**Informatika a programování**

*1. Programování – neimperativní, imperativní, praktické programování.*

*2. Algoritmy a datové struktury (reprezentace objektů, serializace, techniky algoritmizace) – pole, seznamy, objekty, algoritmy typu rozděl a panuj, dynamické programování…*

*3. Grafy a stromy, grafové a stromové algoritmy.*

*4. Reprezentace čísel a znaků v počítači.*

*5. Databáze, jazyk SQL.*

*6. Značkovací jazyky – (X)HTML, XML, DTD, …*

*7. Algoritmická složitost – lineární, polynomiální a exponenciální algoritmy, složitost algoritmu a složitost problému, třídy složitosti.*

*8. Formální jazyky a automaty – regulární jazyk, bezkontextový jazyk, konečný automat, formální gramatika, typy gramatik podle Chomského.*

*9. Regulární výrazy, CQL.*

*10. Umělá inteligence (jaké problémy řeší a jaké algoritmy používá).*

# 1. Programování – neimperativní, imperativní, praktické programování

* ***programování*** – proces algoritmizace dané úlohy, vytváření postupu, jenž vede k řešení dané úlohy
	+ proces zahrnující činnosti od návrhu [algoritmu](https://cs.wikipedia.org/wiki/Algoritmus), psaní, [testování](https://cs.wikipedia.org/wiki/Testov%C3%A1n%C3%AD_softwaru) a [ladění](https://cs.wikipedia.org/wiki/Lad%C4%9Bn%C3%AD_%28programov%C3%A1n%C3%AD%29) [zdrojového kódu](https://cs.wikipedia.org/wiki/Zdrojov%C3%BD_k%C3%B3d) [počítačového programu](https://cs.wikipedia.org/wiki/Po%C4%8D%C3%ADta%C4%8Dov%C3%BD_program) ([software](https://cs.wikipedia.org/wiki/Software)), včetně následné údržby
* ***program*** – zápis algoritmu v programovacím jazyce
* ***programovací jazyky*** – jazyka sloužící k tvorbě počítačových programů (programování); prostředek pro zápis algoritmů, jež mohou být převedeny na počítači
* XML, HTML – jsou to značkovací jazyky, nikoliv programovací
* ***dělení programovacích jazyků***
	+ **podle míry abstrakce:** *vyšší a nižší*
		- **nižší** – primitivní
			* instrukce víceméně přesně odpovídají příkazům procesoru
			* jsou závislé na svém procesoru a nepřenositelné na jiný procesor, např. jazyk symbolických adres (assembler) nebo strojový kód (to, co uvidíme, když se nám podaří natáhnout exe soubor do textového editoru)
			* strojově orientované
			* rychlé, ale náročné a složité na programování, ne příliš srozumitelný zápis
			* používají se dnes spíše výjimečně, např. při psaní rychlých programů pro řídicí jednočipové mikropočítače
			* psaní jader operačních systémů
		- **vyšší** – srozumitelnější – algoritmus se zapisuje strukturovaně
			* nejsou závislé na strojových principech počítače
			* do strojového kódu se převádějí kompilátorem
			* formální jazyky (definované formálně, matematicky)
			* všechny ostatní jazyky (Python, Pascal, C, Prolog, …)
			* dělí se na imperativní (procedurální) a neimperativní (neprocedurální)
	+ **vyšší programovací jazyky podle způsobu programování:** *imperativní, logické a funkcionální*
		- **imperativní – procedurální**
			* k řešení úlohy se používá algoritmu
			* algoritmus se zapisuje posloupností příkazů, cyklů a větvení (podmínek)
			* využívají se proměnné
			* pomocí algoritmu počítači říkáme, co přesně má program dělat
			* např. Pascal, C/C++, Basic, Perl, Java, Python, php
		- **neimperativní – deklarativní**, neprocedurální
			* zadáme vstupní údaje a je nám jedno, jakým způsobem si to program udělá
			* **logické** – programátor popíše problém pomocí logických výroků, program z nich vyvozuje požadované informace, program je tedy zpravidla množina odvozovacích pravidel, používají se zejména v umělé inteligenci a expertních systémech, k praktickému programování nejsou využitelné, např. *Prolog*
			* **funkcionální** – vše se popisuje pomocí funkcí, program je množina (ne posloupnost) funkcí, práce se seznamy dat, většinou neexistují proměnné je obecně nepřehledný, blízký klasické matematice, ale pro některé úlohy může být vhodnější než imperativní, např. *Lisp, Haskell*
	+ **podle způsobu překladu:** *interpretované a kompilované*
		- **interpretované**
			* překládány až za běhu programu, nedochází k samostatnému překladu, nevzniká samostatně spustitelný programem
			* spouští se přes interpret – program, který interpretuje (provádí) zdrojový kód řádek po řádku
				+ přečte jeden řádek kódu, provede syntaktickou kontrolu a ihned zajistí vykonání kódu
				+ na chyby v zápise se tak přijde až za běhu programu (v takovém případě se běh programu přeruší)
			* výhody
				+ při změně zdrojového kódu není potřeba překladatele
				+ pokud existují interprety pro různé operační systémy, pak je snadná distribuce nových verzí programů uživatelům
			* nevýhody
				+ pomalý běh programů
				+ zjištění syntaktických chyb až při běhu
			* např. Basic, Python, JavaScript
		- **kompilované/kompilační**
			* text programu ve vyšším programovacím jazyce = zdrojový kód
			* zdrojový kód je vstupem do překladače
			* překladač (kompliátor) = program, který převádí zdrojový kód do strojového kódu, tj. do formy samostatně spustitelného souboru (exe/com v operačních systémech Microsoft, ELF formát v Linuxu)
			* překladač provádí syntaktickou kontrolu celého zdrojového kód
				+ zda je program zapsán podle pravidel jazyka
				+ pokud se vyskytne chyba v zápisu programu, překlad je přerušen a seznam chyb zobrazen uživateli
			* výhody
				+ syntaktická kontrola celého textu najednou, výsledný program je možné spustit samostatně bez speciálního programového prostředí
				+ běžící samostatný program je rychlý (rychlejší než u interpretovaného jazyka)
			* nevýhody
				+ při jakékoliv změně programu (zdrojového kódu) je nutné provést překlad znovu
			* např. Pascal, C/C++
		- moderní jazyky kombinují oba přístupy

# 2. Algoritmy a datové struktury (reprezentace objektů, serializace, techniky algoritmizace) – pole, seznamy, objekty, algoritmy typu rozděl a panuj, dynamické programování, …

* ***algoritmus*** – schematický postup řešení určitého problému prováděný pomocí konečného množství přesně definovaných kroků
* ***způsob zápisu algoritmů***
	+ *lineárním vyjádřením*
		- pomocí přirozeného jazyka doplněného klíčovými slovy (metajazyk, pseudokód)
		- zápisem v některém z programovacích jazyků
	+ *grafickým zápisem*
		- v podobě vývojového diagramu
		- v podobě strukturogramu
* ***vlastnosti algoritmů***
	+ *konečnost (finitnost)* – algoritmus má konečné množství kroků
	+ *určitost* *(determinovanost*)
		- každý krok algoritmu musí být jednoznačně a přesně definován
		- v každé situaci musí být naprosto zřejmé, co a jak se má provést
		- pro stejné vstupy dostaneme pokaždé stejné výsledky
	+ *korektnost* – algoritmus skončí pro libovolná (korektní) data správným výsledkem v konečném množství kroků
	+ *obecnost (univerzálnost)* – řeší všechny úlohy daného typu
	+ *výstup (resultativnost)* – má alespoň jeden výstup a vede od zpracování hodnot k výstupu
	+ *elementárnost* – skládá se z konečného počtu jednoduchých (elementárních) kroků
* ***dělení algoritmů***
	+ *rekurzivní a iterativní*
		- rekurzivní – opakuje kód prostřednictvím volání sebe sama, lze převést do iterativní podoby, snadno čitelný a kompaktní zápis, ale je třeba dodatečných systémových prostředků pro udržení jednotlivých rekurzivních volání (např. merge sort, quicksort)
		- iterativní – spočívá v opakování určité své části (bloku); (např. bubble sort, insertion sort)
	+ *deterministické a nedeterministické*
		- deterministické – v každém svém korku má právě jednu možnost, jak pokračovat
		- nedeterministické – má více možností
	+ *sériové, paralelní a distribuované*
		- sériový – vykonává všechny kroky jeden po druhém v sérii
		- paralelní – vykonává kroky zároveň ve více vláknech
		- distribuovaný – vykonává kroky zároveň na více strojích
* ***etapy řešení problému*** – potřebujeme vědět, s jakými údaji budeme pracovat (vstupní data) a podmínky, za jakých lze docílit správného výsledku, řešení problému probíhá v několika etapách:
	+ *specifikace (definice) problému* – znalost vstupních dat a požadavky na výstup
	+ *analýza problému* – volba vhodné metody řešení
	+ *sestavení algoritmu* – posloupnost na sebe navazujících kroků
	+ *kódování* – zápis algoritmu v programovacím jazyce
	+ *testování* – ověření správnosti algoritmu
* ***datové typy*** – určují, jakých hodnot může nabývat objekt daného datového typu a množinu přípustných operací nad daným typem
	+ *konstanta* – veličina, která během řešení problému nemění hodnotu
	+ *proměnná –* veličina, která během řešení problému může měnit hodnotu
	+ *výraz –* je tvořen operátory, operandy a speciálními znaky
	+ *funkce*
* ***řídicí struktury***
	+ *jednoduché příkazy*
	+ *strukturované příkazy*
		- sekvence – posloupnost jednoho nebo více příkazů prováděných v pevně daném pořadí
		- selekce – provedení dalšího příkazu závisí na splnění zadané podmínky (if/else)
		- cyklus – část algoritmu, která je opakovaně prováděna za splnění řídící podmínky (for, while)
* ***serializace*** – převedení datové struktury nebo instance objektu uložené ve vnitřní paměti počítače na posloupnost bitů, kterou lze uložit na nějaké úložiště neb přenést po síti

Techniky algoritmizace

* ***metody návrhu algoritmu***
	+ *shora dolů* – postup řešení rozkládáme na jednodušší operace, až dospějeme k elementárním krokům
	+ *zdola nahoru* – z elementárních kroků vytváříme prostředky, které nakonec umožní zvládnout požadovaný problém
	+ *kombinace obou* – obvyklý postup shora dolů doplníme částečným krokem zdola nahoru tím, že se např. použijí knihovny funkcí, vyšší programovací jazyk nebo systém pro vytváření programů
	+ nejrozšířenější metody návrhů algoritmů jsou typ **rozděl a panuj, hladový algoritmus, dynamické programování a backtracking**
* ***algoritmus typu rozděl a panuj***
	+ případ aplikace postupu shora dolů
	+ problém dělí na menší podproblémy, na něž se rekurzivně aplikují až po triviální podproblémy, které lze vyřešit přímo
	+ dílčí řešení se sloučí vhodným způsobem
	+ postup *rozděl* (rozčlenění úlohy na několik poúlloh stejného typu ale menší velikosti) → *vyřeš* (řešení podúloh) → *sjednoť* (sloučení řešení do řešení původní úlohy)
	+ binární vyhledávání, quicksort
* ***hladový algoritmus***
	+ přímočarý přístup k řešení určité třídy optimalizačních úloh
	+ hledáme podmnožinu množiny údajů, která vyhovuje určitým podmínkám a přitom optimalizuje předepsanou funkci, vyhovující množina se nazývá *přípustné řešení*
	+ *optimální řešení* – nejlepší z přípustných řešení
	+ algoritmus prochází jednotlivé prvky množiny údajů a v každém kroku se rozhoduje, zda se daný prvek hodí do optimálního řešení, používá výběrovou proceduru založenou na nějaké optimalizační funkci
	+ např. hledání nejkratší cesty v grafu
* ***dynamické programování***
	+ podobné hladovému algoritmu, ale zkoumá všechny posloupnosti, které by mohly být optimální a vylučuje ty, které optimální nebudou
	+ používá dílčí výsledky získané během výpočtu
	+ překrývající se podproblémy jsou na sobě závislé
	+ nejjednodušší je *metoda hrubé síly,* kdy se vygenerují všechny posloupnosti a z nich se vybere ta nejlepší; dynamické p. však automaticky negeneruje možnosti, které určitě nejsou optimální
	+ např. grafové úlohy a algoritmy
* ***hledání s návratem (backtracking)***
	+ založeno na prohledávání stavového prostoru problému
	+ metoda pokusů a oprav, zpětného sledování, prohledávání do hloubky
	+ hledáme n-tici, která optimalizuje nějakou účelovou funkci
	+ vytváříme jednu n-tici za druhou a pro každou novou složku se kontroluje, zda by mohla být optimální a splňovat zadané podmínky
	+ pokud jsou hodnoty nějaké složky vyčerpány, vrací se algoritmus o krok zpět a zkouší další možnou hodnotu
	+ problém osmi dam, chůze koně celou šachovnicí

Datové struktury

Datová struktura je konkrétní způsob uložení dat v paměti počítače. Kvůli různým technickým omezením není možné vytvořit jednu univerzální a za všech okolností efektivní datovou strukturu. Proto bylo navrženo mnoho rozličných datových struktur, které jsou používány tam, kde se zdají být nejvhodnější.

Požadavky na datové struktury jsou často protichůdné. Někdy je prioritou rychlost přístupu k datům, jindy je přednější maximální úspora paměti. Volba té či oné datové struktury by měla být vždy učiněna s přihlédnutím k účelu, k němuž má být struktura nejčastěji použita.

Operace na datových strukturách – přístup k *n-*tému prvku, vložení prvku, odstranění *n*-tého prvku.

**Indexování**

Index je databázová konstrukce, sloužící ke zrychlení vyhledávacích a dotazovacích procesů v databázi, definování unikátní hodnoty sloupce tabulky nebo optimalizaci fulltextového vyhledávání.

Když např. na Googlu zadám nějaký dotaz, identifikuje mi na 200 000 000 výsledků a zobrazí prvních deset do čtvrt sekundy. Na Googlu však nepostupují tak, že by, když něco hledám, načítali všechny stránky výsledků a procházeli jejich obsahem. Oni už mají stránky předem zanalyzované a termín, který hledám, porovnávají oproti nějakému indexu, který se pouze odkazuje na původní obsah.

* klíč – jednoznačný identifikátor záznamu
* pro rychlejší vyhledávání nad klíčem vybudovat index (index = seřazená posloupnost hodnot pro jeden klíč, s ukazateli na záznam)
* vybudování indexu ⇒ rychlé vyhledávání
* příkl.: korpus

***Datové struktury*** – programové konstrukce sloužící k efektivní správě dat v počítačovém softwaru, abstraktní datové typy

*Pole (array)*

* patří k **nejjednodušším datovým strukturám**
* **proměnná stejného typu**
* uložení hodnot v souvislé paměťové oblasti; **souvislý způsob uložení do paměti.** Paměť je jako kdyby velká tabulka, má spoustu políček, do kterých si můžeme něco dát.
* přístup k prvkům pole je určen udáním hodnoty **indexu** a není závislý na přístupu k jinému prvku → pole je strukturou s přímým nebo náhodným přístupem
* nesetříděné pole – všechny operace využívají sekvenčního vyhledávání s lineární složitostí i v nejhorším případě
* setříděné pole – vyhledávání pomocí binárního vyhl. se složitostí O(log n)
* rychlejší přístup, pomalejší modifikace
* výhody: **indexace**
* nevýhody: když chci vložit prvek třeba na druhé místo, musím nejdříve vše přesunout, teprve potom můžu vložit. Musím v paměti najít volné místo.



*Seznam (list)*

* uložení hodnot v nesouvislé paměťové oblasti lineárním způsobem
* pořadí prvku je určeno ukazateli mezi prvky seznamu (na následníka)
* jednoduchá, pružná reprezentace všech typických operací s dynamickými množinami
* **rychlé přidávání prvků na počátek/konec seznamu** (*O(N)*)
* vhodný pro **procházení prvků popořadě**
* **uspořádaný**
* **vyšší paměťové nároky** než pole
* pomalejší přístup, rychlejší modifikace



Pole vs. seznam:

* pole: rychlejší přístup, pomalejší modifikace
* seznam: pomalejší přístup, rychlejší modifikace
* vhodnost použití závisí na datech

*Objekt*

* v objektově orientovaném programování objekty představují komponenty nehmotného charakteru. Jsou abstrakcí a zjednodušením svých hmotných protějšků.
* vykazují určité vlastnosti a chování.
* tvořen:
	+ **datovou strukturou:** představována proměnnými různých datových typů. Ovlivňuje vlastnosti objektů. Tato část objektu bývá označována jako soukromá, z vnějšku je před jinými objekty ukryta (princip zapouzdření).
	+ **metodami:** definují operace, které je možno s daty provádět. Určují chování objektů. Tato část objektu bývá označována jako veřejná



*Zásobník*

* přesně určený způsob vkládání a mazání prvků
* princip **LIFO** (last in, first out) – prvek, který byl poslední vložen, je jako první ze zásobníku vyzvednut
* pracovat lze pouze s prvkem, který je na vrcholu zásobníku (jako batoh)
* implementace pomocí pole nebo spojového seznamu
* operace push (vložení) a pop (vyjmutí)
* typická časová složitost operací O(1)
* hodí se pro implementaci procházení do hloubky
* používá se především pro dočasné ukládání dat v průběhu výpočtu, při procházení grafů, vyhodnocování výrazů, rekurzi

**

**

*Fronta*

* mechanismus přístupu **FIFO** (first in, first out) – jako první je z fronty odebrán prvek, který byl do ní první vložen
* pracovat lze pouze s prvkem, který je na čele fronty
* operace put (vložení) a get (odebrání)
* pro implementace fronty potřebujeme ukazatele na hlavu (začátek) a ocas (konec) fronty
* implementace pomocí pole nebo spojového seznamu
* typická časová složitost operací O(1)
* hodí se pro implementaci procházení do šířky
* použití: zpracování příchozího požadavku serverem

**

**

*Hashovací tabulky*

* slouží k ukládání dvojic klíč-hodnota
* kombinuje vyhledávání pomocí indexu a procházení seznamu
* používá hashovací funkci
	+ konzistentně vrací pro stejné objekty stejné adresy (deterministická)
	+ je rychlá
	+ uniformní – výstupní hodnoty mají přibližně stejnou pravděpodobnost, využívá celého prostoru adres se stejnou pravděpodobností
	+ nezaručuje, že pro dva různé objekty vrátí různou adresu
	+ minimální změna vstupu vyvolá velkou změnu výstupu
	+ je obtížně reverzibilní

# 3. Grafy a stromy, grafové a stromové algoritmy

* teorie grafů zkoumá vlastnosti struktur, které se nazývají grafy
* graf je uspořádaná dvojice množiny vrcholů V a množiny hran H

V = {1, 2, 3, 4, 5, 6}

E = {{1,2},{1,5},{2,3},{2,5},{3,4},{4,5},{4,6}}

* každá hrana je dvouprvkovou podmnožinou množiny vrcholů

Další pojmy

* *sousední vrcholy* – vede mezi nimi hrana
* *stupeň vrcholu* – počet hran, které z daného vrcholu vychází
* *úplný graf* – mezi každými dvěma vrcholy vede hrana



* *podgraf* – tvoří jej vybrané vrcholy a hrany původních grafu



* *izomorfismus* – grafy jsou shodné, až na pojmenování vrcholů, jde o bijekci, vrcholy dokážeme přejmenovat tak, že z jednoho grafu dostaneme druhý



* *silné souvislé komponenty* – maximální podgraf orientovaného grafu, mezi každými dvěma vrcholy existuje cesta tam i zpět



***Typy grafů***

* *kružnice (cyklus)* – má stejný počet vrcholů a hran, všechny vrcholy jsou stupně 2 a nákres grafu tvoří kružnici
* *cesta* – kružnice s chybějící hranou
* *cyklus v grafu* – podgraf, který je kružnicí
* *cesta v grafu* – podgraf, který je cestou
* *klika v grafu* – podgraf, který je úplným grafem
* *souvislý graf* – existuje cesta mezi každými dvěma vrcholy
* *acyklický graf* – neobsahuje cyklus (také les, jeho souvislé části jsou stromy)
* *strom* – acyklický souvislý graf

***Rozšíření pojmu graf***

* *orientovaný graf* – hrany jsou orientování, rozeznáváme zdrojový a cílový vrchol, množina hran je množinou uspořádaných dvojic, kde je jednoznačně dán první a druhý vrchol (zdrojový a cílový)
	+ na graf můžeme nahlížet jako na binární relaci na množině vrcholů a reprezentovat jej jako tabulku relace, tzv. matici sousednosti
* *ohodnocený graf* – přiřazuje každé hraně reálné číslo (ohodnocení), využití pro modelování vzdáleností, kapacit síťových kanálů apod.
* *multigraf* – povoluje více hran mezi stejnou dvojicí vrcholů, včetně smyček (hrany začínající a končící ve stejném vrcholu)

***Další pojmy***

* *souvislé komponenty v grafu* – největší souvislé podgrafy, vzájemně nejsou spojeny žádnou hranou, mezi každými vrcholy takových podgrafů vede cesta
	+ v případě orientovaných grafů mluvíme o silně souvislých komponentách a cesta vede mezi dvěma vrcholy tam i zpět
* *délka cesty* – počet hran v této cestě nebo součet ohodnocení hran cesty
* *vzdálenost mezi vrcholy* – délka nejkratší cesty mezi dvěma vrcholy
* *kořen* – jeden význačný vrchol, v němž graf "začíná", volba je dána konvencemi
* *listy stromu* – všechny vrcholy stupně 1, které nejsou kořenem
* *kostra grafu* – podgraf, který obsahuje všechny vrcholy původního grafu a zároveň je stromem, vzniká po odstranění všech cyklů v grafu
* *minimální/maximální kostra* – má minimální/maximální možný součet ohodnocení hran

**Grafové algoritmy**

* nejjednodušší jsou algoritmy procházení (prohledávání) grafu
* ***procházení do hloubky (depth-first search)***
	+ označíme určený vrchol, přesuneme se na libovolný sousední neoznačený vrchol a provedeme znovu celý algoritmus s novým počátečním vrcholem
	+ pokud neexistuje sousední neoznačený vrchol, vracíme se postupně na vrcholy, které jsme již navštívili v opačném pořadí
* ***procházení do šířky (breadth-first search)***
	+ označíme určený vrchol a všechny neoznačené sousední vrcholy přidáme do seznamu
	+ dokud není seznam prázdný, vybereme vždy první vrchol seznamu, smažeme jej odtud, označíme a všechny neoznačené sousední vrcholy přidáme na konec seznamu
* ***Kruskalův algoritmus* – hledání minimální kostry**
	+ vstupem je ohodnocený graf, výstupem minimální kostra
	+ nejdříve setřídí všechny hrany vzestupně podle ohodnocení
	+ zkouší je postupně přidat do výsledného grafu a v každém kroku kontroluje, zda nevznikl cyklus
	+ pamatuje si a v každém kroku aktualizuje seznam souvislých komponent výsledného grafu a testuje, zda je nově přidaná hrana nespojuje
* ***Dijkstrův algoritmus –* nalezení nejkratší cesty mezi zadanou dvojicí vrcholů**
	+ počítá všechny vzdálenosti z počátečního vrcholu do všech ostatních vrcholů grafu
	+ vstupem je graf ohodnocený nezápornými čísly a počáteční vrchol s
	+ výstupem je pole vzdáleností z vrcholu s do všech dalších vrcholů grafu
	+ po celou dobu běhu si udržuje aktuální nejmenší známé vzdálenosti do všech vrcholů
	+ na začátku je všem vrcholům přiřazena hodnota nekonečno
	+ udržuje si množiny navštívených a nenavštívených vrcholů a pokračuje tak dlouho, dokud množina nenavštívených vrcholů není prázdná



**Strom**

* každý prvek má nejvýše jednoho předchůdce a může mít více než jednoho následníka
* strom je tvořen vrcholy (uzly)
* **neorientovaný souvislý graf, který neobsahuje žádnou kružnici**
* *typy uzlů*
	+ listy – uzly bez následníka, není k nim připojen žádný podstrom
	+ kořeny – uzel bez předchůdce, existuje právě jeden
	+ vnitřní uzly – nejsou listem ani kořenem
* patří mezi rekurzivní datové struktury – každý uzel je současně kořenem stromu a zároveň listem stromu vyšší úrovně
* *dělení stromů*
	+ unární – stupeň uzlu 1
	+ binární – stupeň uzlu 2
	+ ternární – stupeň uzlu 3
	+ vyšší řády se zpravidla neužívají
* *binární strom*
	+ neobsahuje žádný uzel
	+ je složen ze tří disjunktních množin uzlů: kořene, levého podstromu a pravého podstromu (oba jsou taktéž binární stromy)

**trie**

* prostředek k reprezentaci morfémové struktury
* prefixový strom
* vyhledávací strom
* strom, kde jsou lexikograficky uspořádány jednotlivé řetězce
* efektivní implementace morfologického lexikonu
* struktura trie:
	+ uspořádaný strom nad danou abecedou A
	+ v každém uzlu je různé písmeno z abecedy A
	+ klíč je v trie uložen jako cesta od kořene
* trie v každém uzlu obsahuje podřetězce, kterými může pokračovat řetězec v dosud prohledané cestě
* všichni následníci uzlu mají společný prefix, který je shodný s řetězcem přiřazeným k danému uzlu
* výhody:
	+ sdílení společných preﬁxů
	+ v každém případě nalezení nejdelšího shodného preﬁxu
	+ rychlejší nalezení klíče
	+ potřebuje menší paměť
* jeden kmen je společný pro řadu sufixů a intersegmentů

**Syntaktická struktura**

Důležitým prostředkem pro grafické vyjádření struktury věty (její derivace) je graf-strom, který se nazývá derivační nebo **syntaktický strom věty**. Je to orientovaný acyklický graf s následujícími vlastnostmi:

* 1. existuje jediný uzel, tzv. kořen stromu, do něhož nevstupuje žádná hrana
* 2. do všech ostatních uzlů vstupuje právě jedna hrana
* 3. uzly, z nich žádná hrana nevystupuje, se nazývají koncové (terminální) nebo také listy
* 4. při kreslení se zachovává konvence, že kořen je nejvýše a všechny hrany jsou orientovány směrem dolů
* 5. uspořádání hran zachovává slovoslednou relaci, tj. pořadí slov ve větě (zleva doprava)

Uzly syntaktického stromu

****Označení uzlu (název neterminálu) podle zvoleného přístupu reprezentuje:

* **gramatická role (gramatická funkce)**
	+ charakterizují vztahy mezi větnými složkami na povrchové úrovni
	+ určujeme, zda daný větný člen je NP v roli podmětu, NP v roli předmětu, ADVP určující lokaci atd.
	+ v češtině (a jazycích se systémem gramatických pádů) pomáhá k určení gramatické role právě informace o pádu
	+ ovšem přiřazení gramatických rolí ke gramatickým pádům a naopak není zdaleka jednoznačné.
* **tematická role** (též hloubkový/sémantický pád)
	+ na rozdíl od gramatické role se jedná o sémantickou kategorii
	+ určujeme např.:
		- Agens – kdo je životným původcem nějaké cílevědomé činnosti
		- Patiens – co hraje roli entity, na kterou se působí
		- Donor – osoba, která dává

**Složkový strom**

* skupiny slov tvoří větné jednotky, které jsou označovány jako fráze, a jako větné členy (složky, constituents) formují větu
* např. podstatné jméno – součást jmenné fráze (noun phrase – NP), jmenná fráze spolu s předložkou – tvoří předložkovou frázi (prepositional phrase – PP)
* syntaktická struktura věty je zachycována jako **složkový strom**
* modré věci neterminály/konstituenty – každý modrý uzel popisuje nějakou frázi, která může k něčemu být
* každému slovu je přiřazena nějaká složka (neterminál) = lingvistická intepretace, kterou zachycuji svou znalost o struktuře věty
* Chomsky psal o tom, jestli je to univerzální, jestli ve všech jazycích máme jmenné skupiny, slovesa… (oficiální čínština nemá sloveso, tuto funkci slovo nabývá až v kontextu
* popisuje vnořené fráze, dává jim označení
* složkový popis využívá synt
* fráze jsou nositeli lepšího významu než samostatná slova. Syntaktickou informaci můžeme použít jako vstup nějaké další analýzy. Aplikace je třeba identifikace frází → kvalitnější vyhledávání, odpovídání na otázky… (něco, co není fráze, na to se uživatel neptá)
* je třeba morfologická analýza, syntaktická analýza na ní závisí
* pozn.: deskriptivní adekvátnost – jak přesně popisujeme strukturu věty x explikativní adekvátnost

**Závislostní formalismus**

* jeden člen vazby je označován jako řídící, druhý jako závislý
* např. přídavné jméno závisí na řídícím podstatném jménu
* syntaktická struktura věty je zachycována pomocí **závislostního stromu**:
	+ uzly odpovídají elementárním jednotkám vstupu (často slovům)
	+ hrany označují vztahy závislosti mezi elementárními jednotkami
* závislost není relací mezi jednotlivými slovy, ale obecně relací mezi jedním slovem a frází řízenou druhým slovem. např. vazba mezi konkrétním slovesem a podmětem nebo vazba mezi slovesem a předmětem věty
* technicky vzato, závislostní relace je vztahem mezi uzly a podstromy (uzlem a všemi uzly, které na tomto uzlu závisí)
* informaci kóduje jako „čáry“ mezi vstupními slovy
* v základním stromě nejsou neterminály, jen slova a šipky mezi nimi
* závislost je vyjádřena čárami mezi slovy
* vyjadřuje to relaci závislosti a podřízenosti
* každý podstrom je jedna platná fráze
* u závislostního formalismu problém s doplňky (nelze závislost označit dvěma čarami – Pivo je dobré studené.)
* ze závislostního formalismu nejsme dobře schopni určit hranice vět – nevíme, ve které větě je to sloveso. Rozpoznávání hranic vět je dost složitý problém

Závislostní je starší

V současnosti se preferuje závislostní, hlavně v Americe

Složková analýza Chomsky

Závislostní – Šmilauer, Bloomfield

# 4. Reprezentace čísel a znaků v počítači

Reprezentace čísel v počítači

**Vše uložené v počítači jsou jen nuly a jedničky**.

Počítače používají tzv. **dvojkovou soustavu**, jsou to digitální zařízení. Tento způsob se technicky dobře realizuje pomocí elektrického signálu (není napětí = 0; je napětí = 1) i pomocí mechanických prostředků (0 = není prohlubeň; 1 = je prohlubeň nebo výstupek).

Jedna nula nebo jednička (něco je nebo není) je také **nejmenší jednotka informace**, která říká, který ze dvou stejně pravděpodobných stavů nastal.

BITY A JEJICH POČET

Této *nejmenší jednotce informace* (je [1] nebo není [0]) se říká ***1 bit*** (značka malé *b*).

Tvůrci počítačů počítali, kolik různých znaků lze zakódovat pomocí kolika (jak dlouhého řetězce) nul a jedniček. Pokud by byla k dispozici pouze jedna nula či jednička, mohli byste zakódovat jen dva znaky. Např. písmeno A by bylo 0, B pak 1. BABA = 1010 => pro praktickou potřebu málo.

BAJT (B)

Kolik nul a jedniček na jeden znak je tedy potřeba, aby bylo možné **zakódovat celou abecedu**, malá i velká písmena, číslice a ještě zbude rezerva? Tvůrci počítačů se shodli na 8 bitech (00000001). Počet různých kombinací (možností) z 8 nul a jedniček: 256 (28 -> obecně platí, že kombinací zakódovaných dvěma znaky je 2N, kde N je počet bitů).

Této kombinací 8 nul a jedniček dali jméno **bajt** (z anglického byte) se značkou velké **B**.

***Jeden bajt je tedy řetězec osmi bitů.***

Protože bajtů se do počítače vejde hodně, používají se násobné jednotky:

* **1 kilobajt (KB)** je 1 024 bajtů (B)
* **1 megabajt (MB)** je 1 048 576 B, tedy 1 024 KB
* **1 gigabajt (GB)** je 1 073 741 824 B, tedy 1 048 576 KB, tedy 1 024 MB
* **1 terabajt (TB)** je 1 073 741 824 KB, tedy 1 048 576 MB, tedy 1 024GB

Pro jednoduchost stačí většinou uvažovat zaokrouhlené hodnoty, např. KB jako tisíc bajtů, MB jako milion bajtů a GB jako miliardu bajtů nebo tisíc MB.

ALU se nejlépe pracuje s čísly ve dvojkové (binární) soustavě s omezenou, pevnou délkou. Tuto posloupnost jedniček a nul nazýváme slovo (word).

Rozsah hodnot závisí na délce slova (počtu bitů/bajtů) exponenciálně: historické 8bitové slovo pojme 28 = 256 různých hodnot (v přirozených číslech tedy rozsah 0–255), současné 64bitové unese 264 = hodně moc :-)

Číslo 45 zapsané jako 8bitové slovo vypadá takto: 001011012, číslo 93: 010111012.

[**ALU – Arithmetic-logic Unit (aritmetickologická jednotka)**: jednotka provádějící veškeré aritmetické výpočty a logické operace. Obsahuje sčítačky, násobičky (pro aritmetické výpočty) a komparátory (pro porovnávání)]

Vyjádření záporných čísel

S celými čísly se díky metodě dvojkového doplňku (two’s complement) počítá stejně jako s přirozenými, jen využijeme **nejvyšší (první) bit jako znaménko** (nula je plus a jednička minus). Tento systém má jednu nulu a jedno tak trochu „divné“ číslo.

Stále s 8 bity: nejmenší nezáporné číslo je 000000002 = 0, nejvyšší je 011111112 (127; máme o jeden bit méně prostoru). Záporná čísla začínají opět od „nuly“; ovšem už s nastaveným znaménkem: 100000002 a končí hodnotou 111111112. Mezi nejmenším a největším záporným číslem opět můžeme rozdělit 128 hodnot, jako jsme to udělali u kladných (0−127). Končíme ‑1, a proto začínáme ‑128.

Negace a sčítání (odčítání?)

Člověk si opatří opak čísla 45 převrácením bitů a *přičtením jedničky*. V binárním zápise inverze (doplněk) cifer 001011012 vyjde 110100102 a po inkrementaci získáme výsledek 110100112 = −45.

Na negaci −128 nemáme místo; při přičítání jedničky k inverzi 011111112 dochází k přetečení, kterého když si nevšimneme, získáváme opět −128.

Součet 93 a −45 získáme klasickým sčítáním pod sebou, jako známe pro desítkovou soustavu, jen přenášíme dvojku místo desítky.

010111012 93

110100112 −45

001100002 48

Čísla s plovoucí řádovou čárkou (floating-point numbers)

Hodnoty příliš velké nebo neceločíselné se ukládají způsobem, který kopíruje tzv. vědeckou notaci. Například palec v centimetrech (2,54) by se zapsal jako 254 ⋅ 10−2. Při ukládání v počítači stačí rozdělit přidělené místo (např. 64 bitů) na dvě (nestejné) části, do první (větší) zapsat mantisu 254 a do menší části za ní exponent −2.

DIGITALIZACE PODROBĚJI, POTŘEBNÝ POČET BITŮ

Digitalizace = vzorkování původního (analogového) signálů a jeho záznam pomocí toku nul a jedniček.

V počítačových zařízeních se bity používají k záznamu všeho – textů, zvuků i videa. Jak již bylo zmíněno, zápisem pomocí dvou znaků o délce N pozic můžeme zakódovat 2N možností, kde N je počet bitů. Třemi bity můžeme zapsat 23, tedy 8 možností, čtyřmi bity pak 16 možností, pěti 32 možností atd. Ve výpočetní technice se nejčastěji setkáme s těmito hodnotami:

* **8 bitů** (tedy 1 B) umožňuje záznam 28 = 256 možností. Jeden B se dlouho používal pro záznam písmen v tzv. fontech True type, dnes používaná typová písma Open type většinou využívají 3 bajty na znak.
* **16 bitů** (tedy 2B) umožňuje záznam 216 = 65 536 možností. Dnes se tato hodnota používá jen výjimečně.
* **24 bitů** (tedy 3B) umožňuje záznam 225 = 16,8 milionů možností. 24bitová barevná hloubka se často používá při **záznamu obrázků** i **snímků videa**.
* **32 bitů** (4 B) nabízí cca 4,3 miliardy možností, používá se nejčastěji pro **kódování barev**.

DVOJKOVÁ, DESÍTKOVÁ A ŠESTNÁCTKOVÁ SOUSTAVA

Způsob kódování čísel pomocí dvou znaků se v matematice nazývá dvojková nebo také binární soustava. Při práci se setkáme se zápisem i v šestnáctkové (hexadecimální soustavě).

V obvyklé **desítkové soustavě** se čísla zapisují pomocí deseti číslic (0..9). Např. číslo 1 623 je součtem 1 × 1 000 + 6 × 100 + 2 × 10 + 3 × 1. Číslo se tedy skládá z mocnin (řádů) deseti násobených jejich pozičními hodnotami.

Ve **dvojkové soustavě** bude stejné číslo vyjádřeno pomocí mocnin čísla 2 násobených opět jejich pozičními hodnotami, ty však nyní mohou nabývat pouze stavů, 0 a 1.

V **šestnáctkové soustavě** se používá 16 znaků, číslic 0..9 a písmena A..10, B..11, C..12, D..13, E..14, F..15. Číslo 1 623 je vyjádřeno takto:



Neboli 110010101112 v dvojkové soustavě je 162310 v desítkové soustavě a 65716 v šestnáctkové soustavě.

***Převod čísla z desítkové do dvojkové soustavy*** není složitý: stále číslo dělíme dvěma a zbytek po dělení zapisujeme zprava jako vyjádření čísla ve dvojkové soustavě. Např. 1623:2=811, zbytek 1 (zapíšeme 1), 811:2 = 405, zbytek 1 (zapíšeme 1); 405:2 = 202, zbytek 1 (111), 202:2=101, zbytek 0 (0111), 101:2=50, zbytek 1 (10111); 50:2=25, zbytek 0 (010111), 25:2=12, zbytek 1 (1010111), 12:2=6, zbytek 0 (01010111), 6:2=3, zbytek 0 (001010111), 3:2=1, zbytek 1 (1001010111), 1:2=0, zbytek 1 (11001010111), **162310 = 110010101112.**

***Převod čísla z desítkové do šestnáctkové soustavy*** probíhá stejně jako u dvojkové, pouze samozřejmě původní číslo dělíme 16 a opět zapisujeme zbytek po dělení zprava.

Reprezentace znaků v počítači

Se znaky (písmeny, číslicemi, interpunkcí, mezerami, řídícími znaky atd.) pracujeme jako s čísly, každému přiřadíme systematicky jednu pozici v potřebném rozsahu.

Např. americký standard **ASCII** si vystačil se 128 pozicemi (7 bity, osmý se používal při přenosu dat pro kontrolu), ale **neobsahoval diakritiku**. Jazyky s ideogramy naopak potřebují uložit tisíce znaků. Evropané se v mnoha svých standardech vešli buď do 7, nebo do 8 bitů.

*Znakovými sadami myslíme přiřazení čísel znakům*, např. v ASCII se „e“ skrývá pod číslem 101, mezera má číslo 32, rovnítko 61 a zalomení řádku 10. V našem prostředí se používalo mnoho sad, ale stále přežívá microsoftí CP1250 (code page) a v mailech standard ISO-8859-2.

Celosvětově se ale prosazuje obří **znaková sada** **Unicode** (nadmnožina ASCII), která zatím zahrnuje přes 110 000 znaků ze všech možných „abeced“.

Kódováním dáváme číslu, které zastupuje znak, formu. 7- a 8bitové znakové sady stačí zakódovat do sekvence bajtů. Text v Unicode může sestávat z latinky spolu s rozsypaným čajem a výslovností v IPA. Existují kódování s pevným, nebo proměnlivým počtem bajtů.

**Zastaralé kódování UCS-2** používá 16 bitů, zaznamenává proto pouze podmnožinu 65 536 znaků. **Populární kódování UTF-8** zapisuje každý znak 1 bajtem, pokud má znak číselnou hodnotu do 127 (a je v tomto kompatibilní s ASCII), anebo až 4 bajty, kde první má vždy hodnotu větší než 128. České znaky s háčky a čárkami se vejdou do 2 bajtů, valná většina čínštiny do 3, ve 4 jsou jen rarity.

Textové informace jsou vyjadřovány pomocí kódu, které každému znaku přiřadí jedinečný vzor bitů

* ***norma ASCII –*** ústav ANSI (American National Standards Institute)
	+ pomocí sedmibitových posloupností vyjadřuje alfanumerické znaky, interpunkční znaménka a řídící znaky
	+ rozšířením na osm bitů je možné přidat symboly jiných abeced s příslušnou diakritikou
	+ dodatečných 128 bitových vzorů však nestačilo na znaky asijských a dalších jazyků
	+ pro znaky české abecedy se používají rozšíření ISO 8859-2/ISO Latin 2 (unixové systémy), Windows 1250, Mac CE (Apple)
* ***Unicode***
	+ reagoval na nedostatek ASCII
	+ symboly kóduje pomocí 16 bitů, pokrývá všechny existující abecedy
	+ globální standard pro kódování znaků, používá různé znakové sady
	+ ***UTF-8*** – způsob kódování řetězců znaků
		- používá proměnnou délku znaku od 1 do 6 bitů podle pozice v Unicode
		- znaky ASCII jsou kódovány jedním bajtem
		- byl navržen pro zpětnou kompatibilitu s ASCII
		- prostorově úsporné, používá se pro přenos dat
		- při zpracování je nevýhodou nestejná délka znaků
	+ ***UTF-16*** – kóduje řetězce do posloupností 16bitových slov (2 bajty)
		- znaky jsou buď reprezentovány 16bitovým číslem nebo párem čísel
		- znaky nemají pevnou šířku
	+ ***UTF-32*** – kóduje řetězce do posloupnosti 32bitových slov (4 bajty)
		- každý znak je přímo reprezentovaný 32bitových číslem
		- všechny znaky mají stejnou délku
		- vysoké nároky na paměť

# 5. Databáze, jazyk SQL

* ***databáze*** – systém, který převádí velké sady dat na abstraktní nástroje, díky čemuž lze pohodlně vyhledávat a načítat požadované informace
	+ vícerozměrné soubory, které umožňují přístup k informacím z různých perspektiv
	+ využití např. při integrování informací o zaměstnancích, objednávkách apod. při tvorbě webových stránek
* ***schéma*** – popis celé databázové struktury, pomocí něhož databázový software spravuje příslušnou databázi, *dílčí schéma* – popisuje jen část databáze, která souvisí s potřebami určitého uživatele
* ***systém řízení databáze***
	+ *aplikační software* – zajišťuje komunikaci s uživatelem
	+ *systém řízení databáze* – zajišťuje vlastní manipulaci s databází (DBMS)
	+ jejich oddělení má několik výhod
		- můžeme konstruovat a používat abstraktní nástroje, podrobnosti o skutečném uspořádání databáze jsou uloženy izolovaně v systému řízení
		- můžeme řídit přístup k databázi
		- nezávislost dat – můžeme měnit organizaci databáze beze změn aplikačního softwaru
* ***databázový model*** – způsob uložení dat a vazeb mezi nimi
	+ *relační model* – databáze jako logická sada tabulek
		- data jsou prezentována formou tabulek – r e l a c í
		- řádek relace označujeme jako n – t i c i (t u p l e)
		- sloupce označujeme jako a t r i b u t y
		- každý položka ve sloupci popisuje určitou vlastnost/atribut entity, kterou představuje příslušná n-tice
		- v jednom systému můžeme různé typy informací rozdělit do několika relací
		- relační operace
			* SELECT – výběr n-tic s určitými vlastnostmi a jejich umístění do nové relace
			* PROJECT – extrakce sloupců
			* JOIN – sjednocení více relací do jedné
	+ *objektově orientované databázové modely* – obsahují objekty, které jsou vzájemně propojeny tak, aby to odráželo jejich vztahy
		- propojení mezi objekty udržuje systém řízení
		- umožňuje navrhnout celý softwarový systém (aplikační software, systém správy databáze i samotnou databázi)
		- můžeme skrývat a zapouzdřit některé části či aspekty objektů
		- můžeme ukládat inteligentní objekty, které obsahují metody, jak bude reagovat na zprávy týkajících se jeho obsahu a relací
* ***SQL*** – Structured Query Language, dotazovací jazyk většiny systémů relačních databází
	+ popis požadovaných informací (není to řada činností, které je třeba vykonat)
	+ příkazy select, from, where, insert into, delete from, update

# 6. Značkovací jazyky – (X)HTML, XML, DTD, …

* ***značkovací jazyky*** – slouží k obohacení textu o dodatečné informace o významu, struktuře a způsobu zobrazování jednotlivých částí
	+ *značky (tags), příkazy (commands), direktiva* – slouží pro vkládání dodatečných informací přímo do textu
	+ výsledný zdrojový text je ukládán ve formě obyčejného textového souboru
	+ musí obsahovat speciální konstrukce pro zobrazení znaků zvláštního významu (<, >, &, …)
	+ díky otevřenému textovému formátu nevyžadují speciální programové vybavení k editaci
	+ ***popisné/deskriptivní jazyky*** (XML, HTML) – slouží k popisu informací v dokumentu
	+ ***jazyky výkonné/procedurální***(TeX, PostScript) – obsahují navíc instrukce na úrovní programovacího jazyka, umožňují popis vizuální charakteristiky výstupu
	+ ***prezentační jazyky*** (RTF) – soustředí se na to, jak bude označkovaný text zobrazen
* ***SGML*** – Standard Generalised Markup Language
	+ mezinárodní standard, zobecněný jazyk pro značkování
	+ popisuje metody, jak specifikovat aplikačně nezávislou strukturu dokumentu
	+ *elementy* – slouží pro definici struktury
		- každý má definováno, jaké jiné elementy může obsahovat
		- jsou vyznačeny počáteční a koncovou značkou
	+ *atributy* – mohou být definovány jednotlivým elementům
		- jsou jím přiřazovány hodnoty, např. pro kategorizaci elementů
	+ můžeme používat pouze alfanumerické znaky a některé interpunkční znaky
	+ *entity* – slouží k zápisu ostatních znaků, začínají *&* a končí *;* (např. *&nbsp;*)
	+ je poměrně složitý, i pro jednoduché věci je potřeba psát samostatné programy, které využívají nějaký SGML parser

******

* ***XML*** – Extensible Markup Language, reakce na složitost SGML
	+ podmnožina SGML, je určen především pro výměnu dat mezi aplikacemi a publikování dokumentů (nezabývá se však vzhledem ale pouze obsahem)
	+ neobsahuje předdefinované značky, je třeba definovat si vlastní, např. pomocí DTD
	+ ******v nejjednodušších případech nevyžaduje ani DTD, pro popis XML dokumentů se používají i jiné jazyky, např. XML Schema
	+ existuje pro něj více parserů, jsou menší a rychlejší
	+ spravován organizací W3C, otevřenější
	+ snadno převoditelný do jiných formátů, např. pomocí CSS pro zobrazení na obrazovce
* ***DTD*** – Document Type Definition
	+ jazyk pro popis struktury [XML](https://cs.wikipedia.org/wiki/Extensible_Markup_Language) případně [SGML](https://cs.wikipedia.org/wiki/Standard_Generalized_Markup_Language) dokumentu
	+ omezuje množinu přípustných dokumentů spadajících do daného typu nebo třídy. DTD tak například vymezuje jazyky [HTML](https://cs.wikipedia.org/wiki/HyperText_Markup_Language) a [XHTML](https://cs.wikipedia.org/wiki/Extensible_HyperText_Markup_Language)
	+ definice a popis struktury SGML/XML dokumentu
	+ obsahuje všechny přípustné elementy i se svými atributy, jejich strukturu a všechny povolené entity
	+ *SGML parsery* – čtou text SGML a kontrolují, zda odpovídá zadanému DTD
	+ struktura třídy nebo typu dokumentu je v DTD popsána pomocí popisu jednotlivých značek (nebo též elementů) a atributů. Popisuje, jak mohou být značky navzájem uspořádány a vnořeny. Vymezuje atributy pro každou značku a typ těchto atributů
	+ DTD je poměrně starý a málo expresivní jazyk

******

******

*DTD a XML*

* ***HTML***
	+ nejrozšířenější značkovací jazyk, navržený 1990 **T. B. Leem**
	+ používá se pro tvorbu **webových stránek**
	+ aplikace SGML
	+ obsahuje *tagy* a *atributy*
	+ značky se ukládají mezi < >
	+ obsahuje párové a nepárové značky

******

******

* ***XHTML***
	+ r. 2000 vydalo W3C doporučení, aby HTML dokumenty vycházely z XML
	+ nově musí být všechny elementy uzavřené, např. <br />
	+ všechny hodnoty atributů ve značkách musí být v uvozovkách
	+ všechna jména značek a atributů musí být malými písmeny, XML je case sensitive a takto jsou deklarované i v DTD

# 7. Algoritmická složitost – lineární, polynomiální a exponenciální algoritmy, složitost algoritmu a složitost problému, třídy složitosti

* ***složitost algoritmu*** – vztah dané metody k daném prostředkům, které máme k dispozici a kterými jsme omezeni (čas, paměť, počet registrů atd.)
	+ ***časová*** – funkce, která každé množině vstupních dat přiřazuje počet operací vykonaných při výpočtu podle daného algoritmu, tzn. kolik času potřebuje program na své vykonání
	+ ***paměťová*** *–* závislost paměťových nároků algoritmu na vstupních datech, tzn. kolik paměti (úložného prostoru) ke svému běhu program potřebuje
* ***složitost problému*** – složitost nejlepšího algoritmu, který ho řeší. Složitostí problému rozumíme složitost *optimálního* algoritmu konkrétně řešícího zadaný problém.
* ***třídy složitosti problému*** – stanovují obtížnost rozhodnutelnosti daného problému na Turingově stroji
	+ *třída P* – obsahuje problémy rozhodnutelné v polynomiálním čase, jde zpravidla o problémy, které jsou efektivně řešitelné
	+ *třída NP* – obsahuje problémy, které jsou rozhodnutelné pomocí nedeterministického Turingova stroje v polynomiálním čase, tzn. jsme schopni ověřit jejich řešení v polynomiálním čase, nikoliv nalézt řešení
	+ v oblasti informatiky se řeší, zda se tyto třídy rovnají
* ***asymptotická složitost***
	+ nástroj pro porovnání efektivity a rychlosti výkonu algoritmů
	+ složitost vyjadřuje jako funkci vstupu
	+ jak roste složitost algoritmu vzhledem k rostoucímu vstupu
	+ souží ke klasifikaci počítačových algoritmů
	+ zapisuje se pomocí „Omikron notace“, např. *O(N)*
	+ velikost dat *N* se obvykle měří v bitech, bytech nebo buňkách pevné velikosti
	+ můžeme uvažovat složitost v nejlepším případě, průměrném případe a nejhorším případě
	+ *amortizovaná složitost* – určuje časovou složitost algoritmu v sekvenci nejhorších možných vstupních dat
* ***třídy asymptotické složitosti***
	+ *konstantní – O(1) = c*, např. hledání n-tého prvku v poli
	+ *logaritmická – O(log n)*, např. binární vyhledávání
	+ *lineární – O(n)*, např. hledání v neseřazených posloupnostech nebo jeden průchod polem
	+ *lineárnělogaritmická – O(n \* log n),* např. řadicí algoritmy
	+ *kvadratická – O(n2)*, např. průchod všech dvojic v poli, bubble sort
	+ *kubická – O(n3)*, např. násobení matic podle definice
	+ *obecná polynomiální – O(nc)*
	+ *exponenciální – O(cn),* např. problém obchodního cestujícího
	+ *faktoriálová – O(n!),* např. vyhodnocení všech možných permutací *n* prvků
	+ algoritmy pracující s lepší než exponenciální/faktoriálovou složitostí jsou efektivní

# 8. Formální jazyky a automaty – regulární jazyk, bezkontextový jazyk, konečný automat, formální gramatika, typy gramatik podle Chomského

*Formální jazyk*

**Struktura jazyka** zahrnuje informace o:

* co jsou slova (z jakých znaků, jaké slovní tvary a jejich složky)
* jak se slova (větné složky) kombinují do vět
* co slova označují, jaké jsou jejich lexikální významy
* jak se význam věty skládá z významů slov a slovních spojení (větných
* složek)

**Zpracování jazyka dále potřebuje:**

* obecnou (encyklopedickou) znalost světa (ontologie)
* inferenční mechanismus
* znalost komunikační situace

**Roviny analýzy jazyka** – fonetická, morfologická, syntaktická, sémantická, pragmatická; kontextová, znalost zákl. ontologie, jazykové metaznalosti

Přirozený jazyk vs. formální jazyk

***Přirozený jazyk***: přirozený jazyk je jazyk, který je mluvený nebo napsaný lidmi pro univerzální komunikaci

***Formální jazyk:*** množina konečných slov (tj. slov konečné délky) nad určitou abecedou. Místo výrazu „slovo“ se někdy užívá výraz „řetězec“. Definice pojmu *formální jazyk* se může měnit podle toho, v jakém kontextu a v jakém vědním oboru jej používáme.

Pro potřeby matematické teorie jazyků je třeba definovat formální jazyk abstraktně jako matematický model. Využití formálních jazyků: v oblasti programovacích jazyků k definici syntaxe programovacího jazyka; formální jazyky jsou základním kamenem teoretické informatiky. Teorie formálních gramatik se využívá například v oblasti tvorby překladačů a kompilátorů.

Algebra na množině slov nad danou abecedou

*abeceda*

* konečná množina symbolů (znaků, písmen)
* značíme ji (množinu) Σ [sigma]
* prvky abecedy = znaky abecedy (případně taky písmena nebo symboly)
* příkladem abecedy je třeba množina {a, b}, slovem nad touto abecedou je např. ababa (abeceda má jen dvě písmena)
* další příklad abecedy: množina číslic {0, 1,….,9}, nebo prázdná množina Ø

*jazyk*

* jazyk nad abecedou Σ je libovolná množina slov nad Σ (jazyky nad Σ jsou tedy právě podmnožiny Σ\*)
* např. {10,1,011101} je jazyk nad abecedou {0,1}

*slovo (řetězec)*

* libovolná konečná posloupnost prvků abecedy Σ
* značíme w, v
* např. w = aabab

*délka slova*

* počet členů posloupnosti w
* značíme #w, |v|
* počet výskytů znaku a ve slově w značíme #a(w)
* počet znaků b ve slově abaaba: **#b (abaaba) = 2**

*prázdné slovo*

* prázdná posloupnost znaků, která má nulovou délku
* značíme ε

Gramatika

* model, který generuje jazyk; systém pravidel
* gramatika **G je čtveřice (N, Σ, P, S)**, kde:
	+ N – neprázdná konečná množina neterminálů
	+ Σ – konečná množina terminálů – N ∩ Σ = Ø
	+ P – konečná množina pravidel
		- pravidlo (α, β) - zapisujeme ve tvaru α → β (čteme: α přepiš na β)
	+ S – počáteční neterminál (kořen gramatiky)
* *odvození slova z* gramatiky – podle pravidel přepisujeme neterminály, dokud nedostaneme řetězec terminálů – slovo jazyka
* gramatika *G* generuje jazyk *L*, pokud existuje odvození každého slova jazyka *L* z gramatiky *G*

**Pojmy**

* **Terminál:** prvotní symbol daného jazyka; symbol z množiny symbolů jazyka, který už v procesu dané analýzy nenahrazujeme; jde o cílový termín, který je součástí nějakého řetězce jazyka (a, b, c)
* **Neterminál:** prvotní symbol daného jazyka; symbol definovaný pomocí pravidel jazyka (A, B, C)
* **Začáteční symbol:** jeden z neterminálů, ze kterého začíná generování vět jazyka, případně analýza věty
* **Přepisovací pravidla:** jsou základem pro generování vět jazyka; pravidla přepisujeme tak dlouho, dokud nedostaneme řetězec terminálů

**Příklad:**

Mějme gramatiku (N, Σ, P, S), kde:

Σ = {a, b}

N = {S, A}

P = { S → A, A → AA, A→ a }

Příklady odvození z této gramatiky jsou:

S ⇒ A ⇒ a

S ⇒ A ⇒ AA => aA => aAA => aaA => aaa

Chomského hierarchie

* mechanismus vyčerpávajícím způsobem popisující jazyk
* gramatiky rozděleny do 4 skupin (typů) na základě omezení na tvar pravidel
* gramatika vyššího typu je vždy současně i gramatikou nižšího typu
* mechanismus vyčerpávajícím způsobem popisující jazyk
* matematický model gramatiky; nástroj, který lze aplikovat například v lingvistice
* gramatika = systém pravidel; jazyk = množina řetězců
* hierarchie je určena podle omezení kladených na tvar pravidel

Typ 0 (fázové gramatiky, rekurzivně spočetné)

Obsahuje pravidla v nejobecnějším tvaru, libovolná gramatika, neklade žádná omezení na tvar pravidel, povoluje přepis řetězců na řetězce, neomezené přepisovací systémy. Rekurzivně vyčíslitelné, ekvivalentní síle Turingova stroje. Jde jí popsat nejvíc jazyků; obecný přepisovací systém

Typ 1 (kontextové gramatiky, nezkracující)

Pro každé pravidlo α → β platí | α | ≤ | β |, výjimku tvoří S → ε, pokud se S nevyskytuje na pravé straně žádného pravidla. V kontextových pravidlech lze neterminální symbol nahradit řetězce pouze tehdy, je-li jeho pravým kontextem řetězec β a levým kontextem řetězec α. Nepřipouštějí, aby neterminální symbol byl nahrazen prázdným řetězcem. Při generování věty nemůže dojít ke zkracování generovaných řetězců. Víc termů na levé straně (kontext neterminálu), na levé straně se počet termů zmenšuje (ASB → AAaBB). **Umí anbncn**.

Levá strana pravidel nemá více symbolů než pravá, přičemž na evé straně každého pravidla se vyskytuje aspoň jeden neterminální symbol.

Typ 2 (bezkontextové/nekontextové gramatiky)

Každé pravidlo je tvaru A → α, kde | α | ≥ 1, výjimku tvoří S → ε, pokud se S nevyskytuje na pravé straně žádného pravidla. Nahrazení neterminálního symbolu *A* řetězcem lze provést bez ohledu na okolí, v němž by se neterminální symbol mohl vyskytovat. Jsou nejzajímavější pro popis syntaktické stavby přirozených jazyků. Neterminál lze přepsat na cokoliv (S → aSb), jsou ekvivalentní síle zásobníkových automatů, **umí anbn, neumí anbncn**.

Levá strana pravidel má pouze jeden neterminál. Jsou silnější než reglární gramatiky; Touto gramatikou lze popsat slovo, které může místo ahoj obsahovat celou větu znova *(((Ahoj, jak se máš) jak se vede) jak se daří) jak se máš*.

Bezkontextové gramatiky modelují syntaktickou strukturu jazyka tím, že ke každému generovanému řetězci (větě) přiřazují hierarchickou strukturu v podobě tzv. **frázového ukazatele** (bezprostředněsložkového stromu). Jejich pravidla vyjadřují hierarchickou strukturu generovsaných vět (řetězců) a jejich rozklad na tzv. složky nebo fráze (elementy na pravé straně pravidla tvoří bezprostřední složky strany levé). Bezprostřední složky ve frázovém ukazateli zachycují především těsnost, s jakou se k sobě jednotlivá pravidla vážou.

Pro popis přirozených jazyků však slabá generativní síla nekontextových gramatik nestačí.

Typ 3 (regulární gramatiky, konečněstavové)

Každé pravidlo je tvaru A → aB nebo A → a, výjimku tvoří S → ε, pokud se S nevyskytuje na pravé straně žádného pravidla. Tzn. přepisujeme **neterminál → terminál[neterminál]** (S → aS, S → b). Jsou ekvivalentní síle konečných automatů, **neumí anbn**.Jediný možný neterminální symbol na pravé straně pravidla stojí zcela vpravo.

Regulární gramatiky se hodně používají v informatice. Jejich výhoda je v tom, že díky nim lze rychle zjistit, jestli je určitý reg. Jazyk generovaný daný reg. Gramatikou. Reg. Jazykem se **nedá** **zajistit, aby se některá část opakovala.**

Nejrestrikčnější, jde jí popsat nejmíň jazyků; čeština by nešla.

*Pozn.: Nejčastěji se pracuje s regulárními a bezkontextovými gramatikami a jazyky. Tyto typy gramatik jsou efektivně zpracovatelné na počítačích a jako základ se používají i při zpracování textů v přirozeném jazyce.*

*Přirozený jazyk byl dlouho pokládán za bezkontextový, nyní je prokázáno, že obsahuje kontextové prvky.*

Automaty

* abstraktní model charakterizující jazyky
* stavové systémy, které čtou po znacích slovo na vstupu, s přečtením každého znaku změní stav a na konci rozhodnou, zda slovo je akceptováno či nikoliv
* nejjednodušší je *konečný automat*
* **automat A nad abecedou** = {a, b} je znázorněn stavovým diagramem vlevo. Jeho součásti jsou:

**Q** = {q0 , q1 , q2 } … množina stavů

**Σ** = {a, b} … (vstupní abeceda); symbol ε značí prázdné slovo

 **δ** : Q × Σ → Q … přechodová funkce, kde

(viz stavový diagram vpravo): δ(q0 , a) = q0 , δ(q0 , b) = q1 ,

 δ(q1 , a) = q2 , δ(q1 , b) = q1 , δ(q2 , a) = δ(q2 , b) = q1

**q0** … počáteční stav automatu

**F** = {q1 } … množina přijímacích stavů automatu (může jich být více)

příklad:

w1 = ababab

|  |
| --- |
| (q0 , ababab) ⊢zkráceně (v řeči stavů): |

w2 = abba

|  |
| --- |
| (q0 , abba) ⊢ |

*Pozn.: Je-li A konečný automat nad abecedou Σ , pak jazykem přijímaným automatem A míníme množinu L(A) všech slov přijímaných automatem A, tj. L(A) = {w ∈ Σ; A přijímá slovo w}.*

Analýza činnosti konečného automatu

* Je dán konečný deterministický automat A nad abecedou Σ s množinou stavů Q,
* přechodovou funkcí δ : Q x Σ → Q a množinou přijímacích stavů F ⊆ Q.
* Jazyk přijímaný automatem A je L (A).
* Iterace přechodové funkce (označovaná δ\*) působí na množině všech konfigurací jako (stav, slovo) ⊢ \* (stav', slovo'). Takový převod nazýváme **dílčím výpočtem automatu *A***.
* Speciálně pro libovolné slovo w ∈ Σ\* má výpočet automatu *A* na slově w tvar (qo, w) ⊢ \* (qi, ε ); řekneme, že výpočet skončil ve stavu qi. Zřejmě platí: w ∈ L(A) právě když výpočet automatu A na slově w skončí v přijímacím stavu.
* Příklad: *Ukázkový automat A nad abecedou Σ = {a, b} je znázorněn stavovým diagramem vlevo.*
* Ukažme si nejdříve dílčí výpočet našeho automatu na slově

|  |
| --- |
| (q1, aababba) ⊢Zkrácená forma: (q1, aababba) ⊢ \* |

w = aababba odstartovaný ze stavu q1 mající 4 kroky:

Přehled malých automatů nad abecedou Σ = {a, b}

* existují právě dva jednostavové automaty nad Σ
	+ První z nich přijímá všechna slova jazyka Σ\*
	+ druhý z nich žádné
* dvojstavových automatů nad Σ se stavy qo, qi je celá řada, ovšem všechny ty, pro něž je F = {q0, qi} přijímají všechna slova jazyka Σ\*, zatímco ty, pro něž je F = Ø nepřijímají žádné slovo.
* dále zvláštní skupinu tvoří ty automaty, jejichž přechodová funkce dává δ(q0, a) = δ(q0, b) = q0.
* v nich je totiž stav qi *nedosažitelný*, takže se *redukují* na jednostavové (viz úvahy výše).
* tabulka níže zachycuje všechny "podstatné" dvoustavové automaty. Ostatní se z nich získají výměnou rolí znaků *a* a *b* či změnou přijímacího stavu z q1 na q0:





**Zásobníkový automat**

* zásobníkový automat (PDA z anglického pushdown automaton) je teoretický [výpočetní model](https://cs.wikipedia.org/wiki/V%C3%BDpo%C4%8Detn%C3%AD_model) používaný v [informatice](https://cs.wikipedia.org/wiki/Informatika) pro studium [vyčíslitelnosti](https://cs.wikipedia.org/wiki/Vy%C4%8D%C3%ADslitelnost) a obecně [formálních jazyků](https://cs.wikipedia.org/wiki/Form%C3%A1ln%C3%AD_jazyk). Popisuje jednoduchý [počítač](https://cs.wikipedia.org/wiki/Po%C4%8D%C3%ADta%C4%8D), který má jako pracovní paměť vedle konečně stavové jednotky k dispozici [zásobník](https://cs.wikipedia.org/wiki/Z%C3%A1sobn%C3%ADk_%28datov%C3%A1_struktura%29). Zásobníkový automat dokáže rozpoznávat [bezkontextové jazyky](https://cs.wikipedia.org/wiki/Bezkontextov%C3%BD_jazyk)
* zásobníkový automat se v podstatě skládá z [konečného automatu](https://cs.wikipedia.org/wiki/Kone%C4%8Dn%C3%BD_automat), který má navíc k dispozici potenciálně nekonečné množství paměti ve formě zásobníku. Obsah tohoto zásobníku ovlivňuje činnost automatu tím, že vstupuje jako jeden z parametrů do přechodové funkce
* **proces:**
	+ na počátku se automat nachází v definovaném počátečním stavu a zásobník obsahuje pouze počáteční symbol. Dále v každém kroku podle aktuálního stavu, symbolů na vrcholu zásobníku a symbolu na vstupu provede přechod, při kterém vyjme ze zásobníku vrchní symbol, vloží místo něj jiné a ze vstupu přečte další symbol. Toto se opakuje
	+ po dokončení činnosti (po přečtení celého vstupu, pokud do té doby nedojde k chybě) je rozhodnuto, jestli automat vstupní řetězec přijal. K tomu mohou sloužit dvě kritéria:
		- stav, ve kterém se na konci automat nachází, patří do množiny přijímajících stavů, nebo
		- zásobník je na konci prázdný.
	+ pbě definice jsou ekvivalentní, automaty na sebe lze vzájemně převádět (u druhé možnosti je možno z definice automatu zcela vypustit množinu přijímajících stavů).
* příklad:

*Q* = {*q*}

*T* = {a, b, c}

*G* = {A}

*δ* = {

(*q*, a, ε) → (*q*, A),

(*q*, b, ε) → (*q*, ε),

(*q*, c, A) → (*q*, ε),

}

*q*0 = *q*

*Z*0 = ε

* ***bezkontextové jazyky***
	+ jsou akceptovány zásobníkovým automatem
	+ mohou být vygenerovány bezkontextovými gramatikami
	+ každý bezkontextový jazyk lze převést do *Chomského normální formy* – obsahuje pouze pravidla tvaru *X → Y Z | a*, kde *X, Y, Z* jsou neterminály a *a* je terminální symbol
	+ využitelné např. u návrhů programovacích jazyků -> musíme určit, které výrazy do tohoto jazyka patří
	+ vytvoří se gramatika, která definuje, jak může programovací jazyk vypadat
	+ např. napíšeme-li IF .. musí následovat nějaký výraz (např. IF jméno = „Petr“)
* ***regulární jazyky***
	+ nejjednodušší formální jazyky
	+ je akceptovaný deterministickým i nedeterministickým konečným automatem
	+ může být popsán regulárním výrazem
	+ může být vygenerován regulární gramatikou
	+ všechny regulární jazyky jsou bezkontextové
	+ použití např. na webu – ověření toho, zda uživatel zadal správnou e-mailovou adresu

S -> B

S -> „a“ B | „b“ B | „c“ B …. „z“ B | C

C -> „a“ D

D -> „a“ D| „b“ D | „c“ D | … „z“ D | E

E -> „.“| F

F -> „cz“ | „sk“ | „com“

# 9. Regulární výrazy, CQL

* ***regulární výraz*** – řetězec popisující celou množinu řetězců (regulární jazyk)
	+ další způsob formální reprezentace regulárních jazyků; umožňují jej popsat jako výsledek kompozice několika jednoduchých operací nad jazyky (tedy **nerekurzivní popis** – na rozdíl od gramatik a konečných automatů)
	+ využití – vyhledávání textu, manipulace s textem
	+ skládá se z literálů textu, které se mají shodovat, a speciálních znaků, které nejsou součástí hledaného textu, ale slouží pro popis alternativ, množin, počtů výskytů a přepínačů
	+ *metaznaky*
		- **.** (tečka) – libovolný znak, přesně jeden znak
		- \* – libovolný počet opakování předchozího znaku
			* les.\* = les, lesk, lesklý, lesní, lest,
		- + – libovolný počet opakování předchozího znaku > 0
		- [abc] – povolujeme jeden ze skupiny znaků *a, b, c*
			* [bB]ože = bože, Bože
		- [^ ] = odpovídá jednomu znaku, neuvedenému v závorkách ( [^abc] odpovídá libovolnému znaku krom *a, b, c*)
		- skupinu znaků můžeme zapsat také jako interval, např. [1–5], [A–Z]
		- opakování určité sekvence – výraz uzavřeme do závorek (a) a za pravou závorku doplníme kvantifikátor
			* např *ko(ko)?* odpovídá *kos* a *kokos*
		- **|** – slouží jako oddělovač variant, mezi nimiž si můžeme vybrat, např. Petr|Pavel
		- **kvantifikátory**

|  |  |
| --- | --- |
| ? | minimálně 0krát, maximálně 1krát |
| \* | minimálně 0krát (maximálně neomezeno) |
| + | minimálně 1krát (maximálně neomezeno) |
| {n} | právě nkrát |
| {m,n} | minimálně mkrát, maximálně nkrát |
| {m,} | minimálně mkrát (maximálně neomezeno) |

* + - **předdefinované skupiny znaků**

|  |  |
| --- | --- |
| \d | číslice 0-9 |
| \D | jakýkoliv znak kromě číslic 0-9 |
| \w | znaky „slova” (ekvivalentní zápisu [a-zA-Z0-9\_]) |
| \W | jakýkoliv znak kromě znaků „slova” (ekvivalentní zápisu [^a-zA-Z0-9\_]) |
| \s | „bílé” znaky (mezera, tabulátor, znaky pro zalomení řádků) |
| \S | jakýkoliv znak kromě „bílých” znaků |

* + - **příklady**

|  |  |
| --- | --- |
| a+ | sekvence písmen a (1 a více znaků) |
| a\* | sekvence písmen a (0 a více znaků) |
| o?kov | okov či kov |
| tel(efon)? | tel či telefon |
| telef(on|ax) | telefon či telefax |
| [0-9]|[1-9][0-9] | čísla 0 až 99 |
| \d{2} | sekvence dvou číslic desítkové soustavy (00, 01, …,98, 99) |
| [0-9a-fA-F]|[1-9a-fA-F][0-9a-fA-F]+ | hexadecimální čísla |
| (19|20)\d{2} | letopočty 1900-2099 |
| \d{2,6} | sekvence dvou až šesti číslic |
| [^ ,.]+ | neprázdná sekvence znaků mezi nimiž nesmí být mezera ( ), čárka (,) či tečka (.) |
| ^P.\* | řetězec, který začíná písmenem P za nímž následuje libovolný (i nulový) počet libovolných znaků |
| \d+0$ | řetězec, který končí znakem 0 (nula), kterému předchází minimálně jedna číslice |
| a+b | ab, aab, aaab atd. |
| a\+b | a+b |

* ***CQL*** – dotazovací jazyk pro práci s korpusy, dotazy jsou zadávány ve formě [atribut="hodnota"]
	+ *atribut* – word, lemma, tag apod.
	+ *hodnota* – samotný výraz nebo vzor specifikovaný regulárním výrazem
	+ dotaz můžeme omezit na strukturní atributy (věta, doc apod.)
	+ dotaz na více pozic současně vznikne zřetězením dotazů, např. [lemma="mít"][][lemma="srdce"]
	+ další viz <https://www.sketchengine.co.uk/documentation/wiki/SkE/CorpusQuerying>

# 10. Umělá inteligence (jaké problémy řeší a jaké algoritmy používá)

* oblast informatiky, která se snaží budovat autonomní stroje, které dokážou vykonávat složité úkoly bez lidského zásahu, chce naučit počítače vnímat a uvažovat
* ***inteligentní agenti*** – zařízení, které reaguje na podněty ze svého okolí
	+ jsou vybaveni senzory k získání dat ze svého okolí, agent musí rozumně reagovat na přijatá data
	+ roboti, bezpilotní letadla, postavy ve videohře atd.
	+ úrovně reakcí agenta
		- reflexní akce
		- sledování cíle – je třeba mít plán akcí nebo umět vybrat nejlepší možnou akci
		- porozumí informacím dodávaným ze senzorů
		- schopnost formulovat odpovědi ve formě kompatibilní s jejich výkonnými jednotkami (např. generování řeči)
* ***metodiky výzkumu***
	+ *konstrukční* – vytváření systémů, které vykazují lidské chování
	+ *teoretický* – vývoj výpočetních modelů lidské inteligence
* ***Turingův test***
	+ člověk komunikuje s testovanou osobou na dálku pomocí klávesnice, pokud nedokáže svůj protějšek spolehlivě odlišit od člověka, prošel stroj testem a chová se inteligentně
	+ dnes se již nepovažuje za tolik průkazný, dojem inteligence lze simulovat
		- např. program DOCTOR (verze ELIZA)
* ***vnímání obrazu***
	+ *zpracování obrazu* – identifikace vlastností obrazu
	+ *analýza obrazu* – porozumění tomu, co vlastnosti znamenají
* ***zpracování jazyka***
* ***uvažování***
	+ *produkční systémy –* abstraktní entita, která shrnuje skupinu myšlenkových problémů se stejnými vlastnostmi, skládá se z
		- kolekce stavů
			* stav – situace, které může v aplikačním prostředí nastat
			* startovní stav – výchozí, počáteční stav
			* cílový stav – požadovaný, konečný stav
		- sada produkcí (pravidel nebo tahů) – operace, kterými lze v aplikačním prostředí převádět jeden stav na jiný
		- řídicí systém – obsahuje logiku, která řeší problém přechodu ze startovního do cílového stavu, program, který kontroluje aktuální stavy řešení
			* obor problému – souhrn všech stavů, produkcí a předběžných podmínek v produkčním systému, často se znázorňuje v podobě stavového grafu
	+ *expertní systémy* – softwarové sady, které simulují uvažování nad příčinami a následky, např. diagnostika nemocí, volba terapií; využívají principů logického vyvozování a často jsou implementovány v logických programovacích jazycích
	+ *vyhledávací stromy*
		- kořenovým uzlem je startovní stav
		- podřízené položky každého uzlu jsou stavy, které ze z nadřazeného stavu získat pomocí jedno produkce
		- každá cesta od kořene k listu reprezentuje cestu mezi odpovídajícími stavy
	+ *heuristika*
		- k vyhledávání můžeme použít postupy prohledávání do šířky, do hloubky (sleduje slibnější trasy po svislých cestách) nebo algoritmem best-first vyhledávání – tzn. vybíráme tu možnost, která „vypadá“ nejslibněji
		- heuristika je metoda, která slouží k identifikaci nejslibnějšího z několika stavů
		- každému stavu je přiřazena kvantitativní hodnota, která se snaží měřit "vzdálenost" od tohoto stavu k nejbližšímu cíli
		- měla by mít dvě vlastnosti
			* rozumný odhad úsilí, které zbývá vynaložit k řešení problému
			* měla by být snadno vypočitatelná
		- *algoritmus A\** - kromě heuristických hodnot zohledňuje také „akumulované náklady“ na dosažení každého listového uzlu, tzn. počítá celé potenciální vzdálenosti, ne jen zbývající
* ***reprezentace znalostí a manipulace s nimi***
	+ jak ukládat a reprezentovat znalosti, aby s nimi systémy mohly pracovat
	+ způsob uložení musí umožňovat rychlou dostupnost
	+ *metauvažování* – uvažování o uvažování, integrace různých myšlenkových procesů do procesu extrakce znalostí
	+ *předpoklad uzavřeného světa* – pokud nelze výraz odvodit z dostupných informací, je nepravdivý
	+ *problém okna* – problém zachování aktuálnosti znalostí v proměnlivém prostředí
* ***učení***
	+ chceme také, aby agenti dokázali získávat nové informace, uměli se samostatně učit
	+ *učení imitací* – první úroveň, stroj si zaznamenává příslušné kroky řešení, které jsou demonstrovány člověkem
	+ *učení s učitelem* – agent zobecňuje řešení člověkem pro řadu příkladů a vyvozuje z nich obecně uplatnitelný algoritmus, cvičí se na školicí sadě (training set)
	+ *samostatné učení* – agent dostane obecné pravidlo, které mu umožní usoudit, zda při řešení problému uspěl nebo ne a podle toho se zdokonaluje
* ***genetické algoritmy*** – hledají řešení pomocí kombinace náhodného chování se simulací rozmnožování a evolučního procesu přírodního výběru, napodobuje proces přežití nejzdatnějšího tedy nejlepšího řešení
* ***umělé neuronové sítě***
	+ *neuron* – softwarová jednotka, poskytuje výstup 1 nebo 0 podle toho, zda její efektivní vstup přesahuje tzv. prahovou hodnotu
	+ každému spojení jsou přiřazeny *váhy*
	+ neuron vynásobí každou vstupní hodnotu odpovídající váhou spojení a výsledky sečte, aby zjistil efektivní vstup
	+ 1 – excitovaný stav, hodnota přesahuje prahovou hodnotu, 0 – tlumený stav
	+ neuronová síť je jeden z [výpočetních modelů](https://cs.wikipedia.org/wiki/V%C3%BDpo%C4%8Detn%C3%AD_model) používaných v [umělé inteligenci](https://cs.wikipedia.org/wiki/Um%C4%9Bl%C3%A1_inteligence). Jejím vzorem je chování odpovídajících biologických struktur. Skládá se z umělých (nebo také formálních) neuronů, jejichž předobrazem je biologický [neuron](https://cs.wikipedia.org/wiki/Neuron). Neurony jsou vzájemně propojeny a navzájem si předávají signály a transformují je pomocí určitých [přenosových funkcí](https://cs.wikipedia.org/w/index.php?title=P%C5%99enosov%C3%A1_funkce&action=edit&redlink=1). Neuron má libovolný počet vstupů, ale pouze jeden výstup
	+ neuronové sítě se **používají** mimo jiné i pro [rozpoznávání](https://cs.wikipedia.org/wiki/Rozpozn%C3%A1v%C3%A1n%C3%AD) a [kompresi](https://cs.wikipedia.org/wiki/Komprese) obrazů nebo zvuků, předvídání vývoje časových řad (např. burzovních indexů), někdy dokonce k [filtrování](https://cs.wikipedia.org/wiki/Filtrov%C3%A1n%C3%AD_zpr%C3%A1v) [spamu](https://cs.wikipedia.org/wiki/Spam). V [lékařství](https://cs.wikipedia.org/wiki/L%C3%A9ka%C5%99stv%C3%AD) slouží k prohlubování znalostí o fungování [nervových soustav](https://cs.wikipedia.org/wiki/Nervov%C3%A1_soustava) živých [organismů](https://cs.wikipedia.org/wiki/Organismus). Například [perceptronová](https://cs.wikipedia.org/wiki/Perceptron) síť vznikla původně jako simulace fyziologického modelu rozpoznávání vzorů na [sítnici](https://cs.wikipedia.org/wiki/S%C3%ADtnice) [lidského oka](https://cs.wikipedia.org/wiki/Lidsk%C3%A9_oko)
	+ rozpoznávání řeči, řízení auta, rozpoznávání ručně psaného písma
* ***asociativní paměť*** – vyhledávání informací souvisejících s aktuálně zpracovanými (zvuky vyvolávající obrazy apod.)
* ***robotika*** – studium fyzických autonomních agentů s inteligentním chováním, hlavním cílem je osvobodit roboty od závislosti na člověku

**Prolog**

**Prolog** je [logický](https://cs.wikipedia.org/wiki/Logick%C3%A9_programov%C3%A1n%C3%AD) [programovací jazyk](https://cs.wikipedia.org/wiki/Programovac%C3%AD_jazyk). Patří mezi tzv. [**deklarativní programovací jazyky**](https://cs.wikipedia.org/wiki/Deklarativn%C3%AD_programov%C3%A1n%C3%AD), ve kterých programátor popisuje pouze cíl výpočtu, přičemž přesný *postup*, jakým se k výsledku program dostane, je ponechán na libovůli systému.

Prolog se snaží o pokud možno abstraktní vyjádření faktů a logických vztahů mezi nimi s potlačením imperativní složky.

Prolog je využíván především v oboru [umělé inteligence](https://cs.wikipedia.org/wiki/Um%C4%9Bl%C3%A1_inteligence) a v [počítačové lingvistice](https://cs.wikipedia.org/wiki/Po%C4%8D%C3%ADta%C4%8Dov%C3%A1_lingvistika) (obzvláště [zpracování přirozeného jazyka](https://cs.wikipedia.org/wiki/Zpracov%C3%A1n%C3%AD_p%C5%99irozen%C3%A9ho_jazyka), pro nějž byl původně navržen).

Syntaxe jazyka je velice jednoduchá a snadno použitelná pravě proto, že byl původně určen pro počítačově nepříliš gramotné lingvisty.

Prolog je založen na [**predikátové logice prvního řádu**](https://cs.wikipedia.org/wiki/Predik%C3%A1tov%C3%A1_logika_prvn%C3%ADho_%C5%99%C3%A1du); konkrétně se omezuje na [**Hornovy klauzule**](https://cs.wikipedia.org/wiki/Hornova_klauzule). Běh programu je pak představován aplikací dokazovacích technik na zadané klauzule. Základními využívanými přístupy jsou **unifikace,**[**rekurze**](https://cs.wikipedia.org/wiki/Rekurze)**a**[**backtracking**](https://cs.wikipedia.org/wiki/Backtracking).