Radek Hylmar

Programování pro úplné začátečníky

Názorná příručka krok za krokem
Od úplných základů po první prográmky
Jak vytvářet proměnné, podmínky nebo cykly
Praktická cvičení a řešení nejčastějších problémů

CPRESS

Radek Hylmar

Programování pro úplné začátečníky

Computer Press
Brno
2012

Programování pro úplné začátečníky

Radek Hylmar

Ilustrace: Pavlína Vališová
Obálka: Martin Sodomka
Odpovědný redaktor: Martin Domes
Technický redaktor: Jiří Matoušek

Objednávky knih:
http://knihy.cpress.cz
www.albatrosmedia.cz
eshop@albatrosmedia.cz
bezplatná linka 800 555 513
ISBN 978-80-251-2129-0

Vydalo nakladatelství Computer Press v Brně roku 2012 ve společnosti Albatros Media a. s. se sídlem Na Pankráci 30, Praha 4. Číslo publikace 16006.

Dotisk prvního vydání.

ALBATROS MEDIA a.s.

Obsah

Úvodem

Programování se rychle stalo jedním z nejdůležitějších oborů lidské činnosti. Počítačem se dnes řídí téměř vše: od digitálních hodinek, mobilních telefonů a praček, přes požární systémy či ovládání světel až po letadla nebo kosmické rakety.

Stěrače v autě například umějí reagovat na sílu deště a podle toho samy určují frekvenci stírání okna. Jízda je pak pohodlnější, a tím i bezpečnější. Díky ulehčení a zpříjemnění všeho, co děláme, se kvalitně vytvořené počítačové programy dostávají čím dál blíž našemu praktickému životu.

Jak brzo uvidíme, umět programy vytvářet není vůbec těžké. Je pouze třeba osvojit si jistý způsob myšlení, který je nám ovšem už dávno vlastní. Stačí ho jen využít. Pokud navíc zjistíme, jak s počítačem komunikovat, máme ho v hrsti – bude dělat, co mu přikážeme.

Komu je kniha určena

Učebnice je pro všechny, kteří mají nějakou zkušenost s počítači, ale o programování zatím vůbec nic nevědí. Příručka je určena každému, kdo se chce naučit základním dovednostem a technikám programování, ale i těm čtenářům, které programování zajímá jen jako silný fenomén dnešní doby. Pro počítačové programování totiž platí:

- nezáleží na věku, ve kterém začneme programovat,
- nezáleží ani na výši vzdělání, protože v programování jde pouze o způsob přemýšlení a dorozumění se s počítačem,
- nikdy není pozdě s programováním začít.

Pro výuku z této publikace potřebujeme jen dvě věci: počítač a chuť učit se novým věcem.

Forma výkladu

Všechny důležité dovednosti i znalosti si osvojíme doslova krok za krokem a bez zabíhání do zbytečně komplikovaných a nepotřebných detailů. Celý výklad se totiž drží několika základních pravidel:

- začneme zcela od nuly,
- pomalu se budeme učit přemýšlet jako programátor,
- naučíme se mluvit s počítačem tak, aby nás poslouchal,

- vždy upozorníme na užitečné tipy a na možná úskalí,
- text doplňuje mnoho názorných obrázků,
- snadnému pochopení pomáhá i přímé propojení textu s probíraným jevem na obrázku,
- na konci budeme znát všechny nejdůležitější programátorské techniky,
- tyto techniky budeme umět použít v programátorské praxi.

V textu se vyskytuje několik speciálních odstavců, které mají za úkol doplnit základní výklad o další užitečné, podstatné či prostě zajímavé okolnosti:

Poznámka: Poskytuje rozšiřující informace na okraj.

Tip: Doplňuje výklad o zajímavou a prospěšnou informaci.

Důležité: Informace, které byste si měli zapamatovat.

Řešení problému: Pokud hrozí, že se v některém postupu mohou objevit potíže, tento odstavec na ně upozorní a vysvětlí, jak je řešit.

Pro ještě názornější ukázky jsou některé části programových kódů `tučně zvýrazněny`. Tučnost nemá v programu žádný význam – slouží pouze pro zvýraznění právě probíraného pojmu nebo nějaké důležité techniky.

Konkrétní postup výuky

O co při programování jde. Nemůžeme se hned vrhnout ke klávesnici a začít psát programy. Podobně jako plavci si některé základní techniky osvojíme nejprve na břehu. Necháme počítač ještě několik okamžiků vypnutý a v *kapitole 1* si povíme o těch nejzásadnějších věcech: jak se má programátor chovat, jaká dodržovat pravidla a také se naučíme první důležité pojmy. Bez těchto základů bychom se do hlubší vody nemohli pustit.

Potřebné nástroje a první program. S počítačem je potřeba se nějak dorozumět a v *kapitole 2* si řekneme jak. Z Internetu získáme volně dostupný nástroj, který poslouží jako překladač z naší řeči do jazyka počítače a naopak.

Druhá kapitola bude opravdu výjimečná, protože právě v ní se to stane poprvé! Dlouho očekávaný, sice prostinký a jednoduchý, ale krásný a plně funkční prográmek konečně spatří světlo světa. V té chvíli se každý z nás stane programátorem. Nebude cesty zpět.

Počítačová paměť. Bez paměti bychom to daleko nedotáhli. Plavat už umíme, ale proč to nezkusit lodí? V *kapitole 3* se naučíme pracovat s pamětí počítače: zjistíme, jak do ní ukládat věci na pozdější časy a jak je odtud zase získat zpět.

Podmínky a cykly. Dalším krokem na cestě ke skvělým programům bude obyčejné rozhodování a opakování. Zní to jednoduše a je to jednoduché. Přesto jde o nejdůležitější programátorské techniky, bez kterých nelze vyplout na otevřené moře. Podmínky a cykly probereme v *kapitolách 4 a 5*.

Několik jednoduchých programů. Všechno, co již umíme, si prakticky vyzkoušíme na několika programech, ukážeme si pár pěkných triků, jak si zjednodušit práci, budeme vytvářet náhodná čísla a pracovat se soubory. To vše *v kapitole 6*.

Složitější hry s pamětí. Využívat paměť počítače se sice naučíme v kapitole 4, ale na ostřejší kousky si troufneme později. V *kapitole 7* se mimo jiné naučíme tvořit seznamy lidí i s jejich základními údaji (číslo telefonu, adresa apod.). Vzájemně provázané údaje budeme ukládat do paměti a také si ukážeme, jak je z ní jednoduše dostávat.

Využití dříve napsaných programů. Není třeba hotový program tvořit znovu a znovu. Pokud je funkční a v pořádku, využijme ho. Názorně si vše předvedeme v *kapitole 8*. V té se dozvíme i to, jak využívat programy, které napsal někdo jiný. Programátoři celého světa spolupracují a napsané programy běžně poskytují.

Příklady ke knize

Celý balík všech cvičení z knihy si čtenáři mohou stáhnout z adresy *http://knihy.cpress.cz/k1620* – najdete je na záložce **Soubory ke stažení.**

1. Kódy si stáhněte a uložte do počítače.
2. Následně balík rozbalte pomocí některého archivačního programu (např. WinRAR – viz *www.winrar.cz*).
3. Po rozbalení se na disku vytvoří složka *Programovani_PUZ*, v níž naleznete soubory příkladů uložené v jednotlivých složkách podle kapitol.

Ke snadnější orientaci slouží přehledné rozhraní, které spustíte poklepáním na soubor *spustit.html* v hlavní složce. Dále se řiďte pokyny rozhraní.

Kapitola 1
Co je to počítačové programování

Umět programovat *neznamená* prostě vytvářet programy pro počítač – znamená vytvářet co nejefektivnější, co nejrychlejší a nejjednodušší, a přitom dokonale funkční programy. Je určitě možné naučit se komunikovat s počítačem tak, aby nám rozuměl a aby plnil naše rozkazy. I tak by se daly vytvářet jednoduché programy, ale daleko bychom se touto cestou nedostali.

Vytvořit například kalkulačku, jakou známe z nabídky **Start → Všechny programy → Příslušenství**, nebo jakýkoli jiný počítačový program lze mnoha způsoby. Naším cílem bude naučit se budovat programy co možno nejefektivněji. Efektivně pro nás i pro počítač.

Začneme tím nejdůležitějším: v první kapitole se naučíme přemýšlet jako programátor. Všechny naše další snahy v oblasti programování pak budou jednodušší, protože jediným heslem celé učebnice bude: *usnadníme* si práci. Vždy, než se do něčeho pustíme, budeme přemýšlet, jestli by to nešlo udělat nějak lépe; snadněji, rychleji nebo levněji. Jestli by nestálo za to vzít to z druhého konce a celé to provést jinak.

Co je to program a jak ho vytvořit

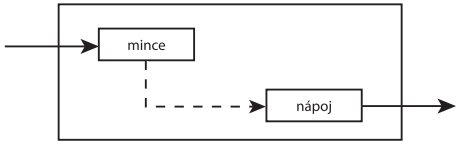
Běžně se snažíme provádět úkoly efektivně; podívejme se na několik obyčejných příkladů. Později zjistíme, že celý smysl jejich efektivity se jasně obráží při programování. Konkrétní příklady nám pomohou zjistit, v čem vězí ono usnadnění – jak přesně přemýšlet, na co se zaměřit, abychom dosáhli lepších výsledků. Všechny příklady dobře známe; ten první, jak uvidíme v dalších částech knihy, je přímo ztělesněním programování.

Nádražní automat

Stojí snad na každém nádraží a za naše peníze nabízí pochybnou kávu. Tedy nejen kávu. Nabízí i čokoládu, čaj, někdy i polévku, zkrátka teplé nápoje. Pokud bychom měli takový automat vytvořit, jak bychom to provedli?

Máme několik možností, ale základem jsou tři věci, které by se nejspíš neměnily:

1. vhození mincí,
2. výběr požadovaného nápoje,
3. kelímek plný zvoleného nápoje.



Obrázek 1.1. Schéma nádražního automatu: automat vyžaduje peníze a výběr nápoje; za to poskytuje vybraný nápoj

Základní schéma automatu ukazuje obrázek 1.1.

Automat je typickým příkladem programu. Vyžaduje od nás aktivitu (vhození požadovaného obnosu a výběr nápoje) a na základě naší konkrétní akce provede nějaký úkon. Tento úkon je jakýmsi jádrem programu. Nakonec vydá nápoj, nebo napíše na svůj displej, že peníze nestačí. A pokud automat zrovna funguje, vrátí nám i drobné.

Důležité: Akci, kterou od nás automat vyžaduje, budeme říkat *vstup*. Do automatu vstupují peníze a potom i náš výběr. To, co z automatu vychází ven, nazýváme analogicky *výstup*. Výstupem je zde kelímek s vybraným nápojem a případně vrácené drobné. Dokonce i informaci na displeji o tom, že peníze nestačí, můžeme považovat za výstup.

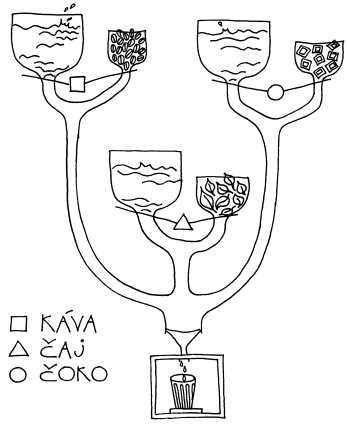
Vstupy a výstupy by se tedy při výrobě automatu nejspíš neměnily. Jaké tedy možnosti při jeho budování máme? Obrázek 1.2 ukazuje nejjednodušší způsob činnosti automatu.

Ale nešlo by to nějak lépe? Nad čím je potřeba přemýšlet, aby se celý proces výroby a podání nápoje zjednodušil? Máme-li na vědomí naše heslo „zjednodušení práce", je důležité myslet na dvě zásady:

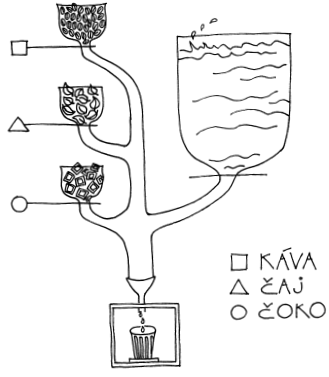
- Abychom nedělali věci víckrát, než je pro výsledek nutné.
- Abychom nevyužívali více zdrojů, než je to pro výsledek nutné.

Druhý bod nás přivádí k myšlence sjednotit všechny nádoby horké vody do jedné. Tím se sníží náklady na výrobu každého automatu a celý proces se zjednoduší. Navíc platí, že čím víc vody je v nádobě, tím pomaleji chladne. Takže se sníží i náklady na ohřívání vody. Obrázek 1.3 ukazuje, jak by náš vylepšený automat vypadal.

Použili jsme extrémně jednoduchý příklad – nejde tu o žádnou revoluční myšlenku; spojení několika nádob do jedné se nabízelo. Ale chtěli jsme ukázat, v čem hledat mezery. Kde hledat případné chyby nebo možnosti šetření a možnosti usnadnění. Stejným směrem by se mělo ubírat naše programátorské myšlení. Dvě výše uvedené zásady bychom si měli pamatovat a při programování využívat. Stačí když budeme jednak šetřit místem či prostředky a také šetřit svými silami. Neboli nedělat věci víckrát, než je to nutné.



Obrázek 1.2. Útroby automatu, který nabízí několik teplých nápojů (každý nápoj má zvláštní zdroj teplé vody)



Obrázek 1.3. Jiný způsob fungování automatu (všechny nápoje využívají jeden zdroj teplé vody)

Fotbal se hraje týmově

Jak uvidíme v dalších kapitolách (zejména v kapitole 8, ale i jinde), umět rozložit síly mezi více složek je důležitou vlastností každého programátora. Není nutné, jak by se mohlo zdát, rozdělovat úkoly mezi více programátorů. Nemusíme psát textovou zprávu kamarádovi, aby přišel pomoci. Vystačíme si úplně sami. Spíše samotný program je třeba naučit spolupracovat s jinými programy.

Na počítačový program se totiž dá pohlížet jako na fotbalový tým při důležitém zápase. Fotbalový tým, neboli celek, jak se říká, se skládá z jedenácti hráčů. Kromě těchto jedenácti mužů čekají na svou příležitost další hráči na střídačce. Hlavním cílem našeho týmu je dát gól.

A teď přichází důležitý okamžik. Jeden skvělý útočník se dostal do pravého rohu na soupeřově území. Před ním ve směru branky, jak ukazuje obrázek 1.4, je doslova hradba obránců. Ale za nimi je zcela nechráněný druhý útočník. Obránci na něj úplně zapomněli. V této chvíli se ukáže, jestli je útočník týmovým hráčem. Pokud by se pokusil projít k bráně sám, zcela zjevně by o míč přišel. Pokud ovšem přihraje, například na hlavičku, je tu reálná šance na gól.

Obrázek 1.4. Dramatická situace před soupeřovou brankou

Počítačové programy jsou na tom velmi podobně. Každý hráč týmu, každá složka programu bude mít jiný, zcela specifický úkol. Obrazně řečeno, určitý hráč má za úkol dopravit míč k rohu na soupeřově území, jiný hráč má za úkol vstřelit branku. A role jednotlivců se nesmějí míchat. Pokud by se náš útočník pokusil o vstřelení branky sám, neuspěl by. Je potřeba předat míč jinému hráči, a tím splnit úkol celého týmu.

Volba správné varianty

U příkladu fotbalového utkání jsme narazili na velmi důležitou věc, kterou je třeba obzvláště zdůraznit. A to přesto, že se jedná o běžnou, každodenní činnost. Jde o *rozhodování*. V počítačovém programu je třeba neustále rozhodovat o nejlepší variantě, o nejkratší cestě k výsledku. Pokud bychom měli fotbalovému hráči, budeme mu říkat číslo1, nařídit, jak se má chovat v popsané situaci před brankou, udělali bychom to nějak takto:

1. V případě, že hráč číslo 1 má míč, je jeho úkolem dostat míč k rohu na soupeřově části hřiště.
2. Ve chvíli, kdy se hráč číslo 1 dostane s míčem na určené místo, má za úkol zjistit, jak vypadá situace před soupeřovou brankou.
3. Pokud před soupeřovou brankou není kromě brankáře žádný hráč, má hráč číslo 1 za úkol dostat se blíž k brance a zkusit míč vsítit.
4. Pokud je před soupeřovou brankou kromě brankáře alespoň jeden (tzn. jeden nebo více) soupeřů a žádný hráč našeho týmu, má hráč číslo 1 za úkol stáhnout se z rohu a najít výhodnější pozici. Třeba i dál od branky.
5. Pokud se před brankou kromě cizích hráčů vyskytuje alespoň jeden náš hráč, má hráč číslo 1 za úkol přihrát mu na hlavičku.

Úkoly pro hráče číslo dvě by byly popsány zvlášť, pro nás je teď důležitější úloha hráče číslo 1. Všimněme si bodů 3–5. V podstatě to nejsou jen úkoly, ale především rozhodování. Teprve po rozhodnutí se provede konkrétní úkol. Dále si povšimněme, že byly popsány všechny možné varianty, aby nehrozilo, že se číslo1 dostane do neřešitelné situace.

Jednotlivé složky programu, jak poznáme v dalších kapitolách, se chovají stejně – je jim potřeba přesně vysvětlit, co mají v jaké situaci provádět. Výsledky rozhodování mohou mít vliv nejen na úspěšné dokončení celkového úkolu, ale i na správné využívání prostředků – šetření. Programování se tak blíží běžnému životu. Také přemýšlíme, jak naložit s nějakým obnosem peněz. A rozhodnutí závisí na tom, co je pro nás hlavním cílem.

Digitální budík – pravidelně opakovaná, jednotvárná činnost

Jak zařídit, aby digitální budík zazvonil ve stanoveném čase? Jak mu říci, aby dřív nezvonil? Zde se bohužel nevyhneme pravidelné, několikrát opakované činnosti. Je totiž potřeba, aby budík po každé sekundě kontroloval, zda už přišel jeho čas, nebo ještě ne.

Budeme-li předpokládat, že hodiny jako takové už fungují a my jsme dostali za úkol zprovoznit budík, můžeme zadat takovouto instrukci:

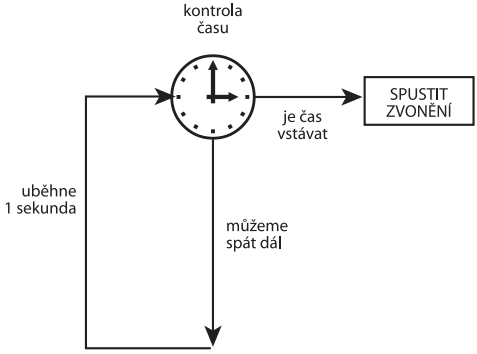
1. Zkontroluj, kolik je hodin.
2. Pokud se čas shoduje s nastaveným časem buzení, začni zvonit a čekej na zamáčknutí.
3. Pokud se čas neshoduje s nastaveným časem buzení, počkej sekundu.
4. Zkontroluj, kolik je hodin.
5. Pokud se čas shoduje s nastaveným časem buzení, začni zvonit.
6. atd.

Vidíme, že bychom takto pokračovali donekonečna, resp. museli bychom vypsat tolik těch tříbodových cyklů, kolik je sekund v jednom dni – to proto, abychom nevynechali žádnou možnost. Je nutné celou akci nějak zautomatizovat, protože zadávat instrukce na tisíce řádků je velmi nepříjemné. Naštěstí máme v kapse zlepšovák, kterému se říká *cyklus*:

1. Zkontroluj, kolik je hodin.

2. Pokud se čas shoduje s nastaveným časem buzení, začni zvonit a čekej na zamáčknutí.
3. Pokud se čas neshoduje s nastaveným časem buzení, počkej sekundu a přejdi k bodu 1.

Nic na tom není – zkrátka jsme přesměrovali další akce tak, abychom je nemuseli všechny vypisovat. Instrukce zůstaly stejné a my si ušetříme námahu.



Obrázek 1.5. Schéma činnosti budíku

Budík je hotov, ale ještě bude důležité připomenout si dva výše zmíněné pojmy: co je vstupem a co výstupem z těchto instrukcí pro budík? To je naprosto nutné vždy vědět, a to ze dvou důvodů:

- Vstup zadáváme proto, že po „programu" něco chceme, např. kávu. Program – automat na kávu – musí umět se vstupem počítat. Musí ho umět využít, vědět, jak s ním naložit. Pokud máme na vědomí všechny možné vstupy, lépe se nám vytvářejí jednotlivé instrukce.
- Výstup je důležitý nejen pro ověření, zda program funguje správně. To by mělo být samozřejmostí – pokud zadáme citronový čaj, neměli bychom dostat horkou čokoládu. Výstup je ovšem důležitý i proto, aby program věděl, kdy skončit svou práci. Když nám automat vydá, co chceme, končí svou činnost a čeká na další instrukce – na další vstup.

Vstupem je tu samozřejmě nastavení přesného času buzení, ale pojďme k výstupu, ke kterému nás přivedl druhý bod a který je v našem příkladu možná o něco důležitější. Je totiž potřeba k ukončení celého procesu, celého programového *cyklu* tří po sobě jdoucích instrukcí. Výstupem je zvonění v daném okamžiku.

Pokud dojde k tomu, že se aktuální čas shoduje s nastaveným časem buzení, je třeba nejen začít zvonit, ale zároveň přerušit ten „věčný" cyklus tří instrukcí. Už není potřeba kontrolovat čas, budík zvoní a program pomalu končí. Nyní se čeká jen na to, až ten spící muž zvedne svou těžkou ruku a otráveně ji nechá spadnout na zvonící budík. To je konec programu.

Stavíme odrazový můstek programování ze čtyř základních kamenů

Jednotlivé příklady, ač se to z jejich všední podstaty nezdá, pomohly k ustavení základních zásad počítačového programování. Pojďme si je zopakovat:

- Je třeba šetřit místem, prostředky a silami – vždy přemýšlejme, jestli by se daná činnost nedala provést snadněji.

- Rozdělujme úkoly pro několik jednotlivců – není třeba, aby jeden hráč zastal práci celého týmu. Od čeho má spoluhráče? Každý z nich bude mít svůj malý, ale jasný, daný úkol.
- Při rozhodování, které je spolu s cyklickým prováděním instrukcí nejčastějším programátorským úkonem, je třeba myslet na všechny varianty, aby se nestalo, že hráč přiběhne k brance a nebude vědět, co má dělat.
- Při tvorbě cyklu je důležité nezapomenout na jeho ukončení. Když už jsme budík jednou zamáčkli, není nutné dál kontrolovat čas.

Tyto pouhé čtyři body jsou skutečně tím odrazovým můstkem, který potřebujeme aby se z nás stali programátoři. Je to základ, na kterém staví své velké programy i zkušení, ostřílení vývojáři.

Poznámka: Počítačovému programování se někdy říká *vývoj* a programátorům vývojáři, protože programy se buď píšou, nebo vyvíjejí.

Než se z můstku odrazíme a než se hlouběji ponoříme do tajů programování, popišme si nejprve, co se z nás potom stane. Ukažme si dopodrobna, jak budeme přemýšlet, až z nás budou kvalitní programátoři. Třeba si to pak celé rozmyslíme a knihu raději odložíme. Kdo ví?

Kdo je programátor

Programátor. Jak by se dal takový člověk popsat? V prvé řadě je trochu líný, protože jediné, co ho zajímá, je jak si usnadnit práci. Jako začínající programátoři se tedy musíme naučit být líní, protože lenost nás donutí pracovat tím nejefektivnějším způsobem. Lenost neznamená nepracovat! Lenost pouze zaručí, abychom nedělali zbytečnou práci, abychom se věnovali jen těm činnostem, které jsou k něčemu, které jsou užitečné.

Jak být líný, to asi nebude těžké se naučit, stačí zkrátka dvakrát měřit, dvakrát si věc nejprve rozmyslet, a potom se na to teprve vrhnout. Ale to není vše, lenost sama o sobě nic neřeší, pokud nevíme, jak přesně na to. Budou tu opět čtyři body, čtyři kroky, kterými při tvorbě programů vždy projdeme.

Dostali jsme úkol vytvořit česko-maďarský a maďarsko-český elektronický slovník. V jednotlivých bodech si řekneme, co všechno je potřeba při tvorbě provést.

Za prvé – Rozbor problému

Je samozřejmě dobré vědět, jak by měl slovník vypadat a jak by měl přesně fungovat. Je totiž nutné celou věc promyslet, abychom pak mohli najít to nejvhodnější řešení. Rozbor neboli analýza problému bude mít dvě části:

- jak bude hotový program vypadat,
- jak bude přesně fungovat.

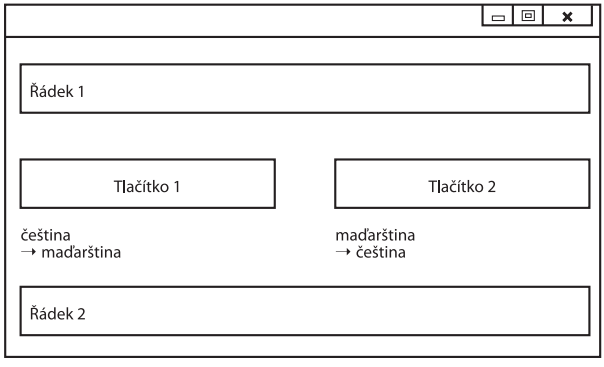
Zadání funkčnosti a vzhledu se při profesionálním programování konzultují se zákazníkem. Tedy s člověkem, který si u nás výrobu programu objednal a vyžaduje konkrétní podobu. Zákazníci mají často přehnané nápady a neuvědomují si, že některé věci prostě udělat nejde.

Poznámka: Možná jste si už všimli, že kromě programátora jsou ve hře další dva lidé. Jedním z nich je zákazník, který má konkrétní požadavky, a druhým je *uživatel*. Uživatel přistupuje k programu nezaujatě, nezná jeho funkčnost, ale chce, aby fungoval. Chce po automatu, aby vydal ten nápoj, který si přeje, chce po slovníku, aby mu přeložil ten pojem, který zadá.

Jak komunikovat se zákazníkem, jak společně analyzovat požadovaný úkol a jak potom vytvářet návrh řešení, jak reagovat na další a další zákazníkovy požadavky i po uzavření návrhu, o tom všem se dočtete v knize *Požadavky na software* (Computer Press, 2008, prodejní kód K1567). Najdete v ní i mnoho vtipných historek z praxe, protože ji napsal skutečný programátor, který má s klienty a jejich požadavky dlouholeté zkušenosti.

My tu zákazníka nemáme, a tak si můžeme sami říci, jak bude náš program vypadat a co bude umět. Buďme konkrétní a stanovme si nejprve vzhled slovníku (v závorkách vždy uvádíme označení daného objektu v programu, abychom se v tom všem vyznali a měli ve všem pořádek).

1. Slovník by měl vypadat jako klasické okno.
2. V okně bude prázdný řádek, do kterého se budou psát slova v češtině či v maďarštině (Řádek 1).
3. Pod Řádkem 1 budou dvě tlačítka:
   - překlad z češtiny do maďarštiny (Tlačítko 1),
   - překlad z maďarštiny do češtiny (Tlačítko 2).
4. Úplně dole bude řádek, ve kterém se bude objevovat překlad do jednoho z jazyků (Řádek 2).



Obrázek 1.6. Ukázka uživatelského rozhraní překladového slovníku

Důležité: Oknu, které jsme právě popsali, říkáme *uživatelské rozhraní*. Uživatelské rozhraní je všechno, co má před sebou uživatel. U automatu na kávu jsou to možnosti výběru nápoje, místo pro vložení mincí, místo, odkud padají vrácené drobné, displej, místo odběru nápoje, zkrátka veškerý viditelný vstup a výstup.

Nyní následuje přesný popis funkčnosti slovníku:

1. Uživatel musí zadat vstupní slovo, které chce přeložit.
2. Uživatel musí klepnout na jedno z tlačítek překladu.
3. Pokud uživatel klepnul na překlad z češtiny do maďarštiny, a pokud se v českém slovníku zadané heslo vyskytuje, vypíše se jeho maďarský protějšek na dolní řádek.
4. Pokud uživatel klepnul na překlad z češtiny do maďarštiny, a pokud se v českém slovníku zadané heslo nevyskytuje, vypíše se na dolní řádek text: „Lituji, toto slovo ne-umím přeložit".
5. Pokud uživatel klepnul na překlad z maďarštiny do češtiny, a pokud se v maďarském slovníku zadané heslo vyskytuje, vypíše se jeho český protějšek na dolní řádek.
6. Pokud uživatel klepnul na překlad z maďarštiny do češtiny, a pokud se v maďarském slovníku zadané heslo nevyskytuje, vypíše se na dolní řádek text: „Lituji, toto slovo neumím přeložit".
7. Nic jiného program nebude umožňovat.

Všimněme si, že popis funkčnosti problému automaticky předpokládá existenci dvou slovníků (česko-maďarského a maďarsko-českého), které jsou kdesi jinde, zvlášť – mimo uživatelské rozhraní. To bude při návrhu řešení potřeba zohlednit. Dále si všimněme posledního bodu – ten je tam pro úplnost, aby bylo jasné, že žádné další body už nepřijdou.

Při analýze problému bychom si měli pamatovat dvě zásady:

- dívejme se z nadhledu,
- buďme předvídaví.

Předvídaví budeme muset být i při návrhu konkrétního řešení, ale už zde musíme myslet na všechno, co by nás mohlo potkat. Proto jsme do jednotlivých bodů zařadili možnosti, při kterých se zadané slovo ve slovníku vůbec nevyskytuje. Uživatel totiž může zadat cokoli. Předvídavost a nadhled se tedy musí týkat především všech možných kombinací uživatelského vstupu.

Za druhé – Návrh funkčního a efektivního řešení

V předchozích odstavcích jsme popsali, jak bude program vypadat a jak by měl fungovat. Nyní je potřeba určit přesné instrukce, jak má vypadat slovník, jak má rozhraní se slovníkem komunikovat a podobné, více méně technické věci. Při rozboru problému se rozhodujeme nad jakousi filozofií celého programu, nyní je třeba řešit konkrétní, velmi detailní aspekty řešení.

Podívejme se nejprve na samotný slovník. To je přece jádro celého programu. Jak by měl slovník vypadat, aby se s ním co nejlépe pracovalo. Je tedy potřeba říci, že slovníky budou dva. To bude nejjednodušší, protože některá česká slova se v maďarštině nevyskytují, nebo mají naopak více možností překladu.

Oba slovníky budou vypadat jako obyčejné tabulky. V levém sloupci bude původní výraz a v pravém jeho překlad (či více možných překladů). Česko-maďarský slovník představuje tabulka 1.1, opačný směr překladu je v tabulce 1.2.

Tabulka 1.1. Česko-maďarský slovník (Slovník 1)

čaj	tea
čokoláda	csokoládé
den	nap
dobrý	jó
dobře	jól
dům	ház
guláš	gulyás
jezero	tó
káva	kávé
město	város
minerálka	ásványvíz
mléko	tej
počítač	számítógép
polévka	lévés
policie	rendőrség
radnice	városháza
slunce	nap
voda	víz
vzdělávat se	művelődik

Tabulka 1.2. Maďarsko-český slovník (Slovník 2)

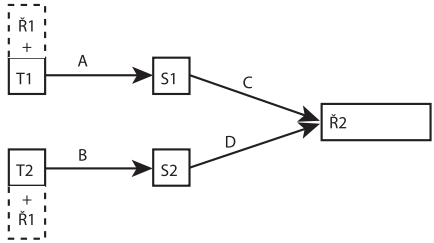
ásványvíz	minerálka
csokoládé	čokoláda
gulyás	guláš
ház	dům
jó	dobrý
jól	dobře
kávé	káva
lévés	polévka
művelődik	vzdělávat se
nap	den, slunce
rendőrség	policie
számítógép	počítač
tea	čaj
tej	mléko
tó	jezero
város	město
városháza	radnice
víz	voda

Jak je vidět u hesel slunce, den a nap, slovníky nejsou stejné. Proto jsou potřeba dva; pro každý směr překladu jeden. Nyní je nejdůležitější propojit oba slovníky s uživatelským rozhraním. Obrázek 1.7 přehledně znázorňuje, o co se budeme snažit:

V podstatě je třeba, aby komunikace probíhala ve čtyřech kanálech:

- **Kanál A:** Od tlačítka „Překlad z češtiny do maďarštiny“ (Tlačítko 1) ke Slovníku 1.

- **Kanál B:** Od tlačítka „Překlad z maďarštiny do češtiny“ (Tlačítko 2) ke Slovníku 2.
- **Kanál C:** Od Slovníku 1 k výstupnímu řádku uživatelského rozhraní.
- **Kanál D:** Od Slovníku 2 k výstupnímu řádku uživatelského rozhraní.



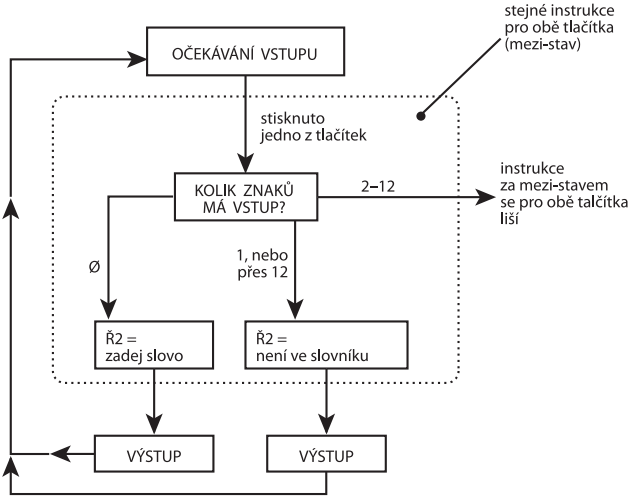
Obrázek 1.7. Komunikace uživatelského rozhraní se slovníky

Máme plně propojeno uživatelské rozhraní s oběma slovníky, nyní si jen pro úplnost zopakujme, co od programu čekáme, co přesně chceme, aby dělal:

- **Vstup:** Text v Řádku 1 a stisknutí jednoho z Tlačítek: 1, či 2.
- **Výstup:** Text, který se posléze objeví na Řádku 2.

Vstupem je ošidné slovo „text“, na které si musíme dát pozor. Je třeba zohlednit nejen situace, kdy zadané slovo není ve slovníku, ale i případ, kdy uživatel klepne na jedno z tlačítek, aniž by něco vepsal do řádku pro vstup.

Nicméně máme už vše potřebné, abychom mohli stanovit jednotlivé instrukce programu. Významné akce se budou vždy odehrávat po klepnutí na jedno z tlačítek, ale abychom nemuseli pro obě tlačítka vypisovat několik stejných instrukcí, vytvoříme jakýsi mezi-stav platný pro obě tlačítka. Mezi-stav znázorňuje obrázek 1.8.



Obrázek 1.8. Díky mezi-stavu bude možné psát několik instrukcí pro obě tlačítka

A nyní již konkrétní instrukce našeho slovníku:

1. Po spuštění program sám o sobě nedělá nic. Očekává uživatelský vstup.
2. V celém programu bude platit, že po výpisu jakéhokoli textu na Řádek 2 se vykonávání programu vrací do bodu 1.

3. Po klepnutí na jedno z tlačítek se spočítá, kolik znaků je na Řádku 1.
4. Pokud je počet znaků na Řádku 1 roven nule (pokud je řádek prázdný), vypíše se na Řádku 2 text: „Zadejte, prosím, slovo, které chcete přeložit".
5. Pokud je počet znaků menší než 2 (nejkratší slova obou slovníků, jó a tó, mají dva znaky) nebo je-li větší než 12 (nejdelší slovo obou slovníků je vzdělávat se, které má i s mezerou 12 znaků), vypíše se na Řádku 2 text: „Zadané slovo bohužel není ve slovníku".

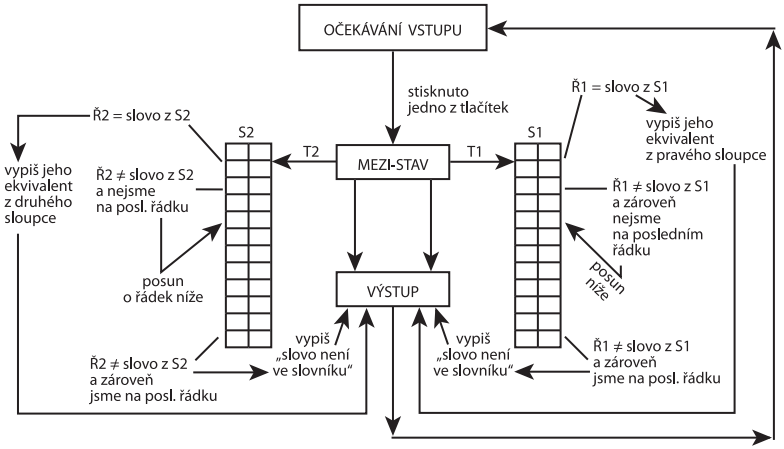
   (Tento bod je zde proto, aby se zbytečně nepřistupovalo ke slovníku, když už zde, v první chvíli, víme, že zadaný text se ve slovníku jistě nevyskytuje.)
6. Pokud má zadaný text alespoň 2 znaky a maximálně 12, vstupme za bránu tvořenou Mezi-stavem:
   - Pokud uživatel klepnul na Tlačítko 1, přistupme k prvními řádku Slovníku 1.
   - Pokud uživatel klepnul na Tlačítko 2, přistupme k prvnímu řádku Slovníku 2.

   Ať už je vykonávání programu v kterémkoli z obou slovníků, je třeba hledat v levém sloupci slovníku zadané vstupní slovo, a to hezky postupně (zde uvidíme pěknou ukázku cyklického provádění instrukcí, nesmíme však zapomenout na nějakou instrukci ukončovací).
7. Pokud se vstupní text shoduje se slovem na řádku levého sloupce, vypíše se na Řádek 2 slovo na stejném řádku, ale v pravém sloupci. Jinak pokračujeme na následující bod.
8. Pokud se vstupní text neshoduje se slovem na řádku levého sloupce a zároveň platí, že nejsme na posledním řádku slovníku, posune se vykonávání programu o jeden řádek níže a pokračujeme bodem 7.
9. Pokud se vstupní text neshoduje se slovem na řádku levého sloupce a zároveň platí, že jsme na posledním řádku slovníku, vypíše se na Řádek 2 text: „Zadané slovo bohužel není ve slovníku".

Protože jsme bodem 9 vyčerpali všechny možnosti a po každé z nich se vypsal nějaký text na Řádek 2, vrací se vykonávání programu do bodu 1, aby vyčkával dalšího vstupu. Celý postup zachycuje schéma na obrázku 1.9.

Poznámka: Nemáme zde možnost ukončení celého programu, nějakou nabídku Soubor a příkaz Zavřít, ale to jen z toho důvodu, že to není nutné k pochopení programátorských technik. Při reálném programování by se s touto možností muselo také počítat.

Pro návrh funkčního řešení si zapamatujme jedinou zásadu: Přemýšlejme o detailech. Nesmíme zapomenout na žádnou možnost vstupů a výstupů programu, je třeba veškerý chod programu popsat do posledního detailu.



Obrázek 1.9. Schéma řešení pro náš slovník

Za třetí – Realizace řešení

Ve chvíli, kdy máme hotový konkrétní a bezchybný postup řešení, nezbývá než ho sdělit počítači. A nebo – pokud by byl cílem automat na kávu – vyrobit automat. Zkrátka realizovat řešení do konkrétní podoby hotového, funkčního programu.

V případě počítačového programování je realizací myšleno vytvořit program tak, jak jsme si ho popsali při prvním kroku. Jak ale sdělit naše myšlenky počítači? Naší řečí je čeština, ale bůhví, jak mluví a čemu rozumí počítač. Jak se s počítačem dorozumět, aby poslouchal naše příkazy, o tom se nejprve zmíníme na konci této kapitoly a hlouběji se tím budeme zabývat ve druhé kapitole.

Konkrétní realizace, to je krok, který ostříleným programátorům vlastně ani nezabere tolik času, co oba kroky předchozí. Ale pro nás bude realizace základním kamenem. Je totiž potřeba naučit se používat tzv. *programovací jazyk*, pomocí něhož budeme s počítačem komunikovat. A naučit se programovací jazyk, to bude stejně těžká práce, jako naučit se anglicky nebo německy.

Nicméně – pro výuku jsme vybrali jeden z vůbec nejjednodušších programovacích jazyků, který je navíc k výkladu základních programovacích principů nanejvýš vhodný, protože přesně k tomu ho Niklaus Wirth vytvořil. Jde o jazyk *Pascal*, který sice v reálné praxi nevyužijeme, ale přechod k jiným jazykům bude velmi snadný. Je to jako umět výborně italsky – přechod např. ke španělštině (která patří do stejné jazykové rodiny) už nebude takový problém jako bez znalosti italštiny.

V dalších kapitolách poznáme, kterou zásadou bychom se měli řídit při realizaci programu a jakou vlastnost nesmíme postrádat: buďme pečliví. Tvořme program od jedné instrukce ke druhé, od jedné funkce programu k další, abychom na nic nezapomněli a nic nezanedbali. Počítač je přesným zařízením a sebemenší chyba se projeví buď podivným chováním programu a nebo kompletní nefunkčností.

S tím souvisí, že zvláště u složitějších programů třetím krokem – realizací – naše práce nekončí. Důležité je program otestovat a vyladit jeho funkčnost.

Za čtvrté – Sledování funkčnosti a vyladění řešení

Na některé chyby z nepozornosti nepřijdeme hned, protože program na první pohled funguje správně. V případě česko-maďarského slovníku můžeme zkusit přeložit třeba slovo „jezero". Na Řádku 2 se vypíše slovo „tó" a my jsme spokojení. Vše je jak má být, jezero se maďarsky skutečně řekne tó.

Navíc jsme podle zásady třetího kroku byli pečliví, takže by neměl být problém. Jenže nikdo z nás není neomylný a na skrytou chybu přijde často až uživatel. Proto je třeba i fungující program trpělivě zkoušet a testovat – nejlépe všechny funkce, které má.

Poznámka: Dalším člověkem, který je často u velkých programátorských projektů, je tester. Ten nedělá nic jiného, než že zkouší jednu funkci programu za druhou a dívá se, jestli vše funguje, jak má.

A co když to vůbec nefunguje

Co když se nám po všech třech krocích a při všech dodržených zásadách nepodařilo přivést stroj k funkčnosti a slovník k překladům? Co když se po klepnutí na jedno z tlačítek na Řádku 2 nic neobjeví; co dělat v případě, že se po vhození správného obnosu peněz a zmáčknutí tlačítka pro kávu nic nestane?

Káva nebude a peníze nám už nikdo nevrátí, ale je dobré, když na problém přijde ještě v procesu tvorby programu sám programátor nebo tester, a ne až uživatel. Řešením je jediná rada: buďme trpěliví a pokusme se systematicky projít všechny instrukce realizovaného programu.

Nejčastějšími chybami jsou totiž právě překlepy při realizaci programu ve třetím kroku, ale i v krocích předchozích jsme mohli něco zanedbat. Proto je dobré při hledání chyby postupovat pozpátku:

1. Nejprve projít hotový program a pokusit se objevit chybu v realizaci.
2. Pokud jsme v bodě 1 žádnou chybu neobjevili, je třeba se blíže podívat na druhou fázi tvorby programu – Návrh řešení. Není třeba problém v definici vstupu a výstupu?
3. Pokud ani ve druhém kroku potíže nenacházíme, mohla by být zrada už v samotné představě programu. Může se stát, že uživatel měl nároky, které šly navzájem proti sobě, a při návrhu řešení ani při realizaci si toho nikdo nevšiml. I to se u větších a složitějších zakázek stává.

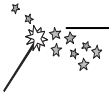
Závěrem k práci programátora

Protože před námi leží pouze menší projekty, protože se budeme zabývat spíše jednodušími programy, budeme sami postupně vykonávat činnosti všech lidí zúčastněných na velkých zakázkách. Konkrétně zastaneme činnosti:

- *zákazníka,* který si určuje, jak by měl program fungovat a jak vypadat,

- *analytika,* jenž musí program konkrétně rozebrat (analyzovat) a navrhnout funkční a efektivní řešení,
- *programátora,* který projekt konkrétně zrealizuje, tzn. napíše ho tak, aby mu rozuměl počítač,
- *testera,* nejtrpělivějšího člena týmu důležitého pro sledování funkčnosti a vyladění řešení.

To vše zvládneme při každém úkolu my sami. Bude potřeba si zejména zvyknout na úkoly před samotnou realizací, před psaním programu. Jedině systematickým rozborem a jedině přesnou definicí všeho, co by měl program umět a mít, určením vstupu a výstupu apod. se v závěru dočkáme bezchybného řešení, které bude čím dál snazší testovat.



Tip: Každá následující část knihy se věnuje jedné z výše popsaných činností. Nikdy se podkapitoly nevěnují analytické i realizační úrovni zároveň. Bude dobré v každé části si uvědomit, komu daný text zrovna patří. Jak jsme již řekli, většinou budou kapitoly určené konkrétní programátorské realizaci, ale protože bez přemýšlivé analýzy se žádný programátor neobejde, bude se většina textu této kapitoly věnovat abstraktnějším návrhům.

Příkazy

Už při návrhu funkčního řešení jsme zadávali programu úkoly. Těmto úkolům pro naše programy budeme běžně říkat *příkazy*, ale všimněme si, že kromě jasných úkolů, jako „vydej kelímek plný kávy", „přihraj míč spoluhráči před bránou", „vypiš slovo ze slovníku" apod. byly naše návrhy plné různých odboček, rozcestí a cyklických činností.

Je třeba si ujasnit, co to ten příkaz vlastně je. V této části kapitoly uvidíme jednak to, proč jsou příkazy tak důležité, jaké by měly příkazy být, aby byly co nejužitečnější, ale také to, že příkazy samy o sobě jsou téměř bezmocné. Povíme si o tom, co je dobré s příkazy kombinovat, aby byly užitečnější.

Co je to příkaz a proč je tak důležitý

Příkazem je myšlen, jak už jsme řekli, konkrétní úkol nějakému programu. Je to jasná, jednoznačná instrukce. Uveďme si hned několik příkladů, co všechno by mohlo být příkazem (i mimo počítačové programy):

1. kopni míč směrem k bráně,
2. vypiš slovo „jezero" na Řádek2,
3. zapni televizi,
4. uvař večeři,
5. dones mi pivo.

Všimněme si jednoznačnosti, která je u příkazů nutná. Tabulka 1.3 ukazuje nesprávné příkazy, důvod jejich chybnosti a také návrh na jejich opravu.

Tabulka 1.3. Chybné příkazy, popis chyby a návrh na opravu

Chybný příkaz	Proč je chybný	Správný příkaz
Teď neuklízej	A kdy by tedy měla uklízet? Co má teď dělat? Příkaz není jednoznačný.	Ukliď tady až po fotbale, do té doby máš volno
Kdy přijdeš?	Otázka samozřejmě není příkazem.	Do půlnoci buď doma
Venku prší, co kdybys zůstala doma	Informace ani podmiňovací způsob také nemůže být příkazem.	Zůstaň doma a ohřej mi polévku
Přepni mi TV program	Pokud máme více než dva kanály, je úkol nejednoznačný.	Přepni mi televizi na Dvojku
Zařiď, aby bylo o víkendu pěkně	Je třeba zadávat reálné úkoly, které náš svět a systém jeho fungování umožňují.	Na víkendový výlet s sebou vezmi deštník
Přelož slovo „mléko“ do finštiny	Tentýž případ jako předchozí – pokud máme jen česko-maďarský slovník, nepůjde to.	Přelož slovo „mléko“ do maďarštiny

Při zadávání příkazů je potřeba myslet na dvě věci:

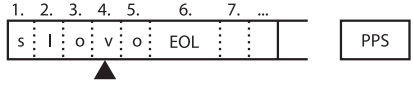
- zadávejme jasné a co nejjednodušší úkony,
- jednoduché úkony se dají spojovat do skupin, ty pak tvoří složitější příkazy.

Jak se dobrat co nejjednodušších příkazů

Když jsme výše popisovali instrukce překladového slovníku, nedalo se prostě říci, „tady máš dva slovníky, přelož mi zadané slovo“. Museli jsme do detailu přesně vypsat, co má daný program dělat. Museli jsme rozebrat úkoly až na ty nejjednodušší instrukce, jako třeba „Po klepnutí na jedno z tlačítek spočítej, kolik znaků je na Řádku 1“.

To je celkem jasný úkol, i když by se dal stále zjednodušovat. Např. tak, že bychom si vytvořili stroj na počítání znaků na řádku. Budou k tomu stačit jen tři věci:

- Řádek, na kterém budeme počítat napsané znaky,
- čtecí hlava, která se bude po Řádku pohybovat,
- paměť pro součty čísel, tj. místo, kam budeme ukládat čísla. Nazveme ho PPS (Paměť Pro Součty).



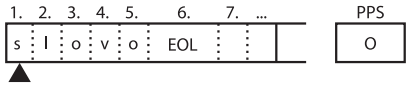
Obrázek 1.10. Schéma stroje na počítání znaků

Můžeme předpokládat, že konec slova (začátek prázdného zbytku řádku) je znakem odlišným od mezery. Tak je tomu běžně i v reálném počítačovém programování. Tento znak se nazývá jednoduše *End of line* (EOL, konec řádku).

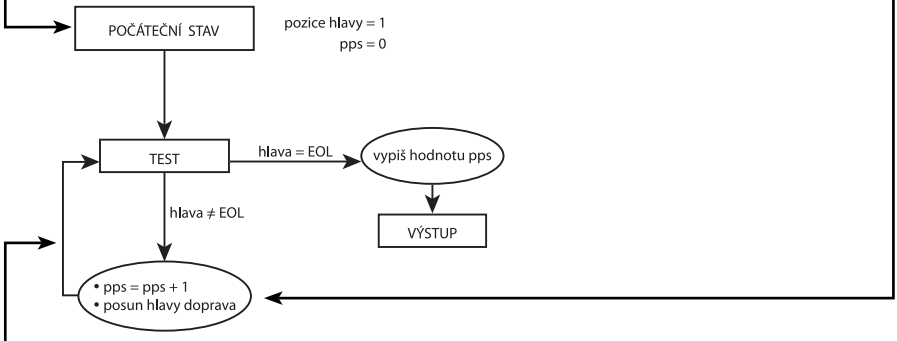
Důležité: Při návrhu instrukcí pro stroj na počítání znaků vidíme body 1–6. Jednotlivým bodům výkonu programu (jakéhokoli) se obyčejně říká *stavy*. I my jim tak budeme říkat, takže náš program má celkem 6 stavů, přičemž ten první nazveme *počátečním stavem*.

Série instrukcí pro stroj na počítání znaků by mohla vypadat nějak takto:

1. V paměti pro součet (PPS) musí být na začátku programu nula. Stroji tedy dáme příkaz: Do PPS napiš číslo 0. A dále zařaď čtecí hlavu na první znak Řádku.
2. Pokud je znak, na který ukazuje čtecí hlava, znakem EOL, běž do stavu 3. Jinak běž do stavu 4.
3. Výstupem je hodnota paměti PPS. Konec programu.
4. Posuň hlavu o jeden znak doprava.
5. Do PPS zapiš hodnotu PPS + 1 (neboli: V paměti PPS přičti jedničku).
6. Pokračuj do stavu 2.



Obrázek 1.11. Počáteční stav stroje na počítání znaků (PPS=0, hlava je na pozici 1)



Obrázek 1.12. Schéma série instrukcí stroje na počítání znaků

To je vše. Nezbývá než ukázat si, že máme pojištěny všechny důležité okolnosti: vstup, výstup a ukončení programu:

- Vstupem může být nejen prázdný řádek, ale třeba i dvě slova, protože mezera mezi nimi není příčinou k ukončení programu.
- Výstupem je vždy číslo, a to konkrétně počet znaků na Řádku.
- Program skončí, jakmile hlava přečte konec řádku, EOL, a vypíše aktuální hodnotu paměti pro součet, PPS.

Budeme teď chvíli testerem a vyzkoušíme, jestli program funguje. Nemůžeme vyzkoušet všechny varianty – jednak proto, že jsme neurčili maximální délku řádku, ale i při standardní délce 256 znaků bychom museli zkoušet alternativy všech písmen na všech pozicích. To by bez počítače trvalo moc dlouho, proto si jen pro demonstraci předvedeme, jak stroj funguje pro zadaný text „Já i ty“:

1. Stroj přejde do tzv. *počátečního stavu*: PPS = 0 a hlava čte první znak řádku.

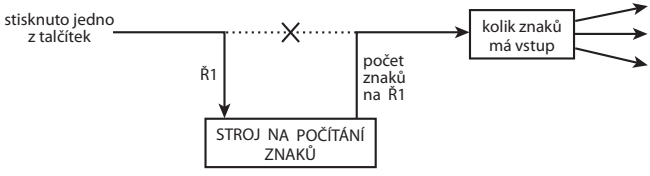
2. Nyní jsme ve vykonávání stavu 2: Hlava čte písmeno „J“, to není EOL, takže přejdeme do stavu 4 výkonu programu.
3. Tam se hlava posune doprava, v dalším kroku se k PPS přičte jednička, takže nyní je hodnota v paměti PPS rovna 1, a přejdeme zpět do stavu 2.
4. Hlava čte písmeno „á“, to opět není EOL, proto se hlava posune doprava, k PPS se přičte jednička (PPS je nyní aktuálně 2) a vracíme se ke čtení ve stavu 2.
5. Hlava čte mezeru, což není EOL, proto se hlava opět posouvá doprava, k PPS se přičte jednička (PPS je aktuálně rovna 3) a výkon programu se vrací ke druhému stavu.
6. Čtecí hlava vidí písmeno „i“. Protože to „i“ není EOL, přejde hlava doprava, PPS se zvýší o 1 (nyní PPS=4) a program opět pokračuje stavem 2.
7. Hlava čte mezeru, PPS = PPS + 1, tedy PPS = 5, hlava se posune dále doprava, a čteme znovu.
8. Písmeno „t“ není rovno EOL, proto PPS = PPS + 1 = 6, hlava se posunuje dál.
9. Nově přečtené písmeno „y“ není EOL, takže PPS = 7, hlava se posunuje doprava.
10. Hlava nyní čte znak konec řádku, EOL, proto vykonávání programu pokračuje do stavu 3.
11. Výstupem je hodnota PPS = 7 a program končí.

Instrukce (stavy) stroje na počítání znaků by se daly stále zjednodušovat, ale my jsme si už jedno zjednodušení ukázali. Na počátku totiž stál příkaz „Spočítej, kolik znaků je na řádku“ a tento příkaz jsme rozepsali do šesti stavů.

Spojování programů

Nyní se ovšem podívejme na věc z druhého úhlu. Představme si, že máme téměř hotový program česko-maďarského a maďarsko-českého slovníku, chybí už pouze nějaký menší program, který by měl jako výstup počet znaků na řádku. Je třeba napsat takový program – to jsme udělali (v šesti stavech) – a zařadit ho do hotového programu slovníku.

V okamžiku, kdy program slovníku potřebuje znát počet znaků na řádku, *předá* stroji na počítaní znaků ten daný popsaný řádek jako vstup programu. Toto je důležitý proces, takže ho pro jistotu demonstrujeme na obrázku 1.13.



Obrázek 1.13. Schéma předávání informací: program slovníku předá stroji na počítání znaků (jako jeho vstup) řádek, o který mu jde, a stroj mu za chvíli vrátí počet znaků na daném řádku. A program slovníku pak může dál pracovat (přesunovat se do dalších svých stavů).

O tomto principu jsme již mluvili při popisu fotbalového utkání. Každý hráč (každý malý program) pomůže svou troškou k úspěchu celého týmu. Jeden hráč má na starosti přihrát, druhý vstřelit branku. Stejně tak naše programy – stroj na počítání znaků je tzv. *podprogramem*. Podprogram v programu je jedním hráčem v týmu, má konkrétní úkol, ten splní a o nic víc se ho tým celého programu neprosí. Na jiné úkoly má zase jiné podprogramy.

Pokud se ty nejjednodušší příkazy naučíme spojovat do vyšších celků, ovládneme jednu z nejdůležitějších dovedností dobrého programátora: umět rozdělit činnosti programu do několika dílčích a pro tyto dílčí činnosti napsat zvláštní (pod)program. I program se může stát podprogramem, pokud jeho vstup a výstup využijeme při nějaké další činnosti.

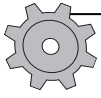
Příkazy samy o sobě nejsou všemocné

O příkazu jsme mluvili jako o jasné a jednoznačné instrukci. Příkazem není otázka ani informace. Příkazem je nějaká rozkazovací věta: napiš, přičti, přejdi do stavu 2 apod. Jak už jsme mnohokrát viděli a jak už jsme se i zmínili, naše programy jsou plné nejrůznějších otázek. „Pokud je znak, který hlava čte, ten a ten, pak…". To přece není příkaz.

Kromě příkazů budeme v našich programech potřebovat několik dalších typů instrukcí, bez nichž by se samotné příkazy v programech neobešly.

Informace

Vzpomeňme na příklad počítání znaků. Celé řešení bylo založeno na tom, že můžeme pracovat s aktuálními informacemi. Jednak jsme mohli zjišťovat, co čtecí hlava zrovna přečetla, ale především jsme měli neustále uložen určitý počet znaků. K uložené hodnotě jsme se mohli vracet a přičítat k ní další hodnoty. Vypadá to jako banalita, ale práce s aktuálními hodnotami patří mezi základní prostředky programování. Více se k ukládání známých hodnot dozvíme v kapitole 3.



Řešení problému: Jak zajistit, aby automat na kávu spolehlivě hlásil nedostatek horké vody pro přípravu nápoje? Jedině tak, že po každém vydaném nápoji zkontroluje aktuální obsah nádrže. Aktuální informace jsou při programování nutností.

Otázky, odpovědi a reakce na odpověď

Potřebujeme-li program rozvětvit a určit další běh programu obou (případně všech) větví, víme, jak to provést. Uložení aktuální informace je důležité, ale neobejde se bez onoho *zjištění* přesného stavu věcí, aby program věděl, kudy běžet dál. Již několikrát jsme se s touto konstrukcí setkali. Je to jednoduché:

1. Na začátku stojí zjišťovací slovo *Pokud*.
2. Potom je třeba položit otázku. Otázka musí být položena tak, aby byla odpověď *ano*, nebo *ne*. Např.: Je přečtené písmeno EOL? Ano, nebo ne? Je před soupeřovou brankou domácí útočník? Ano, nebo ne?
3. Zaznamenáme odpověď.

4. Podle odpovědi je nutná reakce. A zde je již místo pro čistý, jasný příkaz.

Celý konstrukt pak vypadá takto: *Pokud* je přečtené písmeno EOL, *Pak* zobraz výsledek sčítání a skonči. *Jinak* (tzn. pokud přečtené písmeno není EOL), *Pak* posuň hlavu, přičti jedničku a přejdi do stavu 2. O práci s tzv. rozhodovacími konstrukcemi se více dozvíme v kapitole 4.

Opakovaná činnost

Tento zásadní prvek veškerého programování vystihuje staré dobré přísloví: Tak dlouho se chodí se džbánem pro vodu, až se ucho utrhne. Při programování ho můžeme přeformulovat: „Choď pro vodu tak dlouho, dokud se ucho neutrhne." Zde je řečeno vše podstatné. Je nutné:

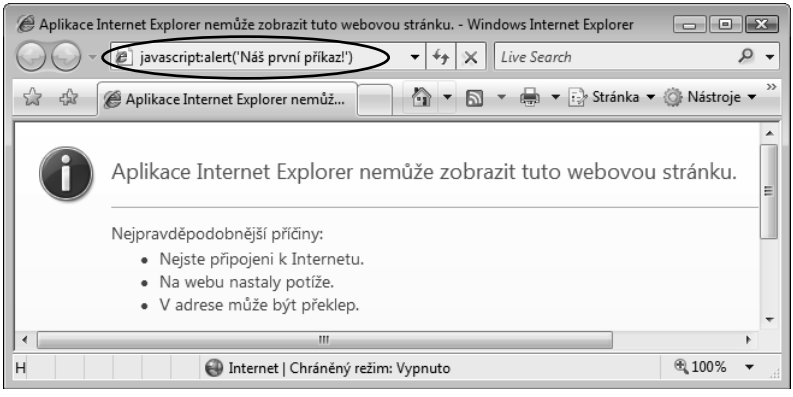
- jasně specifikovat činnost, kterou chceme provádět (Choď se džbánem pro vodu a nalévej ji sem, do našeho velkého džberu),
- zadat ukončovací podmínku (dokud se ucho neutrhne nebo dokud náš džber není úplně plný).

V kapitole 5 se můžeme těšit na hlubší prozkoumání opakovaných činností, ale už zde je jasné, že všechny čtyři prostředky (včetně příkazů) se vzájemně doplňují: pomocí otázek zjišťujeme aktuální informace, ty pečlivě ukládáme a v cyklicky opakovaných činnostech se na ně pak znovu ptáme, abychom přikazovali programu další činnost.

Náš první příkaz pro počítač

Na tomto místě je dobré si uvědomit, že zatím se brouzdáme širým polem teorie. Máme plné právo se ptát, kdy už konečně něco přikážeme počítači. Vzhledem k tomu, že nemáme žádné znalosti jazyka, pomocí kterého bychom s počítačem mohli komunikovat, bude to těžké.

Přesto si můžeme ukázat příkaz, který funguje. Nebudeme mu rozumět, ale uvidíme výsledek. To bude tou největší odměnou po všech pěkných i celkem složitých příkladech, u kterých



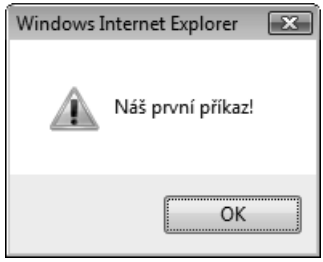
Obrázek 1.14. Internetový prohlížeč Internet Explorer s naším prvním příkazem

však výsledek není vidět. Žádný česko-maďarský překladač tu nemáme. Všechno bylo jen na papíře. Proto nyní postupujme podle následujících kroků k našemu prvnímu příkazu:

1. Spusťme libovolný internetový prohlížeč. Internet Explorer, Mozilla Firefox, Operu či Safari. Kterýkoli. Není nutné, abychom byli připojeni k Internetu, bohatě postačí, když se prohlížeč prostě spustí.
2. Do řádku pro internetovou adresu napíšeme magickou formuli, která se stane naším prvním příkazem (viz obrázek 1.14.):

   ```
   javascript:alert('Náš první příkaz!')
   ```

3. Stiskneme klávesu Enter, jako kdybychom chtěli přejít na zadanou internetovou adresu.
4. Mělo by se objevit okno jako na obrázku 1.15.
5. Klepneme na tlačítko OK nebo stiskneme klávesu Enter a celé okno zmizí.



Obrázek 1.15. Výsledek našeho prvního příkazu

Co jsme to vlastně udělali

Jazyk, kterým jsme s počítačem právě mluvili, se jmenuje JavaScript a používá se na programování webových stránek. Po slovu `javascript` následovala dvojtečka a slovo `alert` následované závorkou, v níž jsou dva apostrofy a mezi nimi text. To celé se dá vyložit jako zpráva počítači: „Budu na tebe mluvit JavaScriptem: vytvoř okno, ve kterém bude text a tlačítko OK. Text zadám do závorky mezi apostrofy."

Tip: Pokud mezi apostrofy v závorce napíšeme jiný text, objeví se ve výsledném okně. Zkusme napsat mezi apostrofy něco jiného a sledujme výsledky.

Algoritmus

Věřte nebo ne, počítačové programování má velmi blízko k vaření. Pokud se totiž rozhodneme vařit nebo péct něco složitějšího, potřebujeme návod, kuchařku. Díky receptu víme, co udělat nejprve, co potom, kolik přimíchat mouky atd. Díky návodu nevynecháme jediný krok, víme, jak dlouho povařit, jak dlouho podusit, na kolik stupňů nastavit teplotu a jakou přísadu kdy přimíchat.

Podrobný návod, postup řešení, recept, to je *algoritmus*. Podle algoritmu, který jako programátoři vytvoříme, se pak chová i program. Algoritmus určuje chování programu. Algoritmus je přesným schématem, které vše řídí a podle něhož se určuje činnost celého naprogramovaného zařízení (např. automatu na kávu).

Čas od času v této knize využijeme pro výklad šikovných matematických nástrojů, důmyslných zařízení a výborných vynálezů jako např. sčítání (+), odčítání (-), rovnítek (=) nebo menšítek (<), a sáhneme dokonce k nejrůznějším zástupným slovům či znakům, nechvalně známým z matematických rovnic, jako jsou písmena *x* a *y*.

Nebudeme je brát do ruky příliš často, ale někdy se jejich použití vyplatí. Není totiž na světě nic přesnějšího než matematika a její nástroje. A protože počítače se řídí výhradně matematickými, tedy velmi přesnými zásadami, je potřeba si zvyknout na rovnítka a zástupné znaky už teď na začátku. Jako programátoři se jimi budeme zabývat dennodenně.

Ale nyní už konečně k receptům zvaným algoritmy. Za okamžik se dozvíme, co to algoritmus přesně je a jak ho co nejlépe vytvořit, aby k něčemu byl.

Poznámka: Slovo *algoritmus* pochází ze jména perského matematika a astronoma, který je tvůrcem systému arabských číslic a základů algebry. Žil na přelomu 8. a 9. století n.l. a jmenoval se Abú Abd Alláh Muhammad Ibn Músá al-Chórezmí Abú Dža'far. To znamená: Otec Abdulláha, Mohameda, syn Mojžíšův, pocházející z města Chwārizm (Chórézm). A kde že se skrývá slovo algoritmus? Jeho jméno se někdy zkracuje na al- Chwārizm. A zde už je původ zřetelnější.

Co musí splňovat algoritmus a proč

Algoritmus není ledajakým postupem. Jak by se nám líbilo, kdybychom měli postupovat podle takového receptu?

1. Do hrnce přidejte ke všemu, co v něm už je, půl kila mouky.
2. Rozpalte troubu na 180°.
3. Do hrnce přidejte 5–15 vajec (podle chuti).
4. Snižte teplotu trouby na 90°.
5. Přidejte do hrnce to, na co máte právě chuť a zalijte vodou.
6. Troubu úplně vypněte.
7. Položte hrnec na plotnu a za stálého míchání přiveďte k varu.
8. Vařte 30 minut.
9. Těsně před varem přisypte tymián, hořčici, kmín, papriku a bazalku.
10. Raději to nejezte.

Trochu to připomíná dort, který si vařili pejsek s kočičkou. Postup je nelogický, chaotický a nejednoznačný. Aby se postup stal algoritmem a byl k něčemu užitečný, musí splňovat několik pravidel.

Konečnost

Co je špatně na těchto příkazech:

- míchejte polévku,
- choď se džbánem pro vodu,
- nalévej do kelímku horký čaj,
- pokud x = 12, násobte x dvěma, dokud má x kladnou hodnotu.

Všechny příkazy by probíhaly do nekonečna. Správný algoritmus (který je vlastně posloupností příkazů) by měl sám o sobě zaručovat, že nedojde k nekonečnému provádění. Konečnost zaručíme vhodnou ukončovací podmínkou:

- vařte polévku 10 minut (celý proces se tedy za 10 minut ukončí) a během varu ji míchejte,
- choď se džbánem pro vodu, dokud se ucho drží džbánu,
- nalij do kelímku přesně deci a půl horkého čaje,
- pokud platí, že x = 12, násobte x dvěma, dokud má x hodnotu menší než 100.

Některé počítačové programy jsou paradoxně založeny na tom, že nikdy nekončí. Ovšem to jsou speciální případy. Nejznámějším z nich je operační systém. Když zapneme počítač, načte se systém, který nemá žádnou ukončovací podmínku. Pokud sám uživatel počítač nevypne, běží systém skutečně donekonečna.

Jednoznačnost

Protože programování probíhá na počítači, musíme být přesní a jednoznační. Nemůžeme dávat počítači na výběr. Počítač není subjektivním tvorem, který by si podle nálady vybral jednu z možností. Nezná výrazy *asi*, *zhruba* nebo *několik*.

O jednoznačnosti jsme již mluvili u příkazů, takže pochopitelně platí i pro celé algoritmy. Jednoznačnost zaručuje, že v každém kroku, v každém stavu algoritmu je jasně dané, jaká činnost se v daném stavu provádí a jak pokračovat dál, tedy do kterého stavu a s jakými informacemi se posunout.

Obecnost

O univerzálnosti, neboli obecnosti jsme ještě nemluvili. Jde zhruba o to, aby byl algoritmus vhodný pro celou třídu obdobných problémů:

- aby algoritmus nebyl postupem na výrobu jednoho psacího stolu, z těchto pěti prken, ale aby byl postupem na výrobu jakéhokoli dřevěného stolu,
- aby postup neurčoval kroky potřebné ke spočítání znaků ve slově „Programování“, ale aby určoval kroky potřebné k součtu znaků jakéhokoli slova. Ať zadáme kterékoli slovo či větu, algoritmus prostě spočítá znaky a řekne nám výsledek,
- aby postup nepopisoval postup výpočtu 3^4, ale aby řešil obecný, univerzální problém x^y,
- aby každý fotbalový hráč měl předem jasno, co má dělat, ve všech situacích, které by mohly nastat, aby nepočítal hloupě s tím, že mu spoluhráč pokaždé přihraje na hlavičku, ale aby bral utkání vcelku – jako soubor mnoha tisíců možností; a v každé z nich aby věděl, co dělat.

Přece jen jsme se o této vlastnosti algoritmu už zmínili. Nejde přece o nic jiného než o množinu vstupních hodnot. Ale to už přece známe. Algoritmus zkrátka musí fungovat pro všechny možnosti vstupu. To je to celé.

Nutnost výstupu

Zde už také víme, oč jde. Algoritmus musí vždy *vrátit* nějaký výsledek. Co to znamená, že program, respektive algoritmus, vrací výsledek: to, co algoritmus vrací, je jeho výstup.

Nejlépe si to lze představit na příkladu automatu na kávu: vhodíme mince, zvolíme nápoj a automat nám vrátí nápoj, který jsme si přáli (protože přesně ten je výstupem algoritmu). Stejně tak je tomu v našem příkladu slovníku: zadáme slovo a stiskneme tlačítko (vstup) a program nám vrátí překlad (výstup). Víme, že výstupem může být i chybové hlášení, např. nápis, že vhozené mince nestačí.

Tento bod říká pouze tolik, že ať už je výsledkem cokoli, algoritmus by nám to měl dát vědět. Třeba i chybovým hlášením.

Jednoznačnost, univerzálnost, konečnost a nutnost vráceného výsledku. Kromě těchto čtyř základních vlastností se někdy uvádějí některé další. Jedním z požadavků, které se kladou na algoritmus, je *správnost výsledku*. Pokud algoritmus počítá mocninu dvou čísel, měl by vracet správný výsledek. Pokud je výstupem algoritmu zvolený nápoj, měl by vracet ten nápoj, který si uživatel/zákazník zvolil.

K čemu se hodí nesprávně fungující algoritmus

Požadavek *správnosti* už nepatří mezi základní, protože *nesprávnost* výsledku by mohla být záměrem programátora. Pokud si programátor usmyslí a požaduje po automatu, aby vracel čaj, když zákazník požaduje horkou čokoládu, pak jestliže se tak automat skutečně chová, pracuje zcela správně podle programátorových instrukcí.

V tomto ohledu se programování blíží k umělecké činnosti. Pokud je programátor dostatečně zručný, může vytvořit jakýkoli stroj; funkčnost stroje je omezena pouze technickými detaily a fyzikálními či chemickými zákony. Vše ostatní je v jeho rukou. Pokud zachová čtyři výše popsané vlastnosti, tvořivosti se meze nekladou. Často sice podobu a funkčnost programu (tím potažmo i algoritmus) určuje zákazník, který programátorovi za jeho práci platí, ale pokud se programátor rozhodne tvořit zdarma, může vše, co sám dokáže.

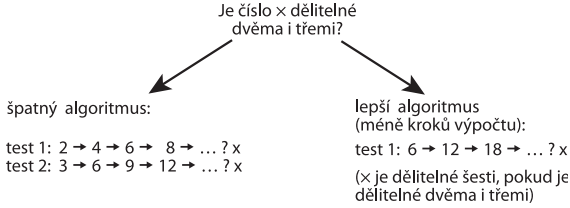
Může si z uživatele utahovat a vracet podivné výsledky, může naprogramovat přehrávač filmů, který přehraje film pozpátku a může také tvořit počítačové viry; ty potom dostat na cizí počítače a vyřadit je nějakým způsobem z provozu. I počítačové viry jsou jen programy, které se chovají přesně podle algoritmu. To, že pak náš počítač, pokud je zavirovaný, nepracuje správně, neznamená, že by byl algoritmus chybný. Právě naopak. Kritérium správnosti je tedy poněkud ošemetné.

Daleko zajímavější je další vlastnost, která se někdy uvádí dokonce jako jedna ze základních požadavků na algoritmus. Je to jeho efektivita, resp. *složitost algoritmu*.

Složitost algoritmu

Složitostí nemyslíme, jak moc je daný problém nebo algoritmus na jeho vyřešení složitý. To by totiž byla dost subjektivní záležitost. Pro každého z nás je to jinak. Někdo algoritmus

prokoukne hned, jiný potřebuje trochu času, aby se mu dostal na kloub. Algoritmickou složitostí však míníme zcela objektivní, měřitelnou veličinu.



Obrázek 1.16. Snažíme se snižovat složitost (graf dobrého a špatného algoritmu)

Složitost určuje časovou a prostorovou zátěž na počítač. Složitost je doba a prostor, který si konání algoritmu vyžádá. Hned si vysvětlíme, o co přesně jde, budeme se však zabývat pouze složitostí časovou. Prostorovou složitostí se zabývat nemusíme; jednak proto, že se definuje úplně analogicky k časové, ale hlavně proto, že se týká operační paměti, kterou algoritmus využívá. Dosud se pohybujeme na suchu, bez počítače, zatím tedy není nutné zabývat se prostorem.

Časová složitost

Protože jednotlivé algoritmy jsou univerzální, protože počítají výsledky pro celou skupinu vstupních dat, je i složitost obecná. Takže časová složitost není, jak by se mohlo zdát, přesná doba činnosti algoritmu. Tato doba totiž závisí na konkrétním vstupu.

Pěkným příkladem by mohl být obyčejný mobilní telefon. Odesílání zpráv SMS nebo fotek se vždy děje po částech, přesněji po balíčcích (paketech). Velká fotografie se rozdělí na několik balíčků a ty se odesílají postupně, aby se v druhém telefonu zase propojily a zobrazily původní fotografii. Stejně je tomu se zprávami, ale protože v normální zprávě je pouze text, je odeslaných balíčků nesrovnatelně méně než při posílání fotografií.

Algoritmus je vždy stejný:

1. Rozděl data do balíčků stejné velikosti.
2. Odesílej postupně všechny balíčky jeden po druhém na danou adresu, dokud nebudou odeslány všechny.

Druhý krok je z časového hlediska zásadní, protože čím více balíčků se musí odeslat, tím delší dobu odesílání potrvá. Pokud se textová zpráva vejde do jednoho balíčku a fotografie se musí rozdělit do deseti balíčků, odesílání fotografie potrvá desetkrát déle než odesílání textové zprávy, protože krok číslo 2 se musí opakovaně (desetkrát) provádět. Z toho můžeme vyvodit dvě důležité věci:

- doba provádění algoritmu se nepočítá v časových jednotkách, které známe (sekundy, minuty), nýbrž je to počet kroků, jež musí algoritmus na cestě k výsledku vykonat (přesná doba se pak dá vypočítat vynásobením počtu kroků a dobou, kterou trvá jeden krok),

- platí, že doba provádění algoritmu na odesílání nějakých dat z mobilního telefonu závisí na velikosti dat, která chceme posílat.

Počty balíčků nebo kroků nemusíme znát konkrétně, stačí vědět, že celková doba provádění závisí na opakovaném provádění druhého kroku – odesílání balíčků. Stokrát větší data, stokrát delší doba provádění. Tisíckrát větší data, tisíckrát delší doba provádění.

Takovéto algoritmy jsou ještě docela rychlé, ale představme si, že by doba provádění závisela na velikosti vstupu takto:

- desetkrát větší data, doba provádění umocněná na desátou,
- stokrát větší data, doba provádění umocněná na stou,
- tisíckrát větší data, doba provádění umocněná na tísícátou.

Důležité: Je jasné, že pro celkovou dobu provádění algoritmu je daleko zásadnější samotný algoritmus, než vstupní data, i když ta samozřejmě celkovou dobu také ovlivňují.

Tak obrovská čísla a tak dlouhý čas provádění je i pro seberychlejší počítač problémem, proto se musíme snažit, aby naše algoritmy byly takové jako v příkladu s mobilním telefonem. Někdy to není možné, ale většinou to chce jen trochu se zamyslet a na světě je rázem mnohem lepší, rychlejší a efektivnější algoritmus.

Tvoříme algoritmy s nízkou složitostí

Jak jsme řešili problém překladu? Vzali jsme zadané slovo a porovnávali jsme ho se všemi slovy v daném slovníku. Hezky jedno po druhém, od shora dolů. Při porovnávání jsme nebyli konkrétní – je třeba porovnat jedno písmeno po druhém, jestli se shodují, nebo ne.

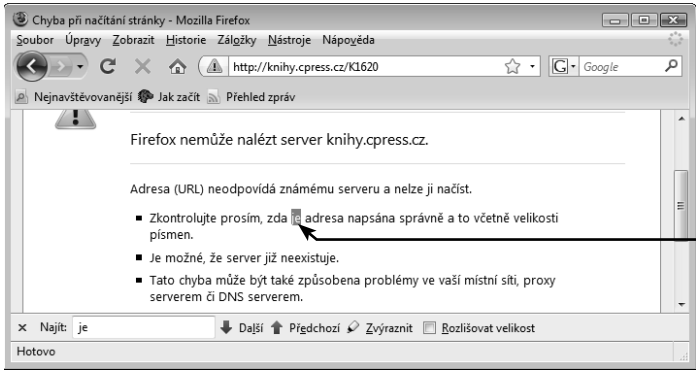
Náš příklad je miniaturní, ale představme si, že by slovník obsahoval tisíce a desetitisíce slov. Pomysleme i na případ databáze obyvatelstva, kde se vyskytují desítky milionů jmen, která navíc nemusí být řazena podle abecedy.

Naše postupné vyhledávání a porovnávání jednoho jména po druhém by zabralo obrovské množství času. Pokud by jedno porovnání (celého slova) trvalo jednu setinu sekundy, trval by průchod celého seznamu všech obyvatel Číny (Číňanů je jedna miliarda) přesně 10 milionů sekund, tedy skoro čtyři měsíce. Podobný problém se řeší při vyhledávání na Internetu, kde jsou rovněž miliony a miliony různých hesel řazených neabecedně.

Vyhledávání v seznamu musí probíhat jinak. Ten způsob dobře demonstruje internetový prohlížeč Mozilla Firefox:

1. Pokud máte Firefox nainstalovaný, spusťte jej a stiskněte kombinaci kláves Ctrl+F.
2. Vlevo dole se objeví pole pro zadání vyhledávání slova.
3. Hledáme-li slovo „jezero“, stiskneme písmeno „j“.
4. Na stránce se označí první výskyt písmena „j“.

5. Stiskneme „e“ a na stránce se označí první výskyt posloupnosti „je“.
6. Atd. Nakonec se buď označí první výskyt posloupnosti písmen „jezero“, nebo se ve vedlejším poli dozvíme, že „Řetězec nenalezen“.



Obrázek 1.17. Vyhledávání v prohlížeči Mozilla Firefox

Poznámka: Se slovem „řetězec“ se ještě mnohokrát setkáme, takže nyní pouze na okraj poznamenáme, že se jedná o posloupnost několika znaků (nejen písmen a číslic, ale třeba i mezer, teček apod.).

Z výše naznačeného algoritmu je vidět, že není potřeba kontrolovat všechna slova. Vyhledávají se pouze výskyty posloupnosti prvních písmen. Pokud je seznam slov, ve kterém se vyhledává, nijak neřazený ani strukturovaný, je to daleko efektivnější způsob než pokaždé kontrolovat rovnost či nerovnost s vyhledávaným slovem.

K čemu je dobrý algoritmus s vysokou složitostí

Je třeba říci, že někdy se bez algoritmů s vysokou složitostí neobejdeme. Obrovskou složitost, kde se počítá s mocninami vstupních dat, mají ty algoritmy, které provádějí své výpočty tzv. *hrubou silou*. Dobrým příkladem je hádání hesla. Nejspolehlivějším, ale také nejnáročnějším způsobem je brát postupně všechny možné kombinace a zkoušet je jako heslo. Není možné, abychom se časem netrefili, ale ono „časem“ může znamenat velmi dlouhou dobu, která velmi rychle roste v závislosti na počtu kláves, z nichž je možné heslo tvořit, a na délce hesla.

U zámku na kolo, který má tři kroužky po devíti číslicích (délka hesla=3, počet kláves/číslic=9), je celkem 9^3 (=729) možností. U zámku, který má devítičíslicových kroužků 5, je to již 9^5 (=59 049) možností. A raději se neptejme, kolik možností je u klasické počítačové klávesnice, když heslo může mít až 256 znaků. Pokud však jiný způsob prolomení hesla neexistuje, nezbývá, než si počkat.

Důležité algoritmy – seřadíme fotbalové hráče podle velikosti

Protože vytvořit efektivní algoritmy pro nejrůznější problémy se lidé snaží od pradávna, je jasné, že máme k dispozici velmi mnoho již vyřešených otázek. Není potřeba znovu vymýšlet to, co už je hotové, a tak si jeden velmi známý algoritmus představíme.

Jmenuje se *BubbleSort* a již za okamžik poznáme proč. Vstupem je množina různě velkých prvků, např. 11 různě vysokých fotbalových hráčů, a výstupem je setříděný celý fotbalový tým podle velikosti. Patří mezi tzv. *třídicí algoritmy* a jednotlivé prvky třídí bubláním.

Hráč probublá na své místo

Tzv. *BubbleSort* je skutečně založený na bublání. Ukažme si celou věc na příkladu:

- Karel měří 190 cm,
- Mirek měří 180 cm,
- Tonda je nejvyšší, měří 210 cm,
- Pepa je nejnižší, měří 170 cm,
- Honza měří 185 cm,
- Ruda měří 195 cm.

Dnes přišlo na trénink jenom šest hráčů (zbývajících pět bohužel leží doma s angínou) a zatím stojí vedle sebe tak, jak je naznačeno ve výčtu (Karel úplně vlevo, Ruda na pravém konci). Právě dorazil i trenér a zařval na hráče, aby se okamžitě seřadili podle velikosti, a to tak, aby vlevo byl nejmenší a vpravo aby byl největší. Chlapci pokrčili rameny a začali se řadit:

1. Nejprve se porovnal Karel s Mirkem. Karel je vyšší, proto se prohodili. Nyní řada našich fotbalových hráčů vypadá takto (píšeme jen jejich začáteční písmena a velikost):

M	K	T	P	H	R
180	190	210	170	185	195

2. Protože se kluci prohazovali, chtějí si to zkontrolovat a porovnají se hráči úplně vlevo: Mirek je nižší než Karel: OK, jedeme dál, porovnávání se posune o jednoho doprava.

3. Karel se nyní porovnává s dalším sousedem, s Tondou. Karel je nižší než Tonda, v pořádku, prohazovat se nemusejí, jedeme dál doprava.

4. Tonda se porovná se svým sousedem vpravo, s Pepou. Tonda je vyšší než Pepa, proto se musejí prohodit. Řada hráčů se změní takto:

M	K	P	T	H	R
180	190	170	210	185	195

5. Protože došlo k prohození, musí se to celé od leva zkontrolovat: Mirek je nižší než Karel, OK, Karel je vyšší než Pepa. To není OK. Musíme je prohodit a po prohození řada vypadá takto:

M	P	K	T	H	R
180	170	190	210	185	195

6. Zase jsme prohazovali, tedy jdeme od začátku: Mirek je vyšší než Pepa. To není v pořádku, musejí se prohodit a řada pak vypadá takto:

P	M	K	T	H	R
170	180	190	210	185	195

Poznámka: Všimněte si, jak Pepa neustálým porovnáváním dobublal až na začátek řady. Stejné to bude za okamžik s nejvyšším Tondou.

7. Došlo k prohazování, proto musíme kontrolovat pěkně od začátku: Pepa je nižší než Mirek, to je OK. Mirek je nižší než Karel, OK. Karel je nižší než Tonda, OK. Tonda je vyšší než Honza: prohodí se tedy mezi sebou a řada vypadá takto:

P	M	K	H	T	R
170	180	190	185	210	195

8. Protože je opět nutné začít od začátku, narazíme na problém u porovnávání Karla s Honzou. Prohodí se mezi sebou a řada vypadá takto:

P	M	H	K	T	R
170	180	185	190	210	195

9. Jdeme opět od leva a při porovnávání Tondy s Rudou zjistíme, že Tonda je vyšší. A tak je prohodíme:

P	M	H	K	R	T
170	180	185	190	195	210

10. Nenechme se mýlit, algoritmus tím nekončí. Protože jsme právě prohodili dva hráče mezi sebou, musíme pěkně od leva až doprava zkontrolovat, že každý pár sousedních hráčů je správně seřazen. Dojdeme až na konec řady, zjistíme, že všechno je v pořádku a v tuto chvíli teprve algoritmus končí a jako výstup nám ukazuje sice jen zdravé jádro fotbalového týmu, zato však seřazené hezky podle velikosti.

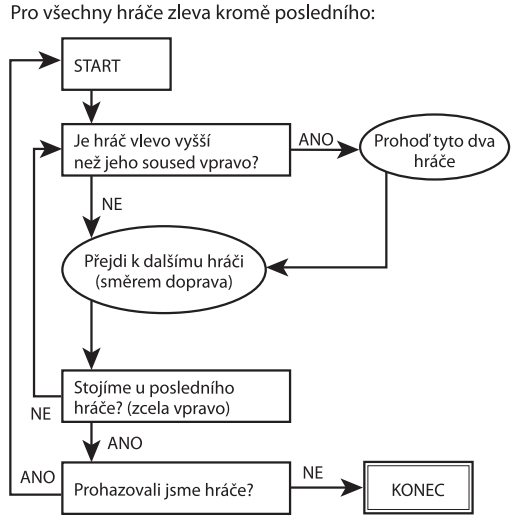
A jak algoritmus vlastně vypadá? Podle postupu jeho provádění se to dá vyvodit, algoritmus bubblesort můžeme tedy zapsat takto:

1. Dokud nenastane situace, že jsou všichni hráči celé řady nižší než jejich pravý soused (kromě toho úplně vpravo, jelikož ten už nemá vpravo souseda), prováděj toto:
2. Pro všechny hráče od levého konce řady až po předposledního (neboli druhého zprava) porovnávej hráče s jejich pravým sousedem:
3. Pokud je hráč vlevo vyšší než jeho soused vpravo, prohoď je.

O něco názornější než samotný zápis algoritmu je obrázek 1.18.

Třídicích algoritmů je celá řada a BubbleSort patří bohužel mezi nejméně efektivní. Je však výborný na ukázku třídění i ukázku algoritmu jako takového. Jiné třídicí algoritmy jsou sice daleko efektivnější, jejich algoritmy jsou však mnohem méně průhledné. Internet je plný nejrůznějších výkladů třídicích algoritmů, v případě zájmu však hledejme ty nejdůležitější:

- SelectSort,
- InsertSort,
- MergeSort,
- QuickSort.



Obrázek 1.18. Schéma algoritmu BubbleSort

Poznámka: Algoritmům se říká *třídicí*, protože se prvky setřídí (seřadí) podle velikosti. Slovo sort v angličtině znamená právě setřídit nebo seřadit; uspořádat.

Každý třídicí algoritmus nakonec vrátí setříděnou množinu prvků (seřazený fotbalový tým), ale každý z nich na to jde jinak. Každý využívá jinou myšlenku. Pochopení jednotlivých algoritmů, jejich výhod i nevýhod, je důležité pro každého, kdo to s programováním myslí vážně. My to s programováním vážně myslíme, ale protože si už chceme co nejdříve všechno vyzkoušet v praxi, musíme nechat další algoritmy na později; až je budeme umět zapsat do počítače. V sedmé kapitole si zkusíme některé další třídicí algoritmy naostro naprogramovat.

Jak to funguje v počítači

Jak počítači sdělit, aby opakoval určitou činnost, jak mu přikázat, aby mezi sebou prohodil dva prvky množiny, nebo jak zařídit, aby si počítač pamatoval nějaké hodnoty? Tyto činnosti nelze nařizovat lidskou řečí.

Pokud chceme počítači něco přikázat, musíme to udělat v jeho řeči, které se jednoduše říká *programovací jazyk*. Takový programovací jazyk je trochu jiný než třeba čeština, protože v češtině neustále vznikají nová slova, ta stará se přestávají používat, některé výrazy se na

Moravě řeknou jinak než v Čechách, a dokonce se časem mění i významy jednotlivých slov.

V programovacím jazyku je naopak všechno dáno navěky a nic se nikdy nemění, slova mají pořád stejný význam, aby jim všechny počítače vždy rozuměly. V dalších kapitolách se naučíme slova jednoho programovacího jazyka, Pascalu, ale programovacích jazyků existuje celá řada. Jejich různé druhy popisuje Příloha A na konci této knihy.

Dorozumívání programátora s počítačem

No dobře, řekneme si, naučíme se jakýsi jazyk, kterému počítač rozumí, ale jak máme na počítač mluvit? Přes mikrofon? To přece ne. Zřejmě se slova budou někam zapisovat, ale kam?

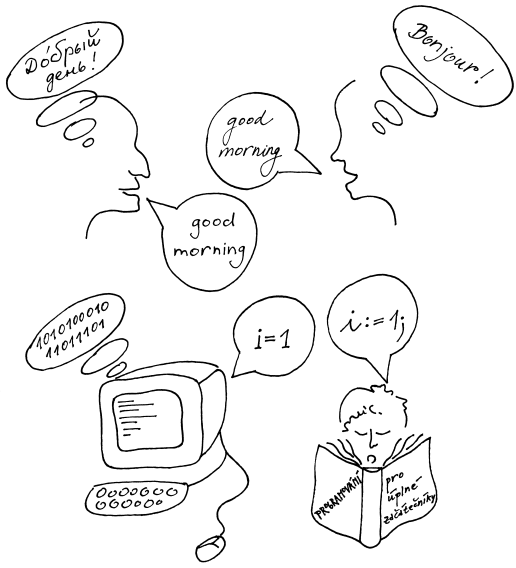
Na to je jednoduchá odpověď. Každý programovací jazyk má své vlastní *rozhraní*. Rozhraní je podobné např. editoru Microsoft Word nebo obyčejnému WordPadu. Nejdůležitější je pole, do kterého se zapisují příkazy pro počítač.

Nyní se oklikou dostáváme tam, kde jsme začali: ke konkrétní realizaci problému:

1. Po analýze problému a návrhu funkčního řešení máme před sebou algoritmus napsaný v češtině tak, jak jsme ho zapsali třeba u bublinkového algoritmu (BubbleSort) nebo u překladového slovníku.
2. Nyní nastává realizace. Ta spočívá:
   - v překladu algoritmu do programovacího jazyka,
   - v zápisu textu v programovacím jazyce do výše zmíněného rozhraní.
3. Po tom, co zapíšeme náš algoritmus v programovacím jazyce do programovacího rozhraní, nezbývá než zkusit takový program spustit.

Po spuštění si počítač převede slova programovacího jazyka do svého vlastního, pro člověka nesrozumitelného jazyka a program vykoná. Je to na vlas stejný princip jako v případě dvou lidí, z nichž jeden mluví jen francouzsky a anglicky a druhý jen rusky a anglicky. Jako dorozumívací řeč používají angličtinu.

Programátor mluví česky a umí i programovací jazyk. Počítač mluví svým složitým jazykem, kterému nerozumíme, ale rozumí i programovacímu jazyku. Díky tomu spolu můžeme komunikovat.



Obrázek 1.19. Dva lidé, kteří se dorozumívají pomocí angličtiny, jejich mateřské jazyky se liší, a člověk s počítačem, kteří se dorozumívají pomocí programovacího jazyka, jejich „mateřské" jazyky se liší

Kapitola 2
První krůčky velkého programátora

V první kapitole jsme nakousli výklad o programovacím jazyku jako dorozumívacím prostředku počítače s programátorem. Také jsme mluvili o rozhraní, do kterého se zapisují příkazy v programovacím jazyce. Z tohoto rozhraní si počítač bere jednotlivé příkazy v programovacím jazyce a překládá si je do svého jazyka. Co nám tedy brání, abychom získali ono tajemné rozhraní, do kterého můžeme zapsat naše první skutečné příkazy?

Protože si počítač jednotlivé příkazy programovacího jazyka překládá do svého vlastního, strojového jazyka, říká se rozhraní, o kterém jsme mluvili, *překladač*, někdy také *kompilátor* – z anglického compiler.

Stažení a instalace překladače

Někdy jako překladač slouží i adresový řádek obyčejného webového prohlížeče. To jsme viděli na příkladu v první kapitole. Zadali jsme příkaz v programovacím jazyce JavaScript a stiskli klávesu ↵Enter. Počítač si pro sebe přeložil náš příkaz do svého jazyka a potom příkaz vykonal. Výsledkem bylo okno s nápisem, který jsme zadali mezi apostrofy v závorce.

Abychom mohli dělat ještě zajímavější kousky, stáhneme si jiný překladač. Jmenuje se Free Pascal a budeme do něj, jak název napovídá, psát příkazy v programovacím jazyce Pascal. Někdy se setkáte i s jiným názvem pro náš překladač, a to *Pascal free IDE*. IDE znamená Integrated Development Environment, česky integrované vývojové prostředí. Té zkratky se nemusíme bát:

- *Integrované* znamená, že ho máme, resp. budeme mít, nainstalovaný v počítači.
- Slovem *vývojové* je míněno, že slouží k vývoji programů, neboli k programování.
- Uvnitř *prostředí* překladače Pascal můžeme psát příkazy v programovacím jazyce Pascal a počítač nám bude rozumět.
- Slovo *free* znamená, že tento překladač je volný, svobodný. To znamená, že je zdarma a my ho na svém počítači můžeme bezplatně využívat.

Překladač Free Pascal získáme snadno. Nejprve je potřeba stáhnout si instalační soubor:

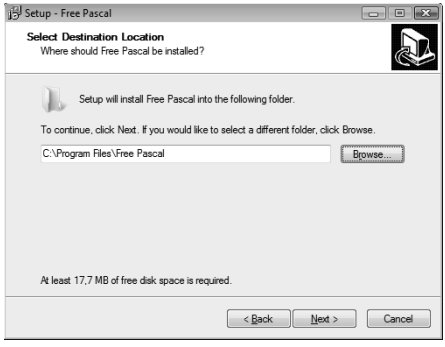
1. Na svém webovém prohlížeči přejdeme na stránku *http://knihy.cpress.cz/programovani-pro-uplne-zacatecniky.html*
2. Klepneme na odkaz *Soubory ke stažení* a poté na odkaz *Instalace Free Pascal 2.2.0*.

3. Archív *Instalace Free Pascal 2.2.0.rar* uložíme.
4. Stažený archív rozbalíme pomocí některého archivačního programu kamkoliv na náš disk.

Poznámka: V dalším textu předpokládáme, že začínající programátor v Pascalu se pohybuje v systému Microsoft Windows. Pro uživatele unixových systémů je návod k instalaci i k používání překladače Free Pascal na těchto stránkách: *http://www.linuxsoft.cz/article.php?id_article=99*.

Pokud jsme úspěšně stáhli instalační soubor, můžeme překladač na náš počítač nainstalovat. Obvykle to spočívá v prostém poklepání na instalační soubor (*fpc-2.2.0.i386-win32.exe*) a postupném procházení jednotlivými instalačními kroky:

1. První okno, které se nám po spuštění instalačního souboru ukáže, nás vítá v instalačním procesu. Vidíme verzi překladače Free Pascal a druhý a třetí řádek nás spravuje o tom, že bychom při instalaci neměli mít spuštěny žádné jiné aplikace. Poslední řádek pouze nabádá, abychom klepli na tlačítko **Next** (Další), pokud chceme pokračovat v instalaci.
2. Dalším krokem je výběr umístění instalace. Nejlepší je dávat všechny instalované programy do adresáře *C:\Program Files*. I my zvolíme tuto cestu a v adresáři *Program Files* nainstalujeme překladač do podadresáře *Free Pascal*. Je potřeba myslet na to, že pro instalaci potřebujeme 17,7 MB volné paměti, jak nám říká text na spodním okraji okna. Pokud tolik místa máme, není problém překladač instalovat a klepneme proto na tlačítko **Next** (Další).

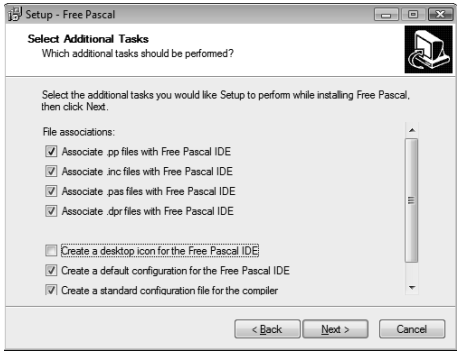


Obrázek 2.1. Druhé okno instalačního procesu

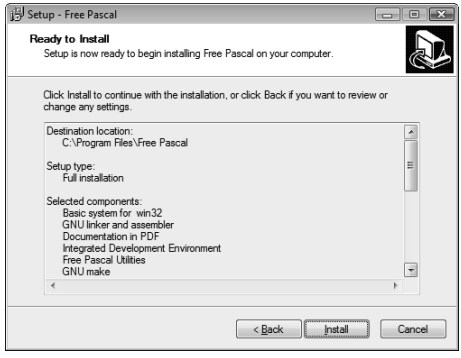
3. Ve třetím okně ponecháme **Full instalation**, neboli plnou instalaci všech potřebných součástí a klepneme na tlačítko **Next** (Další).
4. Zde pouze volíme, jak se bude v hlavním, startovním menu jmenovat odkaz na Free Pascal. Pokud nechceme, aby se nějaký odkaz ve startovním menu tvořil, odškrtneme čtvereček **Don't create a Start Menu folder** (Nevytvářet položku v nabídce *Start*). Klepneme opět na tlačítko **Next** (Další).

5. Zde zaškrtneme všechny možnosti, aby se typy všech vybraných souborů přátelily s naším překladačem. Pouze v případě, že nechceme, aby se vytvořila na pracovní ploše ikona rychlého spuštění překladače Free Pascal, odškrtneme třetí čtvereček odspoda. Potom klepneme na tlačítko **Next** (Další).

6. Přichází zopakování všech podstatných informací, které jsme při instalaci zvolili. Po prohlédnutí klepneme na tlačítko **Install** (Instalovat) a instalace začíná.



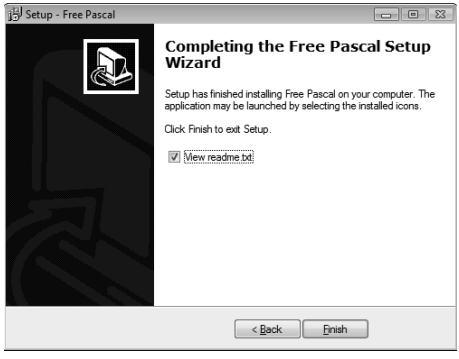
Obrázek 2.2. Páté okno instalačního procesu



Obrázek 2.3. Šesté okno instalačního procesu

7. Předposlední okno zobrazuje informace o novinkách verze 2.2.0 oproti verzím starším. Tentýž text najdete po instalaci v souboru *C:\Program Files\Free Pascal\doc\fpc\whatsnew.txt*.

8. Závěrečné okno pouze informuje, že vše proběhlo v pořádku a vybídne nás k ukončení instalačního procesu. Pokud si chceme přečíst soubor *readme.txt*, který obsahuje základní informace o nainstalovaném překladači a vývojovém prostředí Free Pascal, necháme zaškrtnutý čtvereček **View readme.txt** (Zobrazit soubor čtimě.txt). V opačném případě ho odškrtneme klepnutím myši. Soubor si můžeme později přečíst zde *C:\Program Files\Free Pascal\doc\fpc\readme.txt*. Po klepnutí na tlačítko **Finish** (Dokončit) se instalace dokončí a případně se otevře soubor *readme.txt*.



Obrázek 2.4. Poslední okno instalačního procesu

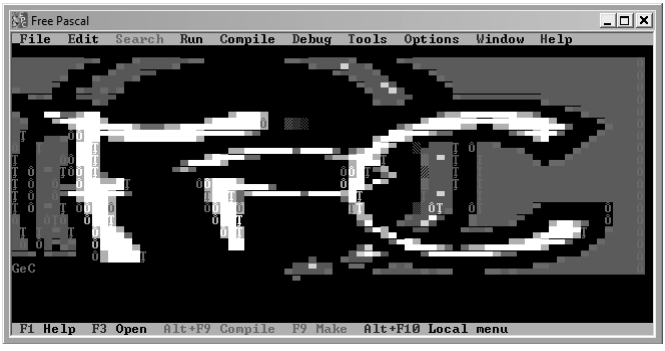
Spuštění vývojového prostředí

Existuje několik možností spuštění prostředí Free Pascal. Vybíráme tu, která se nám zdá nejpohodlnější a nejlepší: poklepeme na ikonu přímo na své pracovní ploše.

Řešení problému: Pokud na své pracovní ploše ikonu nenajdeme, jsou tu další dvě možnosti, jak vývojové prostředí spustit:

- Asi nejkomplikovanější je vstoupit přímo do adresáře *C:\Program Files\Free Pascal\bin\i386-win32* a tam spustit soubor *fp.exe*. Pokud však nemáme ikonu na pracovní ploše a odmítli jsme vložení odkazu do startovního menu, není jiné možnosti.
- Dalším způsobem je klepnout na tlačítko **Start**, zvolit nabídku Všechny **programy**, zde zvolit **Free Pascal** a klepnout na odkaz **Free Pascal**.

Mělo by se otevřít černé okno s nepříliš vábným logem Free Pascal (FPC), na jehož horním i dolním okraji je šedá lišta plná nabídek. Viz obrázek 2.5.



Obrázek 2.5. Úvodní okno překladače Free Pascal

Celý překladač je založen na jednoduchosti. Proto i logo vývojového prostředí je velmi jednoduše vytvořeno pomocí barevných čtverečků, jež se dají nazvat *pixely*.

Další výklad předpokládá, že jsme úspěšně stáhli i spustili jakoukoli verzi překladače a vývojového prostředí Free Pascal. V následujících částech si jednak blíže vysvětlíme rozdíl mezi programovacím jazykem a jeho překladem do jazyka počítače a v návaznosti na to se probereme nejdůležitějšími funkcemi a nabídkami právě spuštěného vývojového prostředí Free Pascal.

Zdrojový kód a jeho překlad

S pojmem *zdrojový kód* se tu sice setkáváme poprvé, ale jde o starou známou věc: zdrojový kód není nic jiného než posloupnost příkazů zapsaná v programovacím jazyce. Zdrojový kód stojí tedy na půl cesty mezi příkazy pronesenými lidskou řečí a příkazy, kterým rozumí jen počítač. Pro celý další výklad bude důležité rozlišovat tři druhy jazyků:

- *Přirozené jazyky.* Těmi mluví lidé (čeština, angličtina, maďarština apod.).

- *Programovací jazyky.* Ty slouží k programování. Je jich mnoho, podobně jako přirozených jazyků, a jejich různé typy zachycuje Příloha A na konci této knihy. Pascal patří mezi imperativní programovací jazyky, což znamená, že je založený na příkazech.
- *Strojové jazyky.* Těm rozumí počítače, ale je dobré si uvědomit, že všechny počítače stvořili lidé, takže někteří lidé strojovému jazyku rozumí. Strojový kód je pro nás sice těžko srozumitelný, ale na druhou stranu je velmi logický a jeho logika se takřka nemění.

Jak to celé funguje

Počítače jsou stroje jako všechny ostatní a jejich funkčnost je založena na stejném principu jako třeba žárovka nebo varná konvice. Zapneme: svítí a hřeje, vypneme: nesvítí a nehřeje. Dvě hodnoty a nic víc na tom není.

Jazyk počítače je stejně jako jazyk jeho jednodušších příbuzných, žárovky či varné konvice, založený pouze na dvou hodnotách. Elektronické vzruchy a signály, které probíhají útrobami počítače, jsou pouze dvojího druhu: ano a ne, zapnuto, nebo vypnuto. Aby se s tím dalo nějak pracovat, řekli si výrobci počítačů, že tyto signály převedou na čísla. Vypnuto bude 0, zapnuto bude 1.

A zde je počátek všeho. Když už máme něco vyjádřeno čísly, dokážeme všechno. Dvojková neboli binární číselná soustava totiž počítá také pouze s dvěma hodnotami, právě s jedničkou a s nulou. A protože se někdy strojový kód v dvojkové soustavě převádí do soustavy šestnáctkové (v ní je 10 čísel, 0–9 a 6 písmen, A–F), může instrukce ve strojovém kódu vypadat třeba takto:

```
3E 08 2E FF 36 40 08
```

Takový kód se dá číst jen velmi těžko a ještě hůře se v něm programuje. Dříve to ovšem jinak nešlo: v dávných dobách počítačových začátků se musela napsat instrukce v podstatě pro každý elektronický signál. Proto si první programátoři kdysi dávno řekli, že by tento kód bylo dobré převést na jiný, srozumitelnější lidskému jazyku.

Od té doby vznikají programovací jazyky čím dál bližší jazyku lidí. Respektive, abychom byli přesnější, blíží se víc a víc k angličtině. Protože programu zapsanému v takovém jazyce počítač nerozumí, je vždy s programovacím jazykem pevně svázáno nejen vývojové prostředí, ve kterém se programuje, ale také překladač daného jazyka do strojového kódu číslic šestnáctkové a dvojkové soustavy.

Klíčová slova

Slova, která se v programování používají, jsou tedy velmi často anglická. V jazyce Pascal tomu není jinak:

- `write` znamená *napiš*,
- `read` znamená *přečti*,
- `begin` je *začátek*,
- `end` je *konec*.

Jako programátoři budeme zanedlouho vytvářet nová slova programovacího jazyka, a to na základě slov původních, již předem nastavených. Těmto původním slovům se říká *klíčová*. S jejich pomocí budeme schopní nejen psát programy, ale tvořit další slova, která nám psaní programů ulehčí.

Představme si to na části programu automatu na kávu:

```
Pokud je vhozený obnos menší než cena vybraného nápoje, potom vypiš na
displeji informaci o tom, kolik ještě chybí.
```

Pojmy `vhozený obnos`, `cena` nebo `vybraný nápoj` se určitě dají nastavit – dají se nějak definovat. Prostě tím, že přesně určíme jejich hodnotu. Cena kávy je 12 korun, proto pokud na automatu stiskneme tlačítko pro kávu, pak pojem `cena vybraného nápoje` aktuálně znamená 12 korun.

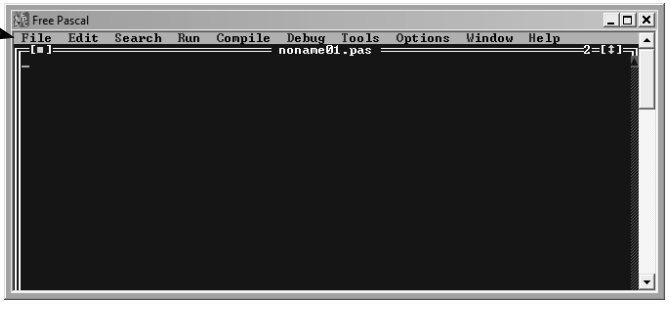
Ale některá slova se nijak definovat nedají. Pojmy `pokud` či `potom` musejí být klíčové. Musejí být předem nastaveny, být původní součástí programovacího jazyka.

Důležité: S klíčovými slovy `pokud` a `potom` se setkáme snad ve všech programovacích jazycích, ovšem protože je většina z nich založená na angličtině, jde o klíčová slova `if` (jestliže) a `then` (pak). Ani v jazyce Pascal tomu není jinak.

Vývojové prostředí jazyka Pascal nám ulehčuje práci tím, že klíčová slova zbarvuje bíle, zatímco většina ostatních slov je žlutě. Obraťme tedy pozornost k vývojovému prostředí a vše si vyzkoušejme v praxi. Rozdíl mezi přirozeným, programovacím a strojovým jazykem nabude ještě jasnějších obrysů.

Seznámení s vývojovým prostředím jazyka Pascal

Po spuštění prostředí Free Pascal klepneme na příkaz **File** (Soubor) v horní šedé liště a pomocí myši nebo klávesnice vybereme z nabídky položku **New** (Nový). Otevře se modré okno, do kterého je možné psát zdrojový kód. Viz obrázek 2.6.



Obrázek 2.6. Nová, čistá plocha připravená pro naše příkazy

Toto prostředí funguje jako klasické, průhledné okno: budeme sem psát příkazy a počítač je z druhé strany uvidí. Stačí potom dát znamení, že si má počítač předložený zdrojový kód přeložit do svého jazyka.