popsat v podstatě nekonečně mnoho různých postupů, které má počítač vykonat, ale k dispozici máme přístroj, který má ze své podstaty značně omezenou paměť. I zde je odpovědí systém. Ten musí být dostatečně jednoduchý, aby jeho zapamatování nebylo pro počítač příliš náročné. Zde jsme však v opačné situaci, nemusíme bádat, o jaký systém se jedná, protože jsou to lidé, kdo jej vytvořili.

Úkol, který na nás v této situaci padá, má povahu tvůrčí, neboť jsme zde ti, kdo pomocí pravidel, která jsme stvořili a učinili je součástí „mozku počítače“, musí počítači vysvětlit, co po něm chceme a jak to má udělat.

Pro tuto činnost se vžil termín programování. Dává nám k dispozici nekonečně mnoho kostiček lega, ale typů kostiček není mnoho. V těchto skriptech se vám pokusím vysvětlit, kdy použít jakou kostičku a jak tyto znalosti použít k tomu, abyste z nich postavili vše, co budete potřebovat.

Takovýchto stavebnic, říkejme jim programovací jazyky, je však celá řada a ne všechny mají stejné kostičky. Naší stavebnicí bude především jazyk Python a jeho blízký příbuzný JavaScript, trávící většinu času na internetu, nebude-li řečeno jinak. Příležitostně si vypůjčíme některé kostičky, které Python ani JavaScript neposkytují, z jazyka Java.

Doufám, že vám tato skripta dopomohou ke kreativitě i tam, kde jste nevěřili, že se může skrývat. Berte však na vědomí, že obsahují celou řadu zobecnění, která pro edukativní účely některá fakta platná pro určité programovací jazyky zanedbávají či zatajují, pokud však pochopíte elementární principy, nemělo by vám činit problémy si tyto mezery doplnit.

Část I.
Základní elementy a konstrukce

1. Co je to programování?

1.1. Jak počítače rozumí programům

1.1.1. Koloběh života programu

Naším prvním úkolem je pokusit se pochopit, co to je vlastně program, jak mu počítač rozumí a jak program vytvořit. V první řadě musíme pochopit, že program jako takový je v podstatě sada instrukcí a instrukce je něco, čemu počítač rozumí a na co reaguje daným způsobem. Dalo by se také říci, že počítač se vždy nachází v nějakém stavu, přičemž tímto stavem rozumíme data, která jsou právě teď v paměti. A tento stav mimo jiné uchovává informaci o tom, která instrukce programu se má vykonat jako další.

Každá instrukce na základě vstupu a/nebo aktuálního stavu změní výstup/stav, čímž získáme stav nový (včetně nové instrukce, která se má vykonat), a takto pokračujeme neustále dokola s novými instrukcemi. S každou změnou stavu se nám tedy informace o tom, která instrukce následuje, může změnit (typicky na další instrukci, v pořadí v jakém jsou psány v programu, ale může se změnit i na libovolnou další nebo instrukci zopakovat). Program končí v okamžiku, kdy stav neobsahuje informaci o další instrukci, respektive tato informace nám říká, že program již dál nemá pokračovat. Tento (zjednodušený) obecný princip běhu programu nám ukazuje Obrázek 1.1.

1.1.2. „Překladatel“ vs. „Tlumočník“

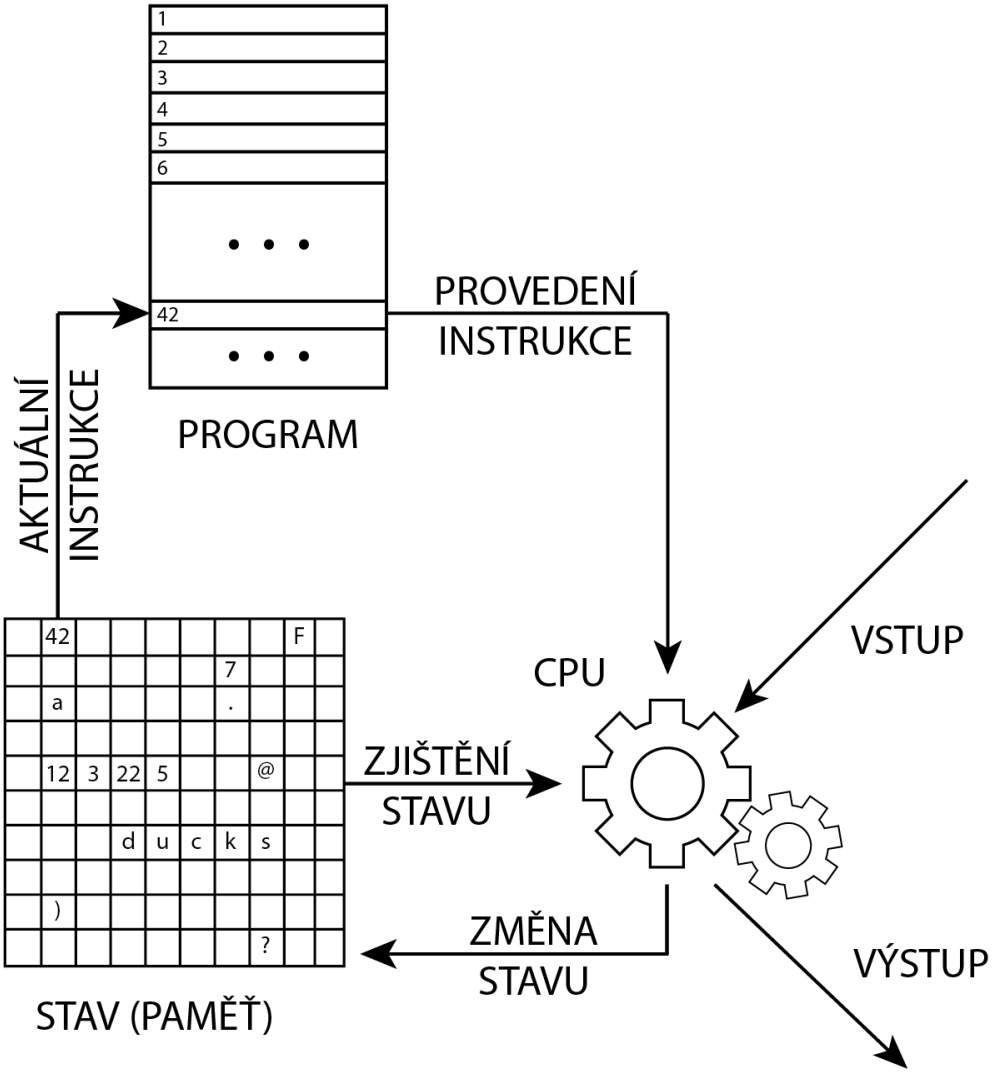
Zmínil jsem, že program jako takový je sada instrukcí. S jistou mírou nadsázky by se dalo říci, že se jedná o dialog s počítačem, z tohoto nám plyne, proč se pravidlům, kterými jsou programy zapsány, říká programovací jazyky. Nastává zde však jeden problém. Každý počítač rozumí pouze elementárnímu „jazyku“ strojových instrukcí, které jsou specifické pro daný přístroj. Pokud bychom chtěli mluvit s počítačem v jeho „mateřském jazyce“, museli bychom tyto instrukce znát a vzhledem k jejich omezenému množství by se jednalo o velice zdlouhavý a komplikovaný rozhovor.

Programovací jazyky jsou jakýmsi kompromisem, nejedná se totiž o jazyky, kterým by počítač přímo rozuměl, spíše o jazyky, kterým je schopen porozumět člověk a zároveň se dají počítači buď přeložit, nebo přetlumočit. Právě to, zda se daný jazyk počítači překládá, nebo tlumočí, rozlišuje základní dva typy programovacích jazyků, a to jazyky kompilované a interpretované.

O jazycích kompilovaných mluvíme tehdy, když je program vždy nejprve celý přeložen překladatelem do řeči počítače (v počítačovém „hantecu“ se tomuto překladateli říká překladač nebo také kompilátor a nejedná se o člověka, ale o program).

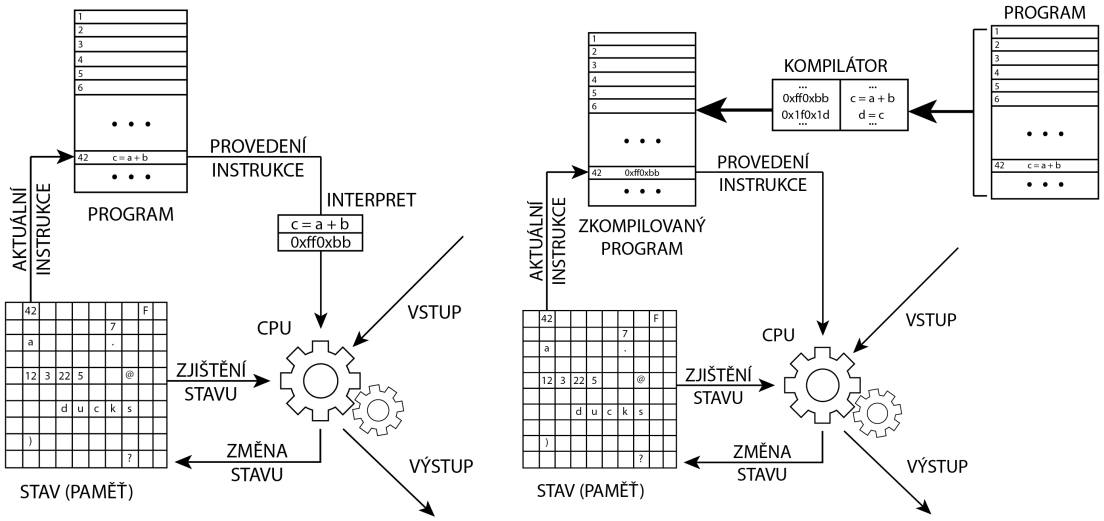
Oproti tomu jazyky interpretované počítači program „simultánně tlumočí“ až při jeho spuštění, a to příkaz po příkazu (tomuto tlumočníkovi se říká interpret).

Jakési hybridní řešení pak používá například Java, která program celý přeloží do tzv. byte codu (jazyk na půl cesty mezi výchozím programovacím jazykem a cílovým jazykem počítače) a poté tlumočí počítači z tohoto mezijazyka. Můžeme si to představit tak, že tlumočník dostane přeložený přepis promluvy ze zdrojového jazyka do angličtiny a poté tlumočí z angličtiny do jazyka cílového.



Obrázek 1.1: Základní cyklus programu I

Obrázek 1.2: Základní cyklus programu II



(a) interpretované jazyky

(b) kompilované jazyky

Jak vypadá schéma z Obrázku 1.1 pro kompilované a interpretované jazyky ukazuje Obrázek 1.2.

1.2. Reprezentace stavu (proměnné)

Jak jsem již zmínil, v programování rozumíme stavem počítače to, co je aktuálně uloženo v jeho paměti, chceme-li paměť organizovat, potřebujeme nějak rozlišit, co máme kde uloženo. Pro tyto účely používáme proměnné.

1.2.1. Pojmenování

Proměnné jsou v podstatě pojmenování nějakého místa v paměti, případně pojmenování reference, která ukazuje na místo v paměti. Proměnná se pak může chovat jako malá paměť a být rovněž dále strukturována a pojmenována, na tomto principu fungují komplexnější typy proměnných, jako jsou seznamy, slovníky, objekty aj., o kterých si povíme v dalších kapitolách. Pro pojmenování proměnných platí určitá pravidla, typicky proměnné mohou obsahovat čísla, písmena a některé speciální znaky (např. _ nebo $), přičemž čísly název nesmí začínat. Chceme-li do proměnné uložit nějakou hodnotu, použijeme operaci přiřazení. Pro tu používáme typicky operátor =, jenž má na levé straně název proměnné a na pravé straně hodnotu, kterou chceme na dané místo uložit. Jednoduché uložení hodnoty do proměnné ukazuje Kód 1.

```
1 |     my_variable = 42
```

Kód 1

Obrázek 1.4: Proměnná jako reference vs. hodnota

(a) reference (b) hodnota

1.2.2. Proměnné jako koncept v odlišných jazycích

Proměnné se mohou v daném jazyce chovat v zásadě dvěma způsoby, jako hodnota nebo jako reference.

Jsou-li proměnné typu hodnota, je proměnná v podstatě pojmenované místo v paměti a změníme-li její hodnotu, změní se hodnota uložená na daném místě.

Mají-li proměnné povahu referenční, je proměnná v zásadě ukazatel na místo v paměti a při přiřazení nové hodnoty se vytvoří nové místo v paměti s přiřazovanou hodnotou a proměnná bude ukazovat nově na toto místo, přičemž data na původním místě zůstanou nezměněna. V případě, že na nějaké místo již neukazuje žádná proměnná, paměť se uvolní pro další použití.

Většina programovacích jazyků oba přístupy kombinuje a v některých případech se chová jako reference a v jiných jako hodnota. Oba přístupy ukazuje Obrázek 1.4.

Jazyky, se kterými se v tomto předmětu setkáme (Python, Javascript a okrajově rovněž Java), tento přístup kombinují, a to tím způsobem, že sice s proměnnými pracují jako s hodnotami (tedy pokud máme dvě proměnné, které ukazují na stejné místo, a jedné z nich změníme hodnotu, hodnota druhé proměnné se nezmění), ale těmito hodnotami jsou ve skutečnosti reference. O důsledcích tohoto přístupu později.

1.3. Zápis instrukcí (výrazy)

1.3.1. Elementární instrukce a stavební bloky

Vytváříme-li program, je třeba si uvědomit, že jazyk, kterému rozumí počítač, je velice omezený a jedná se jen o základní instrukce typu *načti*, *ulož*, *sečti*, *odečti*, *skoč na místo*, *skoč do jiného (pod)programu*, *větší*, *menší*, *rovno*, *a zároveň*, *nebo* a několik málo dalších. Oproti tomu programovací jazyky, jako jsou Java, Python aj., nám poskytují mnohem komplexnější operace.

Pokud bychom sledovali vývoj programovacích jazyků od jejich prvopočátku, zjistíme, že na začátku jsme měli k dispozici pouze tyto elementární instrukce a programy byly podrobné popisy, zatímco modernější jazyky nám obvykle umožňují používat více a více komplexní příkazy, jakými jsou podmínky, cykly, funkce nebo objekty.

Ty jsou však ve skutečnosti jen většími bloky, „prefabrikáty“, které nám autor jazyka připravil, a při překladu nebo interpretaci se nahrazují posloupností instrukcí, ze kterých jsou složeny. Máme tu výhodu, že již nemusíme znovu vynalézat kolo, přesto je však dobré si uvědomovat, nejen jak toto „kolo“ funguje, ale také proč se tak děje, jelikož nám to může pomoci lépe vyřešit situaci, kdy se správně neotáčí. Právě tento překlad nám zajišťují kompilátor a interpret popsané v 1.1.2

1.3.2. Vyhodnocování jednoduchých výrazů

Programovací jazyky nám tedy poskytují celou řadu operací (od aritmetických operací až po cykly), které nám umožňují sestavovat výrazy. Počítač je však ve skutečnosti schopen vyhodnotit v jednu chvíli pouze jednoduchý výraz, tj. jednoduchou operaci s parametry, které mají danou hodnotu, například $x+y$, kdy známe hodnotu x a y. Složitější výrazy se tedy (podobně jako v matematice) vyhodnocují pomocí substituce. Zvolíme si operaci, kterou chceme vyhodnotit, a pokud je na pozici operandů složený výraz, vypočítáme jeho hodnotu a substituujeme za jeho výsledek.

To, jaká operace se vypočítá nejdříve a jak probíhají operace, je řízeno dvěma principy.

Prvním z nich je, že se nejprve vyhodnocují operace s vyšší prioritou. Priorita operací je dána pro každý jazyk, ale bývá intuitivní, typicky se nejprve vyhodnocují proměnné (dosadíme za ně jejich aktuální hodnotu), volání funkcí (provedeme funkci a dosadíme výsledek), poté operace typu mocnění, dále násobení a dělení, sčítání a odčítání, ještě nižší prioritu mají porovnávací operace (větší, menší, rovno) a logické operace (negace, a, nebo).

V případě operací se stejnou prioritou se výraz vyhodnocuje zleva doprava.

Chceme-li pořadí vyhodnocování změnit, použijeme stejně jako v matematice závorky. Celé vyhodnocování si můžeme představit jako složkový strom (znalí formální syntaxe již vědí), přičemž platí, že operandy se nám spolu s operací spojují do výsledku a tento výsledek může být operandem další operace.

Srovnejte Obrázek 1.6 a Obrázek 1.7.[1] Oba uvedené kódy jsou v jazyce Python a v obou případech uvažujeme hodnotu proměnné a jako 10 a hodnotu proměnné b jako 5.[2]

Je důležité si uvědomit, že operace přiřazení jako taková už nemůže být substituována a je to vždy poslední vyhodnocovaný výraz.

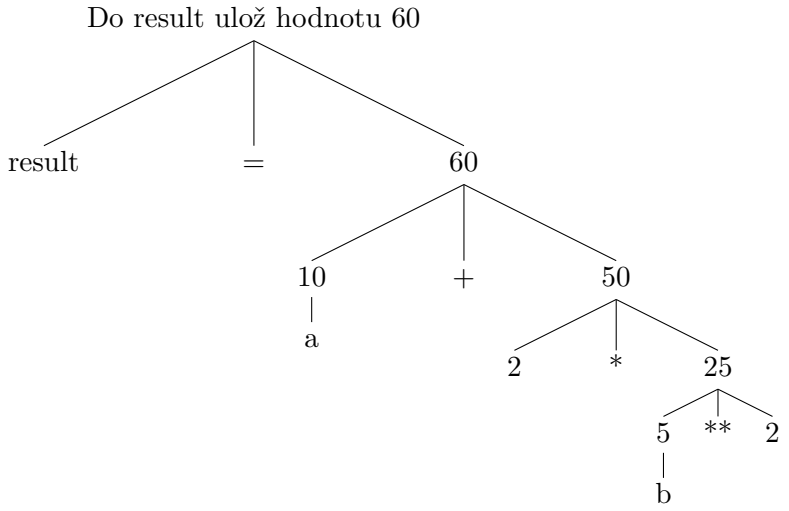
[1] ** → *mocnina*

[2] Pro účely testování výsledků výrazů se vám mohou hodit speciální příkazy pro výpis:

$$\mathbf{print}(\mathbf{a} + \mathbf{2} * \mathbf{b}) \qquad \text{(Python)}$$

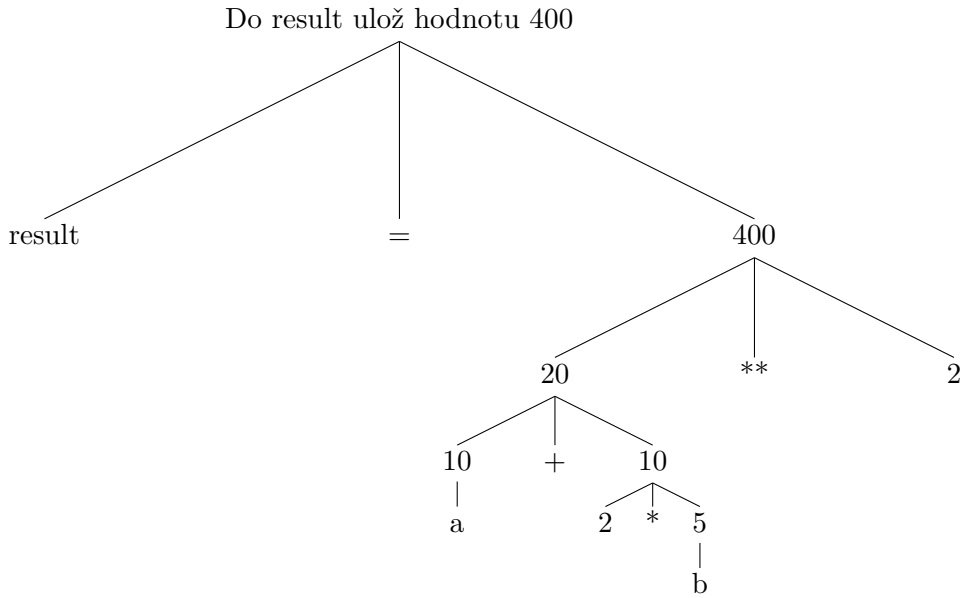
$$\mathbf{console.log}(\mathbf{a} + \mathbf{2} * \mathbf{b}) \qquad \text{(Javascript)}$$

```
result = a+2*b**2
```



Obrázek 1.6: Vyhodnocení složeného výrazu bez uzávorkování

```
result = (a+2*b)**2
```



Obrázek 1.7: Vyhodnocení složeného výrazu s uzávorkováním

Příklady k procvičení

1. **Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu v jazyce s proměnnými typu hodnota?**

```
a = 10
b = a
b = 20
```

2. **Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu v jazyce s proměnnými typu reference?**

```
a = 10
b = a
b = 20
```

3. **Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu v jazyce s proměnnými typu hodnota?**

```
a = 10
b = a + 15
b = 20
```

4. **Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu v jazyce s proměnnými typu reference?**

```
a = 10
b = a + 15
b = 20
```

5. **Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu v jazyce s proměnnými typu hodnota?**

```
a = 10
b = a
a = 7
```

6. **Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu v jazyce s proměnnými typu reference?**

```
a = 10
b = a
a = 7
```

7. **Nakreslete strom vyhodnocení volání výrazu pro následující kód v jazyce Python.**

```
a = 10
b = 20
result = (a + 2) * (10 * a) ** b - 4
```

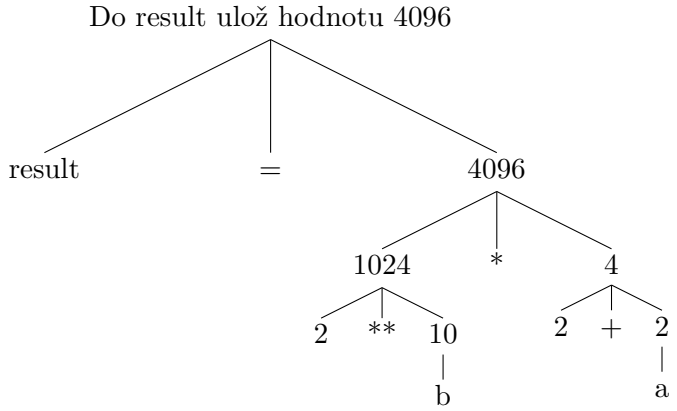
8. Nakreslete strom vyhodnocení volání výrazu pro následující kód v jazyce Python.

```
a = 10
b = 20
result = (a + b**2 - 10) * 4
```

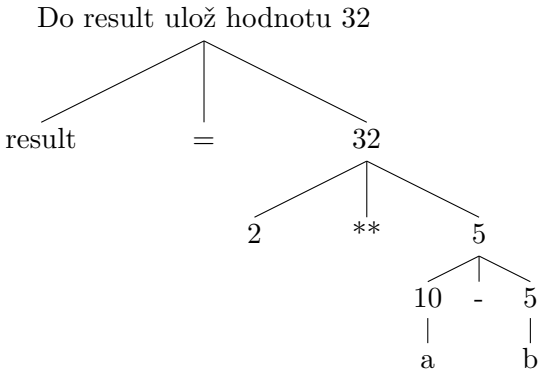
9. Nakreslete strom vyhodnocení volání výrazu pro následující kód v jazyce Python.

```
a = 10
b = 20
result = (a**2 + 10) * a**(3*b)
```

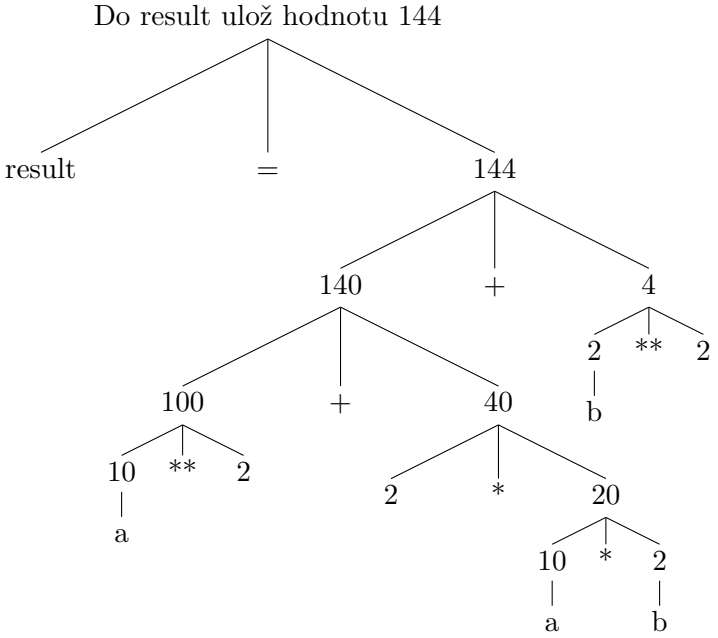
10. Napište kód např. v Pythonu, který odpovídá následujícímu stromu vyhodnocení výrazu:



11. Napište kód např. v Pythonu, který odpovídá následujícímu stromu vyhodnocení výrazu:



12. Napište kód např. v Pythonu, který odpovídá následujícímu stromu vyhodnocení výrazu:



2. Datové typy a kolekce

2.1. Co je uloženo v paměti?

2.1.1. Princip datových typů

V minulé kapitole jsme se naučili, co to jsou proměnné a jak se s nimi pracuje. Pro zjednodušení jsme tiše předpokládali, že se jedná pouze o čísla. Proměnné, respektive data, se kterými chceme pracovat, mohou být ale různého typu. V proměnné může být hodnota číselná (celé nebo reálné číslo), pravdivostní (pravda nebo lež), případně textová (libovolná posloupnost znaků). Toto jsou základní datové typy, které nám poskytuje většina programovacích jazyků. Některé jazyky je dále dělí podle maximálních a minimálních hodnot (například na 32bitová a 64bitová celá čísla), ale těmito detaily se v našem případě nemusíme nutně zabývat.

2.1.2. Jak určit datový typ

Nabízí se tedy otázka, jak poznáme, o jaký typ se zrovna v naší proměnné jedná. Například jak poznáme, zdali hodnotou 7 myslíme číslici ve smyslu textovém, nebo číslo? K této otázce přistupují různé jazyky odlišnými způsoby a jejich práce s datovými typy se postupně dělí podle tří kritérií.

První z nich rozděluje jazyky na typované a netypované, říká nám tedy, zda vůbec umožňují pracovat s různými datovými typy, nebo ne, nicméně s jazyky, které by to neumožňovaly, se dnes nejspíše setkáte už jen v učebnicích historie. Zbylá dvě kritéria se pak aplikují jen na jazyky typované.

Druhé kritérium dělí jazyky na explicitně a implicitně typované a říká nám, zda při vytvoření nové proměnné musíme její typ explicitně uvést, nebo si jej program sám odvodí podle dosazené hodnoty.

Explicitní jsou v podstatě všechny jazyky kompilované, z ostatních (nekompilovaných) známých jazyků je to pak například Java.

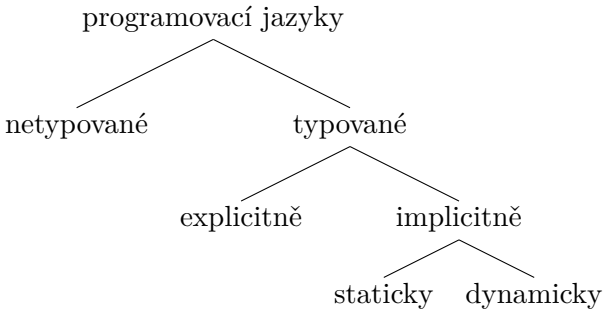
Jak takové typování může vypadat? Kód 2 ukazuje vytvoření různých typů proměnných v jazyce Java:

```
1 int a = 42;
2 String b = "Hello universe!";
3 boolean c = true;
4 float d = 10.0;
```

Kód 2

Oproti tomu implicitní typování umožňuje poznat datový typ podle přiřazené hodnoty. Každý typ má určitý formát, kterým hodnoty zapisuje (čísla se zapisují jako čísla; text se zapisuje jako posloupnost znaků v apostrofech nebo uvozovkách, aby se odlišil od názvu proměnných; reálná čísla jsou čísla celá, obsahující navíc desetinnou část oddělenou tečkou, pravdivostní hodnota obsahuje klíčové slovo $True$ nebo $False$ apod.).

Tento formát má každý programovací jazyk pevně daný a podle toho, kterému formátu odpovídá hodnota proměnné, je určen datový typ. V Pythonu, který je na rozdíl od Javy jazykem typovaným implicitně, bychom tedy jako obdobu Kódu 2 použili Kód 3.



Obrázek 2.1: Členění programovacích jazyků podle typování

```
1 a = 42
2 b = "Hello universe!"
3 c = True
4 d = 10.0
```

Kód 3

Implicitně typovaný je rovněž JavaScript, s tím rozdílem, že zde (jak ukazuje Kód 4) musíme explicitně označit, že vytváříme novou proměnnou, a to klíčovým slovem *var*.

```
1 var a = 42
2 var b = "Hello universe!"
3 var c = true
4 var d = 10.0
```

Kód 4

Zbývá nám třetí kriterium, to dělí jazyky na staticky a dynamicky typované.

Ve staticky typovaných jazycích platí, že jakmile má proměnná jednou přiřazený datový typ, již v ní nemohou být uloženy hodnoty jiného datového typu.

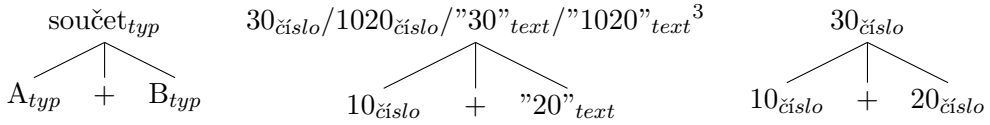
V dynamicky typovaných jazycích je datový typ určen podle právě uložené hodnoty, tudíž pokud uložíme do proměnné hodnotu jiného typu, než byla původní hodnota, změní se datový typ proměnné.

Z těchto vlastností se dá očekávat, že toto dělení se týká především implicitně typovaných jazyků, jelikož u explicitně typovaných jazyků musí být typ určen explicitně při vytvoření, a bylo by tudíž komplikované jej u již vytvořené proměnné měnit. Základní dělení programovacích jazyků podle určování datového typu proměnné shrnuje Obrázek 2.1.

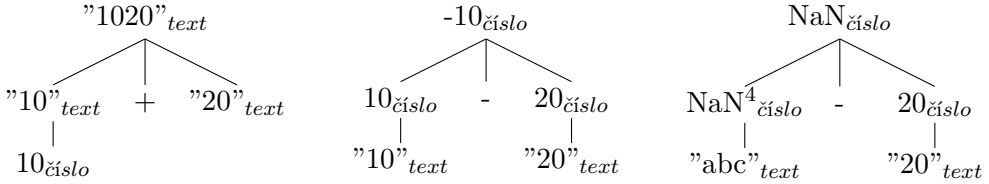
2.1.3. Přetypování a typová omezení

Pokud máme v jazyce nějakou konstrukci, obvykle klade určité požadavky na svoje parametry, případně podle typu parametrů mění svůj význam. Příkladem takových omezení v češtině jsou například shoda nebo pádová omezení.

V programovacích jazycích jsou tato omezení na argumenty operací daná často jako povolené datové typy. Pokud operace (např. sčítání) dostane takovou kombinaci argu-



Obrázek 2.2: Omezení pro operaci součet, kdy $typ \in \{\ číslo,\ text,\ \ldots\ \}$



Obrázek 2.3: Operace součet a rozdíl v JavaScriptu (implicitní přetypování)

mentů, která není povolena (tzn. není definováno její chování), operace, případně i celý program, skončí chybou.

Jako příklad můžeme vzít operaci sčítání. Ta je mimo jiné definována pro dvě čísla jako jejich součet a pro dva řetězce (texty) jako jejich spojení. Jak ukazuje Obrázek 2.2, v Pythonu není definována pro číslo a text ani text a číslo (tyto varianty se mohou lišit, záleží totiž na pořadí argumentů, proto jsou zde uvedeny obě). Oproti tomu v JavaScriptu je tato varianta definována díky takzvanému implicitnímu přetypování.

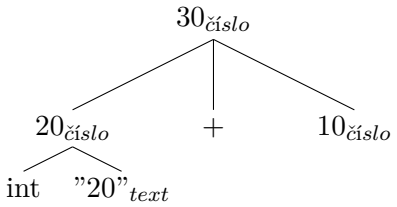
Implicitní (automatické) přetypování je vlastnost některých jazyků, kdy existuje hierarchie typů, které lze implicitně převádět. Nejvýše v této hierarchii je (v našem případě) text a číslo je níže. Výsledný typ, na který jsou veškeré hodnoty převedeny (je-li to možné), je tedy nejvyšší typ v hierarchii, pro který je operace definována.

Například máme-li operaci sčítání pro text a číslo, výsledkem bude text a číslo je převedeno rovněž na text, protože text je hierarchicky výše a operace sčítání je pro text definována. Pokud by operací bylo odčítání, výsledkem bude číslo, protože odčítání není pro text definováno. Všimněte si, že se zde převádí z textu na číslo, což není vždy možné. Jak bychom například reprezentovali řetězec "a" jako číslo? V tomto případě by výpočet skončil chybou, případně by výsledkem byla nějaká speciální hodnota typu NaN (Not a number).

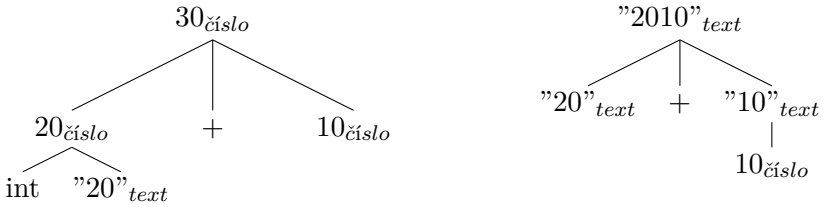
Toto pozorování nám ukazuje zajímavý fakt, který platí obecně pro implicitní přetypování a jeho hierarchii. Typ, který je hierarchicky výše, zahrnuje (nebo je schopen reprezentovat) všechny hodnoty typu, který je níže. Proto je výchozím chováním zvolit typ, který je výše, protože víme, že převod bude možný. Převod směrem dolů nemusí vždy existovat, jak nám ukazuje případ, kdy se snažíme převést "a" na číslo, případně dochází ke ztrátě části informace (například při převodu desetinného čísla na celé), a je tudíž potenciálně nebezpečný. To nám napovídá, proč je v hierarchii text výše než číslo, veškerá čísla totiž dokážeme popsat formou textu. Různé případy pokusu o implicitní přetypování v JavaScriptu nám ukazuje Obrázek 2.3.

[3]Program neví, kterou variantu zvolit, a proto skončí chybou.

[4]„Not a number“ = speciální konstanta, výsledek převodu nevhodné hodnoty na číslo



Obrázek 2.4: Explicitní přetypování v Pythonu



Obrázek 2.5: Explicitní přetypování vs. implicitní přetypování

Ne všechny jazyky nám poskytují možnost implicitního přetypování (například Python), v podstatě všechny jazyky nám ale umožňují přetypování explicitní, které nám umožňují řešit stejné situace. Python není výjimkou, explicitní přetypování se chová v podstatě stejně jako přetypování implicitní s tou výjimkou, že pokud jej chceme použít, musíme to programu říci.

Explicitní přetypování nám obvykle umožňují i jazyky s implicitním přetypováním, přičemž přetypování explicitní má před tím implicitním přednost. Zápis přetypování v Pythonu ukazuje Kód 5, přetypování v Javě pak Kód 6. Odpovídající strukturu ukazuje Obrázek 2.4.[5]

```
1 int("20") + 10
```

Kód 5

```
1 (int) "20" + 10
```

Kód 6

Porovnejte vyhodnocení totožného výrazu s implicitním a explicitním typováním v jazyce (např. JavaScriptu), který poskytuje obě možnosti (Obrázek 2.5).

[5] *int* odpovídá datovému typu celé číslo

2.2. Složitější strukturování paměti

2.2.1. Jak seskupovat hodnoty, které mají něco společného

Do této chvíle jsme o proměnných uvažovali jako o něčem, co uchovává jednoduchou hodnotu, jako je číslo nebo text. Velice často ale chceme uchovávat komplexnější informace. Takovým příkladem může být třeba posloupnost čísel nebo slov nebo výsledky testů jednotlivých studentů. Pro tyto účely nám slouží složené datové typy, základními složenými datovým typy jsou objekty a kolekce. O objektech si povíme ve třetí části těchto skript, nyní se zaměříme pouze na kolekce.

Kolekcí obecně rozumíme více proměnných, které jsou uloženy jako jeden balíček v jiné proměnné. Obvykle rozeznáváme dva základní typy kolekcí, a to kolekce uspořádané a neuspořádané.

2.2.2. Uspořádané, neuspořádané skupiny

Uspořádaným kolekcím se obvykle říká pole nebo seznamy. Seznamy se typicky zapisují jako výčet jednotlivých položek oddělených čárkami uzavřený v hranatých závorkách. Jednotlivé položky jsou pak automaticky označeny čísly od 0 do n v pořadí, v jakém jsou uloženy, tedy číslo pořadí položky v seznamu. Podle typu programovacího jazyka může seznam obsahovat buď pouze položky stejného typu (typické pro explicitně typované jazyky, např. Javu), nebo položky různých typů (typické pro jazyky implicitně typované, např. Python či JavaScript). K jednotlivým položkám přistupujeme pomocí pořadových čísel, kterými jsou označeny, těm se obvykle říká indexy.

Nejobvyklejším neuspořádaným kolekcím se často říká slovníky, mapy či asociativní pole. My se budeme držet pojmu slovník. Od uspořádaných se liší jen nepatrně. Také se zapisují jako položky oddělené čárkou, nicméně nikoliv uvnitř hranatých závorek, ale uvnitř závorek složených. Dalším rozdílem je, že položka zde není pouze hodnotou, ale jedná se o dvojici (jméno položky a hodnota položky) oddělenou dvojtečkou, přičemž jménu položky se často říká klíč a k jednotlivým hodnotám přistupujeme právě pomocí těchto klíčů. Stejně jako u seznamů se i u slovníků liší jazyk od jazyka, zda můžou obsahovat různého typu, a i u slovníků platí, že Python i JavaScript nám toto umožňují.

Dalo by se tedy říci, že neuspořádané kolekce jsou obecnější než kolekce uspořádané, jelikož u uspořádaných kolekcí máme klíče pevně dané pořadím (říkáme jim indexy a jde o číslo pořadí počítané od 0), zatímco u kolekcí neuspořádaných si tyto klíče určujeme sami. Jedná se tedy opět o klasický rozdíl implicitní vs. explicitní vyjádření.

Ke kolekci jako k celku přistupujeme klasicky pomocí názvu proměnné. K jednotlivým položkám přistupujeme tak, že k názvu proměnné přidáme klíč/index dané položky v hranatých závorkách (tak odlišíme, kde začíná název položky a kde končí název proměnné jako celku).

Jelikož kolekce může obsahovat položky libovolného typu (případně více typů), může obsahovat i jiné kolekce. Tímto získáváme tzv. vícerozměrné (někdy se používá i pojem vícedimenzionální) kolekce.

Pokud chceme přistupovat k položce, která je uvnitř jiné položky (jedná se tedy o kolekci uvnitř kolekce), postupujeme stejně, jako když přistupujeme k položce jednoduché,

tedy k názvu kolekce přidáme klíč/index dané položky v hranatých závorkách, s tím rozdílem, že názvem kolekce je zde název prvotní kolekce a klíč/index položky, která obsahuje naši vloženou kolekci. Tento princip můžeme opakovat pro libovolné množství zanoření a je totožný pro přístup k hodnotě i pro ukládání nové hodnoty.

Příklady práce s uspořádanými i neuspořádanými kolekcemi v jazyce Python ukazuje Kód 7.[6]

```
1  seznam = [10, 20, "AHOJ", ["A", 20], {"TEST": 42}]
2  print(seznam[2]) # "AHOJ"
3  seznam[2] = 751
4  print(seznam[2]) # 751
5  print(seznam[3][1]) # 20
6  seznam[3][1] = "B"
7  print(seznam[3][1]) # "B"
8  print(seznam[4]["TEST"]) # 42
9  seznam[4]["TEST"] = 42 * 2
10 print(seznam[4]["TEST"])  # 84
11 slovnik = {"A":"a", "l":[1, 3], "hex":{"A":10, "FF":256}}
12 print(slovnik["A"])  # "a"
13 slovnik["A"] = 42
14 print(slovnik["A"])  # 42
15 print(slovnik["l"][0])  # 10
16 slovnik["l"][0] = [1,2]
17 print(slovnik["l"][0])  # [1, 2]
18 print(slovnik["l"][0][0])  # 1
19 print(slovnik["hex"]["FF"])      # 256
```

Kód 7

2.2.3. Reprezentace složených proměnných v paměti

V 1.2 bylo zmíněno, že některé jazyky se chovají k proměnným jako k hodnotě a některé jako k referenci. A to ve smyslu, že u jazyků, které berou proměnnou jako hodnotu, operátor = mění tuto hodnotu, zatímco u jazyků považujících proměnnou za referenci, operátor = mění hodnotu na místě, kam proměnná referuje.

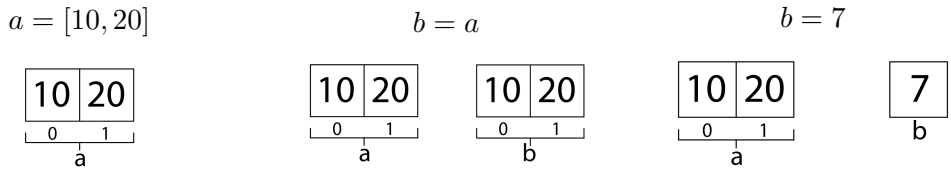
Z tohoto vyplývá, že u hodnotových typů na sobě dvě proměnné nemohou být závislé. To však neplatí pro typy referenční, srovnejte Obrázek 2.6 a 2.7.

Jak bylo rovněž zmíněno v 1.2, Python i JavaScript používají přístup hybridní, který oba výše zmíněné přístupy kombinuje. Pracuje s hodnotami, ale tyto hodnoty jsou referencemi. Důsledkem toho je, že přiřazení měnící hodnotu zároveň mění referenci. Tímto vzniká rozdíl mezi přiřazením hodnoty kolekci a přiřazením hodnoty položce kolekce, jak ukazuje Obrázek 2.8.

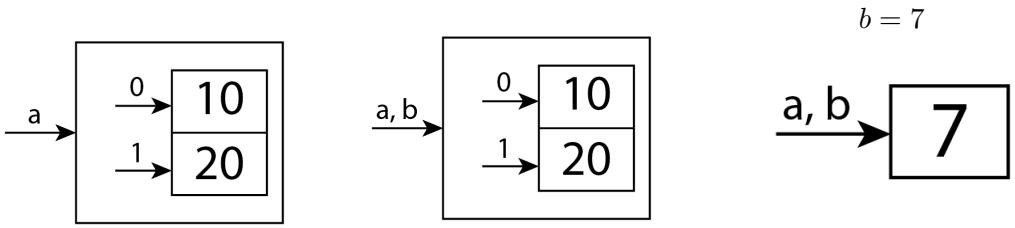
[6] Znak # označuje komentáře, tedy kód, který se počítačem nevyhodnocuje a má pouze informační charakter pro autora. V ukázce jsou použity pro zobrazení výpisu funkce *print*.

Kód by byl v podstatě totožný i v jazyce JavaScript a to s tím rozdílem, že pro deklaraci nové proměnné bychom museli použít klíčové slovo *var*, místo příkazu *print* bychom použili příkaz *console.log* a místo znaku # dvojici znaků //.

Obrázek 2.6: Proměnná jako hodnota

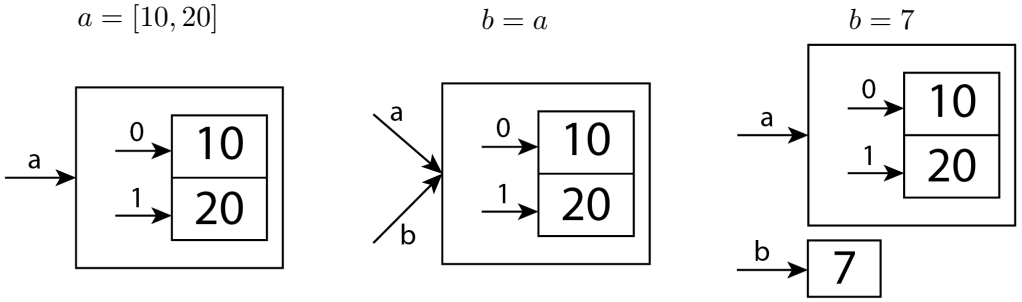


Obrázek 2.7: Proměnná jako reference

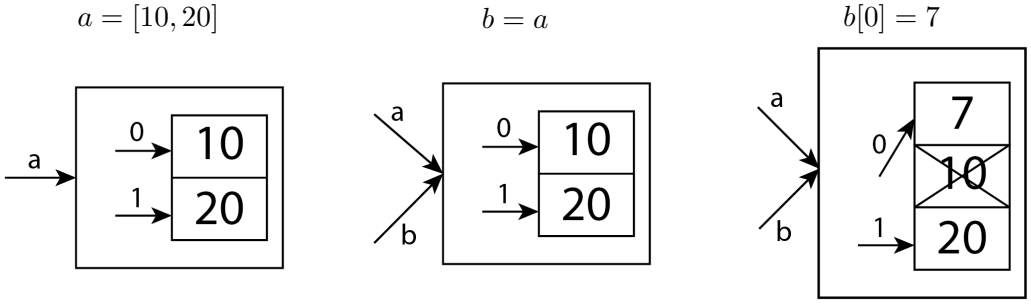


Obrázek 2.8: Proměnná jako reference/hodnota (Python/JavaScript)

(a) Přiřazení hodnoty kolekci



(b) Přiřazení hodnoty položce



Na úrovni jednoduchých typů se tedy proměnné chovají jako hodnota, zatímco položky se v rámci kolekcí chovají jako reference, je důležité brát tuto skutečnost v potaz, protože její neznalost může vést k problémům při porozumění chování programu.

Příklady k procvičení

1. **Jaký je výsledek tohoto kódu v jazyce Python? Nakreslete strom vyhodnocení výrazu.**

```
a = 10
b = 20
c = a + b
```

2. **Jaký je výsledek tohoto kódu v jazyce Python? Nakreslete strom vyhodnocení výrazu.**

```
a = "10"
b = "20"
c = a + b
```

3. **Jaký je výsledek tohoto kódu v jazyce Python? Nakreslete strom vyhodnocení výrazu.**

```
a = "10"
b = 20
c = a + b
```

4. **Jaký je výsledek tohoto kódu v jazyce Python? Nakreslete strom vyhodnocení výrazu.**

```
a = "10"
b = 20
c = int(a) + b
```

5. **Jaký je výsledek tohoto kódu v jazyce Python? Nakreslete strom vyhodnocení výrazu.**

```
a = 10
b = 20
c = a + str(b)
```

6. **Jaký je výsledek tohoto kódu v jazyce Python? Nakreslete strom vyhodnocení výrazu.**

```
a = 10
b = 20
c = str(a) + str(b)
```

7. **Jaký je výsledek tohoto kódu v jazyce Python? Nakreslete strom vyhodnocení výrazu.**

```
a = 10
b = 20
c = str(a + b)
```

8. **Jaký je výsledek tohoto kódu v jazyce Python? Nakreslete strom vyhodnocení výrazu.**

```
a = 10
b = 20
c = str(int(a) + int(b))
```

9. **Jaký je výsledek tohoto kódu v jazyce Python? Nakreslete strom vyhodnocení výrazu.**

```
a = 10
b = "20"
c = a * b
```

10. **Jaký je výsledek tohoto kódu v jazyce Python? Nakreslete strom vyhodnocení výrazu.**

```
a = 10
b = "20"
c = a * int(b)
```

11. **Jaký je výsledek tohoto kódu v jazyce Python? Nakreslete strom vyhodnocení výrazu.**

```
a = "10"
b = "20"
c = a * b
```

12. **Jaký je výsledek tohoto kódu v jazyce Python? Nakreslete strom vyhodnocení výrazu.**

```
a = "10"
b = 20
c = a * b
```

13. **Jaký je výsledek tohoto kódu v jazyce JavaScript? Nakreslete strom vyhodnocení výrazu.**

```
var a = 10
var b = 20
var c = a + b
```

14. **Jaký je výsledek tohoto kódu v jazyce JavaScript? Nakreslete strom vyhodnocení výrazu.**

```
var a = "10"
var b = "20"
var c = a + b
```

15. Jaký je výsledek tohoto kódu v jazyce JavaScript? Nakreslete strom vyhodnocení výrazu.

```
1    var a = "10"
2    var b = 20
3    var c = a + b
```

16. Jaký je výsledek tohoto kódu v jazyce JavaScript? Nakreslete strom vyhodnocení výrazu.

```
1    var a = "10";
2    var b = 20;
3    var c = parseInt(a) + b;
```

17. Jaký je výsledek tohoto kódu v jazyce JavaScript? Nakreslete strom vyhodnocení výrazu.

```
1    var a = 10;
2    var b = 20;
3    var c = a + b.toString();
```

18. Jaký je výsledek tohoto kódu v jazyce JavaScript? Nakreslete strom vyhodnocení výrazu.

```
1    var a = 10
2    var b = 20
3    var c = a.toString() + b.toString()
```

19. Jaký je výsledek tohoto kódu v jazyce JavaScript? Nakreslete strom vyhodnocení výrazu.

```
1    var a = 10
2    var b = 20
3    var c = (a + b).toString()
```

20. Jaký je výsledek tohoto kódu v jazyce JavaScript? Nakreslete strom vyhodnocení výrazu.

```
1    var a = 10
2    var b = 20
3    var c = (parseInt(a) + parseInt(b)).toString()
```

21. Jaký je výsledek tohoto kódu v jazyce JavaScript? Nakreslete strom vyhodnocení výrazu.

```
1    var a = 10
2    var b = "20"
3    var c = a * b
```

22. Jaký je výsledek tohoto kódu v jazyce JavaScript? Nakreslete strom vyhodnocení výrazu.

```
1    var a = 10
2    var b = "20"
3    var c = a * parseInt(b)
```

23. Jaký je výsledek tohoto kódu v jazyce JavaScript? Nakreslete strom vyhodnocení výrazu.

```
1    var a = "10"
2    var b = "20"
3    var c = a * b
```

24. Jaký je výsledek tohoto kódu v jazyce JavaScript? Nakreslete strom vyhodnocení výrazu.

```
1    var a = "10"
2    var b = 20
3    var c = a * b
```

25. Napište příkaz, který v jazyce Python vypíše hodnotu 42 nacházející se v kolekci (proměnné) *data*.

```
1    data = {"A": 42, "B":10}
```

26. Napište příkaz, který v jazyce Python vypíše hodnotu 42 nacházející se v kolekci (proměnné) *data*.

```
1    data = [72, 42, 10]
```

27. Napište příkaz, který v jazyce Python vypíše hodnotu 42 nacházející se v kolekci (proměnné) *data*.

```
1    data = [72, {"A":42, "B":10}]
```

28. Napište příkaz, který v jazyce Python vypíše hodnotu 42 nacházející se v kolekci (proměnné) *data*.

```
1    data = {"A":72,"B":[42, 10]}
```

29. Napište příkaz, který v jazyce Python vypíše hodnotu 42 nacházející se v kolekci (proměnné) *data*.

```
1    data = ["AHOJ", 73, True, {"A":[42,7], "B":34}]
```

30. Napište příkaz, který v jazyce Python vypíše hodnotu 42 nacházející se v kolekci (proměnné) *data*.

```
1    data = ["A", {"A":3, "B": {"A":[{"ANS":42},7]}, "C":10}]
```

31. Napište příkaz, který v jazyce Python vypíše hodnotu 42 nacházející se v kolekci (proměnné) *data*.

```
data = [[[[[42]]]]]
```

32. Napište příkaz, který v jazyce Python vypíše hodnotu 42 nacházející se v kolekci (proměnné) *data*.

```
data = [[[10, [[42]]]]]
```

33. Napište příkaz, který v jazyce Python vypíše hodnotu 42 nacházející se v kolekci (proměnné) *data*.

```
data = {"A":{"A":{"A":{"A":{"A":42}}}}}
```

34. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu hodnota?

```
a = [10, 17]
b = a
b = 3
```

35. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu hodnota?

```
a = {"A":10, "B":17}
b = a
b = 3
```

36. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu hodnota?

```
a = {"A":10, "B":17}
b = a
b["A"] = 3
```

37. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu hodnota?

```
a = [10, 17]
b = a
b[0] = 3
```

38. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu hodnota.

```
a = {"A":[10, 17], "B":42}
b = a
b["A"] = 3
```

39. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu hodnota?

```
a = {"A":[10, 17], "B":42}
b = a
b["A"][1] = 3
```

40. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu hodnota?

```
a = [10, 17, [0, 2, {"A":3}, "A"], {"B":42, "C":[2, 3]}]
b = a
b[2][2] = 3
```

41. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu hodnota?

```
a = [10, 17, [0, 2, {"A":3}, "A"], {"B":42, "C":[2, 3]}]
b = a
b[2][2]["A"] = 42
```

42. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu hodnota?

```
a = [10, 17, [0, 2, {"A":3}, "A"], {"B":42, "C":[2, 3]}]
b = a
b[2] = 42
```

43. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu hodnota?

```
a = [10, 17, [0, 2, {"A":3}, "A"], {"B":42, "C":[2, 3]}]
b = a
b[3]["C"] = 42
```

44. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu reference?

```
a = {"A":10, "B":17}
b = a
b["A"] = 3
```

45. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu reference?

```
a = [10, 17]
b = a
b[0] = 3
```

46. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu reference?

```
a = {"A":[10, 17], "B":42}
b = a
b["A"] = 3
```

47. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu reference?

```
a = {"A":[10, 17], "B":42}
b = a
b["A"][1] = 3
```

48. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu reference?

```
a = [10, 17, [0, 2, {"A":3}, "A"], {"B":42, "C":[2, 3]}]
b = a
b[2][2] = 3
```

49. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu reference?

```
a = [10, 17, [0, 2, {"A":3}, "A"], {"B":42, "C":[2, 3]}]
b = a
b[2][2]["A"] = 42
```

50. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu reference?

```
a = [10, 17, [0, 2, {"A":3}, "A"], {"B":42, "C":[2, 3]}]
b = a
b[2] = 42
```

51. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu reference?

```
a = [10, 17, [0, 2, {"A":3}, "A"], {"B":42, "C":[2, 3]}]
b = a
b[3]["C"] = 42
```

52. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné hybridního typu hodnota/reference (např. v Pythonu nebo JavaScriptu)?

```
a = {"A":10, "B":17}
b = a
b["A"] = 3
```

53. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné typu reference?

```
a = [10, 17]
b = a
b[0] = 3
```

54. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné hybridního typu hodnota/reference (např. v Pythonu nebo JavaScriptu)?

```
a = {"A":[10, 17], "B":42}
b = a
b["A"] = 3
```

55. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné hybridního typu hodnota/reference (např. v Pythonu nebo JavaScriptu)?

```
a = {"A":[10, 17], "B":42}
b = a
b["A"][1] = 3
```

56. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné hybridního typu hodnota/reference (např. v Pythonu nebo JavaScriptu)?

```
a = [10, 17, [0, 2, {"A":3}, "A"], {"B":42, "C":[2, 3]}]
b = a
b[2][2] = 3
```

57. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné hybridního typu hodnota/reference (např. v Pythonu nebo JavaScriptu)?

```
a = [10, 17, [0, 2, {"A":3}, "A"], {"B":42, "C":[2, 3]}]
b = a
b[2][2]["A"] = 42
```

58. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné hybridního typu hodnota/reference (např. v Pythonu nebo JavaScriptu)?

```
a = [10, 17, [0, 2, {"A":3}, "A"], {"B":42, "C":[2, 3]}]
b = a
b[2] = 42
```

59. Jaké jsou hodnoty proměnných a a b po provedení následujícího kódu, jedná-li se o proměnné hybridního typu hodnota/reference (např. v Pythonu nebo JavaScriptu)?

```
    a = [10, 17, [0, 2, {"A":3}, "A"], {"B":42, "C":[2, 3]}]
    b = a
    b[3]["C"] = 42
```

3. Podmíněné příkazy a bloky

3.1. Podmíněné příkazy a větvení programu

3.1.1. Princip podmínek

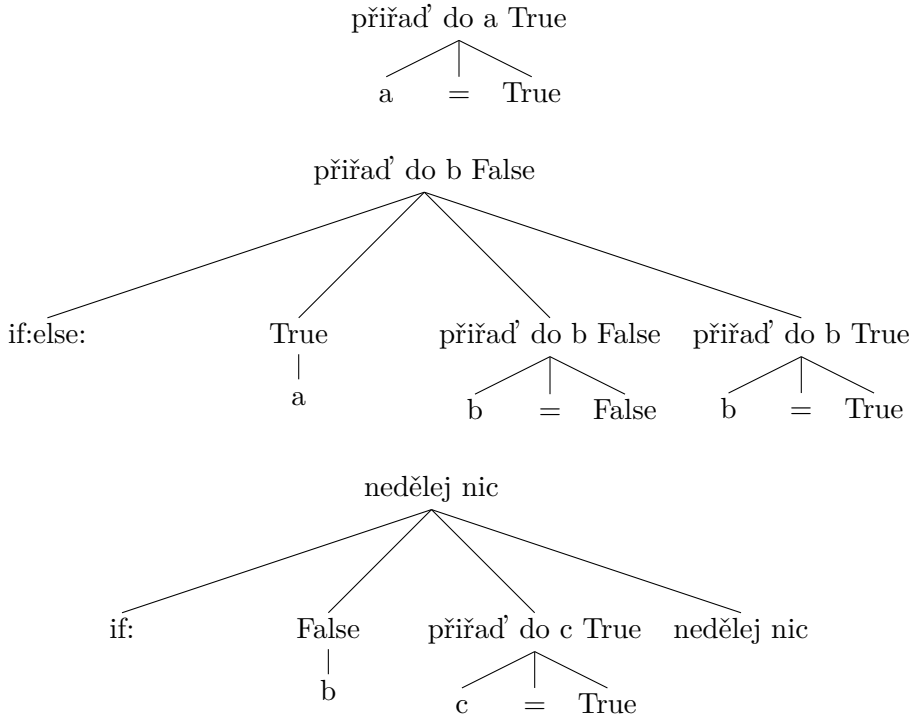
Proměnné nám umožňují popsat obecně vzorce a postupy, které jsou vždy v principu stejné, bez ohledu na zadané hodnoty. To nám však pro popis obvykle nestačí. Popisujeme-li nějaký postup, často chceme, aby se nějakým způsobem choval v jedné situaci a jiným způsobem v situaci druhé. Můžeme si to představit jako jakési větvení programu a samotné rozhodnutí, kterou cestou (větví) se má program vydat, si můžeme představit jako rozcestí. A právě toto rozcestí popisujeme pomocí podmínek.

Podmínku si můžeme představit jako speciální operaci, která na rozdíl od většiny ostatních operací nemá jeden nebo dva parametry, ale má parametry tři, přičemž třetí parametr je volitelný. Protože příkaz + má dva operandy a příkaz jako takový se píše mezi ně, potřebujeme na oddělení parametrů pouze jedno klíčové slovo/znak, a to samotné +. Protože se operátor nachází mezi operandy, říká se této formě zápisu příkazu **in**fixový zápis. Nachází-li se operátor před operandy, jedná se o zápis **pre**fixový, u operátoru za operandy jde o zápis **post**fixový. Podmínky nám stejně jako jednoduché operace umožňují zápis infixový, a to pomocí dvojice znaků ? a :, které oddělují jednotlivé operandy, vzhledem k povaze podmínek je však tato forma používána jen ojediněle pro velice jednoduché podmínky a obvykle se používá forma prefixová.

Protože v prefixové formě se nachází operátor před operandy, potřebujeme o jeden znak/klíčové slovo více. Obvykle se používá trojice if, závorky nebo dvojtečka a $else$. Podoba prefixové formy se v jednotlivých jazycích liší. Zatímco Python používá formu $if\ parametr1 : parametr2\ else : parametr3$, JavaScript používá obvyklejší formu zápisu $if(parametr1)\{parametr2\}\ else\{parametr3\}$. V obou případech se pro přehlednost obvykle druhý a třetí parametr píší na samostatné řádky odsazené tabulátorem. Zároveň, jak bylo již zmíněno, platí, že třetí parametr je nepovinný (a pokud jej nechceme využít, nepíšeme ani příkaz $else$).

Zatím jsme si popsali formu příkazu, to nám ale stále neříká nic o tom, jak příkaz funguje. Obecně můžeme říci, že první parametr je samotná rozhodovací podmínka a zbylé dva parametry jsou ony cesty, na které se nám program rozvětvuje. První parametr má tedy formu logického výrazu, to znamená výraz, který se nám vyhodnotí vždy buď na hodnotu $True$ (v JavaScriptu $true$), nebo $False$ (v JavaScriptu $false$). To, jak takové výrazy vypadají, si ukážeme v 3.1.2, nyní budeme předpokládat pouze výsledek $True/False$. Pokud má první parametr hodnotu $True$, provedeme příkaz daný druhým parametrem, pokud má naopak hodnotu $False$, provedeme příkaz daný třetím parametrem.

Z tohoto nám vyplývají dva důležité poznatky. Tím prvním je, že druhý a třetí parametr mají formu **libovolného** příkazu (nebo, jak si ukážeme v 3.1.3, více příkazů), tedy například přiřazení hodnotě proměnné. Druhým poznatkem je, že zatímco pokud nepovinný třetí parametr uvedeme, jedná se o větvení, tedy proveď dle podmínky buď jeden, nebo druhý příkaz, neuvedeme-li jej, podmínka je interpretována tak, že pokud platí (je rovna True), příkaz se provede, jinak ne.




```
1           a = True
2           if a:
3               b = False
4           else:
5               b = True
6           if b:
7               c = True
```

Obrázek 3.1: Podmíněný příkaz v Pythonu

Příklad interpretace jednoduchých podmínek a způsob jejich zápisu v jazyce Python ukazuje Obrázek 3.1. Všimněte si, že v interpretaci vidíme, že vyhodnocení příkazů je závislé na vyhodnocení předchozích příkazů.[7]

3.1.2. Logické výrazy

To, zda se podmínka vyhodnotí jako *True* nebo jako *False*, můžeme řídit pomocí tzv. logických výrazů. V zásadě jde o výrokovou, respektive predikátovou logiku, tak jak ji můžete znát z matematické logiky. Zároveň zde platí, že princip vyhodnocování je stejný jako u výrazů zmíněných v 1.3.2, s tím rozdílem, že požadujeme, aby výsledkem výrazu byla vždy hodnota True nebo False. Tyto dvě hodnoty dohromady tvoří datový typ,

[7] řádky 2-5 a řádky 6-7 považujeme za jeden složený příkaz, podmínku

který se obvykle nazývá *bool* nebo *boolean* (řídí se pravidly tzv. booleovské/Booleovy algebry).

Pro získání těchto hodnot z hodnot textových nebo číselných, případně i z kolekcí, můžeme použít celou řadu operací. Obecně se jedná o operace porovnávací, tj. ==[8], ! =[9], <=, >=, <, >, které vrací $True$, pokud daná (ne)rovnost platí, a $False$, pokud ne. Další možností (v Pythonu) jsou operace obsažení *in* a *not in*, které slouží k zjištění, zda námi požadovaný prvek je, respektive není v námi dané kolekci. Pravdivostní hodnotu mohou vracet rovněž funkce, ale o těch později. Pravdivostní hodnoty mohou být (jak jsme viděli na Obrázku 3.1) uloženy i v proměnných.

V neposlední řadě pak máme k dispozici tři základní logické operace, tj. konjunkci, disjunkci (nebo chcete-li logické spojky *a*, *nebo*) a negaci, tak jak jsou definovány ve výrokové logice. Ty se používají ke spojování logických výrazů do výrazů složených. Ve většině běžných programovacích jazyků (včetně JavaScriptu) je zapisujeme jako &&, || a !, oproti tomu Python volí názornější (ale delší) zápis *and*, *or* a *not*. Vyhodnocování všech třech operací uvádí Tabulky 2a, 2b a 2c.

Tabulka 1: Tabulky vyhodnocování logických operací

a	*not a*
False	True
True	False

(a) Negace

a	b	*a and b*
False	False	False
False	True	False
True	False	False
True	True	True

(b) Konjunkce

a	b	*a or b*
False	False	False
False	True	True
True	False	True
True	True	True

(c) Disjunkce

Z hlediska priority mají všechny porovnávací operace nižší prioritu než operace matematické, tedy pokud výrazy explicitně neuzávorkujete, vyhodnocují se později. Ještě nižší prioritu mají operace logické, přičemž z nich má nejvyšší prioritu negace, následuje konjunkce (*and*, občas rovněž nazývána logický součin) a nejnižší prioritu má disjunkce (*or*, též logický součet). Pro *and* a *or* zároveň platí pravidlo tzv. zkráceného vyhodnocování. To znamená pokud je podle prvního operandu zřejmý výsledek celého výrazu, druhý operand se nevyhodnocuje. Prakticky to tedy znamená, že pokud má první operand operace *or* hodnotu True, druhý operand se nevyhodnocuje a naopak má-li první operand operace *and* hodnotu False, druhý operand se taktéž nevyhodnocuje.

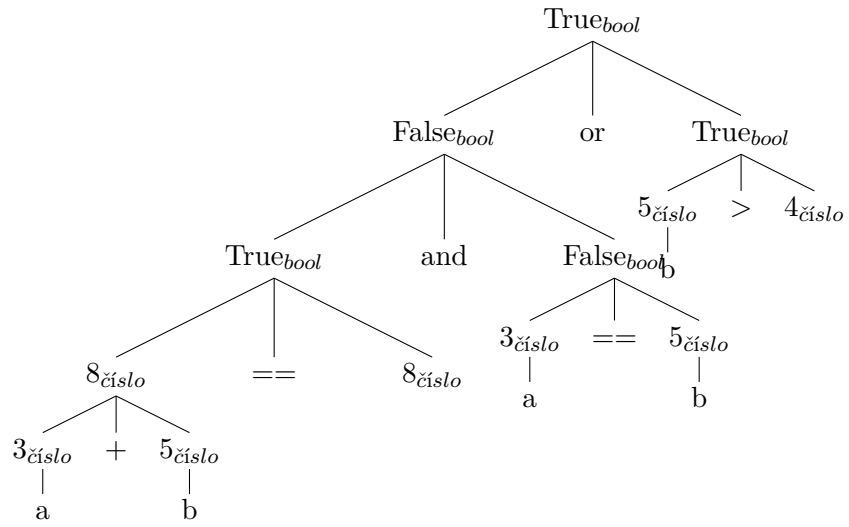
Ukázku vyhodnocení složeného logického výrazu ukazuje Obrázek 3.2, a aplikaci logického výrazu v podmínce pak Obrázek 3.4. V obou výrazech předpokládáme hodnoty proměnných $a = 3$ a $b = 5$.

3.1.3. Vícenásobné podmínky a větvení

V 3.1.1 jsem zmínil, že parametrem podmínky může být jakýkoliv příkaz. Z toho vyplývá, že tímto příkazem může být i jiná podmínka.

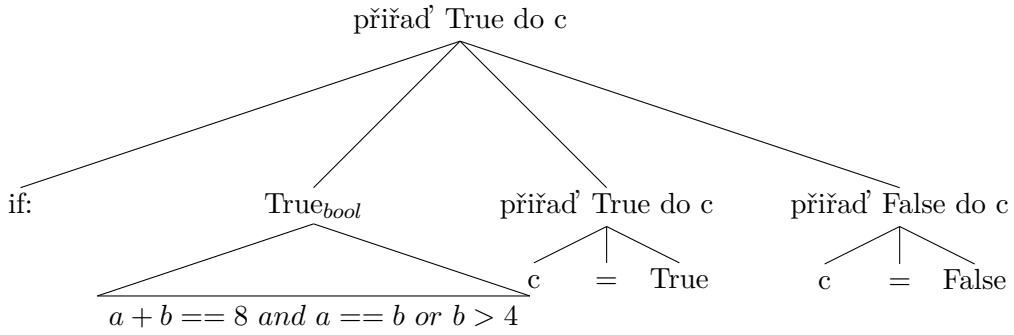
[8] operátor rovnosti == se používá pro odlišení od operace přiřazení, tj. =

[9] operátor nerovnosti



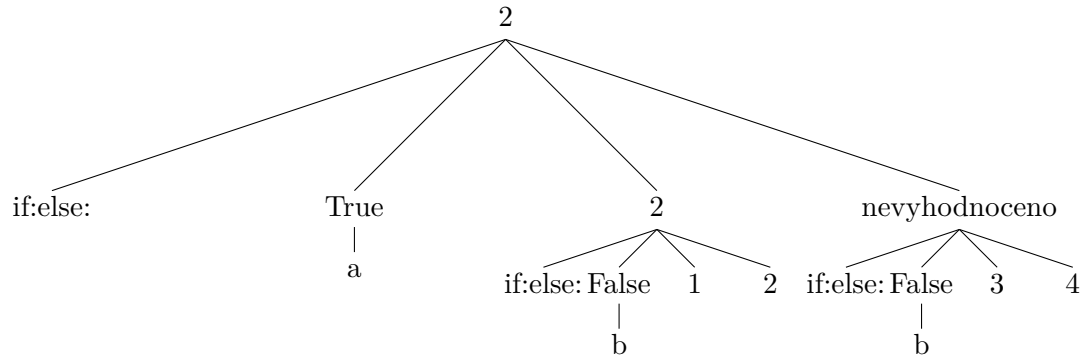

```
a + b == 8 and a == b or b > 4
```

Obrázek 3.2: Složený logický výraz




```
if a + b == 8 and a == b or b > 4:
    c = True
else:
    c = False
```

Obrázek 3.3: Složený logický výraz v podmínce




```
1          if a:
2              if b:
3                  print(1)     #1
4              else:
5                  print(2)     #2
6          else:
7              if b:
8                  print(3)     #3
9              else:
10                 print(4)     #4
```

Obrázek 3.4: Vnořené podmínky

Právě tato vlastnost nám umožňuje vytvářet vícenásobné větvení. Jak takové obecné vícenásobné větvení může vypadat ukazuje Obrázek 3.4[10]. Uvažujeme následující hodnoty proměnných: $a = True, b = False$.

Samozřejmě vnořená podmínka může opět obsahovat podmínku, ne všechny podmínky musí mít třetí parametr, tj. část *else* atd. Jistě vidíte, že tento zápis je značně nepřehledný, a tak pro speciální případ větvení, kdy všechny zanořené podmínky (s výjimkou poslední, nejhlouběji zanořené) mají jako třetí parametr další podmínku, vznikl zkrácený zápis pomocí klíčového slova elif[11]. Plně rozepsanou i zkrácenou formu tohoto vícenásobného větvení ukazuje Obrázek 3.5. Hodnoty proměnných jsou $a = False, b = False, c = True$.

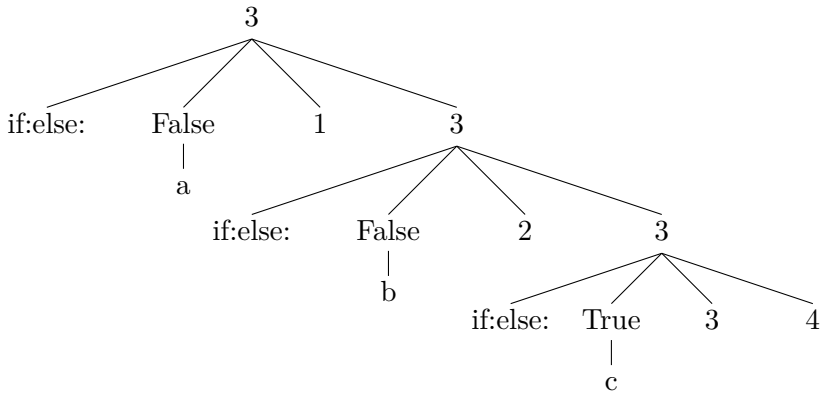
3.2. Bloky kódu

3.2.1. Blok jako nástroj substituce

Je praktické mít možnost rozhodovat na základě aktuálního stavu (dat) o tom, který příkaz se má provést, typicky však potřebujeme za daného stavu provést více než jeden příkaz. Hodila by se nám tedy konstrukce, která by řekla, že daná skupina příkazů se vzhledem ke svému okolí chová jako jeden velký příkaz.

[10] Vyhodnocování je zde zjednodušeno tak, že v kořeni (pod)stromu je vždy příkaz, který byl zvolen podmínkou, příkazy zde nejsou uvedeny a jsou zastoupeny čísly.

[11] V jiných jazycích, např. JavaScriptu, se místo elif používá zápis elseif, případně else if




```
        if a:
            print(1)     #1
        else:
            if b:
                print(2)     #2
            else:
                if c:
                    print(3)     #3
                else:
                    print(4)     #4
```

```
        if a:
            print(1)     #1
        elif b:
            print(2)     #2
        elif c:
            print(3)     #3
        else:
            print(4)     #4
```

Obrázek 3.5: Lineárně vnořené podmínky – elif

To, co potřebujeme, je tedy jakási substituce, kdy několik příkazů substituujeme za jeden příkaz, jehož provedení znamená provést postupně všechny příkazy, které jsou jím substituovány.

Takováto konstrukce ve většině moderních jazyků existuje a obvykle se jí říká blok kódu, v angličtině se rovněž používá název scope.

Bloky mohou být buď izolované (typické u funkcí a objektů, o kterých si povíme později), nebo neizolované.

Izolované bloky mají tu vlastnost, že veškeré proměnné, které v rámci bloku vytvoříme, mimo něj zanikají, a zároveň to, že pokud chceme některou hodnotu existující mimo blok použít uvnitř bloku, musíme to programu explicitně sdělit.

Oproti tomu neizolované bloky mají přístup ke všem hodnotám existujícím mimo blok (pokud jsou dostupné v bloku, uvnitř kterého se blok nachází). Někdy (dle jazyka) pak platí, že proměnné vytvořené uvnitř bloku existují i vně bloku, a někdy tyto proměnné mimo blok zanikají. Pro Python je typické, že proměnné existují dál, zatímco pro JavaScript nikoliv.

Pro podmínky jsou relevantní pouze bloky neizolované, izolované bloky tedy v tuto chvíli nebudeme brát v potaz. Neizolované bloky nám totiž umožňují právě onu substituci více příkazů za příkaz jeden. Vše, co se nachází uvnitř bloku, je totiž vně něj považováno za jeden příkaz.

3.2.2. Bloky kódu v různých programovacích jazycích

Zbývá nám tedy otázka, jak blok kódu vytvořit. Napříč jazyky se vyskytují v podstatě tři možnosti.

Tou nejobvyklejší je označení bloku, respektive jeho uzavření do složených závorek. Tedy vše, co se nachází uvnitř složených závorek, je považováno za součást jednoho bloku, přičemž tyto bloky lze zanořovat. Tento způsob používá například JavaScript, Java a celá řada používaných jazyků (C, C++, C, PHP, ...). Pro přehlednost obvykle platí, že obsah každého zanořeného bloku je navíc vzhledem k okolí odsazen (typicky tabulátorem), závorky jsou však stále nezbytné a odsazení je nepovinné.

Tímto nepsaným pravidlem se inspiroval Python, ten používá odsazení jako označení bloku. Narozdíl od obvyklého přístupu však nepoužívá složené závorky a toto odsazení je povinné.

Třetí způsob je nejstarší a dnes už se používá okrajově. Od prvního způsobu se lisí pouze tím, že místo otevírací a zavírací složené závorky používá klíčová slova *begin* a *end*, rovněž s volitelným odsazením.

```
1  ...
2  {
3      var c = 10;
4      console.log(c);
5      {
6          var a = 20;
7          console.log(a);
8      }
9      var d = 30;
10     console.log(d);
11 }
12 ...
13
14 ...
15     c = 10
16     print(c)
17         a = 20
18         print(a)
19     d = 30
20     print(d)
21 ...
22
23 ...
24 begin
25     c = 10
26     print(c)
27     begin
28         a = 20
29         print(a)
30     end
31     d = 30
32     print(d)
33 end
34 ...
```

Kód 8

Jakým způsobem se liší jednotlivé přístupy nám (v různých jazycích) ukazuje Kód 8.[12] Je dobré také poznamenat, že použití bloků bez příkazu typu *if* (případně jiného, který pracuje s následujícím příkazem jako svým parametrem) nám nijak nemění chování programu, protože příkazy se vyhodnotí jeden po druhém, stejně jako bez použití bloků. Bloky mají smysl jen pokud potřebujeme sdružit příkazy vzhledem k jinému příkazu.

[12]Na řádcích 1–12 je varianta zápisu v JavaScriptu, na řádcích 14-21 ukázka téhož v Pythonu a na řádcích 23–34 ukázka v Ruby; ... reprezentuje okolní kód.
Všimněte si také, že JavaScript používá pro oddělení příkazů středníky (odřádkování je pouze pro přehlednost), ačkoliv umožňuje použít místo nich rovněž odřádkování, zatímco Python i Ruby používají pro oddělení příkazů povinné odřádkování.

Příklady k procvičení

1. **Jaká je hodnota proměnné c? $True$, nebo $False$?**

```
a = 5
b = 10
c = a > 5 and b < 12 or b == a
```

2. **Jaká je hodnota proměnné c? $True$, nebo $False$?**

```
a = 5
b = 10
c = a >= 5 and (b < 12 or b == a)
```

3. **Jaká je hodnota proměnné c? $True$, nebo $False$?**

```
a = 5
b = 10
c = a >= 5 and b < 12 and b == a
```

4. **Jaká je hodnota proměnné c? $True$, nebo $False$?**

```
a = 5
b = 10
c = (a + b > 10 or a != b) and b - a < 3
```

5. **Jaká je hodnota proměnné c? $True$, nebo $False$?**

```
a = 5
b = 10
c = a + b > 10 or a != b and b - a < 3
```

6. **Jaká je hodnota proměnné c? $True$, nebo $False$?**

```
a = 5
b = 10
c = a**2 == a + 2*b and (b - a) * 2 == b
```

7. **Zvolte hodnoty proměnných a a b tak, aby platila rovnost na druhém řádku.**

```
c = a + b == 10 and (b == 5 or b == 3)
c == True
```

8. **Zvolte hodnoty proměnných a a b tak, aby platila rovnost na druhém řádku.**

```
c = a and not b or b and a or b
c == True
```

9. Zvolte hodnoty proměnných a a b tak, aby platila rovnost na druhém řádku.

```
c = a and not b or b and a or b
c == False
```

10. Zvolte hodnoty proměnných a a b tak, aby platila rovnost na druhém řádku.

```
c = a and b and not b or (a or not a)
c == True
```

11. Zvolte hodnoty proměnných a a b tak, aby platila rovnost na druhém řádku.

```
c = a + 2*b == 7 and b/2 - 3 == 15
c == True
```

12. Zvolte hodnoty proměnných a a b tak, aby platila rovnost na druhém řádku.

```
c = a + 2*b == 7 and b/2 - 3 == 15
c == False
```

13. Zvolte hodnoty proměnných a a b tak, aby platila rovnost na druhém řádku.

```
c = a and b and not b or (a or not a)
c == False
```

14. Co vypíše následující kód?

```
a = 2
b = 3
if a + b > 5:
    print("baf")
else:
    print("lek")
```

15. Co vypíše následující kód?

```
a = 9
b = 6
if a == 7 or b >= 5:
    print("abraka")
if b < 12:
    print("dabra")
```

16. Co vypíše následující kód?

```
a = 4
b = 7
c = 12
if a + b < c:
    print("lower")
elif a + b == c:
    print("equal")
else:
    print("higher")
```

17. Co vypíše následující kód?

```
a = 2
b = 5
if a*2 > b:
    b = b + 1
    if b == 7:
        print("7")
    elif b == 10:
        print("10")
elif b*2 > a:
    a = a + 1
    if a == 12:
        print("12")
    else:
        print(a)
```

18. Co vypíše následující kód?

```
a = 42
if a \% 4 == 0:
    print("4k")
elif a \% 4 == 1:
    print("4k + 1")
elif a \% 4 == 2:
    print("4k + 2")
else:
    print("4k + 3")
```

19. Mějme tři proměnné a, b a *operation*, přičemž poslední jmenovaná může nabývat hodnot "$+$", "$-$", "$*$" a "$/$". Napište program, který se bude chovat jako jednoduchá kalkulačka vypíše výsledek dle hodnot daných proměnných.

20. Napište program, který má jako vstup tři proměnné a, b a c odpovídající délkám strany trojúhelníku. Napište program, který vypíše "*Ano*", pokud se může jednat o trojúhelník (platí trojúhelníková nerovnost), nebo "*Ne*", pokud neplatí.

21. Napište program, který vypíše 10 znaků +, pokud je číslo v proměnné a sudé a 10 znaků −, pokud je liché.

22. Mějme proměnnou $track\_number$ odpovídají aktuálnímu číslu skladby, proměnnou $track\_count$ odpovídající počtu skladeb v playlistu a proměnnou $repeat$ mající hodnotu $True$ nebo $False$ podle toho, zda chceme po přehrání poslední skladby pokračovat od začátku. Napište program odpovídající situaci po přehrání skladby s pořadím odpovídajícím hodnotě proměnné $track\_number$. Pokud se nejedná o poslední skladbu, přejděte na následující skladbu a vypište, kolikátou skladbu nyní přehráváte, pokud jste na konci, vypište při vypnutém opakování, že přehrávání skončilo, v opačném případě přejděte na první skladbu (opět vypište pořadí).

23. Mějme proměnnou $track\_number$ odpovídají aktuálnímu číslu skladby, proměnnou $tracks$, což je seznam názvů skladeb v playlistu a proměnnou $repeat$ mající hodnotu $True$ nebo $False$ podle toho, zda chceme po přehrání poslední skladby pokračovat od začátku. Napište program odpovídající situaci po přehrání skladby s pořadím odpovídajícím hodnotě proměnné $track\_number$. Pokud se nejedná o poslední skladbu, přejděte na následující skladbu a vypište název nově přehrávané skladby, pokud jste na konci, vypište při vypnutém opakování, že přehrávání skončilo, v opačném případě přejděte na první skladbu (opět vypište název).

24. Mějme proměnnou $player\_state$, která odpovídá aktuálnímu stavu přehrávače a nabývá hodnot "$playing$" a "$paused$". Napište program odpovídající části obsluhy tlačítka play/pause, která se stará o aktuální stav přehrávače. Tedy stisknete-li tlačítko při přehrávání přejde do stavu pozastaveno, v opačném případě přejde do stavu přehrávání.

4. Cykly

4.1. Obecné cykly s podmíněným opakováním

4.1.1. Podmíněné a nepodmíněné skoky v programu

Zmínil jsem, že počítač ve skutečnosti rozumí pouze elementárním příkazům a příkazy, které používáme v programovacích jazycích, nás od tohoto faktu pouze odstiňují. Abychom lépe porozuměli tomu, jak fungují v programovacích jazycích konstrukty, kterým se obecně říká cykly, bude se nám hodit pochopit, jak pracuje jeden typ těchto elementárních, a to tzv. skoky.

Skokem se rozumí příkaz, který způsobí to, že místo toho, abychom pokračovali následující instrukcí (což je výchozí chování při běhu aplikace), skočíme na námi definovanou instrukci (řádek kódu).

Existují dva typy skoků, přičemž s tím prvním jsme se už setkali, jedná se totiž o podmíněné skoky, a podmínky, o kterých pojednává předchozí kapitola, jsou příkladem právě tohoto typu skoků. Podmíněné skoky se chovají tak, že pokud je splněna námi definovaná podmínka, skočí na definované místo v programu, pokud ne, pokračují dalším příkazem, jako by zde žádný skok nebyl. Podmínka je tedy podmíněný skok, který ve skutečnosti vyhodnocuje opačnou podmínku, než má danou, a pokud je tato opačná podmínka splněna, tak program skočí buď na začátek bloku else (je-li přítomen), nebo za podmínku, pokud není opačná podmínka splněna, pokračuje následujícím příkazem, tj. blokem uvnitř podmínky.

Nás ovšem ve vztahu k cyklům bude zajímat především druhý typ skoků, kterým jsou skoky nepodmíněné. Ty, jak název napovídá, žádnou podmínku nemají a jakmile jsou vyhodnoceny, příkaz jednoduše skočí na dané místo (řádek, instrukci).

Právě pomocí nepodmíněných skoků si můžeme ukázat, jak vlastně fungují cykly.

4.1.2. Princip cyklu

Cykly jako takové můžeme popsat jako opakování určité posloupnosti příkazů do té doby, dokud je splněna námi daná podmínka. Obecný cyklus, který takovéto definici nejlépe odpovídá, se ve většině moderních programovacích jazyků nazývá *while*, což je rovněž klíčové slovo obvykle používané k jeho zápisu.

Zápis cyklu *while* je typicky velice podobný zápisu podmínky, s tím rozdílem, že zde není obdoba bloku *else* a namísto klíčového slova *if* používá klíčové slovo *while*. Příklad tohoto cyklu v jazycích Python a JavaScript ukazují Kódy 9 a 10.

```
1 a = 5
2 while a > 0:
3     print(a)
4     a = a - 1
```

Kód 9

```
1 var a = 5;
2 while(a > 0) {
3     console.log(a);
4     a = a - 1;
5 }
```

Kód 10

Oba kódy nám vypíší postupně čísla 5, 4, 3, 2 a 1. Rozebereme-li si tento kód, vidíme, že dokud platí, že hodnota proměnné a je větší než nula, opakují se nám dva příkazy, a to výpis hodnoty proměnné a následné snížení hodnoty proměnné o jedna.

Pokud využijeme výše zmíněný nepodmíněný skok, můžeme reprezentovat cyklus *while* jako podmínku. Jak by tato reprezentace vypadala v Pythonu, ukazuje Kód 11.[13]

```
1 a = 5
2 if a > 0:
3     print(a)
4     a = a - 1
5     jmp(2)
```

Kód 11

Z tohoto kódu vidíme, že cyklus jako takový se od podmínky liší v tom, že na konci bloku kódu, který se při splnění podmínky vykoná, se nachází skok na první řádek tohoto cyklu, tedy na vyhodnocení podmínky. Po dokončení bloku se nám tedy znovu vyhodnocuje podmínka a v případě jejího opětovného splnění se kód vykoná znovu. Cyklus se zastaví v okamžiku, kdy není splněna podmínka, protože v tu chvíli se přeskočí obsažený blok, a to včetně nepodmíněného skoku na jeho konci.

4.1.3. Závislost příkazů na kontextu – stejný kód, jiný stav

Takto definované cykly mají jednu důležitou vlastnost, opakovaně provádí stejnou posloupnost příkazů, ale mohou při každém opakování docházet (a typicky také dochází) k jiným výsledkům.

Nabízí se otázka, jak je možné dojít stejnou posloupností instrukcí k jinému výsledku. Odpovědí je kontext. Při každém vykonání instrukce se nám totiž mění stav (to, co je uloženo v paměti) a výsledek příkazů je závislý na stavu. V našich příkladech je tímto stavem hodnota proměnné a, tento stav je pak měněn postupným snižováním této hodnoty, kdybychom v cyklu stav nijak neměnili, nemohl by cyklus nikdy skončit, jelikož pokud by byl stav stejný jako v okamžiku prvního spuštění cyklu, byla by zákonitě vždy splněna podmínka opakování. V našem případě by toto bylo způsobeno odstraněním příkazu měnícího hodnotu a, což by mělo za následek neustálé vypisování hodnoty 5, a to až do násilného ukončení běhu programu, jelikož program sám o sobě by ukončení nebyl schopen.

Pokud před sebou máme kód obsahující složitější cykly, můžeme si dopomoci k porozumění tak, že si dosadíme hodnoty do proměnných. Následně si rozepíšeme cykly a

[13] Tento kód v Pythonu nespustíte, protože příkaz jmp v něm neexistuje, jedná se o zápis reprezentující nepodmíněný skok na řádek 2 a slouží pouze pro účely demonstrování principu.

podmínky na skutečně provedenou posloupnost příkazů. Podmínky přepisujeme jednoduchým pravidlem, je-li podmínka splněna, připíšeme její blok příkazů if, není-li splněna, připíšeme blok *else*, je-li přítomen, v opačném případě nepřipíšeme nic a pokračujeme dalším příkazem. Cykly přepisujeme tak, jako bychom znovu a znovu vyhodnocovali podmínky, a tímto vyhodnocováním končíme v okamžiku, kdy podmínka není poprvé splněna, poté pokračujeme dalším příkazem.

Postupně rozepsanou formu Kódu 9, respektive Kódu 11, ukazuje Kód 12.

```
a = 5
if a > 0:
    print(a)
    a = a - 1
if a > 0:
    print(a)
    a = a - 1
if a > 0:
    print(a)
    a = a - 1
if a > 0:
    print(a)
    a = a - 1
if a > 0:
    print(a)
    a = a - 1
if a > 0:
    print(a)
    a = a - 1

#vyhodnoceno
a = 5
print(5)
a = 5 - 1
print(4)
a = 4 - 1
print(3)
a = 3 - 1
print(2)
a = 2 - 1
print(1)
a = 1 - 1
```

Kód 12

4.2. Cykly s pevně daným opakováním

4.2.1. Cyklus typu for

Cyklus *while* bychom mohli nazvat cyklem obecným, můžeme ho použít v podstatě ve všech situacích, které použití cyklů vyžadují, a to bez ohledu na to, zda víme dopředu,

kolikrát chceme dané operace opakovat.

Existují však i další cykly, které jsou více specializované, pro nás je zajímavý jeden, který je rovněž ve většině moderních programovacích jazyků samozřejmostí. Jedná se o cyklus *for*, případně *foreach*, přičemž cyklus *foreach* je v mnoha jazycích (např. Python a JavaScript) pouze variantou cyklu *for* a i v tomto výkladu jej tak budeme popisovat.

4.2.2. Cykly pro procházení kolekcí (foreach)

V našem popisu začneme právě variantou cyklu *for*, která má v některých jazycích k dispozici vlastní příkaz, obvykle nazývaný *foreach*.

Tato varianta slouží k procházení kolekcí, a to jak uspořádaných (seznamů), tak neuspořádaných (slovníků). Cyklus využívá lokální proměnnou, do které v každé iteraci (jeden průchod cyklem) přiřadí prvek na příslušné pozici v kolekci, a s touto proměnnou v příkazech v těle[14] cyklu pracuje, přičemž máme obvykle možnost procházet buď klíče/indexy prvků, nebo hodnoty položek.

Procházení klíčů i hodnot (neuspořádané) kolekce v Pythonu a JavaScriptu ukazuje Kód 13 a 14.

```
1 a = {"a":1, "b":2, "c":3}
2
3 for x in a.values(): #1, 2, 3
4     print(x)
5
6 for x in a.keys():  #a, b, c
7     print(x)
```

Kód 13

```
1 var a = {"a":1, "b":2, "c":3};
2
3 for(var x of a) {   //1, 2, 3 (experimental)
4     console.log(x);
5 }
6
7 for(var x in a) {   //a, b, c
8     console.log(x);
9 }
```

Kód 14

Všimněte si, že zatímco Python používá stejnou syntaxi pro procházení klíčů i hodnot a mění pouze příkaz, který je volán jako parametr, JavaScript pro rozlišení používá dvě klíčová slova, pro klíče je to *in* a pro hodnoty *of*. JavaScript používá totožný přístup i pro kolekce uspořádané. Oproti tomu přístup Pythonu se lehce liší.

Podstata cyklu *for* v Pythonu je totiž procházení uspořádané kolekce, tj. seznamu, a to i v případě procházení neuspořádaných kolekcí. Z tohoto důvodu zde vidíme volání

[14] Tělem cyklu se obvykle nazývá blok příkazů, které se při každém průchodu cyklu vyhodnocují.

a.values() a *a.keys*(), jejich výsledkem je totiž seznam hodnot, respektive klíčů slovníku. Z tohoto nám vyplývá, že procházení seznamu pomocí cyklu *for* je v Pythonu poměrně přímočaré, to ukazuje Kód 15, totéž pro JavaScript ukazuje Kód 16.

```
a = [7, 8, 9]

for x in a: #7, 8, 9
    print(x)

for x in range(len(a)):  #0, 1, 2
    print(x)
```

Kód 15

```
var a = [7, 8, 9];

for(var x of a) {   //7, 8, 9 (experimental)
    console.log(x);
}

for(var x in a) {   //0, 1, 2
    console.log(x);
}
```

Kód 16

Vidíme, že zatímco kód v JavaScriptu se v podstatě nezměnil, v Pythonu došlo k několika změnám, procházení hodnot seznamu se zjednodušilo, procházení klíčů (indexů) seznamu se lehce zkomplikovalo, o příkazech *range* a *len* a o tom, proč je třeba je v tomto případě použít, si povíme později.

Abychom cyklu *for* lépe porozuměli, ukažme si ještě, jak jej lze zapsat pomocí nám již dobře známého cyklu *while*. Řešení všech čtyř úloh (klíče i hodnoty (ne)uspořádaných kolekcí) ukazuje Kód 17.[15]

```
a = {"a":1, "b":2, "c":3}

keys = a.keys() # ["a", "b", "c"]
i = 0
while i < len(keys):
    x = keys[i]
    print(x)
    i = i + 1

values = a.values() # [1, 2, 3]
i = 0
while i < len(values):
    x = values[i]
```

[15]V tomto případě již uvádíme pouze ukázky v Pythonu, v JavaScriptu bychom se řídili obdobným principem, pouze bychom syntax cyklu *while* upravili tak, aby odpovídala JavaScriptu, tj. dodali závorky, klíčové slovo *var*, nahradili funkci *keys*

```
14      print(x)
15      i = i + 1
16
17  b = [7, 8, 9]
18
19  i = 0
20  while i < len(b):
21      x = b[i]
22      print(x)
23      i = i + 1
24
25  i = 0
26  while i < len(b): # varianta 1
27      x = i
28      print(x)
29      i = i + 1
30
31  i = 0
32  indices = range(len(b))
33  while i < len(indices): # varianta 2
34      x = indices[i]
35      print(x)
36      i = i + 1
```

Kód 17

Všimněte si, že se u všech čtyř úloh opakuje totožný princip, příkaz *len* nám vrací číslo odpovídající počtu prvků dané kolekce a proměnná *i*, kterou postupně zvyšujeme, reprezentuje index prvku v poli. Vzhledem k tomu, že index posledního prvku v poli je o jedna menší než počet prvků, podmínka je ve tvaru *menší než*, a nikoliv *menší rovno*.

4.2.3. Klasická varianta cyklu for

Druhým (klasickým) použitím cyklu *for* je situace, kdy chceme zopakovat dané příkazy postupně pro jiné hodnoty proměnných, přičemž je známo o jaké hodnoty se jedná. Hodnoty jsou definovány třemi čísly, první určuje počáteční hodnotu, druhé koncovou a třetí velikost kroku, respektive může se jednat o definicí kroku jako takového. Běžný zápis takovéhoto cyklu používá například JavaScript, jak (spolu s přepisem pomocí cyklu *while*) ukazuje Kód 18.

```
1  for(var i = 0; i < 10; i = i + 2) { // 0, 2, 4, 6, 8
2      console.log(i);
3  }
4  var j = 0
5  while(j < 10) {
6      console.log(j);
7      j = j + 2;
8  }
```

Kód 18

Jak vidíte, v klasickém zápisu jsou jednotlivé části oddělené středníky, a to v pořadí počátek, horní mez a definice kroku. Oproti tomu Python, který v cyklu *for* vždy vyžaduje procházení seznamu, používá příkaz *range*. Ten má totožné parametry, s tím rozdílem, že zadáváme pouze čísla, to mimo jiné omezuje definici pouze na formu součtovou.

Jak tedy tuto funkci použít? Funkce *range* má tři prvky zapisovány následujícím způsobem *range*(*počáteční hodnota*, *horní mez*, *velikost kroku*). Tato funkce nám vygeneruje seznam obsahující právě ty prvky, které popisu budou odpovídat. Například *range*$(0, 10, 2)$, nám vygeneruje seznam obsahující prvky 0, 2, 4, 6 a 8. Vynecháme-li poslední parametr, v každém kroku se bude přičítat číslo jedna, vynecháme-li i druhý parametr, počáteční hodnota se stanoví na číslo 0. Chceme-li tedy projít postupně indexy seznamu, využijeme k tomu právě funkci *range* v kombinaci s funkcí *len*. Volání *range*(*len*(*kolekce*)) nám vrátí seznam obsahující čísla 0, 1, ..., *počet prvků* $-$ 1, tedy právě indexy dané kolekce.

Obdobu Kódu 18 v Pythonu ukazuje Kód 19.

```
1 for i in range(0, 10, 2):
2     print(i)
3
4 j = 0
5 while(j < 10):
6     print(j)
7     j = j + 2
```

Kód 19

Příklady k procvičení

1. **Co bude výstupem následujícího kódu?**

```
for i in range (5):
    if i % 2 == 0:
        print(i)
```

2. **Co bude výstupem následujícího kódu?**

```
for i in [2, 4, 6, 8]:
    print(i // 2)
```

3. **Co bude výstupem následujícího kódu?**

```
col = [1, 2, 3, 4, 5]
for i in col:
    print(i * 2)
```

4. **Co bude výstupem následujícího kódu?**

```
col = {"A":"a", "B":"b","C":"c"}
for i in col.values():
    print(i)
```

5. **Co bude výstupem následujícího kódu?**

```
col = {"A":"a", "B":"b","C":"c"}
for i in col.keys():
    print(i)
```

6. **Co bude výstupem následujícího kódu?**

```
col = {"A":"a", "B":"b","C":"c"}
for i in col.keys():
    print(i + ": " + col[i])
```

7. **Co bude výstupem následujícího kódu?**

```
mad_list = ["Adam", "Eva", "Marek", "Jana", "J  ", "Jitka", "Petr"]
for person in mad_list:
    if person == "J  ":
        print("letadlo")
    else:
        print("bl zen")
```

8. **Co bude výstupem následujícího kódu?**

```
for(var i=1; i<100; i*=2) {
    console.log(i);
}
```

9. **Co bude výstupem následujícího kódu?**

```
for(var i=0; i<100; i*=2) {
    console.log(i);
}
```

10. **Co bude výstupem následujícího kódu?**

```
text = "Hello universe!"
for i in text:
    print(i)
```

11. **Co bude výstupem následujícího kódu?**

```
text = "Hello universe!"
for i in range(0, len(text), 2):
    print(i)
```

12. **Co bude výstupem následujícího kódu?**

```
for i in range(1, 5):
    for j in range(2, 6):
        print(i + j)
```

13. **Co bude výstupem následujícího kódu?**

```
n = 15
while n > 0:
    n = n - 2
    print(n)
```

14. **Co bude výstupem následujícího kódu?**

```
n = 15
while n > 0:
    if n % 2:
        print(n)
```

15. **Co bude výstupem následujícího kódu?**

```
var n = 15;
while(n % 2 != 0) {
    console.log(n);
    n = n / 2
}
```

16. **Napište kód, který vypíše pomocí jednoho nebo více cyklů následující posloupnost:** $2, 4, 6, 8, \ldots$

17. **Napište kód, který vypíše pomocí jednoho nebo více cyklů následující posloupnost:** $2, 5, 11, 23, \ldots$

18. **Napište kód, který vypíše pomocí jednoho nebo více cyklů následující posloupnost:** $1, 4, 9, 16, \ldots$

19. **Napište kód, který vypíše pomocí jednoho nebo více cyklů následující posloupnost:** $True, False, True, False, \ldots$

20. **Napište kód, který vypíše pomocí jednoho nebo více cyklů následující posloupnost:** $True, False, False, True, False, False, True, \ldots$

21. **Napište kód, který vypíše pomocí jednoho nebo více cyklů následující posloupnost:** $1, 1, 2, 1, 2, 3, 1, 2, 3, 4, \ldots$

22. **Napište kód, který vypíše pomocí jednoho nebo více cyklů následující posloupnost:** $1, 2, 3, \ldots, 2, 3, 4, \ldots, 3, 4, 5, \ldots$

23. **Napište kód, který vypíše pomocí jednoho nebo více cyklů prvky kolekce** *col* **společně s cestami k nim.**

```
col = [[1, 3], [2, 7, 5], [8, 2, 9]]
```

24. **Napište kód, který vypíše pomocí jednoho nebo více cyklů prvky kolekce** *col* **společně s cestami k nim.**

```
col = [1, 3, 2, 7, 5, 8, 2, 9]
```

25. **Napište kód, který vypíše pomocí jednoho nebo více cyklů prvky kolekce** *col* **společně s cestami k nim.**

```
col = {"A": 1, "B": 3, "C": 2, "D": 7, "E": 5}
```

26. **Napište kód, který vypíše pomocí jednoho nebo více cyklů prvky kolekce** *col* **společně s cestami k nim.**

```
col = {"A": [1, 3], "B": [2, 7, 5], "C": [8, 2, 9]}
```

27. **Napište kód, který vypíše pomocí jednoho nebo více cyklů prvky kolekce** *col* **společně s cestami k nim.**

```
col = {"A": {"a":1, "b":3}, "B": {"a":2, "b":7, "c":5}}
```

28. **Mějme proměnnou** *tracks* **obsahující název skladeb v playlistu a proměnnou** *repeat*, **která říká, zda chceme po přehrání poslední skladby pokračovat znovu od začátku. Napište pomocí cyklu program, který bude tento playlist „přehrávat“, kdy přehráním jedné skladby se rozumí výpis jejího názvu.**

29. **Mějme seznam** *data*, **který obsahuje jako své položky slovníky, které mají přesně dva prvky – učo a kód předmětu a odpovídají zapsaným předmětům. Napište program, který vytvoří nový slovník, který bude mít jako klíče učo a jako hodnoty seznam zapsaných předmětů. Ukázkový vstup a výstup mohou vypadat následovně:**

```
#input
data = [{"uco": "421234", "code":"PLIN048"},
    {"uco": "421235", "code":"PLIN049"},
    {"uco": "421235", "code":"PLIN048"},
    {"uco": "421234", "code":"PLIN049"},
    {"uco": "421232", "code":"PLIN048"}]
#output
"""
{
    "421232":["PLIN048"],
    "421234":["PLIN048","PLIN049"],
    "421235":["PLIN048","PLIN049"]
}
"""
```

30. **Předpokládejte, že proměnná *tokens* obsahuje text ve formě seznamu jednotlivých slov, respektive tokenů. Napište program, který vypíše veškeré hapaxy (tokeny, které se v textu vyskytují pouze jednou).**

31. **Předpokládejte, že proměnná *tokens* obsahuje text ve formě seznamu jednotlivých slov, respektive tokenů. Napište program, který vypíše deset nejčetnějších tokenů spolu s jejich absolutní a relativní četností.**

Část II.
Procedurální programování

5. Podprogramy – funkce a procedury

5.1. Princip podprogramů

5.1.1. Podprogramy a izolované bloky

Jeden ze základních principů programování se skrývá pod zkratkou DRY neboli *Don't Repeat Yourself*, tedy neopakuj se. Pokud máte ve svém kódu velké části, které jsou stejné nebo se liší pouze v drobnostech, tento princip vám říká, že něco děláte špatně. A co tedy v takovéto situaci dělat?

Obecně platí, že pokud je ve vašem kódu nějaká „rutinní" opakující se činnost, můžete si ji představit jako podprogram. Ten napíšete jednou a poté ho pouze spouštíte. Dříve se používal termín rutina, dnes se setkáte spíš s termíny funkce a procedura (pozor, je mezi nimi drobný rozdíl, o kterém si povíme později). Jak tedy podprogram vytvoříme?

Princip je ve většině jazyků podobný, klíčové slovo pro vytvoření podprogramu následováno jejím názvem, kulatými závorkami obsahujícími parametry a nakonec blok kódu, který obsahuje samotný podprogram. V JavaScriptu používáme klíčové slovo *function*, v Pythonu je to pak *def*. Podprogram bez parametrů (o těch až později) v Pythonu ukazuje Kód 20, jeho obdobu v JavaScriptu Kód 21.

```
1 def hello_everyone():
2     print("Hello world!")
3     print("Hello universe!")
4     print("Hello everyone!")
```

Kód 20

```
1 function hello_everyone() {
2     console.log("Hello world!");
3     console.log("Hello universe!");
4     console.log("Hello everyone!");
5 }
```

Kód 21

Všimněte si, že i v tomto případě se jedná, jak už bylo zmíněno, o blok kódu, který je přiřazen funkci, tento blok se také zapisuje totožně jako u podmínek nebo cyklů, nicméně oproti nim se jedná vždy o blok izolovaný, to znamená, že veškeré proměnné vytvořené uvnitř podprogramu neexistují mimo něj a naopak proměnné vytvořené mimo podprogram (kromě tzv. globálních proměnných, ty však není doporučeno používat) nemáme uvnitř podprogramu k dispozici, pokud mu je explicitně nepředáme pomocí parametrů (o nich později). Můžete si tedy představit, že podprogramy se chovají podobně, jako kdyby se nacházely v samostatných souborech.

S podprogramy můžeme typicky také pracovat jako s proměnnými, a to tím způsobem, že se jedná o proměnnou, která se jmenuje stejně jako funkce (bez klíčového slova a závorek, tedy např. *hello_everyone*), a ta obsahuje funkci. Pokud ji uložíme do jiné proměnné, vytváříme tak vlastně alias. JavaScript i Python nám umožňují vytvářet i

nepojmenované podprogramy, které se přímo přiřadí do proměnných, to se však doporučuje pouze pro velice jednoduché (jednořádkové) podprogramy, viz Kód 22 a 23. Těmto anonymním podprogramům se také říká *closures* (uzávěry).

```
1 hello_world = lambda: print("Hello world!")
```

Kód 22

```
1 hello_world = function() { console.log("Hello world!") }
```

Kód 23

Všimněte si, že zatímco v JavaScriptu pouze vynecháme v definici název, Python používá pro uzávěry speciální klíčové slovo *lambda*, to vychází z tzv. lambda kalkulu, což je formalismus, ve kterém se uzávěry hojně využívají.

5.1.2. Volání podprogramu

Když víme, jak funkci definovat, hodilo by se také vědět, jak ji použít. Naštěstí je to velice jednoduché, příkaz pro zavolání podprogramu je v podstatě totožný jako pro získání hodnoty proměnné, s tím rozdílem, že pokud se jedná o podprogram, dodáme za název kulaté závorky. Princip je stejný ve všech moderních programovacích jazycích, uvažujme výše uvedený podprogram *hello_everyone*, jeho použití v rámci programu spolu s možností vytvoření aliasu ukazuje Kód 24.

```
1 hello_everyone()
2 hello_again = hello_everyone
3 hello_again()
4 hello_everyone()
```

Kód 24

Tento kód zavolá podprogram celkem třikrát, na řádku 1, 3 a 4, přičemž na řádku 3 se volá skrze alias vytvořený na řádku 2.

Ukažme si také, že se podprogram opravdu chová jako izolovaný blok, to nám demonstruje Kód 25.

```
1 def test():
2     a = 10
3     print(a)
4
5 a = 20
6 test()
7 print(a)
```

Kód 25

Kód nejprve na řádku 3 vypíše hodnotu lokální proměnné a, což je 10, následně na řádku 7 vypíše hodnotu 20, protože proměnné a na řádcích 2 a 5 nejsou totožné, první z nich se nachází v izolovaném bloku podprogramu, druhá je definována vně a přiřazení hodnoty do jedné nám neovlivní hodnotu druhé.

5.2. Vstup a výstup podprogramu

5.2.1. Parametry a jejich paměťová reprezentace

Někdy může být užitečné mít napsaný kód, který dělá vždy to samé (není nijak parametrizován) jako podprogram, typicky ale chceme měnit chování podprogramu podle zadaných parametrů v daném kontextu.

Parametry píšeme v definici podprogramu do kulatých závorek jednoduše tím, že napíšeme jméno parametru, jednotlivé parametry oddělujeme čárkou. V Pythonu může mít každý parametr nastavenu výchozí hodnotu, toho dosáhneme tím, že za název parametru připíšeme = a samotnou výchozí hodnotu. Podprogram může mít libovolné množství parametrů. Hodnoty parametrů (v pořadí, v jakém jsou psány v definici) dosazujeme do závorek na místě volání podprogramu, přičemž ty, které mají výchozí hodnotu, můžeme vynechat, definice i volání v Pythonu a JavaScriptu ukazují Kódy 26 a 27.

```
1  def test(a, b, operation="+"):
2      if operation == "+" :
3          print(a + b)
4      elif operation == "-":
5          print(a - b)
6      elif operation == "*":
7          print(a * b)
8      elif operation == "/":
9          print(a / b)
10     else:
11         print("Unknown operation")
12
13 test(10, 20)
14 test(20, 30)
15 test(10, 20, "-")
16 test(10, 20, "*")
17 test(10, 20, "a")
```

Kód 26

```
1  function test(a, b, operation) {
2      if operation == undefined
3          operation = "+"
4
5      if operation == "+"
6          console.log(a + b)
7      else if operation == "-"
8          console.log(a - b)
9      else if operation == "*":
10         console.log(a * b)
11     else if operation == "/":
12         console.log(a / b)
13     else
14         console.log("Unknown operation")
15
```

```
16 test(10, 20)
17 test(20, 30)
18 test(10, 20, "-")
19 test(10, 20, "*")
20 test(10, 20, "a")
```

Kód 27

V obou případech nám první volání vypíše číslo 20, druhé číslo 50, třetí -10, čtvrté 200 a páté *Unknown operation*. Všimněte si, že explicitní uvedení hodnoty parametru má přednost před výchozí (implicitní) hodnotou a že ačkoliv nám JavaScript neumožňuje přímo zadávat výchozí hodnoty parametrů, umožňuje nám stejného efektu dosáhnout pomocí podmínky a speciální hodnoty *undefined*, což je hodnota, kterou má proměnná, pokud nebyla definována (tj. nebyla do ní dosazena žádná hodnota).

Toto nám umožňuje ovlivňovat chování podprogramu na základě parametrů, za tyto parametry však mohou být dosazeny i proměnné, výrazy nebo volání jiných funkcí a v tuto chvíli (především u proměnných) se nabízí otázka, zda pokud je vstupním parametrem proměnná, může dojít k tomu, že se změna hodnoty uvnitř podprogramu projeví i mimo něj. Abychom tuto otázku zodpověděli, je potřeba se důkladněji podívat na to, jaký je vztah mezi předanou hodnotou parametru uvedenou v závorkách na místě volání a parametrem jakožto proměnnou figurující v definici podprogramu.

Princip je jednoduchý, v podstatě se do parametrů přiřadí hodnoty, které předáme během volání, přičemž přiřazení se chová stejně, jako bylo zmíněno v 1.2 a 2.2.3. Tedy v případě Pythonu a JavaScriptu jsou na sobě proměnné nezávislé, pokud jde o jednoduché typy, a v případě složených typů jsou závislé, ale jen do okamžiku prvního přiřazení hodnoty parametru pomocí operátoru =. Za přiřazení se u kolekcí nepovažuje přiřazení hodnoty položce. Jak se interpretuje volání podprogramu ukazuje Kód 28.[16]

```
1  def test(x):
2      print(x[0])
3      x[1] = 10
4      print(x[1])
5      x = [2, 4]
6      x[0] = 5
7      print(x[0])
8      print(x[1])
9
10 y = [1, 3]
11 print(y[0])
12 test(y)
13 print(y[0])
14 print(y[1])
15
16 #interpretace
17 y = [1, 3]
18 print(y[0])
19 x = y
```

[16]Kód je v Pythonu, v JavaScriptu by byl princip totožný a pouze syntax by se mírně lišila.

```
20 print(x[0])
21 x[1] = 10
22 print(x[1])
23 x = [2, 4]
24 x[0] = 5
25 print(x[0])
26 print(x[1])
27 print(y[0])
28 print(y[1])
```

Kód 28

Nejprve tedy vypíšeme hodnotu y_0, tj. 1, následně do x přiřadíme y a vypíšeme x_0, které bude logicky rovněž 1, do x_1 přiřadíme hodnotu 10, protože obě proměnné stále sdílí referenci (do x nebylo zatím přiřazeno), provede se změna i u y. Vypíšeme x_1 a dostaneme hodnotu 10. Přiřadíme do x novou kolekci obsahující 2 a 4, následně do x_0 přiřadíme 5, poté vypíšeme x_0 a x_1 a dostaneme hodnoty 5 a 4. Nakonec vypíšeme y_0 a y_1 a dostaneme hodnoty 1 a 10. Proč? Vše, co se dělo od řádku 23 (respektive v rámci podprogramu od řádku 5) dále, již nemělo vliv na y, protože x získalo novou referenci, změna x_1 ale proběhla dříve, tudíž se projevila i na y.

5.2.2. Princip návratových hodnot

Když už víme, jak ovlivňovat chování podprogramu na základě vstupu, nabízí se otázka, jak ovlivnit program na základě podprogramu, respektive jeho výstupu. V minulé části jsme viděli nepřímý způsob, jak toho dosáhnout, když se nám hodnota kolekce, která se nacházela vně podprogramu, změnila uvnitř podprogramu. Tomuto chování podprogramů se říká vedlejší efekt, tj. chování podprogramu vedle své samotné činnosti způsobí jako vedlejší efekt změnu stavu. To však není vždy žádoucí způsob.

Alternativa, kterou nám podprogramy poskytují, je definovat jejich výstup, v podstatě by se dalo říci, že výstup podprogramu je hodnota podprogramu (ve stejném smyslu jako hodnota proměnné), častěji se však terminologicky setkáte s pojmem *hodnota funkce*. Právě to, jestli má podprogram výstup, nebo ne, rozlišuje dva typy podprogramů, při absenci výstupu jej nazýváme procedura, pokud podprogram výstup má, říkáme mu funkce. Úmyslně jsem proto do této chvíle používal termín podprogram, který zahrnuje obě varianty. Velice často se v širším významu používá termín funkce i pro všechny podprogramy, přičemž za proceduru se považuje funkce s prázdným výstupem. Protože je terminologie takto zažita, budu dále používat namísto pojmu podprogram pojem funkce v jeho širším významu.

Funkce mající výstup se tedy svým chováním blíží funkci tak, jak ji známe z matematiky. Tedy předpisu závislému na vstupech, který se při zadání vstupů změní na vypočítanou hodnotu. Definice funkce je v našem případě oním předpisem a volání je realizací této funkce, zadáním vstupu, pokud tedy někde v kódu zavoláme funkci, která má výstup, můžeme s ní pracovat jako s proměnnou, protože se za ni dosadí (např. do výrazu) její výstup, rozdíl je pouze v tom, že zatímco proměnná se vyhodnotí triviálně dosazením aktuálně přiřazené hodnoty, funkce se musí nejprve provést a za její volání se

teprve dosadí její výsledek.

S tímto přístupem k funkcím jste se již setkali, vzpomeňme například příkazy *len* a *range*, jak z jejich podoby (za jejich názvem se nachází kulaté závorky s hodnotami parametrů) nyní již jistě víte, jedná se ve skutečnosti o funkce. V případě *len* je vstupem kolekce, funkce spočítá, kolik má kolekce prvků, a jako výsledek (výstup) nám vrátí číslo odpovídající počtu těchto prvků, tedy na místě volání v daném výrazu se nám nahradí za toto číslo. Podobně funkce *range* nám dle zadaných parametrů vygeneruje kolekci čísel, kterou vrátí na výstupu jako výsledek, tedy volání je nahrazeno ve výrazu touto kolekcí.

V tuto chvíli je tedy klíčovou otázkou, jak určit výstup funkce. Výrazná část programovacích jazyků (včetně Pythonu, JavaScriptu, ale např. i Java nebo jazyk C) k tomu používá příkaz označovaný klíčovým slovem *return*, tedy návrat z funkce. Základním účelem tohoto příkazu je, že ukončí provádění funkce a vrátí se zpět na místo volání, kde pokračuje dále v programu. Pokud není příkaz *return* uveden, provede se vždy jako poslední příkaz funkce. Kde v tomto příkazu však figuruje onen výstup? Trik spočívá v tom, že *return* má jeden volitelný parametr a tím je právě výstup funkce, tímto parametrem může být v zásadě libovolný výraz, od jednoduché konstanty až po složené výrazy. Tomuto výstupu funkce se vzhledem k použití příkazu *return* říká obvykle návratová hodnota funkce.

Je důležité si také uvědomit, že výstup se typicky pro různé vstupy (a stav paměti) liší. Toho můžeme dosáhnout buď výrazem, např. výraz $a + b$ je závislý na a a b, nebo tím, že příkaz *return* napíšeme ve více variantách, v tomto případě je však důležité si uvědomit, že jakmile se příkaz *return* vyhodnotí, zbytek funkce se neprovede, tedy nemá smysl za sebe psát více příkazů *return*, protože se vyhodnotí pouze první z nich, typické použití více různých návratových hodnot tedy obvykle zahrnuje podmínky, kdy se např. jedna návratová hodnota použije při splnění podmínky a druhá při jejím nesplnění. Použití návratové hodnoty jako výstupu funkce spolu s použitím této funkce ve výrazu a interpretaci téhož bez použití funkcí ukazuje Kód 29.[17]

```
1  def odd_or_even(number):
2      if number % 2 == 0:
3          return "odd"
4      else:
5          return "even"
6
7  print("Number 10 is " + odd_or_even(10) + ".")
8  print("Number 9 is " + odd_or_even(9) + ".")
9
10 #interpretace
11 number = 10
12 if number % 2 == 0:
13     odd_or_even = "odd"
14 else:
15     odd_or_even = "even"
```

[17]Kód je opět v Pythonu, použití v JavaScriptu je obdobné, rovněž používá klíčové slovo *return* a hodnotu oddělená mezerou

```
print("Number 10 is " + odd_or_even + ".")
number = 9
if number % 2 == 0:
    odd_or_even = "odd"
else:
    odd_or_even = "even"
print("Number 9 is " + odd_or_even + ".")
```

Kód 29

Jak vidíte, funkci, která vrací hodnotu, si lze představit jako proměnnou, jejíž hodnota ale závisí na vstupu (daných parametrech). První print nám tedy vypíše „Number 10 is even.“, zatímco výstupem druhého bude text „Number 9 is odd.“

5.2.3. Funkce a typování

V poslední části této kapitoly bych rád krátce zmínil specifika funkcí v typovaných jazycích, jako ukázkový jazyk zde budu používat Javu.

V typovaných jazycích je princip funkcí v podstatě totožný, rozdílem je, že funkce mohou (v Javě musí) být typovány. Typováním funkce se rozumí to, že v definici je určeno, jakého datového typu je návratová hodnota (pokud funkce návratovou hodnotu nemá, použije se prázdný datový typ, v Javě se označuje klíčovým slovem *void*), a zároveň je datový typ přiřazen i každému parametru. Z toho plyne, že typ výstupu i jednotlivých vstupů je striktně omezen na daný datový typ.

Tento přístup má na jednu stranu nevýhodu v tom, že nás určitým způsobem omezuje a pro funkce, které mají fungovat pro více datových typů, nás nutí psát více variant s jinak typovanými parametry (obvykle se tomuto přístupu říká přetěžování funkcí). Metody, jak tato omezení řešit, jsou však nad rámec těchto skript. Na druhou stranu nám tyto restrikce poskytují základní kontrolu správnosti vstupu a zabraňují tomu, abychom jako vstup dostali například text v okamžiku, kdy očekáváme celé číslo. Jak vypadá typovaná funkce v Javě ukazuje Kód 30[18].

Kompromisem je tzv. volitelné typování, které umožňuje funkce i proměnné typovat, ale nevyžaduje to, typování tedy můžeme použít jen tam, kde si to přejeme. Tento přístup používá například Typescript (speciální knihovna pro JavaScript).

```
String odd_or_even(int number) {
    if(number % 2 == 0) {
        return "odd"
    } else {
        return "even"
    }
}
System.out.println("Number 10 is " + odd_or_even(10) + ".")
System.out.println("Number 9 is " + odd_or_even(9) + ".")

```

[18] Zápis je mírně zjednodušen, Java při vytváření funkcí vyžaduje nastavit ještě několik dalších kritérií, ta však nijak neovlivňují chování funkce ve smyslu, který by pro nás byl v tuto chvíli relevantní, a nejsou tudíž uvedeny

```
#interpretace
int number = 10
String odd_or_even = ""
if(number % 2 == 0) {
    odd_or_even = "odd"
} else {
    odd_or_even = "even"
}
System.out.println("Number 10 is " + odd_or_even + ".")
int number = 9
if(number % 2 == 0) {
    odd_or_even = "odd"
} else {
    odd_or_even = "even"
}
System.out.println("Number 9 is " + odd_or_even + ".")
```

Kód 30

Všimněte si, že při definici funkce Java nevyžaduje žádné klíčové slovo, roli klíčového slova *def*, respektive *function*, převzal datový typ návratové hodnoty.

Příklady k procvičení

1. **Mějme funkci** *play* **s parametrem** *tracks* **obsahujícím název skladeb v playlistu, parametrem** *current* **obsahujícím index aktuální skladby a parametrem** *repeat*, **který říká, zda chceme po přehrání poslední skladby pokračovat znovu od začátku. Implementujte funkci tak, že bude tento playlist „přehrávat", kdy přehráním jedné skladby se rozumí výpis jejího názvu.**

2. **Implementujte funkce** *prev* **a** *next* **a druhou jmenovanou použijte v rámci funkce** *play*.

3. **Implementujte jednoduchý EventHandler tj. funkci** *listen* **s parametry** $eventName$ **typu text a** *listener* **typu funkce (s funkcemi lze pracovat jako s proměnnými) a funkci** $fireEvent$ **s parametry** $eventName$ **typu text a** *data* **typu slovník. Pro uložení listenerů použijte slovník klíčovaný podle jména události mající jako hodnotu seznam přiřazených funkcí. Funkce** *listen* **přidává funkci do seznamu podle klíče** $eventName$ **a funkce** $fireEvent$ **spustí postupně všechny funkce uložené pod tímto klíčem.**

4. **Implementujte funkci pro násobení dvou matic ve formátu dvourozměrných seznamů, tj. seznam seznamů. Vyjděte z následujícího vzorečku, kdy** c **reprezentuje výslednou matici a** a **a** b **matice vstupní. Dolní indexy reprezentují indexy v kolekci a** m **je počet sloupců první matice a počet řádků druhé matice.**

$$c_{ij} = \sum_{k=1}^{m} a_{ik} \cdot b_{kj}$$

6. Rekurze (viz Rekurze)

6.1. Vnořené funkce

6.1.1. Volání funkce uvnitř funkce

O funkcích jsem v minulé kapitole mluvil také jako o podprogramech. Toto pojmenování není bezúčelné. Tak jako program existuje ve větším prostředí (kterým je počítač, respektive jeho operační systém a vše, co v něm běží), ze kterého prostřednictvím vstupu získává kontext a prostřednictvím výstupu jej zase informuje o svém stavu, případně výsledku, nachází se v takovém prostředí i podprogram, v tomto případě je však tím prostředím program, jehož je podprogram, tedy funkce, součástí.

To má jeden podstatný dopad, funkci můžeme psát stejným způsobem jako program sám, a tedy můžeme v ní volat další funkce, používat podmínky, cykly kolekce a další. Jediným podstatným rozdílem je, že definice dalších funkcí se nepíšou uvnitř naší funkce[19], aby byly k dispozici i mimo ni. Volání funkce může být dokonce i návratovou hodnotou funkce jiné.

Na začátku může být problematické takto napsanému kódu porozumět, naštěstí i zde se vyhodnocování řídí principy, se kterými jsme se již setkali, v podstatě stejnými, jakými se vyhodnocuje funkce uvnitř programu. Tento obecný postup vyhodnocování by se dal nazvat principem vkládání a dosazování.

6.1.2. Princip vkládání a dosazování

Kód, který se nachází uvnitř definice funkce, lze v kontextu (na místě volání funkce) interpretovat v podstatě dvěma způsoby, a to jako vkládání kódu funkce (s dosazenými vstupy) na konkrétní místo nebo dosazení návratového hodnoty funkce za volání funkce samotné. Obvykle při vyhodnocování funkce s návratovou hodnotou dochází k oběma operacím, nejprve se vloží kód a následně se původní volání nahradí návratovou hodnotou.

S tím, jak se volání funkce nahrazuje návratovou hodnotou, jsme se již setkali a toto se při volání uvnitř jiné funkce nikterak neliší, v podstatě s voláním zacházíme jako s parametrizovanou proměnnou. Otázkou tedy je, jaký kód se vkládá a na jaké místo jej vložíme. I zde jsou obě odpovědi poměrně jednoduché. Vkládá se veškerý kód funkce, který se pro dané parametry provede, tedy od začátku po příkaz *return*, přičemž se bere v potaz, které podmínky jsou pro dané volání splněny, a vkládá se jen ten kód, který se nachází uvnitř splněných podmínek (jsou-li ve funkci nějaké). Místo vložení je vždy stejné, kód se vkládá mezi příkaz, v rámci něhož byla funkce volána, a příkaz, který mu přímo předchází.[20]

Jak vnořené volání funkcí vypadá a také to, jak je interpretováno, nám ukazuje Kód 31.

[19] V JavaScriptu lze zvolit i tento způsob, funkci pak bude možno používat pouze uvnitř funkce, ve které je vytvořena.

[20] Mluvíme-li zde o předcházejícím příkazu, rozumí se tím příkaz, který předchází při vyhodnocování, nikoliv v kódu. Ve skutečném pořadí mohou hrát roli podmínky, cykly a další konstrukce, při analýze pak postupujeme způsobem uvedeným v příslušných kapitolách

```
1  def add(x, y):
2      print(x)
3      print(x)
4      return x + y
5
6  def multiply(a, b):
7      result = 0
8      for i in range(b):
9          result = add(result, a)
10     return result
11
12 multiply(4, 2)
13 multiply(2, 4)
14
15 #interpretace
16 a = 4
17 b = 2
18 result = 0
19 for i in range(b):
20     x = result
21     y = a
22     print(x)
23     print(y)
24     add = x + y
25     result = add
26
27 a = 2
28 b = 4
29 result = 0
30 for i in range(ab):
31     x = result
32     y = a
33     print(x)
34     print(y)
35     add = x + y
36     result = add
```

Kód 31

Cyklus *for* nám v tomto případě zopakuje dané příkazy dvakrát u prvního volání, respektive čtyřikrát u volání druhého.

Volání funkce může být i návratovou hodnotou, vstupem, volání se ve funkci rovněž může nacházet více. V případě funkce jako parametru je nejprve vyhodnocena vnitřní funkce (tj. parametr) a poté funkce vnější. Pravidla pro vkládání kódu a dosazování jsou stejná, ale je potřeba si uvědomit, že kód se nevkládá před volání funkce, ale před celý příkaz (složený výraz), ve kterém se volání nachází. To platí pro vnitřní i vnější funkci, přičemž je zachováno pořadí kódu dle pořadí volání.

Interpretaci a zápis těchto komplexnějších variant vnořeného volání funkce nám ukazuje Kód 32.

```
def neg(z):
    print(z)
    return -z

def text_result(n):
    print(n)
    return "Result: " + str(n)

def add(x, y):
    print(x, y)
    return x + y

def substract(a, b):
    print(a, b)
    return text_result(add(a, neg(b)))

print(substract(10, 5))

#interpretace
a = 10
b = 5
print(a, b)
z = b
print(z)
neg = -z
x = a
y = neg
print(x, y)
add = x + y
n = add
print(n)
text_result = "Result: " + str(n)
substract = text_result
print(substract)
```

Kód 32

Funkce by samozřejmě mohly být komplexnější, obsahovat podmínky a cykly, principy vyhodnocování podmínek či cyklů by však byly totožné s těmi, které byly vyloženy v příslušných kapitolách, a nebudu je zde tudíž opakovat.

6.2. Rekurze

6.2.1. Princip rekurze – přímá a nepřímá

Specifickým případem zanořeného volání funkce je takzvaná rekurze. Rekurzí označujeme situaci, kdy funkce obsahuje (jedno nebo více) volání sebe sama, přičemž toto volání může být buď přímé, anebo nepřímé. Přímým se rozumí, že funkce A volá funkci A, zatímco při nepřímé rekurzi funkce A volá funkci B (ta případně volá funkci C, ta funkci D atd.) a ta volá opět funkci A.

Už z této definice možná vidíte problém, který zde nastává. Pokud funkce volá, ať přímo či nepřímo, sebe sama, kdy toto volání skončí? Bez definovaného omezení se bude volání funkce dále a dále zanořovat a nikdy neskončí[21].

Jak tedy tuto situaci vyřešit? Musíme zajistit to, aby k rekurzi nedocházelo ve všech případech, aby se v určité situaci, v určitém okamžiku zastavila. Kontextu, ve kterém v rekurzivní funkci nedochází k rekurzi, se říká báze nebo také koncová podmínka (nebo podmínky). Ty realizujeme, jak název napovídá, tak, že před samotné rekurzivní volání umístíme podmínku, která v případě splnění funkci ukončí (a typicky vrátí návratovou hodnotu). Alternativním přístupem je umístit do podmínky nikoliv toto ukončení, ale rekurzi samotnou. První přístup se používá typicky u funkcí vracejících hodnotu, druhý pak u funkcí, které hodnotu nevrací (procedur).

6.2.2. Analýza rekurze

Největším problémem rekurzivních funkcí je správně analyzovat interpretaci jejich volání. Princip analýzy je totožný jako u kterýchkoliv vnořených volání funkcí, nicméně kvůli tomu, že se stále vracíme do stejné funkce, je velice snadné se ztratit. Podívejme se tedy nyní na několik běžných typů rekurze a na vhodné způsoby, jak je analyzovat.

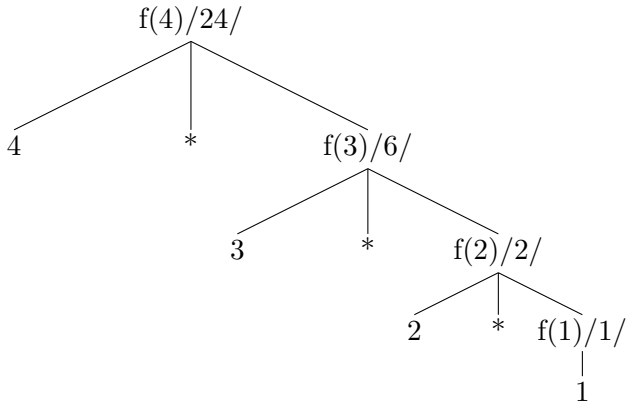
Nejjednodušším typem je takzvaná koncová rekurze (v angličtině tail recursion). Ta má tu vlastnost, že rekurzivní volání je posledním příkazem funkce, a to ať už jde o funkci, která hodnotu vrací, nebo nikoliv, za koncovou lze považovat i takovou rekurzi, kde je rekurze součástí výrazu, který je návratovou hodnotou funkce. Typickým příkladem koncové rekurze je například výpočet faktoriálu. Zvláštní vlastností koncové rekurze je také to, že ji lze vždy bez problémů přepsat pomocí cyklu.

Výpočet faktoriálu pomocí rekurze, cyklu a interpretaci rekurentní varianty ukazuje Kód 33.[22]

```
1  def factorial_recursion(x):
2      if x <= 1:
3          print("stop")
4          return 1
5      else:
6          print(x)
7          return x * factorial_recursion(x - 1)
8
9  def factorial_iteration(n):
10     result = 1
11     for i in range(n, 0, -1):
12         result = result * i
13     return result
14
15 factorial_recursion(4)
16
```

[21] Ve skutečnosti skončí v okamžiku, kdy programu dojde přidělená paměť, v tu chvíli je program násilně ukončen (podobně jako většina postav ve Hře o trůny)

[22] Pro odlišení příkazů podle funkce, která je volá (odlišení zanořených volání), je použito za názvem proměnné podtržítko a hodnota x pro dané volání.



Obrázek 6.1: Faktoriál jako koncová rekurze

```
17 #interpretace
18 x_4 = 4
19 print(x_4)
20 x_3 = 3
21 print(x_3)
22 x_2 = 2
23 print(x_2)
24 x_1 = 1
25 print("stop")
26 factorial_recursion_1 = 1
27 factorial_recursion_2 = x_2 * factorial_recursion_1
28 factorial_recursion_3 = x_3 * factorial_recursion_2
29 factorial_recursion_4 = x_4 * factorial_recursion_3
```

Kód 33

Z interpretace můžeme cyklickou povahu koncové rekurze vidět (pokud vezmeme v potaz, že „indexy“ nejsou součástí názvu proměnných, ale jedná se o měnící se hodnoty téže proměnné, dostaneme v podstatě interpretaci implementace pomocí cyklu. Za povšimnutí také stojí „trychtýřová“ povaha posloupnosti volání. Při rekurzi má vkládání kódu právě tuto povahu, vše, co se nachází před voláním rekurze, se přidává před volání, a to v pořadí od původního volání až do splnění koncové podmínky (proto se mluví o zanořování funkcí). V zanoření splňujícím koncovou podmínku (tj. maximální úroveň zanoření, báze) se přidá kód v pořadí, v jakém je. Následují části kódu obsahující zanoření (s dosazenými hodnotami), případně pokud se nejedná o koncovou rekurzi kód nacházející se za voláním, a to v pořadí opačném než při zanoření, tedy směrem nahoru k prvotnímu volání funkce (proto se v tomto případě mluví o návratu z funkce a návratových hodnotách). Grafickou reprezentaci ukazuje Obrázek 6.1.[23]

Pokud má funkce pouze jedno místo volání, které není koncové (tím pádem se zcela jistě nejedná o návratovou hodnotu), přepis na cyklus nemusí být triviální, v některých

[23]Vyhodnocování zde postupuje nejprve směrem dolů podél levých větví a poté zpět nahoru cestou dosazování výsledku z nižší úrovně (mezi lomítky)

případech ani možný. Interpretace bude nicméně podobná a lišit se bude pouze tím, že mimo samotný kód volání budou směrem nahoru k prvotnímu volání vkládány i příkazy po rekurentním volání následující. Můžete si to představit jako vkládání závorek do sebe. Přesněji to ilustruje Kód 34.

```
def brackets(n):
    print(n)
    print("(")
    if n > 1:
        brackets(n - 1)
    print(")")
    print(n)

brackets(4)

#interpretace
n_4 = 4
print(n_4)
print("(")
n_3 = 3
print(n_3)
print("(")
n_2 = 2
print(n_2)
print("(")
n_1 = 1
print(n_1)
print("(")
print(")")
print(n_1)
print(")")
print(n_2)
print(")")
print(n_3)
print(")")
print(n_4)
```

Kód 34

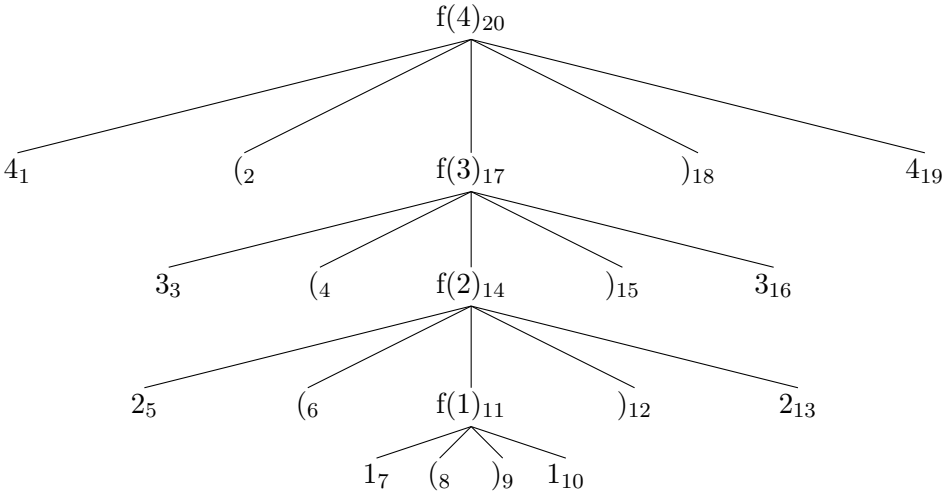
I zde vidíme onu trychtýřovou povahu, výstupem bude posloupnost 4(3(2(1()1)2)3)4. Vizuální reprezentace vyhodnocení je zobrazena na Obrázku 6.2.[24]

Situace je komplikovanější, pokud k rekurentnímu volání dochází na více místech. Toto může být opět v rámci výrazu nebo v rámci různých částí kódu nezávisle na sobě. Podívejme se na oba příklady.

Postup u více nezávislých volání je podobný jako u jednoho, ale dochází k více zanořováním, přičemž se postupuje shora dolů a zleva doprava. Modifikaci Kódu 34 tak, aby ukazoval tento přístup, spolu s jeho interpretací ukazuje Kód 35.[25] Reprezentaci

[24] Indexy u jednotlivých větví indikují pořadí vyhodnocování.

[25] V tomto příkladu byl snížen počet zanoření, protože velikost interpretace zde roste velice rychle s



Obrázek 6.2: Obecná rekurze s jedním místem zanoření

volání ukazuje Obrázek 6.3.

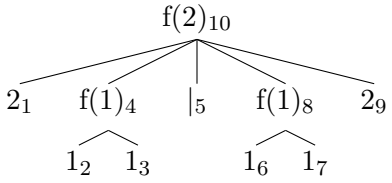
```
1  def brackets(n):
2      print(n)
3      if n > 1:
4          brackets(n - 1)
5          print("|")
6          brackets(n - 1)
7      print(n)
8
9  brackets(2)
10
11 #interpretace
12 n_2 = 2
13 print(n_2)
14 n_1_a = 1
15 print(n_1_a)
16 print(n_1_a)
17 print("|")
18 n_1_b = 1
19 print(n_1_b)
20 print(n_1_b)
21 print(n_2)
```

Kód 35

Posloupnost by v tomto případě vypadala následovně: 211|112, v případě vstupu 4 bychom pak dostali následující posloupnost: 43211|112|211|1123|3211|112|211|11234.

Všimněte si, že i zde se uplatňuje totožný princip vkládání, jen je zde dvakrát vloženo

každým krokem, v podstatě exponenciálně. Zároveň přibylo označení „indexy“ *a* a *b*, abychom rozeznali první a druhé volání a pro účely vykreslování byly vynechány závorky.



Obrázek 6.3: Rekurze s vícenásobným zanořením

totéž oddělené svislítkem. Stejný princip zůstává zachován i pro větší počet rekurentních volání. A podobně bychom řešili i situace, kdy by některé volání bylo omezeno podmínku, nacházelo se v cyklu a podobně, v případě podmínek by k některým voláním nedocházelo vždy, a některá vložení by se tudíž neprovedla, cykly by se chovaly v podstatě obdobně jako Kód 35, ten si můžeme představit (se zanedbáním detailů) jako rozepsaný cyklus délky 2.

Zanoření rovněž nemusí být totožná a mohou mít odlišné parametry, takovým případem je typický reprezentant druhého typu vícenásobného zanoření (tj. více zanoření v rámci jednoho výrazu), kterým je výpočet tzv. Fibonacciho čísel.

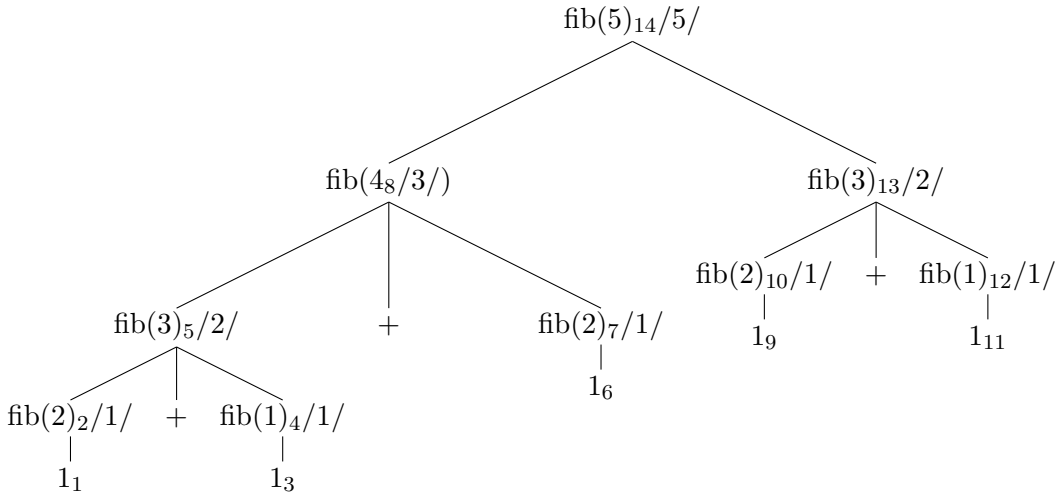
Fibonacciho čísla jsou posloupnost čísel začínající dvěma jedničkami a pro každý její další prvek platí, že je součtem předchozích dvou. Všimněte si, že už z definice se nám rekurzivní řešení přímo nabízí. Tato posloupnost není výjimkou, v podstatě veškeré posloupnosti se dají velice snadno a intuitivně implementovat pomocí rekurze. Důvod, proč se v informatice jako s příkladem setkáte především s Fibonacciho čísly, je, že se jedná o vskutku pozoruhodnou posloupnost, která se vyskytuje v přírodě na místech, která byste nejméně čekali.[26]

Definici tedy známe a nabízí se nám i implementace. Tu spolu s interpretací ukazuje Kód 36, grafickou reprezentaci pak Obrázek 6.4.[27]

```
1  def fibonacci(n):
2      if n <= 2:
3          return 1
4      else:
5          return fibonacci(n-1) + fibonacci(n - 2)
6
7  fibonacci(5)
8
9  #interpretace
10 n_4 = 4
11 n_3_a = 3
12 n_2_a = 2
13 fibonacci_2_a = 1
```

[26]Pro ty z vás, které zajímá, co se za touto posloupností skrývá, doporučuji si ve vyhledávači najít právě termín Fibonacciho čísla, případně k němu přidat ještě druhý úzce související pojem, kterým je zlatý řez. Jedná se o jednu z mála věcí, která i nematematika dokáže přesvědčit o tom, že matematika je fascinující.

[27]Vypočítané hodnoty při cestě zpět jsou opět ohraničeny lomítky a indexy odpovídají pořadí vyhodnocování



Obrázek 6.4: Rekurze s vícenásobným zanořením

```
14 n_1_a = 1
15 fibonacci_1_a = 1
16 fibonacci_3_a = fibonacci_2_a + fibonacci_1_a
17 fibonacci_2_b = 1
18 fibonacci_4 = fibonacci_3_a + fibonacci_2_b
19 n_3_b = 3
20 n_2_c = 2
21 fibonacci_2_c = 1
22 n_1_b = 1
23 fibonacci_1_b = 1
24 fibonacci_3_b = fibonacci_2_c + fibonacci_1_b
25 fibonacci_2_d = 1
26 fibonacci_5 = fibonacci_4 + fibonacci_3_b
```

Kód 36

Povšimněte si, že zanořování je v podstatě totožné jako u nezávislých volání, přičemž hlavní rozdíl je zde způsoben především tím, že jedno volání je voláno s parametrem $n-1$ a druhé s $n-2$, tudíž druhá větev není zcela totožná, ale odpovídá větvi první, s tím rozdílem, že jí chybí jedna úroveň zanoření. To je dobře vidět z toho, že kroky 1 až 5 jsou totožné s kroky 9 až 13, stejně jako kroky 1 a 2 s kroky 6 a 7. Tento fakt ukazuje na riziko rekurze, opakovaně vypočítává i to, co už vypočítala. Jasně vidíme, že fib(1), fib(2), fib(3) atd., by nám stačilo vypočítat pouze jednou. To samozřejmě lze a existuje několik způsobů, jak toho dosáhnout, nejsou však natolik zjevné ze zadání jako řešení rekurzí a nejsou nyní předmětem našeho zájmu.

Především poslední dvě ukázky nás vedou k úvaze nad tím, kdy rekurzi použít.

6.2.3. Použití rekurze

Rekurze je přeneseně řečeno smlouva s ďáblem, většinou vyřeší poměrně snadno řadu problémů a často, jak jsme viděli u Fibonacciho čísel, není tolik obtížné na toto řešení přijít, s použitím rekurze však platíme cenu za její „nenažranost“.

Kdy ji tedy použít? Rozhodně ne v případě rekurze koncové, u té jsme si již řekli, že ji lze snadno přepsat na méně nenažrané cykly. Obecně by se dalo použít poučky „rekurzi použijte tehdy, když vás jiné řešení nenapadá, protože většinou je stále lepší mít pomalé řešení nežli řešení žádné“.

Typickým případem, kdy rekurzi využijeme takřka vždy, jsou úlohy typu rozděl a panuj. Obvykle se jedná o úlohy, které jsou náročné na přímé řešení, a tato metoda se drží principu zmenšovat problém na menší podproblémy, dokud se nedostane do fáze, kdy je schopna každý podproblém triviálně vyřešit, a výsledky následně spojí.

Klasickým případem jsou některé tzv. řadicí algoritmy, o kterých se zmíníme ve čtvrté části skript věnované algoritmizaci. Chcete seřadit dlouhou řadu čísel vzestupně, jak toho dosáhnete co nejrychleji? Příkladem postupu rozděl a panuj je například následující. Postupně si rozděluje řadu napůl, dokud nedostane jedno číslo, to je z podstaty seřazené, pak vezme každá dvě sousední čísla a spojí je ve správném pořadí, následně spojuje dvojice čísel, čtveřice atd., dokud není seřazeno celé pole. Rekurze je zde skryta v tom, že když si postup rozdělí řadu napůl, seřadí obě poloviny řady stejným způsobem, dokud nedojde na řady obsahující pouze jedno číslo.

O dalších zajímavých příkladech si povíme ve čtvrté části skript.

Příklady k procvičení

1. **Napište program na procházení a výpis libovolně zanořené kolekce, využijte rekurzi.**

2. **Napište program, který vypíše n-tý člen tribonacciho posloupnosti. První tři prvky jsou 0, 1, 1 a každý další člen je součtem předchozích tří. Použijte rekurzi.**

3. **Napište program, který vypíše n-tý člen aritmetické posloupnosti na základě tří parametrů a, tj. hodnota prvního prvku, d, tj. diference a n (pořadí prvku), přičemž platí, že $a_n = a_{n-1} + d$.**

4. **Napište program, který vypíše n-tý člen geometrické posloupnosti na základě tří parametrů a, tj. hodnota prvního prvku, q a n (pořadí prvku), tj. kvocient, přičemž platí, že $a_n = a_{n-1} \cdot q$.**

5. **Napište program, který projde seznam pomocí rekurze.**