Manuál

Formátuj, ať je to přehledný

- ideálně používej jednotné formátování pomocí stylů textu (normal text/ heading 1/ heading 2 ...)

Využívej komentáře:

- když si něčím nejsi jistý
- když s něčím nesouhlasíš
- když je potřeba něco doplnit

Jestli je něco očividně špatně, uprav to.
Vnější zdroje pouze s doplňkovými informacemi, vše podstatné přímo do dokumentu.
Otázku, co si vezmeš na starost, označ v nadpisu svým jménem. (JMÉNO)
Když ji dokončíš, tak odstraň ‘ROZDĚLANÉ’

Tady jsou ještě docela poctivě dělaný poznámky z hodin od kluků z nižších ročníků, tak koukni, jestli něco z toho nebudeš moci využít zde: Infoseminář zápisky

Informatika - zpracované otázky

1 Operační systém (Tonda + Filip)

- Spouští a spravuje úlohy
- Zprostředkovává rozhraní s hardwarem
- Zprostředkovává filesystem
- Zprostředkovává uživatelské rozhraní

- Kernel je podřadný operačnímu systému - nemusí splňovat veškeré výše stanovené podmínky

Dělení operačních systémů

- Grafické UI / Textové UI
- Využití: Desktop / server / mobil / …
- License
- Jednoúlohové / víceúlohové
- Jednouživatelský / Mnohouživatelský

Správa úloh

- Každé jádro procesoru dovede pracovat v jeden moment pouze na jedné úloze
- **Interrupt** = přerušení úlohy běžící na jádře, aby se dostalo na další
- Díky interruptu může běžet na procesoru více úloh jak kolik má jader
- **Scheduler** udává úlohám prioritu, dle které potom dostávají výpočetní čas na jádře
- O načasování se stará **clock**

- OS běží v privilegovaném modu, úlohy pouze v uživatelském
- Úlohy mají přístup pouze ke svému dílu paměti, ke zbytku přistupují za použití “system call”

File system je uživatelsky srozumitelnou abstrakcí adresového systému paměti. Zpravidla je hierarchizován a každý soubor má svou unikátní adresu v ní.
Partitions (díly) jsou na sobě nezávislé file systemy.
IPC = Inter Process Communication, jsou tím například sockety a filesystem

BIOS - Basic Input/Output System, zkontroluje hardware a načte bootloader
Bootloader - načte kernel

Historie

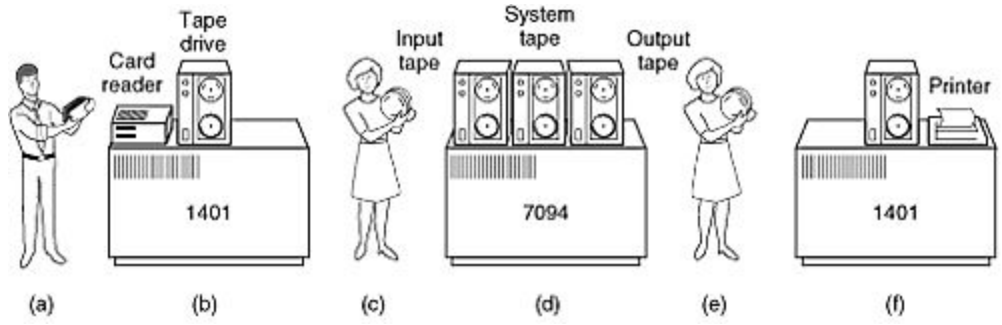
První generace (40. léta až ranná 50. léta)

- Počítače bez operačního systému
- Uživatelé měli typicky přidělený čas, kdy mohli počítač používat
- Děrné štítky, papírové nebo magnetické pásky
- Jenom strojový kód
    - Případně později jazyk symbolických adres (assembly)
    - Později také kompilátory, aby člověk nemusel přepisovat assembly do strojového kódu ručně
- Zárodek operačního systému: Knihovny
    - Také třeba na děrných štítcích

- Uživatelův program mohl využívat už v nich předpřipravených procedur
- Pomáhali třeba se vstupem a výstupem

Druhá generace (1955 až 1965)

- Tranzistory, mainframy (komerčně dostupné počítače)
- Programovalo se už třeba ve FORTRANu
- Byly rychlejší -> Více uživatelů -> Programátoři předali své programy operátorovi a šli si udělat kafe
- Bylo tedy těžší programy debugovat, sledovat kolik sys. zdrojů používají (nebo papíru na tisk), knihovny byly stále složitější, operátoři celý proces zpomalovali



- Předchůdce operačního systému byly tzv. “batch systems”, které házely programy do hlavního počítače za sebou -> není potřeba, aby kolem něho pořád běhal operátor
    - Na obrázku je tohle např. implementované pomocí počítače, který mnoho programů na děrných štítcích přepíše na jednu pásku
    - První batch systems vyvíjeli pro IBM General Motors
- No a později tohle bylo už čistě softwarové
- Btw, v téhle době jsou operační systémy na úrovni toho, čemu dneska říkáme kernel

Třetí generace (1965 až 1980)

- Multitasking
    - Díky tomu bylo možné daleko lépe využívat procesor - viz time-sharing
- System/360
    - K nim OS/360, obrovská věc psaná tisícemi programátorů se spoustů bugů
    - Protože prostě udělat něco, co by fungovalo efektivně na různých velikostech počítačů s různým I/O harwarem a to celé jenom v assembly bylo tehdy moc ambiciózní
- Minipočítače (pořád obrovské ale nezabíraly celou místnost)
    - Např. PDP-1
- Vznikaly time-sharing systémy
    - Místo, aby uživatelé měli přidělená časová okna, kdy mohli k systému přistupovat, přihlásili se všichni přes online terminál
    - Pokud se přihlásilo 20 uživatelů a z nich 17 zrovna nic po počítači nechtělo, ten mohl pracovat na úlohách od těch 3 aktivních a třeba ještě v pozadí pracovat na nějakém velkém výpočtu
    - Nejpoužívanější byl asi MULTICS - na něm pak Ken Thompson založil UNIX
- UNIX
    - Vznikly dvě hlavní nekompatibilní verze: System V (AT&T) a BSD
- POSIX
- MINIX
    - Napsal Tanenbaum, který také napsal "Modern Operating Systems"
    - Původně zamýšleno jako nástroj pro výuku operačních systémů

    - Dneska je to v každém Intel procesoru
- LINUX
    - Torvalds se naštval, že MINIX lze svobodně používat jenom pro potřeby výuky, takže si udělal svůj

Čtvrtá generace

- Osobní počítače
- Nejprve se jim říkalo "mikropočítače"
- Komerčně daleko dostupnější
- IBM PC
- Potřebovalo OS -> MS-DOS
    - Tohle nevím jestli je pravda, ale je to vtipný:
      Bill Gates IBM řekl, že jim udělá OS. Pak šel a chtěl místo toho koupit už hotové OS jménem DOS. Jenomže mu ho neprodali. Takže byl nucen sehnat tým a opravdu OS naprogramovat.
- 1981 je vydán MS-DOS
- Windows původně grafické rozšíření MS-DOSu
- Apple
- GUI
    - Fakt se to vyslovuje "guji"
    - Myslím, že Apple byl první, pak to okopíroval Microsoft

2 GNU/Linux (Honza)

Linus Torvalds

- vývoj linuxu (pouze kernel) 1991 - po vzoru systému UNIX a MINIX
- věří, že je nesmyslné odepsat všechen proprietární software (nepropaguje ho ale linux se snaží podporovat proprietární drivery nvidie)

Richard Stallman

- father of free software a donedávna prezident (GNU = GNU's not unix)
- nevlastní mobilní telefon odmítá používat libovolný proprietární software

GNU/Linux (liGNUx)

- Vznikl jako spojení ekosystému programů GNU (free software kolony základních programů pro UNIX) a Linuxového jádra

Otevřený software - Open Source

Zdrojový kód je veřejně dostupný, můžu ho stahovat a instalovat jak chci.

Svobodný software - Free Software

Je dovoleno ho upravovat a vydávat vlastní pozměněné distribuce
4 základní svobody podle GNU:

- The freedom to run the program as you wish, for any purpose (freedom 0).
- The freedom to study how the program works, and change it so it does your computing as you wish (freedom 1). Access to the source code is a precondition for this.
- The freedom to redistribute copies so you can help others (freedom 2).
- The freedom to distribute copies of your modified versions to others (freedom 3). By doing this you can give the whole community a chance to benefit from your changes. Access to the source code is a precondition for this.

Filosofie UNIXU (a od něho odvozených systémů)

- Vše funguje jako soubor - připojená myš, soubor, klávesnice, tiskárna, disk...
- Program dělá jednu věc a dělá ji pořádně
- Program umí zpracovávat textové proudy
- Výstupy programů lze použít jako vstupy (díky tomu lze kombinovat více jednodušších programů v řešení komplexnějších problémů)

Absolutní a relativní cesty

- **Absolutní cesta** začíná `/`a dále pak může obsahovat i . a ..
- **Relativní cesta** začíná nějakou ze základních relativních cest:
  - . - současná složka
  - .. - rodičovská složka

Zkratky

- ~ - značí domovský adresář

Přístupová práva

- r (read), w (write) x (execute)
- Práva jsou zakódována v jednom trojmístném čísle
- Jednotlivé cifry označují práva pro vlastníka, skupinu a ostatní
- Cifra se vytváří tak, že se sečtou hodnoty přepínačů r(4), w(2), x(1)
    - Např 754 znamená, že:
        - Vlastník může: číst(4) + zapisovat (2) + spouštět(1)
        - Skupina může: číst(4) + spouštět(1)
        - Všichni ostatní mohou: číst(4)

Příkazy:

```
$ # všechno za znakem # je komentář, který interpret ignoruje
$ man "název příkazu"  # otevře manuál zadaného příkazu
$ whoami  # vypíše jméno uživatele
$ pwd  # (Print Working Directory) zobrazí složku ve které se právě nacházíme
- cd [adresář]  # (Change Directory) přesune mě do určené složky (pomocí absolutní nebo
relativní cesty)
- echo [něco]  # vypíše něco
- cp [co] [kam]  # kopírovat
- mv [co] [b]  # přesunout
- rm  # smazat (-r - rekurzivně) (-f smazat i chráněné soubory)
- ls  # vypíše obsah složky (-a - i skryté soubory) (-l - s podrobnými informacemi)
- cat [název]  # čte soubory (-n čísluje řádky)

- vim [file]  # otevře vim
- chmod [3 ciferná přístupová práva] [file]  # mění práva
- chown [username] [file]  # změní vlastníka
- chgrp [group name] [file]  # změní skupinu
- sort  # řadí soubory podle switchů
- head [jméno]  # vypíše prvních 10 řádků souboru (-[n] vypíše n řádků)
- tail [jméno]  # vypíše posledních 10 řádků souboru (-[n] vypíše n řádků)
- sed -e "s/x/y/g" [soubor]  # nahradí znaky x znaky y
- touch [jméno]  # vytvoří soubor
- mkdir [jméno]  # (make directory) vytvoří složku
- grep [text] [file]  # najde všechny výskyty textu v souboru
- du [dir]  # určování velikosti souborů ve složce
- wc [soubor]  # počítání řádků
- clear  # smaže příkazový řádek

- příkaz > "soubor"  # přepíše soubor výstupem příkazu
```

```
- příkaz >> "soubor"  # přidá na konec souboru výstup příkazu
```

3 Verzovací systémy (Filip)

- *Version control systems (VCS)*
- **Verzování** je uchovávání historie veškerých změn provedených v informacích nebo datech.
- **Proč: Historie souborů, kontrola verzí, kooperace**
    - Před verzovacími systémy: Dva lidi začali psát do souboru na serveru, pak ho postupně uložili a zůstaly změny jenom druhého člověka
- Všechny níže uvedené VCSka jsou opensource, ale existují samozřejmě i proprietární programy

Centralizované verzovací systémy

CVS = Concurrent Version System

- K sobě si stahuješ akorát jednotlivé soubory, poslední verze souborů jsou na serveru
- Nevýhoda CVS: Nemá *atomic commits*
    - Pokud *pushnu* na server, soubory se budou aktualizovat postupně. Když se v půlce tohoto procesu pokazí připojení, na serveru zůstane jenom část mého commitu -> fuck, teď je na serveru nefungující verze programu.

SVN = Subversion

- Neukládají se změněné soubory, **ukládají se změny** na nich (ve větších projektech pomalé)
    - Když člověk chce celý projekt dát dohromady a ozkoušet, musí si nechat **dopočítat všechny soubory** a výsledek si stáhnout na svůj počítač
    - Díky tomu je **méně náročný na úložiště**, což je ale dnes nepodstatná výhoda.
        - Dokud do repa nedáváš binární soubory, místo problém není.
- **Zamezuje editaci dvěma osobám zároveň** - editor pošle lock souboru, po editaci unlock

Výhody

- Jsou **jednoduší na pochopení**. Co když je třeba spolupracovat s někým, kdo se normálně v IT nepohybuje.
- Je jednoduší **spravovat, kdo má přístup** k jakému souboru.
- Pokud nechceš, vůbec nemusíš řešit **mergování**.

Distribuované verzovací systémy

Git

- Linus Torvalds
- Mnoho programů
- U sebe máš kompletní repozitář
- Git do historie **ukládá celé soubory**

HG = Mercurial

- Jedna binárka

Výhody

- **Rychlejší**, jelikož člověk nemusí čekat, až někdo jiný doupravuje soubor a uvolní na něm lock
    - **+ Nemusím soubory stahovat**
- **Větvení a mergování je přirozenější.** Systém je založený na tom, že si pracuješ na vlastní branchi a pak to mergneš k tomu, kdo má master kopii. Kdyby člověk chtěl využít větvení a mergování na centralizovaném systému, bude to asi složitější setupnout.
- **Nemusíš být připojený k internetu**, abys mohl pracovat

Používání Gitu

- Vzniká **lokální historie souborů**
    - Při nahrání na cloud git zkoumá, jaký je rozdíl mezi souborem na cloudu a od uživatele
- Repozitář, commit, revize, větev (branch)

Příkazy pro git

- **git clone [url]** vytvoří propojení s vytvořeným projektem
- **git add [file]** přidá soubory do “balíku”
- **git commit** zabalení
    - popř. **-m [commit zpráva]**
- **git push** odešle “balík”
- **git pull** přijme změny z repozitáře
- **git status** zjistí stav lokálního repozitáře
- **git checkout [branch]** přepne na jinou branch
- **git log** ukáže historii commitů s jejich zprávami a daty

4 Programovací paradigma (Honza)

Paradigma

Programovací paradigma je způsob jakým se zapisuje kód, buď imperativně nebo deklarativně

Imperativní programování

- Říkáme počítači co přesně má udělat - základní forma programování - psaní assembleru.
- Není zde žádná abstrakce (co se týče programovacího paradigma)
- Příklady imperativních programovacích jazyků:

  Assembly, C, C++, Python, Java, JavaScript, BASIC, Ruby, ALGOL, ...

Neprocedurální programování

- Neříkáme jak výsledek získat ale co má splňovat výsledek. Například u SQL popíšeme jaké řádky chceme ale už nemusíme psát jak že se mají získat (o to se postará magie uvnitř databáze)
- Obvykle se ve výsledku bude vykonávat imperativní kód (assembler) ale programátor to bude zapisovat deklarativně.
- Hlavní výhodou tohoto přístupu je, že to je bližší tomu jak přemýšlíme a nemusíme řešit detaily a problémy imperativního programování.

Funkcionální

- program = množina funkcí - Logické
- program = zpravidla množina odvozovacích pravidel, např. logických formulí
- Funkce nějak neinteragují s vnějškem (nemají vedlejší účinky)
- Ze stejného vstupu bude vždycky stejný výstup (deterministické)
- Jazyky: Haskell, F#, funkcionálně lze zároveň programovat ve většině vyšších imperativních jazyků (python, javascript) a využívají toho některé frameworky

Deklarativní

- Deklarujeme jaký chceme dostat výsledek a program (černá krabička) se nám postará o to, že ho dostaneme.
- Jazyky: SQL, matematické definice
- Příklad SQL:

```
SELECT SupplierName
FROM Suppliers
WHERE EXISTS (SELECT ProductName FROM Products WHERE Products.SupplierID =
Suppliers.supplierID AND Price < 20);
```

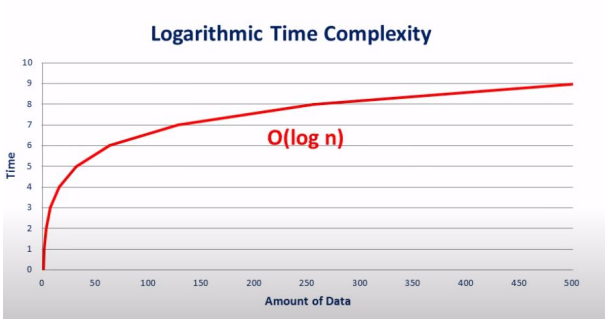
5 Základy teorie složitosti (Tonda)

- Pro zájemce podrobnější článek od KSP (https://ksp.mff.cuni.cz/kucharky/slozitost/ 10 min)

Bylo potřeba vymyslet univerzální způsob jak posoudit efektivitu algoritmu. Nemohlo to být měřený časem, protože to závisí na přístroji na kterém to běží.

O(), asymptotická složitost, algoritmu je tudíž definovaná tím, jak se se doba běhu algoritmu se zvětšujícím se vstupem n v momentě nejhoršího možného případu (dejme tomu že hledáme určitý prvek, tak počítáme s tím, že ten prvek najdeme vždy jako poslední).
Doba běhu se dá počítat jako počet kroků, které musí algoritmus udělat, aby došel k výsledku. V případě **O()** se zbavujeme koeficientů a ponecháváme pouze nejrychleji rostoucí člen. Tj.:
Doba běhu = an + b, O() je rovno tudíž O(n), protože *a* a *b* jsou koeficienty.

Polynomiální složitosti
O(1) je konstantní doba běhu, nezávislá na n (třeba zjištění, jestli je číslo sudé)
O(n) lineární doba běhu v závislosti na n (hledání maxima z N čísel)
O(N · log N) lineárně-logaritmická složitost (nejlepší algoritmy na třídění pomocí porovnávání)
O(n^2) kvadratická doba běhu v závislosti na n (BubbleSort)
O(log n) logaritmická doba běhu v závislosti na n



Binární vyhledávání (binary search) má logaritmickou dobu běhu.
Důkaz:
Binární vyhledávání je hledání hodnoty v se seřazeném seznamu. Dělá to tak, že seznam půlí a pak srovnává v které polovině se může hodnota nacházet. Dobu vyhledávání tudíž bude určená tím, kolikrát musí algoritmus rozpůlit seznam o velikosti n na dvě půlky, aby se dostal k jediné hodnotě. Vyjádřeno s dobou vyhledávání k:

$$n/(2^k) = 1$$

Úpravy

$$2^k = n$$

$$k = log_2 n$$

Základ logaritmu se vynechá, tudíž **O(log n)**.

Nepolynomiální složitosti
O(2^n) exponenciální doba složitosti (nalezení všech posloupností délky N složených z nul a jedniček)
O(n!) faktoriálová doba složitosti (nalezení všech permutací N prvků, tedy například všech přesmyček slova o N různých písmenech)

Přehled rychlostí a délek běhu programu pro jednotlivá n:

funkce / n	10	20	50	100	1 000	10^6

$\log_2 n$	3.3 ns	4.3 ns	4.9 ns	6.6 ns	10.0 ns	19.9 ns
n	10 ns	20 ns	30 ns	100 ns	1 µs	1 ms
$n \cdot \log_2 n$	33 ns	86 ns	282 ns	664 ns	10 µs	20 ms
n^2	100 ns	400 ns	900 ns	100 µs	1 ms	1 000 s
n^3	1 µs	8 µs	27 µs	1 ms	1 s	10^9 s
2^n	1 µs	1 ms	1 s	10^{21} s	10^{292} s	≈ ∞
$n!$	3 ms	10^9 s	10^{23} s	10^{149} s	10^{2558} s	≈ ∞

Pro představu: 1 000 s je asi tak čtvrt hodiny, 1 000 000 s je necelých 12 dní, 10^9 s je 31 let a 10^{18} s je asi tak stáří Vesmíru. Takže nepolynomiální algoritmy začnou být velmi brzy nepoužitelné.

O() není univerzální měřítko pro posouzení efektivity kódu, nezohledňuje totiž faktory jako přehlednost algoritmu, povahu vstupních dat, etc.
Stejně jako je **O()** pro posouzení časové složitosti, může být také **O()** pro posouzení paměťové složitosti.

“Metoda kouknu a vidím“, kterou můžeme použít na určování časové složitosti u těch nejjednodušších algoritmů. Spočívá jen v tom, že se podíváme, kolik nejvíc obsahuje náš program vnořených cyklů. Řekněme, že jich je k a že každý běží od 1 do N. Potom za časovou složitost prohlásíme Nk.

6 Datové struktury (Honza)

Pole

wiki

- array - [0,0,0] ; [1142,1231,123123]
- Vzniklo za účelem reprezentace vektorů
- Operace:
    - Čtení prvku **O(1)**
    - Úprava prvku **O(1)**
    - Odebrání prvku
        - Z prostřed **O(n)**
        - Z konce **O(1)***
    - Přidání prvku
        - Doprostřed **O(n)**
        - Nakonec **O(1)***

    * Aby se v počítači mohlo pole zvětšit, tak se musí vytvořit nové v čase **O(n)** ale pomocí amortizace (ve zkratce stačí, když při každém zvětšování velikost zdvojnásobíme, tím pádem pudou některá přidávání pomalá ale vždy nejvýš každá **O(n)** - tá)

Spojový seznam

wiki

- Jiná implementace “pole”, která umí rychle přidávat a ubírat prvky.
- Ke každému prvku v seznamu je přiložený taky odkaz na další prvek v seznamu (single linked list), či dokonce k předchozímu prvku v seznamu (doubly linked list - logicky méně paměťově efektivní)
    - K tomu se logicky váže i časová složitost čtení a zápisu na index závisející na počtu prvků v seznamu
    - Manipulace s prvkem v seznamu se dotýká pouze sousedních prvků, narozdíl např. Od pole
- Tail - konec seznamu
- Head - začátek seznamu
- Operace:
    - Čtení/úprava prvku
        - Z konce/začátku **O(1)**
        - Z prostřed **O(n)**
    - Odebrání/přidání prvku (pokud máme pozici sousedních prvků v paměti): **O(1)**

Fronta

wiki

- FIFO - First in, first out
- Odebíráme data ve stejném pořadí, v jakém jsme je uložili - pracovat lze pouze s čelním prvkem
- Operace:
    - Přidání prvku **O(1)**

    - Odebrání prvku **O(1)**

Zásobník

wiki

- LIFO - Last in, first out
- Odebíráme data v opačném pořadí, než v jakém jsme je uložili - pracovat lze pouze s prvkem,
- který je na vrchu zásobníku
- Použití: náhrada rekurze, vyhodnocování výrazů
- Operace:
    - Přidání prvku **O(1)**
    - Odebrání prvku **O(1)**

Prioritní fronta

wiki

- Každý prvek má přiřazenou prioritu, dostává se tím pádem na začátek
- Operace (v případě implementace pomocí Haldy):
    - Přidání prvku **O(log n)**
    - Odebrání prvku **O(log n)**

Množina (set / multiset)

wiki

- Pamatuje si pouze jestli daný prvek obsahuje (nebo kolikrát ho obsahuje) neumí si zapamatovat pořadí.
- Operace (implementace pomocí hashování):
    - Přidání prvku: **O(1)** *
    - Odebrání prvku: **O(1)** *

  * Pouze průměrné složitosti - pokud budeme mít smůlu a budou nám kolidovat hashe, tak mohou být operace v nejhorším případě až **O(n)**
- Operace pomocích vyhledávacích stromů (BST) jsou v čase **O(log N)**

Mapa / slovník / asociativní pole

wiki

- klíč - hodnota, h(klíč)
- Funguje podobně jako Množina, jen si ke každému prvku (klíčí) pamatuje nějakou hodnotu
- Operace (implementace pomocí hashování):
    - Přidání prvku: **O(1)** *
    - Odebrání prvku: **O(1)** *

  * Pouze průměrné složitosti - pokud budeme mít smůlu a budou nám kolidovat hashe, tak mohou být operace v nejhorším případě až **O(n)**
- Operace pomocích vyhledávacích stromů (BST) jsou v čase **O(log N)**

Strom

wiki

- Každý prvek má nejvýše jednoho předchůdce a může mít více než jednoho následníka
- Uzly - vrcholy
- ‘otcové’ a ‘synové’, ‘stromy’ a ‘listy’ - uzly bez následníka, ‘kořen’ - uzly bez předchůdce
- Unární, binární a ternární stromy (počet max. následníků 1-3). Více než ternární se zpravidla nepoužívá
- ## Halda

  wiki
  - Vyvážený strom - vždy celý zaplněný
  - Rodič uzlu nese stejnou nebo vyšší hodnotu než uzel sám
  - Lze jednoduše implementovat pomocí pole
  - Používá se v algorytmu HeapSort
  - Operace:
    - Vytvoření **O(n)**
    - Přidání prvku **O(log n)**
    - Odebrání prvku v kořenu (maximum) **O(log n)**
- ## BST - Binary search tree

  wiki
  - Oproti haldě je přesně dané, kde prvek leží
  - Rekurzivně platí, že v levém podstromu jsou všechny prvky menší a pravém jsou všechny prvky větší (nebo obráceně)
  - Je nutné řešit vyvažování stromu aby se z toho nestala nudle (potom by hledání trvalo **O(n)**)
  - Vyhledávání prvků je potom **O(log n)**
- ## B-stromy
  - V uzlu se ukládá více než jeden prvek a mohou mít více než 2 potomky
  - Optimalizace kvůli HW
    - například z HDD se čte po blocích (obvykle 4 kb nebo i více) a potom je neefektivní číst malé množství dat, protože se čte stejně celý blok
    - U B-Stromů vyplní jeden vrchol celý blok a tak jsou o něco rychlejší (stejná časová složitost ale nižší konstanta)

7 Grafy (Honza)

Graf reprezentuje objekty (vrcholy) a vztahy (hrany) mezi nimi

Formální definice

Graf (neorientovaný) je dvojice $G = (V, E)$, kde V je množina objektů (vrcholů) a E je množina vztahů (hran), kde vztah (hrana) je množina 2 objektů (vrcholů).

Orientovaný graf od normálního grafu (neorientovaného) se liší tím, že hrany nejsou množiny ale seřazené dvojice (záleží na pořadí vrcholů na hraně).

Pograf S grafu G je graf jehož vrcholy jsou podmnožinou vrcholů grafu G a jehož hrany jsou podmnožinou hran vrcholu G .

Stupeň vrcholu značí kolik hran obsahuje tento vrchol

Různé specifické druhy grafů

Souvislý graf - mezi každýma dvěma vrcholy existuje cesta
Cesta - souvislý - 2 "krajní" body mají stupeň 1 a všech ostatních $n-2$ vrcholů má stupeň 2
Úplný graf - mezi každou dvojicí vrcholů vede hrana
Cyklus - alespoň 3 vrcholy - všechny vrcholy mají stupeň 2
Cyklický graf - obsahuje nějaký podgraf, který je cyklus
Acyklický graf - není cyklický
Souvislá komponenta - takový podgraf, který je souvislý a zároveň z něj nevede cesta do žádného vrcholu, který nepatří do tohoto podgrafu
Strom - souvislý acyklický graf - viz datové struktury

Způsoby procházení grafů

Mějme nějakou datovou strukturu, kam si ukládáme vrcholy, které chceme projít.
Na začátku tam je jen jeden vrchol (ten v kterém začínáme)
Potom vždy vezmeme nějakým způsobem vrchol z datové struktury, označíme zo prohledaný (případně na něm uděláme nějaký výpočet) a přidáme do datové struktury, všechny jeho sousedy, kteří tam ještě nejsou.

Breadth first search

- Prohledávání do šířky
- Jako datovou strukturu používáme frontu FIFO.
- Nejdříve projdeme všechny prvku vzdálené 1 hranu, potom všechny vzdálené 2 hrany, 3, 4, 5, …..

Depth first search

- Prohledávání do hloubky
- Jako datovou strukturu používáme zásobník FILO

- Funguje jako backtracking, vždycky si vybíráme nějakou cestu a až když nikam nemůžeme, tak se vracíme, dokud nenarazíme na nějakou cestu, kterou se můžeme vydat.

8 Softwarový návrh - ROZDĚLANÉ

9 Objektově orientované programování (Tonda)

Zdroj

Opakem je procedurálně orientované programování.
Paradigma preference dat nad činností. Akci předchází definice datových typů.

Class (třída) předloha souboru dat
    Field datová hodnota určitého typu
    Method související funkce

Object/Instance existující soubor dat

Objekty/Instance se tvoří na základě třídy. Vytvořený objekt/instance má své vlastní, od ostatních oddělené, fieldy. Metody jsou zůstávají stejné pro všechny objekty jedné třídy.

Princip encapsulation fieldy třídy by měly být spravovány pouze metodami třídy.
Příklad: `cat.lives = cat.lives - 1` *je špatně,* `cat.kill()` *je správně.*
Public členové třídy by měly být přístupné odevšad, tj. Optimálně pouze metody
Private členové třídy by měly být přístupné pouze v rámci třídy, tj. Optimálně fieldy a některé metody

Princip dědičnosti dědící třída dostává veškeré členy od vyšší třídy. Navázáno na sebe může být více tříd. Pro označení vztahů mezi třídami používáme pojmů subtype/supertype, descendant/ancestor a v případě přímé návaznosti parent/child.

V metodě je vždy dostupná jako argument instance třídy. *V Javě* `this`*, v Pythonu* `self`*.*

Constructor je metoda, která se zavolá při tvorbě nové instance. Zpravidla jen nastavuje defaultní hodnoty fieldů, ale může dělat víc.

Abstraktní třída netvoří instance a pouze se z ní dědí.

10 Logické obvody (Filip)

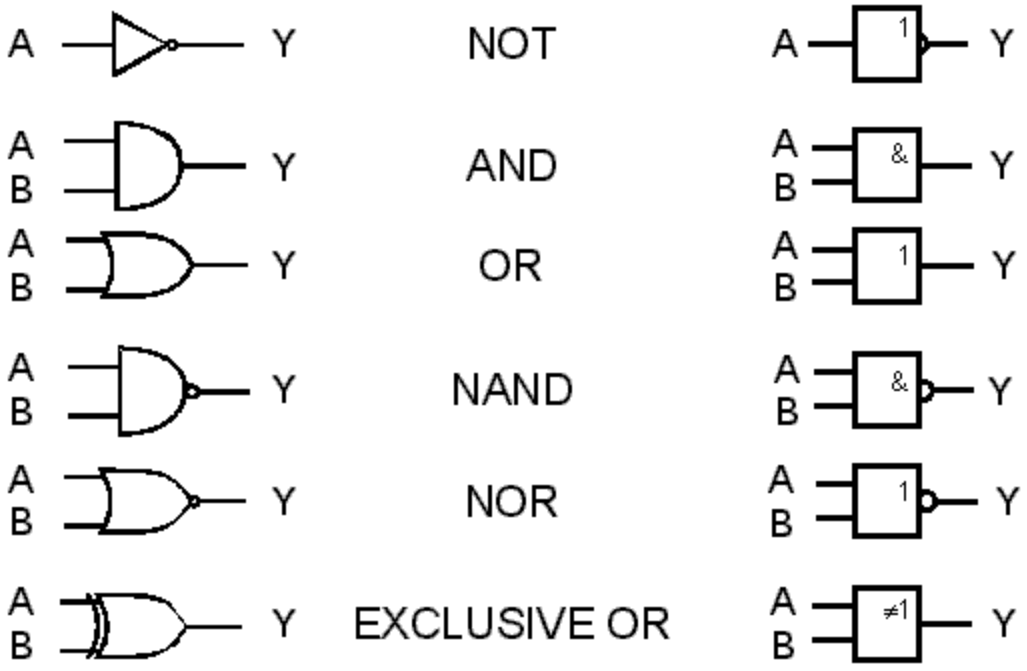
- Logická funkce bere jako vstup jednu nebo více binárních hodnot ano/ne a následně dává ano/ne hodnotu jako výstup
    - Skládá se z logických členů neboli hradel
- **Kombinační obvody** berou ohled jenom na vstupy. **Sekvenční obvody** již pracují i se svým předchozím stavem (jinak řečeno: mají paměť) (to už se ale dostáváme ke konečným automatům).

Hradla

- Hradlo je možné realizovat zapojením aktivních součástek, tranzistorů, diod, rezistorů či dalších pasivních součástek.
- Často se lze setkat s logickými členy ve formě integrovaných obvodů.

Druhy

- NOT - Invertor
- Opakovač/Repeater
- AND - Konjuktor (logické násobení)
- NAND - Invertovaný AND
- OR - Disjunktor (logický součet)
- NOR - Invertovaný OR
- XOR - Exkluzivní logický součet
- XNOR - Exkluzivní invertovaný logický součet



- Jednotlivá hradla jdou složit z jiných hradel
    - Jenom pomocí NANDů, NORů nebo XORů lze složit všechna ostatní hradla
    - Váš procesor je nejspíš složen jenom z NANDů, protože ty jsou nejlevnější
      (tvrdil Šimon nebo Emil)

Karnaughova mapa

- Používá se ke zjednodušení logické funkce
    - Abychom nemuseli skládat složitý obvod, když menší a tudíž levnější dává pro stejné vstupy stejné výstupy

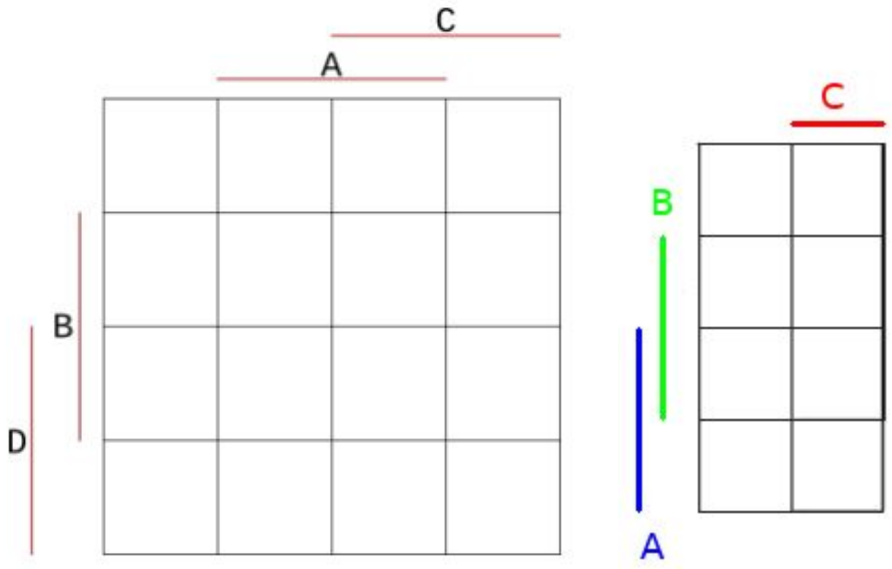
Postup

Všichni máte v emailu Emilův návod, jak použít Karnaughovu mapu na minimalizaci driveru segmentu osmisegmentového displaye, ale já to tady stejně popíšu znovu, stručněji a snad tak, abyste cross-referencí těchto dvou zdrojů z toho všeho třeba byli moudřejší

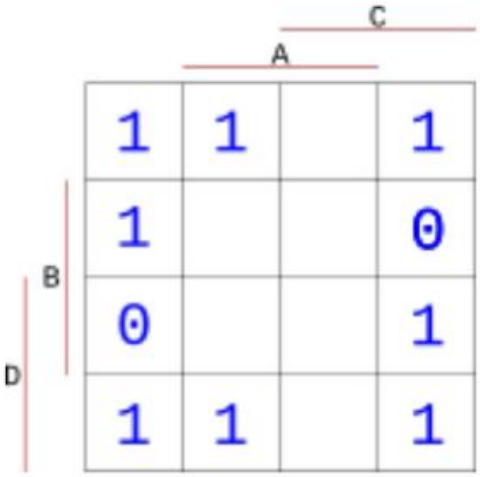
1) Udělám si tabulku všech možných kombinací vstupů a k nim, jaký výstup mají dávat
    - Všimněte si, že v Emilově příkladu (zde) není 16 hodnot (tedy všechny možné kombinace), ale jenom 10. To proto, že jemu na zbylých 6 hodnotách nezáleží.

x	A	B	C	D	Q
0	0	0	0	0	1
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	1
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	1
8	1	0	0	0	1
9	1	0	0	1	1

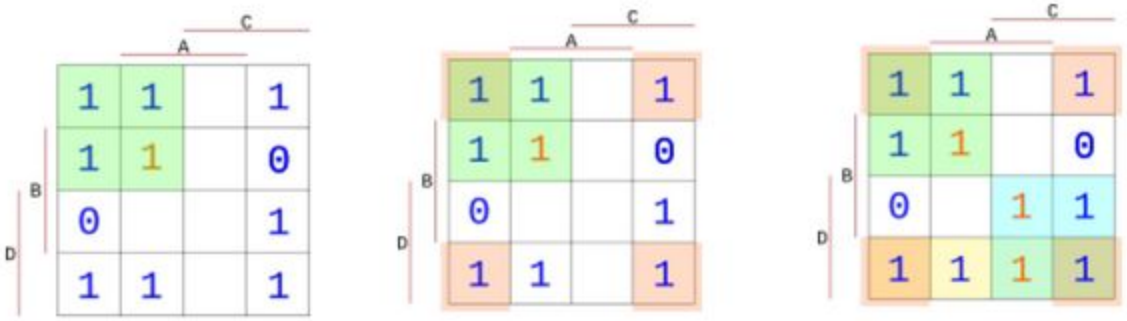
2) Načrtneme si 4x4 Karnaughovu mapu
    - Protože máme 4 vstupy
    - Kdybychom měli třeba vstupy 3, načrtli bychom si 4x2 mapu
    - Jde nám prostě o to, mít políčko pro každou kombinaci vstupních hodnot

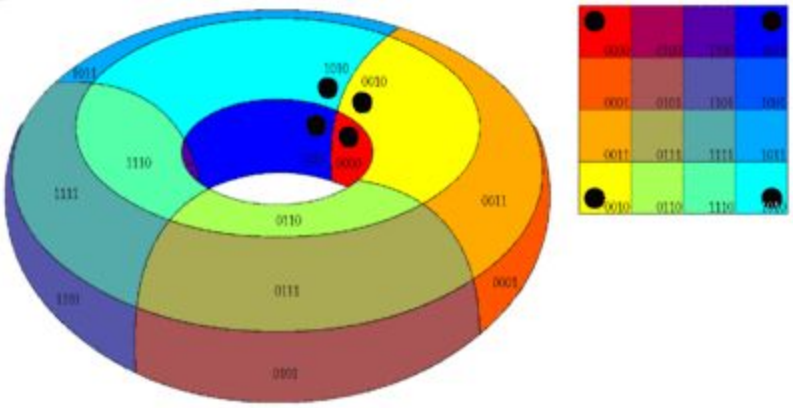


3) Mapu vyplníme
    - Každé políčko náleží nebo nenáleží každé ze 4 vstupních hodnot
        - Všechna políčka v druhém a třetím sloupci náleží A a tedy reprezentují stavy, kde vstup A je 1
    - Podíváme se tedy na tabulku vstupů a výstupů, kterou jsme si udělali v prvním kroku a podle ní do mapy doplníme buď jedničky nebo nuly



4) Jedničky seskupíme do bloků
   - Jsou vždy obdélníkové
   - Mohou se loopovat přes okraje/rohy tabulky (viz obrázek donutu)
   - V zájmu minimalizace to musíme vymyslet tak, aby bloků bylo co nejméně a aby byly ideálně 2x2 nebo 4x1





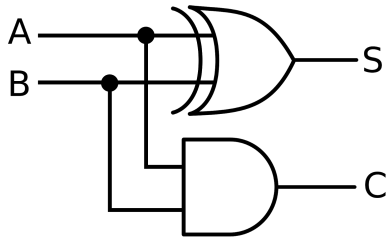
5) Podle bloků sepíšeme výslednou logickou funkci
   - Nejprve vyjádříme jednotlivé bloky pomocí logických operací
     - Např. oranžový blok bude $o = \bar{A} * \bar{B}$
   - Výsledná funkce pak bude logickým součtem jednotlivých bloků

$$f = \bar{C} * \bar{D} + \bar{A} * \bar{B} + D * C + D * \bar{B}$$

Sčítačka

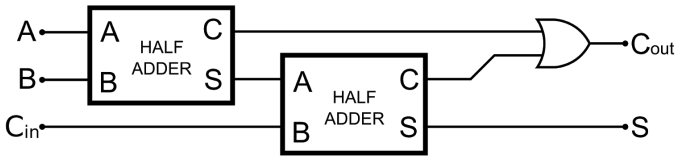
- Přítomná v každém procesoru, jelikož tímto obvodem se provádí sčítání binárních čísel.

Poloviční sčítačka

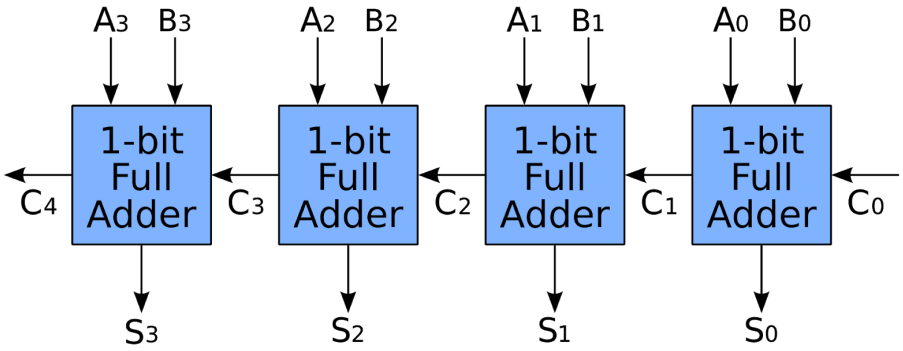


- Jako vstup bere dva sčítané bity
- Jako výstup dává výsledek a carry bit (“1 + 1 je 0, *držím si jedničku*”)
- Nemá ale žádný carry vstup

Úplná sčítačka



- Už bere jako vstup i carry bit
- Když těchhle sčítaček dáme více vedle sebe a propojíme jejich carry vstupy a výstupy, můžeme pomocí nich sčítat i vícebitová binární čísla (první carry in a poslední carry out zůstanou nezapojené)



11 Reprezentace čísel v paměti - ROZDĚLANÉ (Bohouš)

========== od Honzy, předělám ↓↓↓

DEC → lib. soustava

- opakuj, dokud nedostaneš 0:
- číslo vydělím vždy základem soustavy - doprava napíšu zbytek - doleva napíšu podíl - výsledné číslo čteme odspodu

unsigned - x bajtové číslo bez znaménka

- převedu číslo, doplním zleva na x*8 bitů <0; 255>

signed - x bajtové číslo se znaménkem

- kladné číslo:
  - převedu číslo, doplním zleva nulami na x*8bitů <-127; 127> - záporné číslo:
- převedu číslo, doplním zleva nulami - Odečtu 1 - číslo zneguji - tzn. že kladná čísla mají na začátku 0 a záporná 1

desetinná čísla

- funguje stejně jako v dec - opakuj dokud nenajdeš periodu nebo 0:
- vynásob základem soustavy - sepiš a odečti celou část - př. 3+(5⁄8) = 11,101 (1 polovina, 0 čtvrtin, 1 osmina) - (1⁄3) = - 21/10 = 2+(1/10) = 10, 0011 0 - 5⁄8 = 0,101 - 1/7 = 0,001 - 1/9
- 0, 0 0 0 1 1 1 0 - 1/9 2/9 4/9 8/9 16/9 14/9 10/9 2/9 ⇒ 0,000111 - IEEE 754 - online converter

endianita

- určuje, v jakém pořadí bude vícebitové číslo - resp. pořadí bitů - jednotlivé bity se vždy čtou zleva do prava - Big endian - “zleva doprava” - little endian - “zprava doleva”

Float
Double

========== markdown, předělám ↓↓↓

Reprezentace čísel v paměti

>> tabulka "každý s každým" - složitost o(n^2)

- dělíme dvěma v oboru celých čísel
  - když je zbytek, napíšu 1
  - když není zbytek, napíšu 0

- unsigned integer
- signed integer
- float [IEEE 754] (https://cs.wikipedia.org/wiki/IEEE_754)

převod z jakékoliv soustavy do jakékoliv jiné soustavy
číslo/Z = a
a/Z = b , sepíšu zbytek
b/Z = c , sepíšu zbytek
c/Z ...
- zbytky čtu zdola
- zapíšu do řady s významem z^4, z^3, z^2, z^1, z^0

========== markdown, předělám ↓↓↓

Reprezentace čísel v paměti

Číselné soustavy

- převod na soustavy dělením základem a vypisováním zbytku \color{#fff}{foto}*foto*

Formáty

char

1B = 8b

short

2B = 16b

signed

\+ nebo \-

pokud začíná 1 => znegovat a přičíst 1

Necelá čísla ve dvojkové soustavě

- převod na soustavy násobením základem a vypisováním celé části výsledku \color{#fff}{foto}*foto*

Endianita

Big endian

- osmice bitů (Byty) jsou v "normálním" pořadí

Little endian

- Byty jsou v opačném pořadí

12 Kódování znaků (Honza)

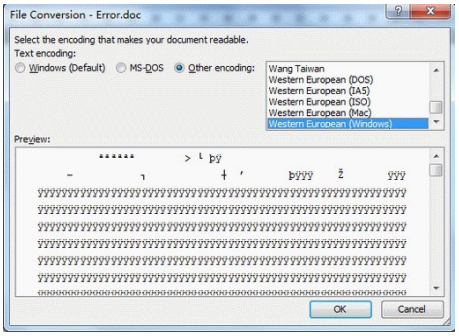
- Libovolné kódování textu (pro různé účely)
    - Patří sem i Morseova abeceda a Braillovo písmo
- Nejznámější ASCII a Unicode

ASCII

- (American Standard Code for Information Interchange) 1960
- 7 bitů 128 znaků (byte byl poprvé zmíněn 1956)
    - 1960 se stále bral za základní datovou jednotku 1 bit a každý si vytvářel vlastní kódování (třeba po 6 nebo 5 bitech)
- každý znak je reprezentován číslem 0 - 127
- číslo je pak zakódováno do bitů pomocí dvojkové soustavy:

Binary	Oct	Dec	Hex	Glyph		
				1963	1965	1967
010 0000	040	32	20	space		
010 0001	041	33	21	!		
010 0010	042	34	22	"		
010 0011	043	35	23	#		
010 0100	044	36	24	$		
010 0101	045	37	25	%		

- písmena (a-zA-Z), čísla (0-9), znaky (!#$% ...), řídící signály (návrat na začátek řádku, nová řádka, výstražný zvonek ...)
- známé rozsahy:
    - 48 - 57 … ‘0’ - ‘9’
    - 65 - 90 … ‘A’ - ‘Z’
    - 97 - 122 … ‘a’ - ‘z’
- kvůli potřebě kódovat i speciální znaky si mnoho národů a výrobců vytvořilo pro své potřeby upravenou ASCII tabulku (někdy i více)
- byla spousta problémů se špatným výběrem kódování (verze tabulky):



Unicode

- Pouze standard z kterého vychází jednotlivá kódování (UTF-8, UTF-16, UTF-32)

    - Hodnotu znaku (číslo) určuje standard unicode ale to jak se toto číslo zakóduje určuje kódování
- 1988 - snaží se vyřešit problémy s ASCII
- Snaží se reprezentovat všechny znaky všech jazyků a nějaké emojis k tomu (1 112 064 znaků)

UTF-8

- Je založen na bytech a každý znak má proměnlivou délku 1 - 4 byte
- Nejpoužívanější kódování na webu a v *nixovém světě (OSX & GNU/Linux)
- První byte obsahuje informaci o délce a navazující byte vždy začíná prefixem 10:

Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF[13]	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

UTF-16

- Proměnlivá délka 2 nebo 4 byte
- Používá se pro zakódování čistého textu v ekosystému microsoft windows (na OSX a linuxu skoro vůbec)
- Některé programy ho používají interně

UTF-32

- Neproměnlivá délka 4 byte
- Neproměnlivá délka je velkou výhodou pokud chceme najít n-tý znak řetězce, tak umíme jednoduše spočítat jeho pozici v poli 4*n
- Není však příliš efektivní (prvních 11 bitů je vždy 0)
- Některé programy používají UTF-32 interně (většina potom prvních 11 nul šikovně ořízne nebo nějak využije)

13 Architektura počítačových systémů - ROZDĚLANÉ (Bohouš)

Von Neumannova architektura

- návrh programovatelného univerzálního počítače - Části:
- ALU - arithmetic logic unit - provádí výpočty - operační paměť - výstupní zařízení - vstupní zařízení - řadič
- ALU + řadič + paměti = dnešní CPU - DMA - Direct memory access
    - přenos dat rovnou mezi op. pamětí a vstupem

Harvardská architektura

- části stejné jako u Von Neumanna - Avšak paměť je rozdělena na:
    - datovou paměť - data ukládaná programem
    - programovou paměť - ukládání programu

Multitasking

- řadič přepíná běh více programů - nikdy neběží 2 programy zároveň

14 Procesor (Tonda)

CPU = Central processing unit, obecně známo jako procesor
Procesor - obvod obsahující zejména logické součástky - historicky elektronky, nyní tranzistory. V rámci zmenšení se se začaly vyrábět jako integrované obvody (IC).

Assembly

Procesor obsahuje Instrukční sadu + speciální instrukce pro optimalizaci - opcode (operation code)
Assembly je lidsky stravitelnější verze opcodu, tudíž se jedná o low-level programovací jazyk.
Příkaz v procesoru se skládá z opcodu a adres.

High level Program *-kompilátor->***Low level** Assembly *-assembler->* Opcode -> Binární kód

Základní operace

Assembly jazyky jsou specifické pro počítačové architektury

- LOAD - načte hodnotu na adrese z ram a uloží do registru
- STORE - uloží hodnotu z registru na adresu
- ADD - sečte hodnoty dvou registrů a uloží do jednoho z nich
- SUB - odečte hodnoty dvou registrů a uloží do jednoho z nich
- JUMP - přesune se na adresu, varianta JUMPIF provede přesun za splnění dané podmínky závisejících na flags ALU
- HALT - zastaví program

Stavba

RAM - Random Access Memory
- Každá hodnota v RAM má svou adresu, přístup k hodnotám trvá konstatní dobu.
- Fyzická úložiště hodnot v RAM jsou poskládána v síti. O zajištění přístupu k nim se stará logický komponent zvaný **Multiplexor**.
- RAM je přímo napojena na CU a pracovní registry. CU s ní pracuje Read-enable a Write-enable kabely.

CU - Control unit
- Zpracovává operace, které načítá z paměti a řídí ostatní komponenty
- V příkazovém registru (IR) se nachází aktuální příkaz skládající se z opcodu a adres registrů/paměti
- V Program Counteru (PC) registeru je uložená nadcházející adresa příkazu

ALU - Arithmetic Logic Unit
- Provádí aritmetické a logické operace, z procesoru má jako input opcode a dvě hodnoty
- Outputuje výsledek operace a mimo jiné vrací i “flags” - binární informace o výsledku, např.: je 0, vstupy se rovnají, první vstup je menší než druhý vstup, overflow

Pracovní registry
- Označeny A, B, C, D, …
- Ukládají se do nich hodnoty, se kterými CU aktuálně pracuje

Hodiny
- Vysílá pravidelné impulsy, čímž synchronizuje zpracovávání instrukcí na jednotlivých komponentech

Sběrnice

- Spojuje jednotlivé komponenty. Aby nebylo nutné na každý komponent navádět vlastní set drátů, všechny jsou napojeny na ten stejný set a jsou aktivovány read-enable/write-enable dráty.

Cache

- Leží přímo na CU, je po registrech druhým nejrychlejším úložištěm
- Kopíruje se do ní kusy dat z RAM, se kterými CU aktuálně pracuje
- Změny mezi Cache a RAM se musejí synchronizovat

15 Paměťová hierarchie (Gabriel) - ROZDĚLANÉ

- Paměťové zařízení umožňuje ukládání a čtení dat. Pro ukládání se obvykle používá binární číselná soustava (dva logické stavy), protože se snadno realizuje v elektronických obvodech. 1 bit - 1 číslice binární soustavy.
- Fyzikálně tedy většinou: přítomnost nebo velikost el. náboje, směr nebo přítomnost magnetického toku (magnetická vrstva disku nebo pásky), propustnost nebo reflexe světla (např. CD ROM)
- Volatilní / nevolatilní - k uchování dat vyžaduje / nevyžaduje napájecí napětí

Druhy pamětí

- Magnetické (například páska[1], diskety, HardDiscDrive[2])
- Elektronické (například Flash, RAM, ROM, WOM)
- Optické (například laserdisc, CD, DVD, Blu Ray)
- Mechanické (například děrné štítky, gramofonové desky)

- Vnitřní - registry, cache, RAM
- Vnější - pevné disky

ROM = Read Only Memory

- např. paměť na základovce - používá se pro uložení Firmware

- PROM - jednou data uložím, vypálí se do paměti
- EPROM - PROM přepsatelná UV
- EEPROM - PROM přepsatelná elektrickým nábojem (- dnes frekventované, bývá tam uložený např. BIOS)
- Např. obyčejné CD a DVD, Blu Ray, ROM

REGISTRY

- Malé ale rychlé paměťové buňky přímo v mikroprocesoru
- K ukládání operandů a výsledků aritmetických a logických operací
- Nejrychlejší paměť připojená k procesoru (stejně rychlá, jako procesor)

CACHE

- vyrovnávací paměť v procesoru, urychlení komunikace s pamětí
- Rychlá statická paměť
- Zařazena mezi dva subsystémy s různou rychlostí a vyrovnává tak rychlost přístupu k informacím - většinou berou data z RAM

[1] Dodnes nejbezpečnější druh paměti
[2] (disc - kulatý, drive - hranatý -> DiscDrive)

RAM

- polovodičové paměti s přímým přístupem umožňující čtení i zápis.
- pomalejší než procesor, rychlejší než vnější paměti - složena z tranzistorů, kondenzátorů
- DRAM - dynamic RAM - obsahují 1 tranzistor - musí se častěji refreshovat - jsou ale jsou levnější, menší, jednoduše vyrobitelné
- SRAM - static RAM - obsahuje 6 tranzistorů - nemusí se tak často refreshovat, rychlejší - je ale dražší (víc kondenzátorů ap.), je větší
- Ve von Neumannově architektuře používána pro programy i pro data

HDD

- Nejpomalejší
- Data jsou na pevném disku uložena pomocí magnetického záznamu. - Stopy - poloměr - Sektory - dokola

FLASH

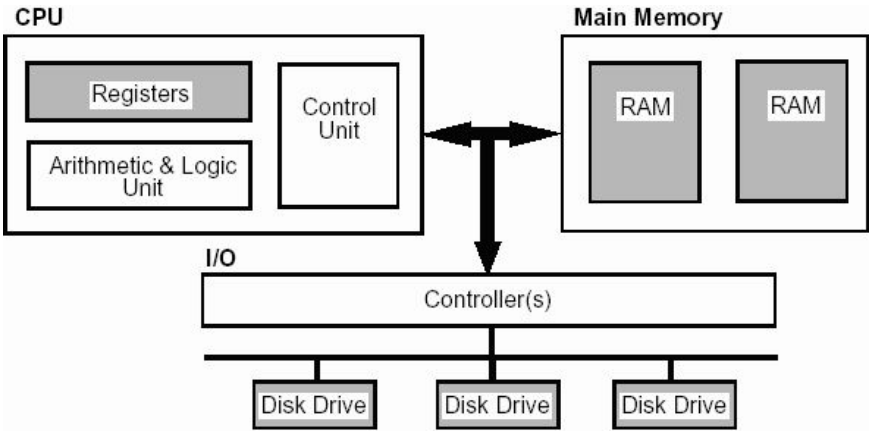
- Nevolatilní, vnitřně organizována po blocích - každý blok lze plnit samostatně. Lze přepisovat jednotlivé bloky.
- Podobně jako RAM, nemá bit select tranzistor - SLC uloží 1 bit do 1 buňky - DLC uloží 2 bity do 1 buňky - nejpoužívanější - TLC uloží 3 bity do 1 buňky - QLC uloží 4 bity do 1 buňky. Hustota bitů v buňce má ale někdy za následek stabilitu a rychlost čtení.

SSD

- Paměť typu flash - čistě elektronické, vysoká rychlost, malá spotřeba, menší maximální počet zápisů do jednoho místa - rovnoměrné rozmisťování dat

CD

- Citlivá vrstva, do které laser vypálí díry – 1 a 0 - přemazatelné - fotopolymer, jehož teplotou se mění odrazivost světla



16 Teorie jazyků a gramatik (Tonda)

Jedná se o výpisky definic z dokumentu od Šimona. Pro hlubší vysvětlení doporučuji přečíst celý text ZDE. Dokument prosím na žádost Šimona nešiřte.

Formální jazyk

Σ Abeceda, konečná množina platných znaků

ε Prázdný řetězec

ω Slovo, konečný řetězec znaků

Σ^* Množina všech možných slov včetně prázdného řetězce

Σ^+ Množina všech možných slov bez prázdného řetězce

$L \subseteq \Sigma$ Jazyk je podmnožinou všech možných řetězců znaků a má nějakou svou charakteristiku.

Např.:
Slova mají vždy sudý počet symbolů nebo *Slovo vždy začíná symboly "xXX" a končí symboly "XXx"*

$G = \{N, \Sigma, P, S\}$ Gramatika popisuje pravidly charakteristiku formálního jazyka

N Konečná množina neterminálních znaků

Σ Množina terminálních znaků, množiny N a Σ se nepřekrývají

P Konečná množina přepisovacích pravidel v základním tvaru:

$(\Sigma \cup N)^* N (\Sigma \cup N)^* \rightarrow (\Sigma \cup N)^*$

S Počáteční znak, z množiny N

Při generování řetězců začíná gramatika s počátečním symbolem S. Následně je vybráno nějaké přepisovací pravidlo obsahující na levé straně neterminální symbol S. To se aplikuje, tedy dojde k přepsání znaku S na pravou stranu pravidla. Tento postup se opakuje až do chvíle, dokud není řetězec tvořen pouze terminálními symboly.

Terminální symboly se nikdy nemohou nacházet samostatně na levé straně pravidel.

Chomského hierarchie

0. neomezená, jestliže pravidla vyhovují obecné definici gramatiky.

1. kontextová, jestliže každé pravidlo z P má tvar $\gamma A \delta \rightarrow \gamma \alpha \delta$, kde $\gamma, \delta \in (N \cup \Sigma)*$,

$\alpha \in (N \cup \Sigma)+$ a $A \in N$, nebo tvar $S \rightarrow \varepsilon$, pokud se S nenachází na pravé straně jiného pravidla.

2. bezkontextová, jestliže každé pravidlo z P má tvar $A \rightarrow \alpha$, kde $A \in N$ a $\alpha \in (N \cup \Sigma)*$.

3. regulární, jestliže každé pravidlo z P má tvar $A \rightarrow aB$, nebo $A \rightarrow a$, kde $A,B \in N$ a $a \in \Sigma$, nebo tvar $S \rightarrow \varepsilon$, pokud se S nenachází na pravé straně jiného pravidla.

Hierarchie je přirozeně propustná ve směru odspodu nahoru, tedy každá regulární gramatika je jistě zároveň bezkontextová, kontextová i neomezená. V opačném směru toto platit nemusí, tedy neplatí, že každá kontextová gramatika je zároveň bezkontextová.

Na základě hierarchie gramatik lze podobně klasifikovat i formální jazyky:

0. rekurzivně spočetný, jestliže existuje neomezená gramatika, která ho generuje.
1. kontextový, jestliže existuje kontextová gramatika, která ho generuje.
2. bezkontextový, jestliže existuje bezkontextová gramatika, která ho generuje.
3. regulární, jestliže existuje regulární gramatika, která ho generuje.

Jazyky a stroje

Stroje, které lze využít na ověření platnosti řetězce v daném jazyku, dle složitosti jazyka:

Gramatika	**Jazyk**	**Stroj**
Neomezená	Rekurzivně spočtený	Turingův stroj
Kontextová	Kontextový	Lineárně omezený Turingův stroj
Bezkontatová	Bezkontaxtový	Zásobníkový automat
Regulární	Regulární	Konečný automat

Turingův stroj je výpočetní model, který je tvořen nekonečnou páskou, množinou stavů a čtecí hlavou. V každém kroku je přečten jeden znak z pásky. Na základě přečteného znaku a aktuálního stavu počítače dojde ke změně stavu, zapsání nového symbolu na přečtenou pozici pásky a pohybu čtecí hlavy o jednu pozici na libovolnou stranu.
Lineární Turingův stroj se od normálního Turingova stroje liší pouze v tom, že nemá neomezeně dlouhou pásku, ale jeho páska je lineárně omezena délkou vstupu.
Zásobníkový automat je opět teoretický výpočetní model obsahující množinu stavů. Jeho program je zadán jako množina přechodů závisející na obsahu zásobníku, čteném znaku a aktuálně čteném znaku.
Konečný automat *popsán v další otázce*

17 Konečné automaty - ROZDĚLANÉ (Gabr)

Konečný automat je teoretický výpočetní model používaný v informatice pro studium formálních jazyků.

- Primitivní počítač, skládající se z několika stavů a přechodů, mezi stavy přechází na základě symbolů, které čte ze vstupu.

- Množina stavů je konečná (odtud název) - konečný automat nemá žádnou další paměť, kromě informace o aktuálním stavu.

- Dokáže rozpoznávat pouze regulární jazyky

- Používají se při vyhodnocování regulárních výrazů, např. jako součást lexikálního analyzátoru v překladačích.

- V informatice se rozlišuje kromě základního deterministického či nedeterministického automatu také automat Mealyho a Mooreův.

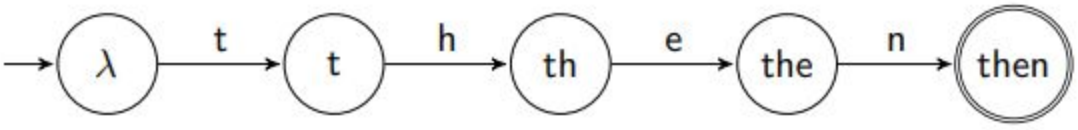
Automat je:

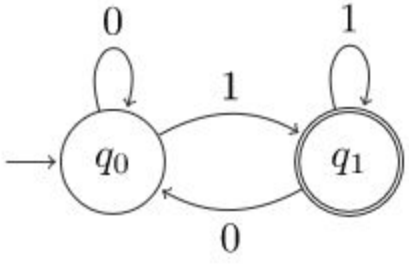
- abstraktní - nezajímá nás realizovatelnost
- diskrétní - má pevně dané vnitřní stavy mezi kterými přechází
- sekvenční - řeší pouze jeden vstup postupně za sebou
- dynamický - jeho stav se mění v čase
- deterministický - když předložím stejné vstupy, mohu očekávat vždy stejné výstupy

Formálně je konečný automat definován jako uspořádaná pětice (S, Σ, σ, s, F), kde:

- S je konečná neprázdná množina *stavů*.
- Σ je konečná neprázdná množina vstupních symbolů, nazývaná *abeceda*.
- δ je přechodová funkce (též přechodová tabulka), popisující pravidla přechodů mezi stavy. Pro deterministický automat: δ: Q x $\Sigma \rightarrow$ Q, pro nedeterministický automat: δ: Q x $\Sigma \rightarrow 2^Q$, viz níže.
- s je *počáteční stav*, $s \in S$.
- F je množina *přijímajících - finálních stavů*, $F \subseteq S$.

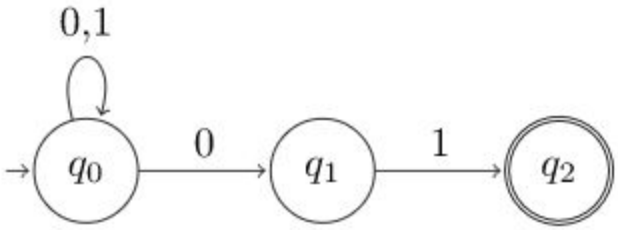
PŘÍKLAD





- Automat pracující s binární abecedou -> Σ = {0,1}, přijímající pouze slova končící na 1
- Dva stavy - počáteční q0, koncový q1
- Pokud slovo nekončí na 1 automat skončí v nekoncovém stavu q0

NFA - nedeterministický konečný automat



- Není zřejmé, co bude automat dělat - z jednoho stavu může vést pro jeden symbol více než jeden přechod.
- přechodová funkce vrací množinu stavů a automat si nedeterministicky vybere
- neboli, akceptuje stavy, kde jeden vstup vede na dvě různá místa - můžu mít i nedefinované vztahy funkce - stav (daná funkce ve stavu neexistuje)
- Na obr. Příklad nedeterministického automatu

DFA - deterministický konečný automat

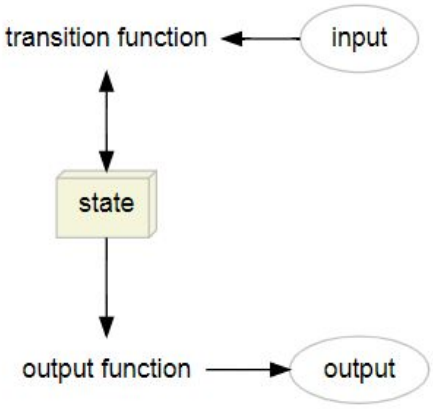
- V každé situaci je jasné, co bude automat dělat - pro každý stav a každý symbol pouze jeden přechod
- Přechodová funkce vrátí pouze jeden výsledek - daný vstup nás vždy pošle na stejné místo

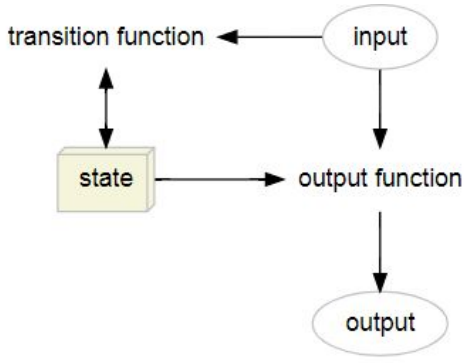
Převod NFA na DFA

- Napíšu si tabulku, kde bude první sloupeček reprezentovat stavy, první řádek přechodové funkce a vnitřek tabulky reprezentovat, do jakého stavu při jaké přechodové funkci půjdu. Jelikož některé přechodové funkce vedou do dvou stavů, musíme automat upravit na DFA - když funkce vede do dvou stavů, vytvoříme nový stav (např. stavy A, B ⇒ AB) - kam tento stav bude přecházet určíme sloučením stavů výchozích - pokud alespoň jeden původní stav byl koncový, i nový stav je koncový - toto opakujeme, dokud nebudeme mít popsány všechny stavy - pokud se při nějaké přechodové funkci z nějakého stavu nešlo, tak musím vytvořit stav nový, který bude jako černá díra - již se z něj nedostanu. (video)

Moorův automat

- Zařízení s konečným počtem vnitřních stavů, mezi kterými se přechází na základě vstupních symbolů. Každý vnitřní stav má definovaný právě jednu hodnotu na výstupu, musí mít dále definovaný výchozí vnitřní stav, ve kterém se nachází před zadáním prvního vstupního symbolu a pravidla pro přechody mezi jednotlivými stavy.
- > Pracuje pouze s pamětí, výstup vypadne až ve stavu - uzlu grafu





Mealyho automat

- Zařízení s konečným počtem vnitřních stavů
- Output je ovlivněn vnitřním stavem i vstupem - pracuje jak s pamětí, tak s kombinační logikou - závisí na přechodech - předchozí stav + aktuální vstup - rychlejší. Výstup vypadne už při přechodu (hrana grafu).

18 Kompilátor (Filip)

Kompilátor

= překladač

- Překládání znamená přepsat zdrojový kód napsaný v programovacím jazyku nejprve do assembly a potom do strojového kódu, kterému rozumí procesor
    - Potřebujeme různé kompilátory (třeba GCC je kolekce kompilátorů) na různé procesorové architektury a pro různé operační systémy
- Vedlejší (avšak žádoucí) vlastností kompilace je, že z výsledného binárního kódu nelze bez značného úsilí zpět získat zdrojový kód
- Kromě překládání mnohdy umí v programu vyhledat chyby
    - Lukáš se třeba neustále chlubí tím, že když se ti něco podaří bez varování zkompilovat v Rustu, tak to nejspíš bude fungovat i v runtimu (▀Ĺ_▀)

Interpret

- Interpret spouští kód řádek po řádku
- Každý prohlížeč má v sobě embedovaný interpret Javascriptu
- Některé interprety ti spustí i kód s chybou a chybu zjistí teprve, když se dostanou na dotyčný řádek
    - Proto jsou kompilované jazyky většinou spolehlivější
- Zdrojový kód je kompatibilní se všemi zařízeními, na kterých běží interpret

Java a C#

- Java/C# jsou hybridní
    + Scala, Kotlin, Jython - protože se kompilují do java bytekódu
- Rychlejší protože "kompilováno", nezávislé na platformě protože JVM

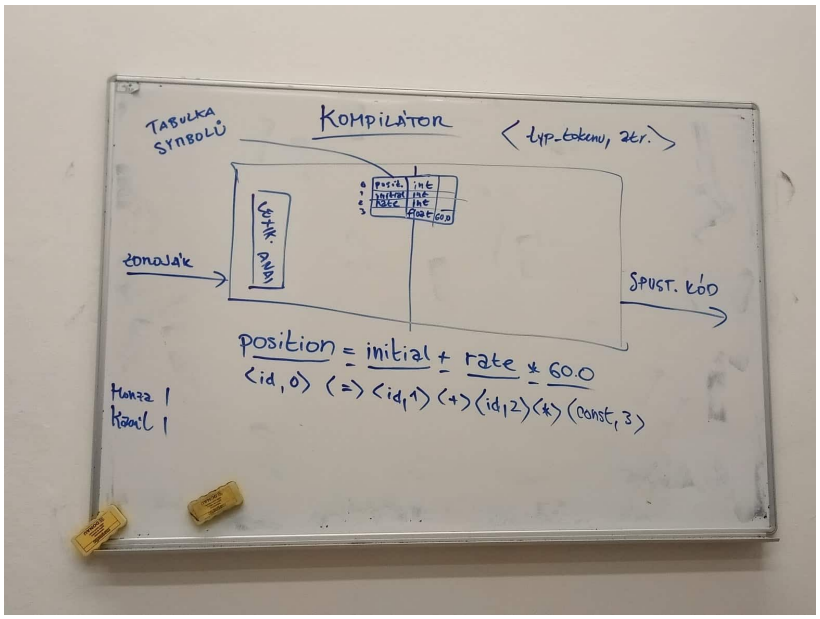
Z čeho se skládá kompilátor

- Frontend a backend
    - Frontend z jazyku udělá strukturu standardizovanou pro více možných jazyků
        - Neboli vygeneruje strom
    - Backend pak ze stromu vytvoří spustitelný bytekód
    - Podle programovacího jazyku fronted, podle cílové architektury backend

Postup kompilace

Lexikální analýza

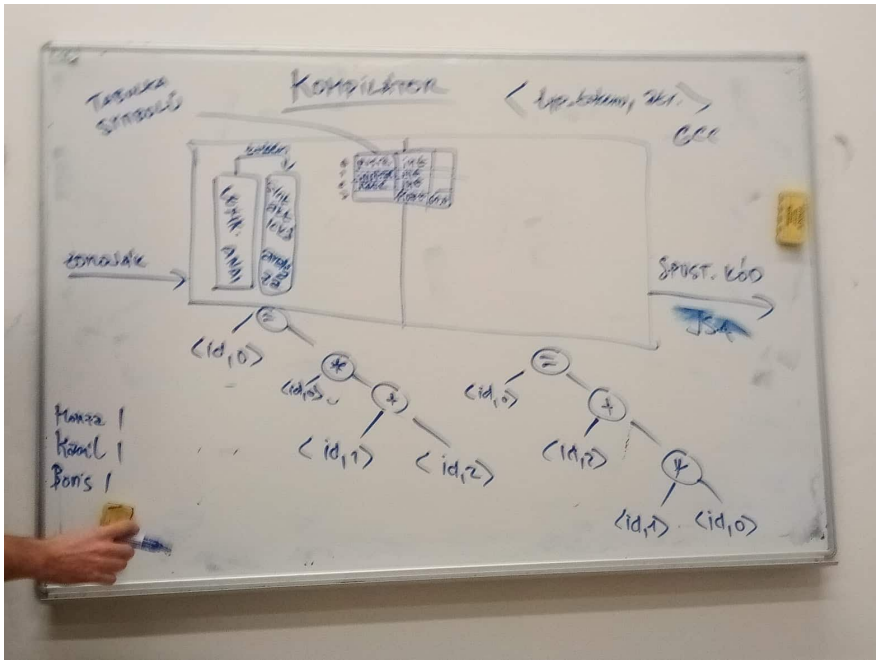
- Projde zdroják a z každého řetězce, u kterého to dává smysl, udělá token <typ tokenu, atribut>
- Error najde, když nějakému řetezci nelze přiřadit token



- Tabulka symbolů
    - Jméno proměnné (pokud je to proměnná), typ, hodnota (pokud má hodnotu)

Syntaktická analýza

- syntaktický analyzátor = parser
- Má k dispozici už vygenerovanou tabulku symbolů
- Vezme tokeny a staví z nich strom
- Vystavím ho tak, aby nejvíc ve spodu byly operace, které chci provést jako první

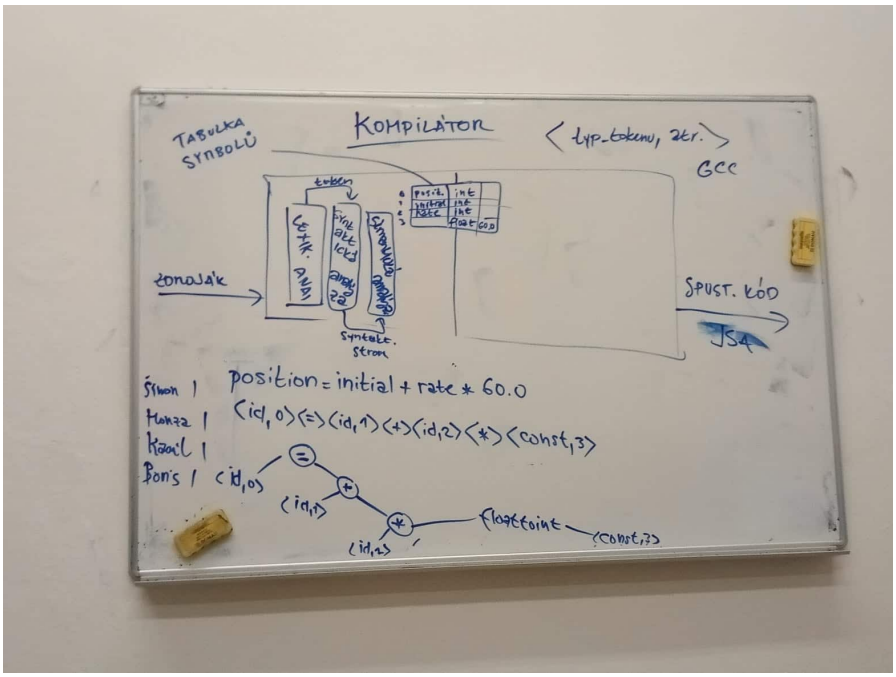


(správně je spíš ten druhý strom, ten první prohazuje prioritu sčítání a násobení podle matematiky)

- Na téhle úrovni se odchytí chyby typu "3 *+ 4"

Sémantická analýza

- Na téhle úrovni se odchytí chyby typu 'sčítám int se stringem'
  - Cčko třeba ale umí sčítat int s floatem
  - To se řeší tak, že během sémantické analýzy do stromu přidám na konec další operace, které zkonvertují jednotlivé hodnoty tak, aby to dávalo smysl
- Výstupem je tedy případně-upravený strom



Tříadresový kód (mezikód)

- Tuhle část procesu používá např. GCC
- Na každém řádku jenom jedna operace
- Používá něco jako registry, jenom jich má neomezené množství

```
t1 = float-to-int(60.0)
t2 = id2 * t1
t3 = id1 + t2
id1 = t3
```

Optimalizace mezikódu (tady už backend)

- Následně se mezikód optimalizuje

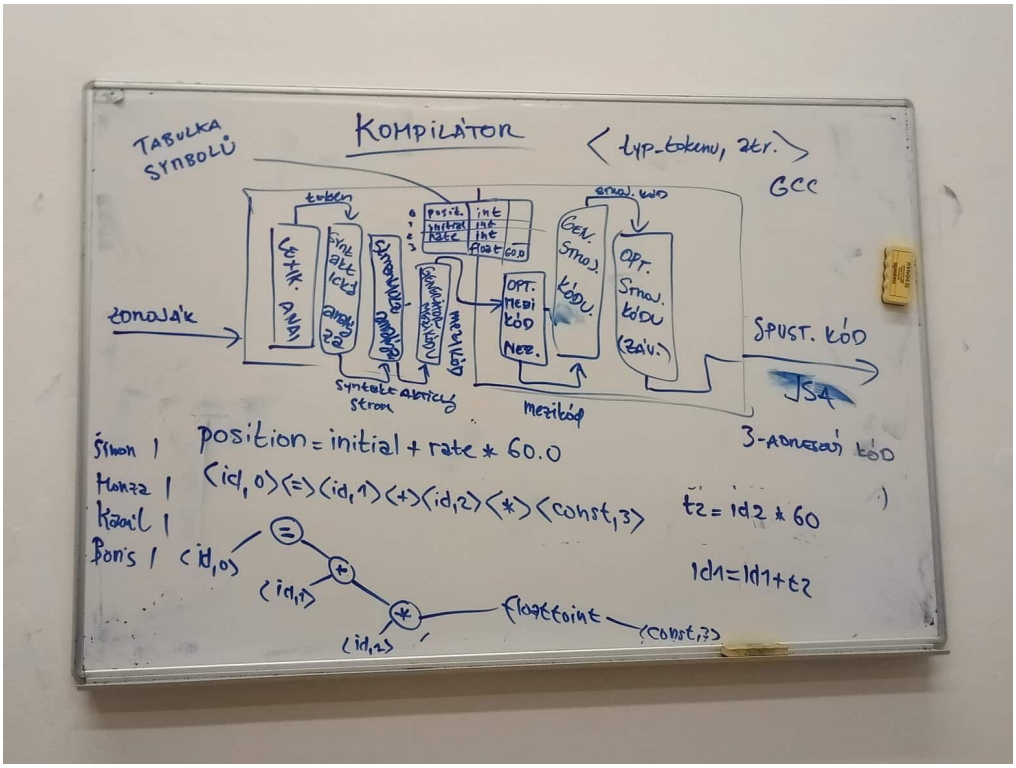
```
t2 = id2 * 60
```

Generátor strojového kódu

- Podle platformy z mezikódu udělá strojový kód

Optimalizace strojového kódu

- Podle platformy ještě dále optimalizuje strojový kód
- Samozřejmě ho nepotřebuji, ale hodí se



Pozn.

- JSA (Jazyk symbolických adres) pak zpracovává program, kterému se říká Assembler
- Pro různé jazyky mi stačí napsat různé frontendy (napíšu si lexikální analýzu, syntaktickou a sémantickou)
    - (F)LEX generuje lexikální analyzátory
    - YACC nebo BISON generují syntaktické a sémantické analyzátory
- Pro různé architektury mi stačí napsat akorát různé generátory strojového kódu (a případně optimalizátor)
- Kompilátor tedy musí udělat s programem mnoho úkonů → trvá to dlouho

Kompletní cesta programu

- Zdrojový kód
- Preprocesor
- Kompilátor
- Linker a loader
    - Zkombinovat zdrojové soubory a nalinkovat knihovny
    - Když se program spouští, zavést ho do paměti - říct mu, kde má svoje místo v paměti

19 Počítačové sítě - ROZDĚLANÉ (Bohouš)

popis jednotlivých vrstev TCP/IP modelu, MAC adresa, IP adresa, rozdíly mezi IPv4 a IPv6, rozdíly mezi TCP a UDP, příklady aplikačních protokolů, DHCP, DNS

MAC adresa

- unikátní identifikátor zařízení

IP adresa

- adresa zařízení v síti “IP adresa slouží k rozlišení síťových rozhraní připojených k počítačové síti.”

IPv4

- 32 bitů
- např. 192.168.1.1
- v dnešním počítačovém světě nedostatečné
- 4,29*10^9

IPv6

- 128 bitů
- 3.4*10^38
- **::** znamená, že do konce ip jsou 0

TCP/IP

- síťová architektura založená na standardu ISO/OSI
- sada protokolů pro komunikaci v počítačové síti
- **ARP** request - zjišťuje mac adresy zařízení podle IP
- 4 vrstvy síťového rozhraní

Aplikační vrstva

- obsahuje protokoly , které slouží k přenosu konkrétních dat. Např.: HTTP (komunikace s www servery), FTP (file transfer protocol), SMTP (simple mail transfer protocol) IMAP (internet message access protocol) a další.

Transportní vrstva

- vytváří spojení a směruje datový tok k příslušným aplikacím. Na výběr jsou protokoly TCP a UDP.

TCP (Transmission Control Protocol)

- zajišťuje spojově orientovanou spolehlivou službu, která je náročnější na režii. Naváže spojení mezi počítači teprve po ověření existence a dostupnosti všech prvků na přenosové cestě a druhá strana vysílá zpět potvrzení o správnosti přijatých dat.
- Spojení je definováno pomocí dvojice tzv. soketů, což je kombinace IP adresy a čísla portu. Navíc umožňuje řízení toku dat – například přijímací strana vyšle požadavek vysílací straně, že nestíhá a potřebuje přenos zpomalit.
- hodí se např k přenosu souborů, videa na YT...

UDP (User Datagram Protocol)

- začne vysílat data i bez ověření samotné existence cílového počítače a bez řízení toku dat. Nemá žádnou režii a hodí se třeba na živě streamované video, online rádia, online multiplayer hry...

Síťová vrstva

- je realizována protokolem IP, který poskytuje také nespolehlivou službu. Vrstva směruje pakety podle IP adres v hlavičce. Dnes používané IPv4 jsou 32bitové, může jich tedy být kolem 4 miliard, což je nedostačující. Řešením je IPv6 se 128bitovým adresováním, které navíc nabízí řízení kvality služby (QoS).

Vrstva síťového rozhraní

- závisí na typu sítě a je různá pro Ethernet, ATM, Token Ring nebo telefonní síť. V naprosté většině lokálních sítí je především díky své jednoduchosti používán Ethernet. Princip byl založen na sběrnicové technologii, kdy se na jeden přenosový kabel snaží dostat svá data všechny počítače, a to často současně – neví, že ve stejném okamžiku se chystá vysílat i někdo jiný.

========== markdown, předělám ↓↓↓

Sítě
![sitove_vrstvy.img](https://upload.wikimedia.org/wikipedia/commons/thumb/c/c0/Tcpip_zapouzdreni.svg/600px-Tcpip_zapouzdreni.svg.png "síťové vrstvy")

Vrstva síťového rozhraní
- MAC adresa v hlavičce
- CRC (cyclic redundancy check - podle hashe) v patičce
Ethernet
DSL
- modem (modulátor-demodulátor) - převádí digitální data na analogový zvuk a zpět
- data tečou po telefonní síti
- původně jen pro účely pevné linky
- pomalý přenos dat
- jen tam, kde není zavedený Ethernet
- ADSL - asymetrická rychlost přenosu dat (většinou rychlejší odesílání než příjem)
WiFi
- data lítaj vzduchem
PPP

Síťová vrstva
IP
- IPv4
    - došly adresy pro zařízení (4,7 miliard adres)
    - dříve se nedostatek adres řešil
- IPv6
ICMP
- PING
DHCP
- rozděluje IP adresy v síti pro jednotlivé MAC adresy
ARP (Adress Resolution Protocol)
- pomocí ARP requestu se zjišťuje MAC adresa z IP adresy
- (ptám se zařízení, která znám, jestli znají cílové zařízení)

Transportní vrstva
TCP
- všechna data přijdou (stahování souboru)
- příjemce musí potvrdit, že přijal správná data
UDP
- proud dat (živé vysílání, online hra...)

Aplikační vrstva
HTTPS
POP3
SSH
FTP
DNS (DNSSec)
- přiděluje doménové názvy k IP adresám
- kořenová organizace IANA poskytuje domény poskytovatelům (v česku cz.nic)
- DNS server vidí všechny stránky, které se snažím navštívit
- .cz .io .gov .org .co .info

========== markdown, předělám ↓↓↓

Topologie sítí

- dřív se používala kruhová topologie

- dnes už jen hvězdicová topologie

Hardware pro sítě

Bridge

- převodník

Modem (MODulátor-DEModulátor)

- převádí digitální data na analogový zvukový signál a zpět

Hub

- celý hub je kolizní doména, spojuje všechny porty

- když spolu komunikují dvě zařízení, nemůže komunikovat žádná jiná dvojice

- na všech portech je slyšet, co si ona dvojice říká

Switch

- jakmile zjistí MAC adresy, nechá spolu komunikovat jen relevantní dvojice

Router

- "chytřejší switch"

- umí pracovat s MAC i IP adresami zařízení

20 Databáze - návrh - ROZDĚLANÉ (pokusí se Gabr)

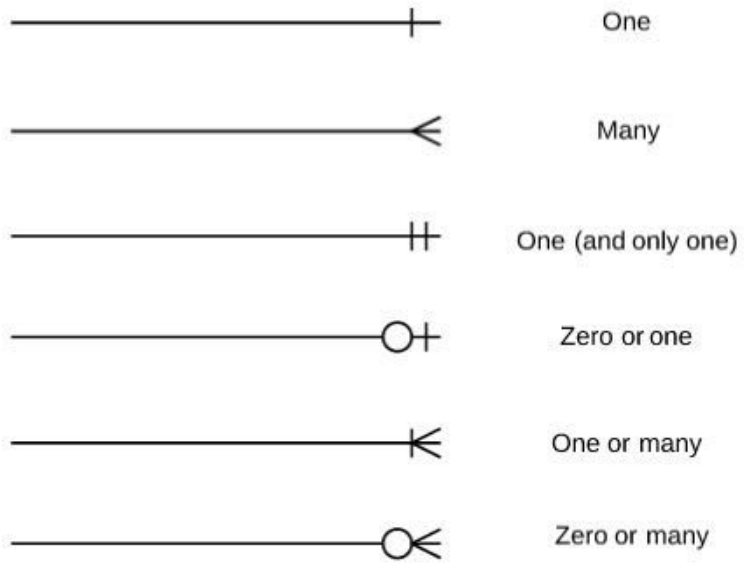
Prvními systémy byly navigační databáze, jako je hierarchická databáze (která se opírala o stromový model a umožňovala pouze vztahy typu one-to-many – „jeden k více“) a síťová databáze (pružnější model, který umožňoval různé vztahy). Tyto první systémy byly jednoduché, ale nepružné. V 80. letech získaly popularitu relační databáze a v 90. letech následovaly objektově orientované databáze. V poslední době vznikly databáze NoSQL jako reakce na růst internetu a potřebu vyšší rychlosti a zpracování nestrukturovaných dat.

Databáze - pojmy

- **Databáze** - je systém souborů s pevnou strukturou záznamů. Tyto soubory jsou mezi sebou navzájem propojeny pomocí klíčů.
- **Datové entity** - jsou objekty v databázi, např. tabulky, indexy, procesy
- Tabulky jsou rozděleny do **atributů** - sloupců - např. ID, adresa, jméno,..., a **záznamů** - to jsou jednotlivé řádky
- **Primární klíč** - je atribut, jenž je pro každý záznam unikátní - např. ID, rodné číslo. Nelze upravovat.
- **Cizí klíč** - slouží k vyjádření vztahů - relací mezi entitami. Např. identifikuje souvislost mezi jednotlivými tabulkami
- **Integrita databáze -** znamená, že data v databázi uložená jsou v osnově definovaných pravidel - např. musí splňovat datový typ definovaný pro danou oblast. K tomu slouží integritní omezení - správnost typu dat, zamezení poškození či smazání dat.
- **Super key** - je set jedné nebo více hodnot, které dokážou unikátně identifikovat záznam - řádek tabulky. může být složen z více sloupců
- **Candidate key** - je jako super key, ale žádná jeho součást nesmí být sama superkey. Ořezané o zbytečné míchání super keys, které by nám stačily samy o sobě
- **Systém řízení báze dat** - je software poskytující uživateli práci s databázemi - upravováním, vytvářením a udržováním je softwarové vybavení zajišťující práci s databází - tvoří rozhraní mezi aplikačními programy a uloženými daty. Např. Oracle, SQLite, MySQL

Relace entit

- Podle poměru záznamů v jedné a druhé databázové tabulce (1:1, 1:N, M:N - many to many)



- One and only - záznam “náleží” jen a pouze jednomu jinému záznamu

- **ERD** - Entity Relationship Diagram - slouží k vizuálnímu zobrazování databází - vztahů entit

Normalizace

Je proces optimalizace navržených struktur databázových tabulek. Cíl - minimální počet redundantních - přebytečných dat. Zjednodušenost databázových struktur lze ohodnotit tzv. normálními formami. Hodnocením normalizace jsou tzv. normální formy - 0NF - 5NF, 5NF je nejlepší možný stav normalizovanosti databáze.
Např.
- Více řádků (zápisů) v tabulce obsahuje stejné informace
  - např. u každého studenta mám napsané jméno a telefon třídního učitele
  - Pokud smažu studenty, nevím nic o třídím
  - Těžko se mi zjišťuje kteří studenti mají stejného učitele
  - Pokud se číslo učitele změní, musím updatovat zápisy všech studentů
  - Data redundancy - přebytečnost
- Řešení: Rozdělím tabulku na dvě
  - např. tabulka 'studenti' a tabulka 'třídní'
- Předpokladem každé normal form je předchozí normal form

NoSQL

- Bez daného schématu
- Možné mít identické atributy ve více entitách - duplikace - při změně atributu nutná změna ve všech entitách
    - Složitější upravování, hrozba nekonzistentních dat atd., ale výrazně rychlejší čtení - což je dnes čím dál tím častěji lepší (soc. sítě etc.)
- Scaling vertically vs. Scaling horizontally
    - Vertically - SQL - s rostoucím množstvím dat rostou požadavky po jednotlivých serverech - entity nejdou jednoduše rozdělit
    - Horizontally - NoSQL - s rostoucím množstvím dat se úložiště může rozrůstat počtem serverů, nikoliv jejich kapacitou - data lze jednoduše distribuovat - úprava počtu serverů je v dnešní době velmi jednoduchá
- Protože zápisy nemají přesné podmínky, je potřeba ověřovat u klienta, jestli jsou stahovány správná data
- Hodí se zmínit zde DRY vs WET solutions:
    - DRY - “don't repeat yourself” - všechna data musejí mít pouze jedno zastoupení v paměti, snaha o omezení opakování
    - WET - “write every time” - pravý opak DRY
        - V případě databází může být někdy DRY řešení, kvůli vysokému počtu logických spojení, horší, než WET

databáze - systém řízení báze dat

- Lineární
    - např. jpg, magnetofonová páska - Hierarchické - stromové - např. html - Síťové
- malá redundance - Např. (**W**orld**W**ide**W**eb) - Relační
    - možnost hledání/filtrování prvků podle atributů - možnost přidávat měnit prvky - Relační databáze má entity
        - entita = nějaký objekt, o kterém si chceme ukládat informace - entita má atributy - jednotlivé údaje
        - povinné atributy značíme * - nepovinné atributy značíme o - primární klíč / identifikátor = atribut / sada atributů - Značí se # - vztahy mezi entitama
            - např. - one to many 1:N
                - “slepičí pařát” - lékař má více pacientů
                - pokud nemusí, u pařátu (u pacienta)
        - značíme o - one to one 1:1
    - jedna věc má druhou - velmi těsný vztah - many to many M:N
        - “dvojitý slepičí pařát” - Více kin promítá více filmů

další dump nezpracovaných zápisků

Databáze

Co říct o databázích

- Primary key
  - Unique, uneditable, non-null
  - *Composite PK*
    - Pokud mám entitu, která je závislá na dvou (více) různých entitách.
    - Její PK tedy můžu vytvořit spojením dvou (více) jejích FK
    - Je to vlastně blbost. Stejně tak můžu téhle entitě dát vlastní PK.
- Foreign key
  - Víc na jednu entitu
- *Bridge table*
  - Občas se hodí, když mám many-to-many vztah
    - Často se stává, že samotná kombinace dvou položek má nějaké vlastnosti
    - Ty pak můžu reprezentovat tabulkou "mezi" těmito dvěmi tabulkami
- Int, varchar, **money**, ...

Normalizace databází

- Více řádků (zápisů) v tabulce obsahuje stejné informace
  - např. u každého studenta mám napsané jméno a telefon třídního učitele
  - Pokud smažu studenty, nevím nic o třídím
  - Těžko se mi zjišťuje kteří studenti mají stejného učitele
  - Pokud se číslo učitele změní, musím updatovat zápisy všech studentů
  - Data redundancy
- Řešení: Rozdělím tabulku na dvě
  - např. tabulka 'studenti' a tabulka 'třídní'
- Předpokladem každé normal form je předchozí normal form

SQL (jenom retrievovací příkazy)

- SELECT column1[, column2] FROM table [WHERE column3 = column4]
  - Kde = je libovolné porovnávací znaménko (<> je nerovnost)
- SELECT column1[, column2] FROM table1 JOIN table2 ON table1.column3 = table2.column4
- SELECT column1[, column2] FROM table1 UNION SELECT column2[, column3] FROM table2
  - Ty sloupce by měly mít stejný datový typ
  - Nedá sloupce 'vedle sebe', ale 'na sebe'
- SELECT column1, sum(column2) as soucet FROM table1 GROUP_BY column1
  - Hodnoty 'column2' sdruží podle toho, které mají stejné odpovídající hodnoty v 'column1'. Na každou skupinu uplatní funkci 'sum'. Výsledkem query bude tabulka 'column1' a 'soucet', kde v sloupci 'soucet' budou právě výsledky funkce 'sum'.
  - SQL má víc funkcí jako 'sum'

ACID transactions

- Transakce = soubor více queries, které chceme spustit najednou po sobě
- **Atomic** = All or nothing
  - Pokud jedno query selže, ostatní se nestanou
- **Consistent** = Nestane se žádná transakce, která by zanechala databázi v invalidním stavu
  - např. záporný zůstatek na účtu
- **Isolated** = Je garantováno, že query jsou vyplněné po sobě (nebo to tak alespoň z venčí vypadá)
- **Durable** = Jakmile proběhla transakce, změny jsou uloženy *napevno*
  - Ne že to je jenom v ramce a pak vypadne proud

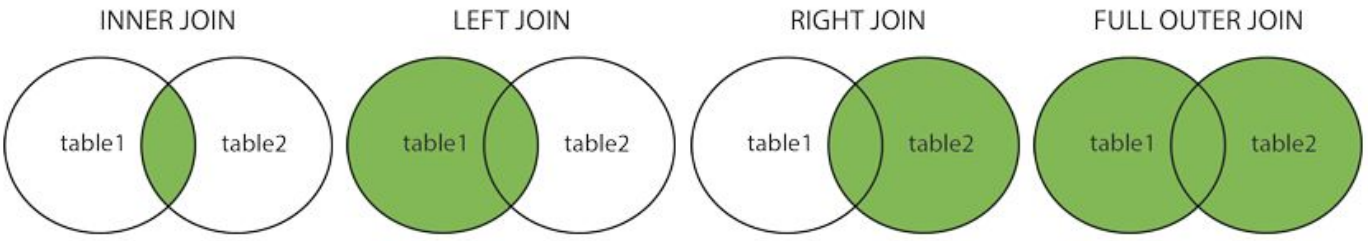
21 Databáze - deklarativní programování (Honza)

- Pro náročné tu je kompletní reference na w3schools - https://www.w3schools.com/sql/default.asp
    - Zároveň mají docela hezký simulátor ke všem příkazům (pokud si nejste jistí doporučuji vyzkoušet)

- Databázi posílám tzv. queries (dotazy) a dostávám od ní odpovědi, se kterými třeba dále pracuji
- Každý dotaz končím středníkem ;

SQL Příkazy

- SELECT sloupec1, sloupec2 FROM tabulka
    - Základní příkaz, který vypíše všechny hodnoty určitých sloupců v tabulce
    - Místo abych vyjmenovával všechny sloupce, můžu použít wildcard *
- sloupec1 AS jmeno, sloupec 2 AS prijmeni
    - Sloupce si mohu nechat vypsat pod jiným jménem
- WHERE sloupec1 <; >; = hodnota;
    - Mohu databázi udat podmínky (např. porovnávací operátory >, <, =)
    - Vypsány pak budou jen řádky, které podmínky splňují
- ORDER BY sloupec1
    - Mohu určit podle kterého sloupce budou řádky odpovědi seřazeny.
- JOIN tabulka2 ON tabulka2.sloupec3 = tabulka1.sloupec1
    - V odpovědi dostanu dvě tabulky spojené do jedné
    - Spojení proběhne pomocí specifikovaných sloupců
    - “tabulka1.sloupec1” by pravděpodobně obsahoval PK tabulky1 a “tabulka2.sloupec3” tyto PK jako FK tabulky2
    - Více typů (výchozí je INNER JOIN):



- LIMIT x
    - Vypíšu pouze prvních “x” sloupců
    - Používá se s ORDER BY
    - Pokud chci např. 4 - 8, napíšu LIMIT 5 OFFSET 3
- GROUP BY a SQL funkce
    - Viz příklady dotazů

Příklady dotazů

- SELECT politik.jmeno FROM politik JOIN strana ON politik.id_strana = strana.id WHERE strana.jméno = “KSČM”;
    - Vypíše politiky z KSČM
- SELECT jmeno, prijmeni FROM politik WHERE oblibenost > 0,05;
    - Vypíše jména a příjmení politiků, jejichž oblíbenost je větší než 5%
- SELECT **sum(volic)** as soucet_volicu FROM politik
    - Vypíšu součet voličů všech politiků
    - Jako výstup dostanu tabulku s jediným sluncem “soucet_volicu” a jediným řádkem - to bude ten součet
    - SQL má více funkcí jako je sum (count, avg, min, max, …)
- SELECT id_strana, sum(volic) as soucetvolicu FROM politik **GROUP BY** id_strana
    - Vypíše součty voličů pro každou stranu
    - Jako výstup dostanu tabulku se dvěma sloupci “id_strana” a “soucet_volicu”

22 Bezpečnost - (Tonda)

Veškeré prvky 23 Šifrování a hashování souvisí také s tímto tématem

Nejčastější příčinou selhání bezpečnosti je uživatelská chyba:

”

Greetings. The Master Control Program has chosen you to serve your system on the Game Grid. Those of you who continue to profess a belief in the Users will receive the standard substandard training, which will result in your eventual elimination. Those of you who renounce this superstitious and hysterical belief will be eligible to join the warrior elite of the MCP. You will each receive an identity disk. Everything you do or learn will be imprinted on this disk. If you lose your disk or fail to obey commands, you will be subject to immediate de-resolution. That will be all.

“ - Sark

Obor se zabývá úkoly

- Ochrana před neoprávněným manipulováním se zařízeními
- Ochrana před neoprávněnou manipulací s daty
- Ochrana informací před krádeží (nelegální tvorba kopií dat)
- Bezpečná komunikaci a přenos dat (kryptografie)
- Bezpečné uložení dat
- Celistvost a nepodvrhnutelnost dat

Kroky

1. Prevence – ochrana před hrozbami
2. Detekce – odhalení neoprávněných (skrytých, nezamýšlených) činností a slabých míst v systému
3. Náprava – odstranění slabého místa v systému

Hrozby

Backdoors
Část systému obcházející běžnou bezpečnostní kontrolu. Zpravidla implementována vývojáři pro debugging, nebo může být vytvořena útočníkem.

Denial of service
Computerphile
Útoky zajisťující nedostupnost síťového zařízení. Oběť je přehlcena requesty. DDoS útoky pochází z více různých IP adres, je tudíž mnohem těžší je blokovat. Existují sítě “zombie počítačů”, virem napadených zařízení, které mohou být v potřebné chvíli nasměrovány na cíl a bombardovat ho requesty.
Zesílené útoky jsou zprostředkovány nevinnými systémy, jimž byl poslán request s falešnou zpětnou adresou (adresou oběti) a oni na něj odpoví.

Útoky s přímým přístupem
Neoprávněný uživatel, který získá fyzický přístup k počítači. Těžko s tím něco udělat, může nabootovat z vlastního disku.

Odposlech
Většinou zprostředkován vládou vynucenými backdoory. Aktuálně je významný americký vládní program PRISM ve spolupráci s Microsoftem, Facebookem, Applem a Googlem.

Spoofing (padělání)
Postup, ve kterém je komunikace odeslána z neznámého zdroje, který se tváří jako zdroj známý přijímači.

Tampering (zasahování)
Škodlivá modifikace produktů.

Phishing
Snaha získat citlivé informace, jako jsou uživatelská jména, hesla či informace o kreditní kartě, přímo od uživatelů. Phishing se obvykle provádí pomocí falešných e-mailových zpráv nebo zneužitím instant messagingu. Často se snaží uživatele přesvdčit k zadání podrobností na falešných webových stránkách, které se zdají (téměř) totožné s legitimními stránkami. „Lovením" důvěřivých obětí lze phishing klasifikovat jako nelegální využití sociálního inženýrství.

Sociální inženýrství
Snaha přesvědčit uživatele, aby dobrovolně prozradil své citlivé údaje, například hesla pro přístup k soukromým službám (e-mail, sociální sítě, internetové bankovnictví, ...), čísla bankovních karet, PIN kód a podobně, například tím, že se vydává za existující instituci (banku, poskytovatele služeb, ...), ale i za kamaráda, příbuzného a podobně. Typické je naléhání na rychlé "řešení", aby oběť neměla čas zjistit si o útoku podrobnosti a tím ho odhalit.

Zabezpečení

Zabezpečení fyzického přístupu
Spočívá v zabránění přístupu nepovolaných osob k částem počítačového systému. Na toto zabezpečení se používají bezpečnostní prvky jako přidělení rozdílných práv zaměstnancům, elektronické zámky, poplašné zařazení, kamerové systémy, autorizační systémy chráněné hesly, čipovými kartami, autentizační systémy na snímání otisků prstů, dlaně, oční duhovky, rozpoznání hlasu, auditovací systémy na sledování a zaznamenávání určitých akcií zaměstnanců (vstup zaměstnanců do místnosti, přihlášení se do systému, kopírování údajů atd.).

Zabezpečení počítačového systému
Zabezpečení počítačového systému spočívá v zabezpečení systému před útokem crackerů, škodlivých programů (viry, červy, trojské koně, spyware, adware, ...). do této části patří i zaškolení zaměstnanců, aby se chovali v souladu s počítačovou bezpečností a dodržovali zásady bezpečného chování na síti.

Zabezpečení informací
Zabezpečení informací spočívá v bezpečném zálohování dat. Záloha dat by měla být vytvořena tak, aby ji neohrozil útočník ani přírodní živelní pohroma (požár, záplavy, pád letadla, ...). Zálohovaná data je také potřeba chránit proti neoprávněné manipulaci použitím vhodného šifrovacího systému. Záloha dat má být aktuální.
Desatero bezpečného internetu

Desatero bezpečného internetu

Nedávej nikomu adresu ani telefon. Nevíš, kdo se skrývá za monitorem na druhé straně.

Neposílej nikomu, koho neznáš, svou fotografii a už vůbec ne intimní. Svou intimní fotku neposílej ani kamarádovi nebo kamarádce - nikdy nevíš, co s ní může někdy udělat.

Udržuj hesla (k e-mailu i jiné) v tajnosti, nesděluj je ani blízkému kamarádovi.

Nikdy neodpovídej na neslušné, hrubé nebo vulgární maily a vzkazy. Ignoruj je.

Nedomlouvej si schůzku přes internet, aniž bys o tom řekl někomu jinému.

Pokud narazíš na obrázek, video nebo e-mail, který tě šokuje, opusť webovou stránku.

Svěř se dospělému, pokud tě stránky nebo něčí vzkazy uvedou do rozpaků, nebo tě dokonce vyděsí.

Nedej šanci virům. Neotevírej přílohu zprávy, která přišla z neznámé adresy.

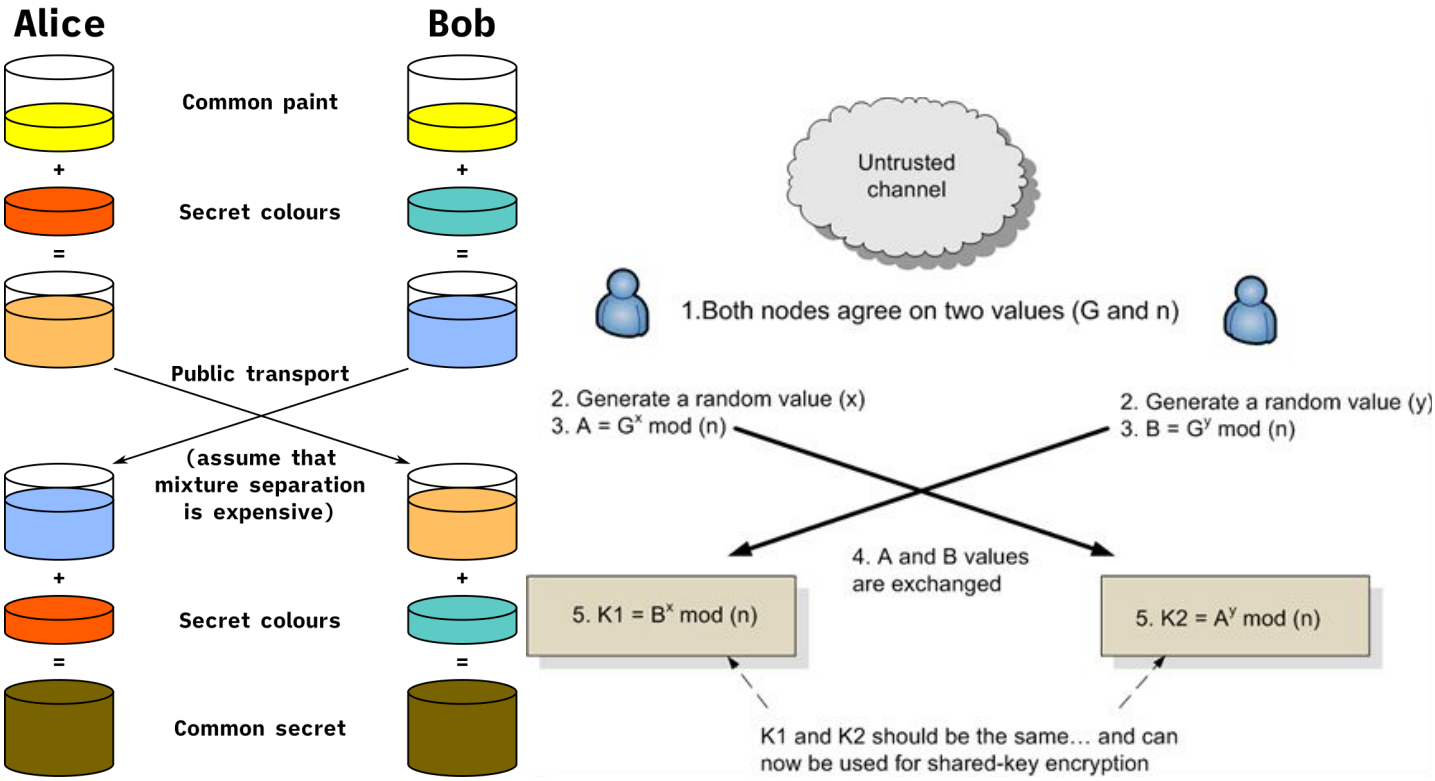
Nevěř každé informaci, kterou na internetu získáš.

Když se s někým nechceš bavit, nebav se.

23 Šifrování a hashování - (Tonda)

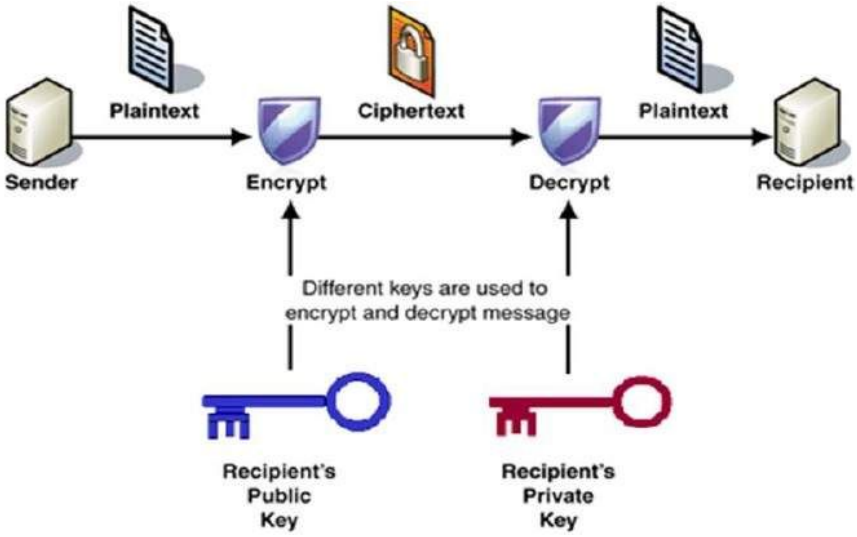
Kryptografie zajišťuje soukromou komunikaci za přítomnosti třetí osoby
Klíč je informace díky které můžeme data zašifrovat/odšifrovat

Symetrická šifra
vyžaduje aby obě strany měly stejný klíč.
K získání společného klíče slouží **Diffie-Hellman key exchange**.
Problém s Diffie-Hellman je, že neověřuje autenticitu komunikujících, tudíž nezabraňuje **Man-In-The-Middle útokům**, kdy se útočník vloží jako další článek mezi komunikujícími a s oběma stranami si vytvoří společný klíč.



YT analogie barev
YT matematické vysvětlení

Asymetrická šifra
využívá soukromého a veřejného klíče. Odesílatel zašifruje data pomocí veřejného klíče příjemce a příjemce data pomocí svého soukromého klíče odšifruje. Díky ní fungují i podpisy, jimiž se ověřuje autenticita odesílatele dat.
Veřejný klíč je produktem vynásobení dvou prvočísel. Asymetricé šifrování je založené na tom, že získání těchto dvou prvočísel z veřejného klíče (faktorizování) je příliš složité.
Nejrozšířenějším příkladem asymetrického šifrování je RSA wiki.



Digitální podpisy

Dovolují příjemci ověřit si autenticitu odesílatele. Odesílatel zašifruje data svým soukromým klíčem a příjemce si to ověří tak, že data odšifruje veřejným klíčem odesílatele.

Hashování

- Hash má danou velikost.
- Z hashe nesmí jít získat vstup.
- Při malé změně vstupu musí dojít k velké změně hashe.
- Unikatní vstup by měl tvořit unikátní hash (za předpokladu, že hash je větší nebo roven vstupu).

Hash se využívá k ověření, zda-li došlo k úspěšnému přenosu souboru. Odesílatel vytvoří hash souboru, který pošle spolu se souborem. Příjemce si vytvoří vlastní hash na základě přijatého souboru a pokud není totožný s hashem, který přišel se souborem, je jasné, že je něco s příchozím souborem špatně.

Hashování se také používá k ukládání hesel, aby nebyly čitelný, když dojde k breachu databáze (riziko je v tom, že lidi používají totožné heslo pro více aplikací). Hash vytvořený na základě zadaného heslo během přihlašování se pak porovnává s hashem v databázi. Problém s tímhle je, že pro ty nejpoužívanější hesla jsou hashe spočítaný v tzv. **Rainbow tables**, tudíž útočník je může snadno dohledat.

Vyřešeno je to pomocí **salt**, pro každého uživatele unikátní a veřejné fráze, která je k heslu před zahashováním přimíchána a vytvoří tak pro totožná hesla různých uživatelů unikátní hash.

24 Testování a dokumentace - ROZDĚLANÉ (Gabr + Honza)

Testování

- Testování kódu je část kódu, která za nás aplikaci vyzkouší (interaguje s ní a kontroluje jestli se chová správně)
- Test Driven Development
- Znamená to, že nejdřív napíšu test a pak až implementuji tu funkci, tak aby splnila test
    - Vyvíjím ve dvou krocích
        1. Napíšu test na novou funktionalitu
        2. Upravím kód, tak aby prošel testem

Dokumentace

- Základním cílem dokumentace je popis funkcionality kódu (stejná funkce jako komentáře)
    - Často se automaticky generuje z komentářů v kódu
- Zjednodušuje programátorovi práci s danou částí kódu (jednodušší pochopení a použití kódu)

25 Právo - ROZDĚLANÉ (Bohouš)

Proprietární software

- nepřístupný zdrojový kód
- Windows, MAC OS, WinRAR

Svobodný software

- můžu používat za jakýmkoliv účelem
- možnost studovat a měnit zdrojový kód
- mohu distribuovat program (i za peníze, i jeho modifikace)

Licence

GPL - General Public License

- mohu volně používat, upravovat
- při další distribuci nesmím změnit licenci
- příklad copyleftové licence

MIT, BSD

- permisivní
- “Dílo je k dispozici tak jak je, udělal jej XY a zříká se jakékoli přímé nebo nepřímé odpovědnosti za cokoli, co se stane. S dílem můžete dělat cokoli, pokud zachováte informaci o autorovi a informaci o zřeknutí se odpovědnosti.”
- v určitých případech dovoluje program šířit pouze v binární formě
- licence musí být zachována (ale autoři se mohou řetězit)

Creative commons (CC)

- soubor licencí
- vyberu si, co má licence obsahovat → vytvoří se nějaká licence

Open Source

- volně přístupný zdrojový kód
- neříká, jaký mám typ licence

NDA

- dohoda o utajení (např. Součást pracovní dohody)
- smlouva mezi alespoň dvěma stranami o sdílení důvěrných materiálů, znalostí nebo informací, které však nebudou sdíleny s jakoukoliv třetí stranou

- využívá se např. pro zachování know-how, obchodního tajemství, nevynášení informací o klientech atd.

SLA

- dohoda o úrovni poskytovaných služeb
- dohoda mezi uživatelem a poskytovatelem služby
    - např. připojení k internetu, distributor software atd.
- představuje formalizovaný popis služby, kterou poskytuje dodavatel zákazníkovi
- definuje:
    - rozsah, úroveň a kvalitu služby
    - kompenzace v případě nedodržení některého z umluvených bodů

========== markdown, předělám ↓↓↓