PROGRAMOVÉ STRUKTURY: ÚVOD DO PŘEDMĚTU

Mise předmětu, podmínky absolvování

14. února 2013

© jstein

PGS 2013 – personální obsazení

- Přednášky:
  - UP-104: 12:05–14:40 (dvě části 12:05–13:15 + 13:25–14:30)
  - Josef Steinberger
    - UL418
    - ÚH
      - Úterý 9:20-10:05
      - Středa 9:20-10:05
    - E-mail: jstein@kiv.zcu.cz
- Cvičení:
  - UL-409
  - Úterý 11:10 + 13:00 + středa 16:40: Martin Zíma
  - Středa 11:10 + 13:00: Josef Steinberger

Mise předmětu

- Rozšířit obzory v programování
- Komparativní studie programovacích jazyků
- Znáte Javu, objektové programování, možná procedurální, ale existuje řada dalších druhů programování, se kterými se můžete v budoucnu setkat (komponentové, aspektové, logické, funkcionální, atd.) – každé má své silné stránky
- Cílem je, aby jste získali přehled + vyzkoušeli si prakticky několik základních konstrukcí v určitých reprezentativních jazycích
- Cílem není naučit se programovat v 5 jazycích

Program přednášek

1. Úvod, historie programovacích jazyků, paradigmata programování
2. Paradigmata programování, syntaxe, sémantika
3. Paralelní programování (převážně vlákna v Javě)
4. Paralelní programování 2
5. Skriptovací jazyky (převážně Python)
6. Skriptovací jazyky 2
7. Skriptovací jazyky a XML
8. Logické programování (Prolog)
9. Logické programování (Prolog II)
10. Funkcionální programování (Lisp)
11. Funkcionální programování II
12. Porovnání vlastností imperativních jazyků
13. Úvod do překladačů

Program cvičení

1. Úvod
2. Regulární výrazy – výukové
3. Vlákna v Javě – výukové
4. Vlákna v Javě – výukové
5. Vlákna v Javě – ověřovací
6. Vlákna v Javě – ověřovací
7. Python – výukové
8. Python – výukové
9. Python – ověřovací
10. Python – ověřovací
11. Prolog – výukové
12. Lisp – výukové
13. Zápis zápočtů

Podmínky absolvování

- Zápočet
  - Jednodušší způsob získání zápočtu
    - Úspěšné absolvování ověřovacích cvičení – program tvořen jednotlivcem
  - Složitější způsob získání zápočtu
    - Úspěšné absolvování oprav v zápočtovém týdnu – program tvořen jednotlivcem
  - Singularity (doložené – např. dlouhodobá nemoc)
    - Vytvoření semestrálních prací z neabsolvovaných ověřovacích cvičení dle zadaní vyučujícího
- Zkouška
  - Písemná
  - Bez pomůcek
  - ½ teoretické otázky – odpověď jednou větou, popř. výčtem, výrazem
  - ½ příklady (reg. výrazy, vlákna, Python, Prolog, Lisp)
  - Nutno získat polovinu bodů z teoretické i praktické části (1=85%, 2=70%, 3=50%)

Zdroje

- Materiály PGS na Webu
- Catalog of Free Compilers and Interpreters. http://www.idiom.com/free-compilers/
- Free electronic book http://www.mindview.net/Books
- http://www-cgi.cs.cmu.edu/afs/cs.cmu.edu/user/dst/www/LispBook/index.html
- http://www.lisp.org/alu/res-lisp-education.clp
- http://clisp.cons.org
- R.W.Sebesta: Concepts of Programming Languages
- H.Schildt: Java2 Příručka programátora
- Ježek, Racek, Matějovič: Paralelní architektury a programy
- Herout: Učebnice jazyka Java + další díly
- Návrat, Bieliková: Funkcionálne a logické programovanie
- http://www.hitmill.com/programming/dotNET/csharp.html#courses
- http://www.skil.cz/python/cztutintro.html
- Daryl Harms: Začínáme programovat v jazyce Python, Comp.Press 2006
- Odkazy ze souboru Languages.mht (portal)

PROGRAMOVÉ STRUKTURY: PARADIGMATA

Historie programovacích jazyků, paradigmata programování, globální kritéria na programovací jazyk, syntaxe, sémantika, překladače, klasifikace chyb

© jstein

První počítačový program (Ada Byron)

Number of Operation	Nature of Operation	Variables acted upon	Variables receiving results	Indication of change in the value on any Variable	Statement of Results	Data ${}^1V_1$	Data ${}^1V_2$	Data ${}^1V_3$	Working Variables ${}^0V_4$	${}^0V_5$	${}^0V_6$	${}^0V_7$	${}^0V_8$	${}^0V_9$	${}^0V_{10}$	${}^0V_{11}$	${}^0V_{12}$	${}^0V_{13}\cdots$	Result Variables ${}^1V_{21}$	${}^1V_{22}$	${}^1V_{23}$	${}^0V_{24}\cdots$
						○ 0 0 1 [1]	○ 0 0 2 [2]	○ 0 0 4 [n]	○ 0 0 0	○ 0 0 0	○ 0 0 0	○ 0 0 0	○ 0 0 0	○ 0 0 0	○ 0 0 0	○ 0 0 0	○ 0 0 0	○ 0 0 0	○ B in a dec. fract. [B_1]	○ B in a dec. fract. [B_3]	○ B in a dec. fract. [B_5]	○ 0 0 0 [B_7]
1	×	${}^1V_2 \times {}^1V_3$	${}^1V_4, {}^1V_5, {}^1V_6$	$\left\{ {}^1V_2 = {}^1V_2 ; {}^1V_3 = {}^1V_3 \right\}$	$= 2n$		2	n	$2n$	$2n$	$2n$											
2	−	${}^1V_4 - {}^1V_1$	${}^2V_4$	$\left\{ {}^1V_4 = {}^2V_4 ; {}^1V_1 = {}^1V_1 \right\}$	$= 2n - 1$	1			$2n - 1$													
3	+	${}^1V_5 + {}^1V_1$	${}^2V_5$	$\left\{ {}^1V_5 = {}^2V_5 ; {}^1V_1 = {}^1V_1 \right\}$	$= 2n + 1$	1				$2n + 1$												
4	÷	${}^2V_5 \div {}^2V_4$	${}^1V_{11}$	$\left\{ {}^2V_5 = {}^0V_5 ; {}^2V_4 = {}^0V_4 \right\}$	$= \frac{2n-1}{2n+1}$				0	0						$\frac{2n-1}{2n+1}$						
5	÷	${}^1V_{11} \div {}^1V_2$	${}^2V_{11}$	$\left\{ {}^1V_{11} = {}^2V_{11} ; {}^1V_2 = {}^1V_2 \right\}$	$= \frac{1}{2} \cdot \frac{2n-1}{2n+1}$		2									$\frac{1}{2} \cdot \frac{2n-1}{2n+1}$						
6	−	${}^0V_{13} - {}^2V_{11}$	${}^1V_{13}$	$\left\{ {}^2V_{11} = {}^0V_{11} ; {}^0V_{13} = {}^1V_{13} \right\}$	$= -\frac{1}{2} \cdot \frac{2n-1}{2n+1} = A_0$											0		$= -\frac{1}{2} \cdot \frac{2n-1}{2n+1} = A_0$				
7	−	${}^1V_3 - {}^1V_1$	${}^1V_{10}$	$\left\{ {}^1V_3 = {}^1V_3 ; {}^1V_1 = {}^1V_1 \right\}$	$= n - 1 (= 3)$	1		n							$n - 1$							
8	+	${}^1V_2 + {}^0V_7$	${}^1V_7$	$\left\{ {}^1V_2 = {}^1V_2 ; {}^0V_7 = {}^1V_7 \right\}$	$= 2 + 0 = 2$		2					2										
9	÷	${}^1V_6 \div {}^1V_7$	${}^3V_{11}$	$\left\{ {}^1V_6 = {}^1V_6 ; {}^0V_{11} = {}^3V_{11} \right\}$	$= \frac{2n}{2} = A_1$						$2n$	2				$\frac{2n}{2} = A_1$						
10	×	${}^1V_{21} \times {}^3V_{11}$	${}^1V_{12}$	$\left\{ {}^1V_{21} = {}^1V_{21} ; {}^3V_{11} = {}^3V_{11} \right\}$	$= B_1 \cdot \frac{2n}{2} = B_1 A_1$											$\frac{2n}{2} = A_1$	$B_1 \cdot \frac{2n}{2} = B_1 A_1$		B_1			
11	+	${}^1V_{12} + {}^1V_{13}$	${}^2V_{13}$	$\left\{ {}^1V_{12} = {}^0V_{12} ; {}^1V_{13} = {}^2V_{13} \right\}$	$= -\frac{1}{2} \cdot \frac{2n-1}{2n+1} + B_1 \cdot \frac{2n}{2}$												0	$\left\{ -\frac{1}{2} \cdot \frac{2n-1}{2n+1} + B_1 \cdot \frac{2n}{2} \right\}$				
12	−	${}^1V_{10} - {}^1V_1$	${}^2V_{10}$	$\left\{ {}^1V_{10} = {}^2V_{10} ; {}^1V_1 = {}^1V_1 \right\}$	$= n - 2 (= 2)$	1									$n - 2$							
13	−	${}^1V_6 - {}^1V_1$	${}^2V_6$	$\left\{ {}^1V_6 = {}^2V_6 ; {}^1V_1 = {}^1V_1 \right\}$	$= 2n - 1$	1					$2n - 1$											
14	+	${}^1V_1 + {}^1V_7$	${}^2V_7$	$\left\{ {}^1V_1 = {}^1V_1 ; {}^1V_7 = {}^2V_7 \right\}$	$= 2 + 1 = 3$	1						3										
15	÷	${}^2V_6 \div {}^2V_7$	${}^1V_8$	$\left\{ {}^2V_6 = {}^2V_6 ; {}^2V_7 = {}^2V_7 \right\}$	$= \frac{2n-1}{3}$						$2n - 1$	3	$\frac{2n-1}{3}$									
16	×	${}^1V_8 \times {}^3V_{11}$	${}^4V_{11}$	$\left\{ {}^1V_8 = {}^0V_8 ; {}^3V_{11} = {}^4V_{11} \right\}$	$= \frac{2n}{2} \cdot \frac{2n-1}{3}$								0			$\frac{2n}{2} \cdot \frac{2n-1}{3}$						
17	−	${}^2V_6 - {}^1V_1$	${}^3V_6$	$\left\{ {}^2V_6 = {}^3V_6 ; {}^1V_1 = {}^1V_1 \right\}$	$= 2n - 2$	1					$2n - 2$											
18	+	${}^1V_1 + {}^2V_7$	${}^3V_7$	$\left\{ {}^2V_7 = {}^3V_7 ; {}^1V_1 = {}^1V_1 \right\}$	$= 3 + 1 = 4$	1						4										
19	÷	${}^3V_6 \div {}^3V_7$	${}^1V_9$	$\left\{ {}^3V_6 = {}^3V_6 ; {}^3V_7 = {}^3V_7 \right\}$	$= \frac{2n-2}{4}$						$2n - 2$	4		$\frac{2n-2}{4}$								
20	×	${}^1V_9 \times {}^4V_{11}$	${}^5V_{11}$	$\left\{ {}^1V_9 = {}^0V_9 ; {}^4V_{11} = {}^5V_{11} \right\}$	$= \frac{2n}{2} \cdot \frac{2n-1}{3} \cdot \frac{2n-2}{4} = A_3$									0		$\left\{ \frac{2n}{2} \cdot \frac{2n-1}{3} \cdot \frac{2n-2}{4} \right\} = A_3$						
21	×	${}^1V_{22} \times {}^5V_{11}$	${}^0V_{12}$	$\left\{ {}^1V_{22} = {}^1V_{22} ; {}^0V_{12} = {}^2V_{12} \right\}$	$= B_3 \cdot \frac{2n}{2} \cdot \frac{2n-1}{3} \cdot \frac{2n-2}{4} = B_3 A_3$											0	$B_3 A_3$			B_3		
22	+	${}^2V_{12} + {}^2V_{13}$	${}^3V_{13}$	$\left\{ {}^2V_{12} = {}^0V_{12} ; {}^2V_{13} = {}^3V_{13} \right\}$	$= A_0 + B_1 A_1 + B_3 A_3$												0	$\{A_0 + B_1 A_1 + B_3 A_3\}$				
23	−	${}^2V_{10} - {}^1V_1$	${}^3V_{10}$	$\left\{ {}^2V_{10} = {}^3V_{10} ; {}^1V_1 = {}^1V_1 \right\}$	$= n - 3 (= 1)$	1									$n - 3$							
Here follows a repetition of Operations thirteen to twenty-three																						
24	+	${}^4V_{13} + {}^0V_{24}$	${}^1V_{24}$	$\left\{ {}^4V_{13} = {}^0V_{13} ; {}^0V_{24} = {}^1V_{24} \right\}$	$= B_7$																	B_7
25	+	${}^1V_1 + {}^1V_3$	${}^1V_3$	$\left\{ {}^1V_1 = {}^1V_1 ; {}^1V_3 = {}^1V_3 \right\}$ ${}^5V_6 = {}^0V_6$; ${}^5V_7 = {}^0V_7$	$= n + 1 = 4 + 1 = 5$ by a Variable-card. by a Variable-card.	1		$n + 1$			0	0										

Historie programovacích jazyků

- Konec 40. let
  - Odklon od strojových kódů
  - Pseudokódy:
    - Pseudooperace aritm. a matem. funkcí
    - Podmíněné a nepodmíněné skoky
    - Autoinkrement. registry pro přístup k polím

Historie programovacích jazyků (2)

- 50. léta
  - První definice vyššího programovacího jazyka (efektivita návrhu programu)
  - FORTRAN (formula translation - Backus), vědeckotechnické výpočty, komplexní výpočty na jednoduchých datech, pole, cykly, podmínky
  - COBOL (common business lang.), jednoduché výpočty, velká množství dat, záznamy, soubory, formátování výstupů
  - ALGOL (algorithmic language - Backus, Naur), předek všech imperativních jazyků, bloková struktura, rekurze, volání param. hodnotou, deklarace typů
  - Nové idee - strukturování na podprogramy, přístup ke globálním datům (Fortan), bloková struktura, soubory, …
  - Stále živé - Fortran90, Cobol

Historie programovacích jazyků (3)

- První polovina 60. let
  - Začátek rozvoje neimperativních jazyků
  - LISP (McCarthy) - založen na teorii rekurzivních funkcí, první funkcionální jazyk, použití v UI (symbolické manipulace)
  - APL - manipulace s vektory a s maticemi
  - SNOBOL (Griswold) - manipulace s řetězci a vyhledávání v textech, podporuje deklarativní programování
  - Vzniká
    - potřeba dynamického ovládání zdrojů,
    - potřeba symbolických výpočtů

Historie programovacích jazyků (4)

- Pozdní 60. léta
  - IBM snaha integrovat úspěšné koncepty všech jazyků - vznik PL/1 (moduly, bloky, dynamické struktury)
  - Nové prvky PL/1 - zpracování výjimek, multitasking. Nedostatečná jednotnost konstrukcí, komplikovanost
  - ALGOL68 - ortogonální konstrukce, první jazyk s formální specifikací (VDL), uživatelsky málo přívětivý, typ reference, dynamická pole
  - SIMULA67 (Nygaard, Dahl) - zavádí pojem tříd, hierarchie ke strukturování dat a procedur, ovlivnila všechny moderní jazyky, corutiny
  - BASIC (Kemeny) - žádné nové konstrukce, určen začátečníkům, obliba pro efektivnost a jednoduchost, interaktivní styl programování (naivní paradigma)
  - Pascal (Wirth) - k výuce strukturovaného programování, jednoduchost a použitelnost na PC zaručily úspěch

Historie programovacích jazyků (5)

- 70. léta
  - Důraz na bezpečnost a spolehlivost
  - Ustálení základních paradigmat
  - Abstraktní datové typy, moduly, typování, práce s výjimkami
  - CLU (datové abstrakce), Mesa (rozšíření Pascalu o moduly), Concurrent Pascal, Euclid (rošíření Pascalu o abstraktní datové typy)
  - C (Ritchie) - efektivní pro systémové programování, efektivní implementace na různých počítačích, slabé typování
  - Scheme - rozšířený dialekt LISPu
  - PROLOG (Colmeraurer) - první logicky orientovaný jazyk, používaný v UI a znalostních systémech, neprocedurální, „inteligentní DBS odvozující pravdivost dotazu“

Historie programovacích jazyků (6)

- 80. léta
  - Modula2 (Wirth) - specifické konstrukce pro modulární programování
  - Další rozvoj funkcionálních jazyků - Scheme (Sussman, Steele, MIT), Miranda (Turner), ML (Milner) - typová kontrola
  - ADA (US DOD) syntéza vlastností všech konvenčních jazyků, moduly, procesy, zpracování výjimek
  - Průlom objektově orientovaného programování - Smalltalk (Key, Ingalls, Xerox: Datová abstrakce, dědičnost, dynamická vazba typů), C++ (Stroustrup 85- C a Simula)
  - Další OO jazyky - Eiffel (Mayer), Modula3, Oberon (Wirth)
  - OPS5, CLIPS - pro zpracování znalostí

Historie programovacích jazyků (7)

- 90. léta
  - Jazyky 4.generace, QBE, SQL - databázové jazyky
  - Java (SUN) - mobilita kódu na webu, nezávislost na platformě
  - Vizuální programování (programování ve windows) - Delphi, Visual Basic, Visual C++
  - Skriptovací jazyky
    - Perl (Larry Wall - Pathologically Eclectic Rubbish Lister)
      - Nástroj pro webmastery a administrátory systémů
    - JavaScript - podporován v Netscape i v Explorer,
    - VBScript,
    - PHP, Python
  - 2000 C Sharp

Generace programovacích jazyků

- První generace - strojový kód: 0 a 1
  - První počítače: přepínače, nikoliv text
  - Absolutní výkon
  - Závislý na hardware
  - Příklad: 10110000 01100001
- Druhá generace – Assembler (assembly language)
  - Závislý na hardware
  - Příklad: mov al, 61h
- Třetí generace – čitelný a snadno zapsatelný lidmi
  - Většina moderních jazyků
  - Příklad: let b = c + 2 * d
- Čtvrtá generace – reportovací nástroje, SQL (structured query language), domain-specific languages
  - Navržené pro konkrétní účel
  - Příklad: SELECT * FROM employees ORDER BY surname
- Pátá generace – synonymum pro vizuální programování nebo označení vývoje pomocí definic omezení
  - stroj sám má zkonstruovat algoritmus.

Paradigmata programování

- **Paradigma**

  = Z řečtiny – vzor, příklad, model – určitý vzor vztahů či vzorec myšlení

  = Souhrn způsobů formulace problémů, metodologických prostředků řešení,metodik, zpracování a pod.
- **Programovací paradigma**
  - Výchozí imaginární schematizace úloh
  - Soubor náhledů na obor informační/výpočetní problematiky
  - Soubor přístupů k řešení specifických úloh daného oboru
  - Soustava pravidel, standardů a požadavků na programovací jazyky
  - Jednotlivá paradigmata mohou zřetelně ulehčit práci v rámci svého určení a komplikovat nebo úplně odsunout neohniskové úlohy do zcela nekompatibilních dimenzí (tak, že např. určité jazyky nelze použít k výpočtům nebo jiné k interakci s uživatelem).

Paradigmata programování (2)

- Procedurální (imperativní) programování
- Objektově orientované programování
- Generické programování
- Komponentově orientované programování
- Deklarativní programování
  - Funkcionální programování
  - Logické programování
  - Programování ohraničeními (constraint prog.)
- Událostní programování (event-driven prog.)
- Vizuální programování
- Aspektově orientované programování
- Souběžné programování
  - Paralelní
  - Distribuované

Procedurální (imperativní)

- Imperativní přístup je blízký i obyčejnému člověku (jako kuchařka)
- Popisuje výpočet pomocí posloupnosti příkazů a určuje přesný postup (algoritmus), jak danou úlohu řešit.
- Program je sadou proměnných, jež v závislosti na vyhodnocení podmínek mění pomocí příkazů svůj stav.
- Základní metodou imperativního programování je procedurální programování, tyto termíny bývají proto často zaměňovány.
- Strukturované
- Modulární

Objektově orientované programování

- Imperativní programování – problém znovupoužitelnosti, modifikace řešení, pokud nalezneme lepší
- Program je množina objektů
- Objekty mají stav, jméno, chování
- Předávají si zprávy
- Zapouzdřenost – data a metody
- Dědičnost
- Polymorfismus

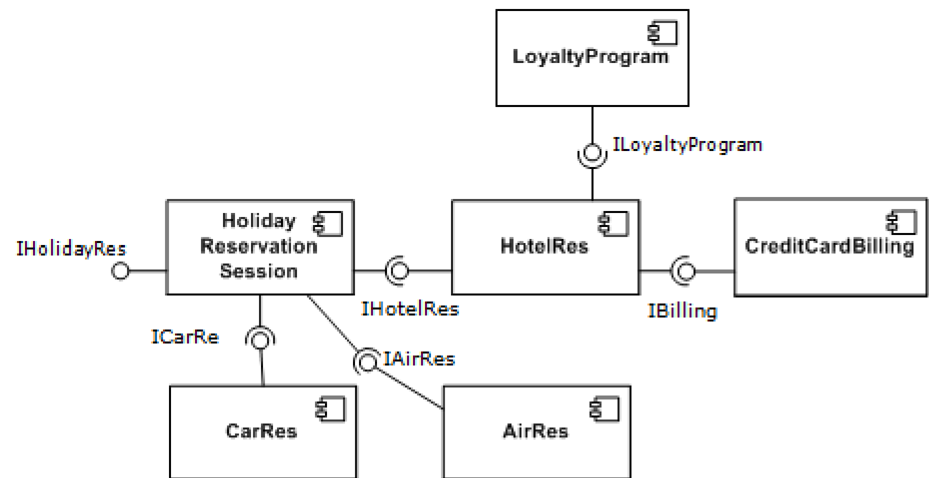
Generické programování

- Rozdělení kódu programu na algoritmus a datové typy takovým způsobem, aby bylo možné zápis kódu algoritmu chápat jako obecný, bez ohledu nad jakými datovými typy pracuje. Konkrétní kód algoritmu se z něj stává dosazením datového typu.
- U kompilovaných jazyků dochází k rozvinutí kódu v době překladu. Typickým příkladem jazyka, který podporuje tuto formu generického programování je jazyk C++. Mechanismem, který zde generické programování umožňuje, jsou takzvané *šablony (templates)*.

```
template<class T> class Stack {
        private: T* items; int stackPointer;
        public:
                Stack(int max = 100) {items = new T[max];
                stackPointer = 0; }
                void Push(T x) { items[stackPointer++] = x; }
                T Pop() { return items[--stackPointer]; }
};
```

Komponentově orientované programování

- Souvisí s komponentově orientovaným softwarovým inženýrstvím
- Využívání prefabrikovaných komponent
  - Násobná použitelnost
  - Nezávislost na kontextu
  - Slučitelnost s ostatními komponentami
  - Zapouzdřitelnost



- Softwarová komponenta
  - Jednotka systému, která nabízí předdefinovanou službu a je schopna kor
  - Jednotka nezávisle nasaditelná a verzovatelná
- Rozdíly oproti OOP
  - OOP – software modeluje reálný svět – objekty zastupují podstatná jména a slovesa
  - Component-based – slepování prefabrikovaných komponent
  - Někteří je slučují a tvrdí, že pouze popisují problém z jiného pohledu

Deklarativní programování

- Kontrast s imperativním programováním
- Založeno na popisu cíle – přesný algoritmus provedení specifikuje až interpret příslušného jazyka a programátor se jím nezabývá.
- Díky tomu lze ušetřit mnoho chyb vznikajících zejména tím, že do jedné globální proměnné zapisuje najednou mnoho metod.
- K předání hodnot slouží většinou návratové hodnoty funkcí.
- Nemožnost program široce a přesně optimalizovat takovým způsobem, jaký právě potřebuje.
- Navíc při deklarativním přístupu je velmi často využíváno rekurze, což klade vyšší nároky na programátora.

Programování ohraničeními

- Constraint programming
- Vyjadřují výpočet pomocí relací mezi proměnnými
- Např. celsius = (fahr -32)*5/9 --definuje relaci, není to přiřazení
- Forma deklarativního programování
- Často kombinace constraint logic programming
- Modeluje svět, ve kterém velké množství ohraničení (omezení, podmínek) je splněno v jednu chvíli, svět obsahuje řadu neznámých proměnných, úkolem programu je najít jejich hodnoty
- Doména proměnných :: jakých hodnot můžou nabývat

Programování ohraničeními (2)

- Hádanka SEND+MORE=MONEY

```
sendmore(Digits) :-
   Digits = [S,E,N,D,M,O,R,Y], % Create variables
   Digits :: [0..9], % Associate domains to variables
   S #\= 0, % Constraint: S must be different from 0
   M #\= 0,
   alldifferent(Digits), % all the elements must take different values
   1000*S + 100*E + 10*N + D
   + 1000*M + 100*O + 10*R + E
   #= 10000*M + 1000*O + 100*N + 10*E + Y, % Other constraints
   labeling(Digits). % Start the search
```

Logické programování

- Použití matematické logiky v programování
- Rozděluje řešení problému je rozděleno mezi
  - Programátora – zodpovědný jen za pravdivost programu vyjádřeném logickými formulemi
  - Generátora modelu (řešení) – zodpovědný za efektivní vyřešení problému (nalezení řešení)
- Využití v umělé inteligenci, zpracování přirozené řeči, expertních systémech
- Významný zástupce – PROLOG

```
nsd(U, V, U) :- V = 0 .
nsd(U, V, X) :- not(V = 0),
                Y is U mod V,
                nsd(V, Y, X).
```

Funkcionální programování

- Výpočet řízen vyhodnocováním matematickým funkcí
- Funkcionální vs. Imperativní – aplikace funkcí vs. změna stavu (funkce mohou mít vedlejší efekty)
- Významný zástupce – LISP
- V minulosti spjat s umělou inteligencí
- Dialekty se používají v Emacs editoru, AutoCADu

```
(defun nsd (u v)
  (if (= v 0) u
      (nsd v (mod u v))))
```

Událostní programování

- Tok programu je určen akcemi uživatele nebo zprávami z jiných programů

Př. verze sečtení čísel dávkově

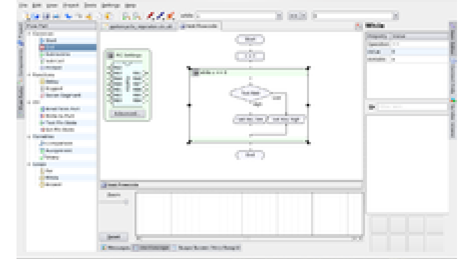
```
read a number (from the keyboard) and store it in variable A[0]
read a number (from the keyboard) and store it in variable A[1]
print A[0]+A[1]
```

Př.event-driven verze téhož

```
set counter K to 0
repeat {
   if a number has been entered (from the keyboard)
         { store in A[K] and increment K
           if K equals 2 print A[0]+A[1] and reset K to 0
         }
}
```

Vizuální programování

- Specifikuje program interaktivně pomocí grafických prvků (ikon, formulářů), např. LabVIEW. Microsoft Vis.St. (VB, VC#,..) nejsou vizuální jazyky, ale textové.
- „boxes and arrows“
  - Boxy reprezentují entity a šipky relace
  - Icon-based, form-based, diagram
- Programování datovým tokem
  - Možnost automatické paralelizace



Aspektově orientované programování

- Motivace:

```
void transfer(Account fromAccount, Account toAccount, int
   amount) {
   if (fromAccount.getBalance() < amount) {
        throw new InsufficientFundsException();
   }
   fromAccount.withdraw(amount);
   toAccount.deposit(amount);
}
```

- Takový transfer peněz má vady (nezkouší autorizaci, není transakční)
- Pokus ošetřit vady rozptýlí kontroly přes celý program (obv. metody, moduly)

Aspektově orientované programování (2)


```
if (!getCurrentUser().canPerform(OP_TRANSFER)) {
        throw new SecurityException();
}
if (amount < 0) {
   throw new NegativeTransferException();
}
if (fromAccount.getBalance() < amount) {
   throw new InsufficientFundsException();
}
Transaction tx = database.newTransaction();
try {
   fromAccount.withdraw(amount);
   toAccount.deposit(amount); tx.commit();
   systemLog.logOperation(OP_TRANSFER, fromAccount, toAccount, amount);
}
catch(Exception e) { tx.rollback();
}
```

Aspektově orientované programování (3)

- AOP se snaží řešit problém modularizováním těchto záležitostí do **aspektů**, tj. separovaných částí kódu (modulů), ze kterých se tyto záležitosti pohodlněji spravují.
- Aspekty obsahují **pokyn** (advice) – kód připojený ve specifikovaných bodech programu a **vnitřní deklarace typů** (inner-type declarations) – strukturální prvky přidané do jiných tříd
- Př. Bezpečnostní modul může obsahovat pokyn, který vykoná kontrolu bezpečnosti před přístupem na bankovní účet
- **Bod řezu** (pointcut) definuje **body připojení** (join points), tj. místa, kde dochází k přístupu k účtu, a kód v těle pokynu definuje, jak je bezpečnostní kontrola implementována
- Jak samotná kontrola, tak i to, kdy se vykoná, je udržováno v jednom místě
- Navíc, pokud je bod řezu dobře navržen, lze předvídat budoucí změny v programu

Aspektově orientované programování (4)

- AspectJ – rozšíření Javy o AOP
- Body napojení obsahují volání funkcí, inicializaci objektů atd., neobsahují např. cykly
- Body řezu
  - Např. `execution(* set*(*))` zabere, pokud spouštíme metodu, jejíž jméno začíná na `set` a má jeden parametr jakéhokoliv typu
  - Složitější: `pointcut set() : execution(* set*(*) ) && this(Point) && within(com.company.*);`
    - Zabere, pokud spouštíme metodu, jejíž jméno začíná na set a `this` je instance třídy Point v balíku `com.company`
    - Na tento bod řezu se potom můžeme odkazovat jako `set()`
- Pokyn popisuje, kdy (např. `after`), kde (body napojení určené bodem řezu) a jaký kód se má spustit

```
after() : set() {Display.update();}
```

- Vnitřní deklarace typu
  - Přidání metody acceptVisitor do třídy Point

```
Aspect DisplayUpdate {void Point.acceptVisitor(Visitor v) {v.visit(this);}}
```

Souběžné programování

- Program je navržen jako kolekce interagujících procesů
- Paralelní x distribuované
- Jeden procesor x více procesorů
- Komunikace: Sdílená paměť x předávání zpráv
- Procesy operačního systému x množina vláken (jeden proces OS)
- Souběžné programovací jazyky používají jazykové konstrukce
  - Multi-threading
  - Podpora distribuovaného zpracování
  - Předávání zpráv
  - Sdílené zdrojů

Globální kriteria na programovací jazyk

1. Spolehlivost
2. Efektivita – překladu, výpočtu
3. Strojová nezávislost
4. Čitelnost a vyjadřovací schopnosti
5. Řádně definovaná syntax a sémantika
6. Úplnost v Turingově smyslu

Spolehlivost

- Typová kontrola (type system)
  - Typované x netypované jazyky
    - Typované – specifikace každé operace definuje typy dat, na které je aplikovatelná
      - Např. chyba při dělení čísla řetězcem (při kompilaci/runtime)
      - Speciální případ – jazyky s jedním typem (např. SGML)
    - Netypované – všechny operace nad jakýmikoliv daty
      - Sekvence bitů – Assembler
  - Statické/dynamické typování
    - Statické – všechny výrazy mají určený typ před spuštěním programu
      - Programátor musí explicitně určit typy (manifestně typované) – Java x kompilátor odvozuje typ výrazů (odvozeně typované) – ML
    - Dynamické – určuje typovou bezpečnost při běhu programu
      - Bez explicitní deklarace typů
      - Proměnná může obsahovat hodnotu různých typů v různých částech programu
      - Složitější ladění – chyba typu nemůže být odhalena dříve než při spuštění chybného příkazu
      - Lisp, JS, Python, PHP

Spolehlivost (2)

- Slabé x silné typování
  - Slabé – dovoluje nakládat s jedním typem jako jiným (např. nakládat s řetězcem jako s číslem)
    - Někdy se hodí, problém odhalení chyb
  - Silné – zakazuje nakládat s jedním typem jako jiným (typově bezpečné – type-safe)
    - Operace s jiným typem => chyba
  - Varianta slabého typování – velké množství implicitních konverzí typů (C++, JS, Perl)
- Zpracování výjimečných situací

Čitelnost a vyjadřovací schopnosti

- Jednoduchost (vs. př.: C=C+1; C+=1; C++; ++C)
- Ortogonalita (malá množina primitivních konstrukcí, z té lze kombinovat další konstrukce. Všechny kombinace jsou legální)
  - vs. př. v C: struktury mohou být funkční hodnotou, ale pole nemohou
- Strukturované příkazy
- Strukturované datové typy
- Podpora abstrakčních prostředků
- Strojová čitelnost

  = existence algoritmu překladu s lineární časovou složitostí

  = bezkontextová syntax
- Humánní čitelnost – silně závisí na způsobu abstrakcí
  - abstrakce dat
  - abstrakce řízení
- Čitelnost vs. jednoduchost zápisu

Řádně definovaná syntax a sémantika

- Syntax = forma či struktura výrazů, příkazů a programových jednotek
- Sémantika = význam výrazů, příkazů a programových jednotek
- Definici jazyka potřebují
  - návrháři jazyka
  - Implementátoři
  - uživatelé

Úplnost v Turingově smyslu

- Turingův stroj = jednoduchý ale neefektivní počítač použitelný jako formální prostředek k popisu algoritmu
- Programovací jazyk je úplný v Turingově smyslu, jestliže je schopný popsat libovolný výpočet (algoritmus)
- Co je potřebné pro Turingovu úplnost?

  Téměř nic: Stačí

  - ✓ celočíselná aritmetika a
  - ✓ celočíselné proměnné spolu se sekvenčně prováděnými příkazy zahrnujícími
  - ✓ přiřazení a
  - ✓ cyklus (While)

Syntax

- Formálně je jazyk množinou vět
- Věta je řetězcem lexémů (terminálních symbolů)
- Syntax lze popsat:
  - Rozpoznávacím mechanismem – automatem (užívá jej překladač)
  - Generačním mechanismem – gramatikou (to probereme)
- Formální gramatika
  - Prostředek pro popis jazyka
  - Čtveřice (N, T, P, S) – Neterminální symboly, Terminální symboly, Přepisovací pravidla, Startovací symbol
- Bezkontextová gramatika
  - Všechna pravidla mají tvar neterminál -> řetězec terminálů/neterminálů
  - Přepis bez ohledu na okolní kontext

Syntax (2)

- Backus Naurova forma (BNF)
  - Metajazyk používaný k vyjádření bezkontextové gramatiky

```
<program> → <seznam deklaraci> ; <prikazy>
<seznam deklaraci> →
           <deklarace> |
           <deklarace>;<seznam deklaraci>
<deklarace> → <spec. typu> <sez. promennych>
```

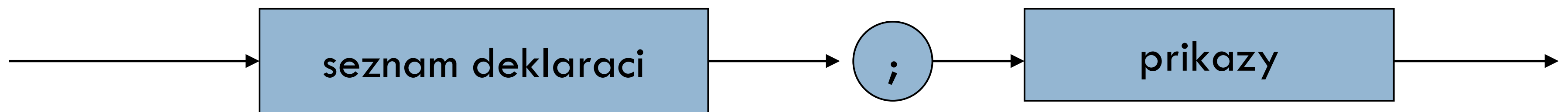
Syntax (3)

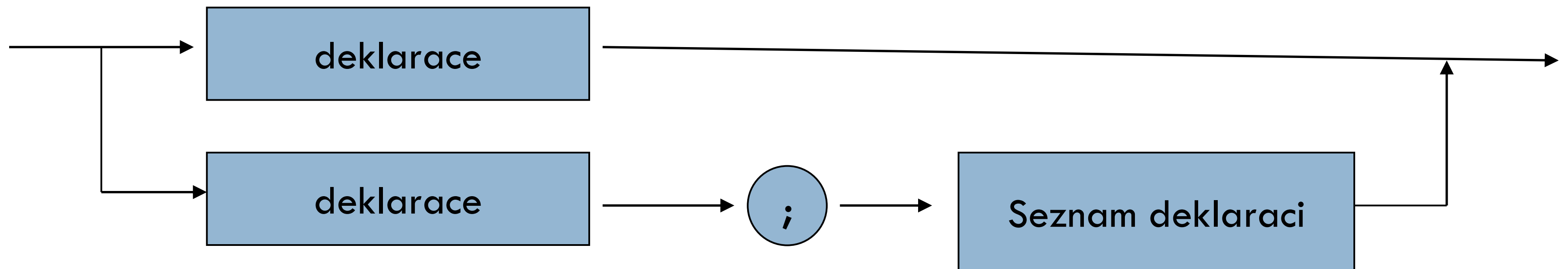
- BNF sama o sobě se dá specifikovat pomocí pravidla BNF následujícím způsobem

```
<syntax> ::= <rule> | <rule> <syntax>
<rule> ::= <opt-whitespace> "<" <rule-name> ">"
<opt-whitespace> "::=" <opt-whitespace> <expression>
  <line-end>
<opt-whitespace> ::= " " <opt-whitespace> | ""
<expression> ::= <list> | <list> "|" <expression>
<line-end> ::= <opt-whitespace> <EOL> | <line-end> <line-
  end>
<list> ::= <term> | <term> <opt-whitespace> <list>
<term> ::= <literal> | "<" <rule-name> ">"
<literal> ::= '"' <text> '"' | "'" <text> "'"
```

Syntax (4)

- Syntaktické diagramy





Syntax – derivace, derivační strom

- Proces lexikální analýzy (parsing) = proces analýzy posloupnosti formálních prvků s cílem určit jejich gramatickou strukturu vůči předem dané formální gramatice
- na vstupu je řetězec, máme vytvořit posloupnost pravidel
- Aplikace pravidla = derivace
- Postupně vznikne strom – derivační (syntaktický) strom (příklad)
- Pokud nahrazujeme vždy nejlevější neterminál – levá derivace
- Rozdíl mezi levou a pravou derivací je důležitý, protože ve většině parsovacích transformací vstupu je definován kus kódu pro každé pravidlo gramatiky. Proto je důležité při parsování se rozhodnout, zda-li zvolit levou nebo pravou derivaci, protože ve stejném pořadí se budou provádět části programu.

Sémantika

- Studuje a popisuje význam výrazů/programů
- Aplikace matematické logiky
- Statická sémantika – v době překladu
- Dynamická sémantika – v době běhu
- Jiná syntax, ale stejné sémantika:
  - x += y (C, Java, atd.)
  - x := x + y (Pascal)
  - Let x = x + y (BASIC)
  - x = x + y (Fortran)
  - Sémantika: aritmetické přičtení hodnoty y k hodnotě x a uložení výsledku do proměnné nazvané x

Metody popisu sémantiky

- Slovní popis nepřesný
- Formální popis:
  - Operační sémantiky
    - Vyjádření významu programu posloupností přechodů mezi stavy
    - Příkaz aktualizace paměti: Pokud se výraz E ve stavu s redukuje na hodnotu V potom program L:=E zaktualizuje stav s přiřazením L=V

$$\frac{\langle E, s\rangle \Rightarrow V}{\langle L := E, s\rangle \rightarrow \left(s \overset{+}{\cup} (L \mapsto V) \right)}$$

  - Denotační sémantiky
    - Vyjádření významu programu funkcemi
  - Axiomatické sémantiky
    - Vyjádření významu programu tvrzeními (např. predikátová logika)

Překlad jazyka

- Kompilátor: dvoukrokový proces překládá zdrojový kód do cílového kódu. Následně uživatel sestaví a spustí cílový kód
- Interpret: jednokrokový proces, „zdrojový kód je rovnou prováděn"
- Hybridní: např. Java Byte-code – soubory *.class

Klasifikace chyb

- Lexikální – např. nedovolený znak
- Syntaktické – chyba ve struktuře
- Statické sémantiky – např. nedefinovaná proměnná, chyba v typech. Způsobené kontextovými vztahy. Nejsou syntaktické.
- Dynamické sémantiky – např. dělení 0. Nastávají při výpočtu, neodhalitelné při překladu. Nejsou syntaktické.
- Logické – chyba v algoritmu

- Kompilátor schopen nalézt lexikální a syntaktické při překladu
- Chyby statické sémantiky může nalézt až před výpočtem
- Nemůže při překladu nalézt chyby v dynamické sémantice, projeví se až při výpočtu
- Žádný překladač nemůže hlásit logické chyby
- Interpret obvykle hlásí jen lexikální a syntaktické chyby když zavádí program

PROGRAMOVÉ STRUKTURY: PARALELNÍ PROGRAMOVÁNÍ

Amdahlův zákon, jazykové konstrukce pro paralelní výpočty, paralelismus

21. Února 2009

© jstein

Amdahlův zákon

- Když se něco dělá paralelně, může se získat výsledek rychleji, ale ne vždy (1 žena porodí dítě za 9 měsíců, ale 9 žen neporodí dítě za 1 měsíc)
- **Amdahlův Zákon**
  - Určuje urychlení výpočtu při užití více procesorů
  - Urychlení je limitováno sekvenčními částmi výpočtu

Obecně: $$\frac{1}{\sum_{k=0..n} \frac{P_k}{S_k}}$$

P_k % instrukcí, které lze urychlit
S_k multiplikátor urychlení
n počet různých úseků programu
k je index úseku

- Je-li v programu jeden paralelizovatelý úsek s P % kódu, pak

$$\frac{1}{(1-P)+\left(P/S\right)} = \frac{1}{(1-P)/1 + P/S}$$

Amdahlův zákon (2)

Př. Paralelizovatelný úsek zabírá 60% kódu a lze jej urychlit 100 krát. ⇒
celkové urychlení je 1/((1-0,6)+(0,6/100)) = 1/(0.4 + 0,006) ≈ **2.5**

Př. 50 % kódu lze urychlit libovolně ⇒
celkové urychlení je 1/(1 – 0.5) = **2**

Př. úseky

P1 = 11 %,	P2 = 48 %	P3 = 23 %	P4 = 18 %
S1 = 1	S2 = 1,6	S3 = 20	S4 = 5

Urychlení je 1 / (0,11/1 + 0,48/1,6 + 0,23/20 + 0,18/5) ≈ **2,19**

Kromě urychlení zajišťují paralelní konstrukce také spolupráci výpočtů

Paralelismus

- Paralelismus se vyskytuje na:
  - Úrovni strojových instrukcí – je záležitostí hardware
  - Úrovni příkazů programovacího jazyka – toho si všimneme
  - Úrovni podprogramů – to probereme
  - Úrovni programů – je záležitostí Operačního Systému
- Vývoj multiprocesorových architektur:
  - konec 50. let – jeden základní procesor a jeden či více speciálních procesorů pro I/O
  - polovina 60. – víceprocesorové systémy užívané pro paralelní zpracování na úrovni programů
  - konec 60. – víceprocesorové systémy užívané pro paralelní zpracování na instrukční úrovni
- Druhy počítačových architektur pro paralelní výpočty:
  - SIMD architektury
    - stejná instrukce současně zpracovávaná na více procesorech
    - na každém s jinými daty
    - vektorové procesory
  - MIMD architektury
    - nezávisle pracující procesory, které mohou být synchronizovány

Paralelismus (2)

- Někdy se rozlišuje
  - Parallel programming = cílem je urychlení výpočtu
  - Concurrent progr.= cílem je správná spolupráce programů
- Paralelismus implicitní (zajistí překladač) nebo **explicitní** (zařizuje programátor)
- Realizace buď konstrukcemi jazyka nebo knihovny (u tradičních jazyků)

Paralelismus na úrovni podprogramů

- Sekvenční výpočetní proces je v čase uspořádaná posloupnost operací =vlákno.
- Definice vláken a procesů se různí
- Obvykle proces obsahuje 1 či více vláken a vlákna uvnitř jednoho procesu sdílí zdroje, zatímco různé procesy ne.
- Paralelní procesy jsou vykonávány paralelně či pseudoparalelně (pseudo=nepravý)
- Kategorie paralelismu
  - Fyzický paralelismus (má více procesorů pro více procesů)
  - Logický paralelismus (time-sharing jednoho procesoru, v programu je více procesů)
  - Kvaziparalelismus (kvazi=zdánlivě, př. korutiny v některých jazycích)
    - Korutiny – speciální druh podprogramů, kdy volající a volaný jsou si rovni (symetrie), mají více vstupních bodů a zachovávají svůj stav mezi aktivacemi.
- Paralelně prováděné podprogramy musí nějak komunikovat
  - Přes sdílenou paměť (Java, C#), musí se zamykat přístup k paměti
  - Předáváním zpráv (Occam, Ada), vyžaduje potvrzení o přijetí zprávy

Problémy paralelního zpracování

- Nové problémy
  - rychlostní závislost
  - uvíznutí (vzájemné neuvolnění prostředků pro jiného),
  - vyhladovění (obdržení příliš krátkého času k zajištění progresu),
  - livelock (obdoba uvíznutí, ale nejsou blokovány čekáním, zaměstnávají se navzájem (after you - after you efekt)

Př. Z konta si vybírá SIPO 500,-Kč a obchodní dům 200,-Kč

Ad sériové zpracování

Zjištění stavu konta - Odečtení 500 - Uložení nového stavu - Převod 500 na konto SIPO

...

Zjištění stavu konta - Odečtení 200 - Uložení nového stavu - Převedení 200 na konto obch. domu

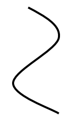
...

Výsledek bude OK

Rychlostní závislost

- Ad paralelní zpracování dvěma procesy

Zjištění stavu konta
Odečtení 500
Uložení nového stavu
Převod 500 na konto SIPO



Zjištění stavu konta
Odečtení 200
Uložení nového stavu
Převedení 200 na konto obch.domu

- Pokud výpočet neošetříme, může vlivem různých rychlostí být výsledný stav konta: (Původní stav -500) nebo (Původní stav -200) nebo (Původní stav - 700)
- Operace výběrů z konta ale i vložení na konto musí být prováděny ve vzájemném vyloučení. Jsou to tzv. **kritické sekce programu**
- Jak to řešit?
  - 1. řešení: Semafor = obdobnou funkci jako klíč od WC nebo návěstidlo železnice (jen jeden může do sdíleného místa).
  - Operace: zaber(semafor) a uvolni(semafor)

Semafory

Proces A		Proces B
Zaber(semafor S)		Zaber(semafor S)
Zjištění stavu konta	odtud	Zjištění stavu konta
Odečtení 500	jsou	Odečtení 200
Uložení nového stavu	kritické	Uložení nového stavu
Převod 500 na konto SIPO	sekce až sem	Převedení 200 na k. OD
Uvolni(semafor S)		Uvolni(semafor S)

- Výsledný stav konta bude (Původní – 700)
- Nebezpečnost semaforů:
  - Opomenutí semaforové operace (tj. ochránění krit. sekce)
  - Možnost skoku do kritické sekce
  - Možnost vzniku deadlocku (viz další), pokud semafor neuvolníme

Uvíznutí (deadlock)

Př. Procesy A, B oba pracují s konty (soubory, obecně zdroji) Z1 a Z2. K vyloučení vzniku nedeterminismu výpočtu, musí požádat o výlučný přístup (např. pomocí semaforů)

Pokud to provedou takto:

Proces A	Proces B
...	...
Zaber(semafor S1)	Zaber(semafor S2)
Zaber(semafor S2)	Zaber(semafor S1)
...	...

- Bude docházet k deadlocku. Každý z procesů bude mít jeden ze zdrojů, potřebuje ale oba zdroje. Oba procesy se zastaví.
- Jak tomu zabránit? (např. tzv. bankéřův algoritmus nebo přidělování zdrojů v uspořádání = pokud nemáš S1, nemůžeš žádat S2, …)

Monitor

- 2. řešení Monitor
- Monitor je modul (v OOP objekt), nad jehož daty mohou být prováděny pouze v něm definované operace.
- Provádí-li jeden z procesů některou monitorovou operaci, musí se ostatní procesy postavit do fronty, pokud rovněž chtějí provést některou monitorovou operaci .
- Ve frontě čekají, dokud se monitor neuvolní a přijde na ně řada.

Př. Typ monitor konto -data: stav_ konta

-operace: vyber(kolik), vlož(kolik)

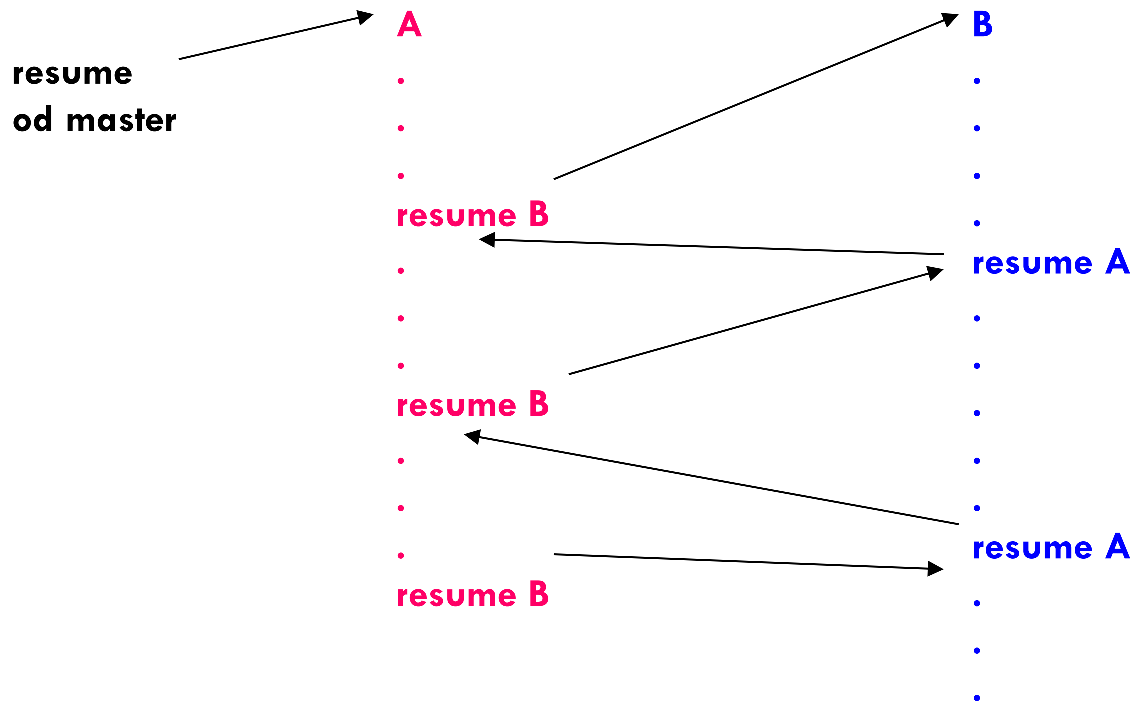
Instance: Mé_konto, SIPO_konto, Obchodům_konto

Proces A	Proces B
Mé_konto.vyber(500)	Mé_konto.vyber(200)
SIPO_konto.vlož(500)	Obchodům_konto(vlož(200)

- Pozn. V Javě jsou monitorem objekty, které jsou instancí třídy se synchronized metodami (viz později)

Korutiny (kvaziparalelismus)

- Kvaziparalelní prostředek jazyků Modula, Simula, Interlisp
- Představte si je jako podprogramy, které při opětovném vyvolání se nespustí od začátku, ale od místa ve kterém příkazem resume předaly řízení druhému



Korutiny (kvaziparalelismus) (2)

- Speciální druh podprogramů – volající a volaný nejsou v relaci „master-slave“
- Jsou si rovni (symetričtí)
  - Mají více vstupních bodů
  - Zachovávají svůj stav mezi aktivacemi
  - V daném okamžiku je prováděna jen jedna
- Master (není korutinou) vytvoří deklarací korutiny, ty provedou inicializační kód a vrátí mastru řízení.
- Master příkazem resume spustí jednu z korutin
- Příkaz resume slouží pro start i pro restart
- Pakliže jedna z korutin dojde na konec svého programu, předá řízení mastru

Paralelismus na úrovni podprogramů

- Procesy mohou být
  - nekomunikující (neví o ostatních, navzájem si nepřekáží)
  - komunikující (např. producent a konzument)
  - soutěžící (např. o sdílený prostředek)
- V jazycích nazývány různě:
  - **Vlákno výpočtu** v programu (thread Java, C#, Python) je sekvence míst programu, kterými výpočet prochází.
  - **Úkol** (task Ada) je programová jednotka (část programu), která může být prováděna paralelně s ostatními částmi programu. Každý úkol může představovat jedno vlákno.
- Odlišnost vláken/úkolů/procesů od podprogramů
  - mohou být implicitně spuštěny (Ada)
  - programová jednotka, která je spouští nemusí být pozastavena
  - po jejich skončení se řízení nemusí vracet do místa odkud byly odstartovány

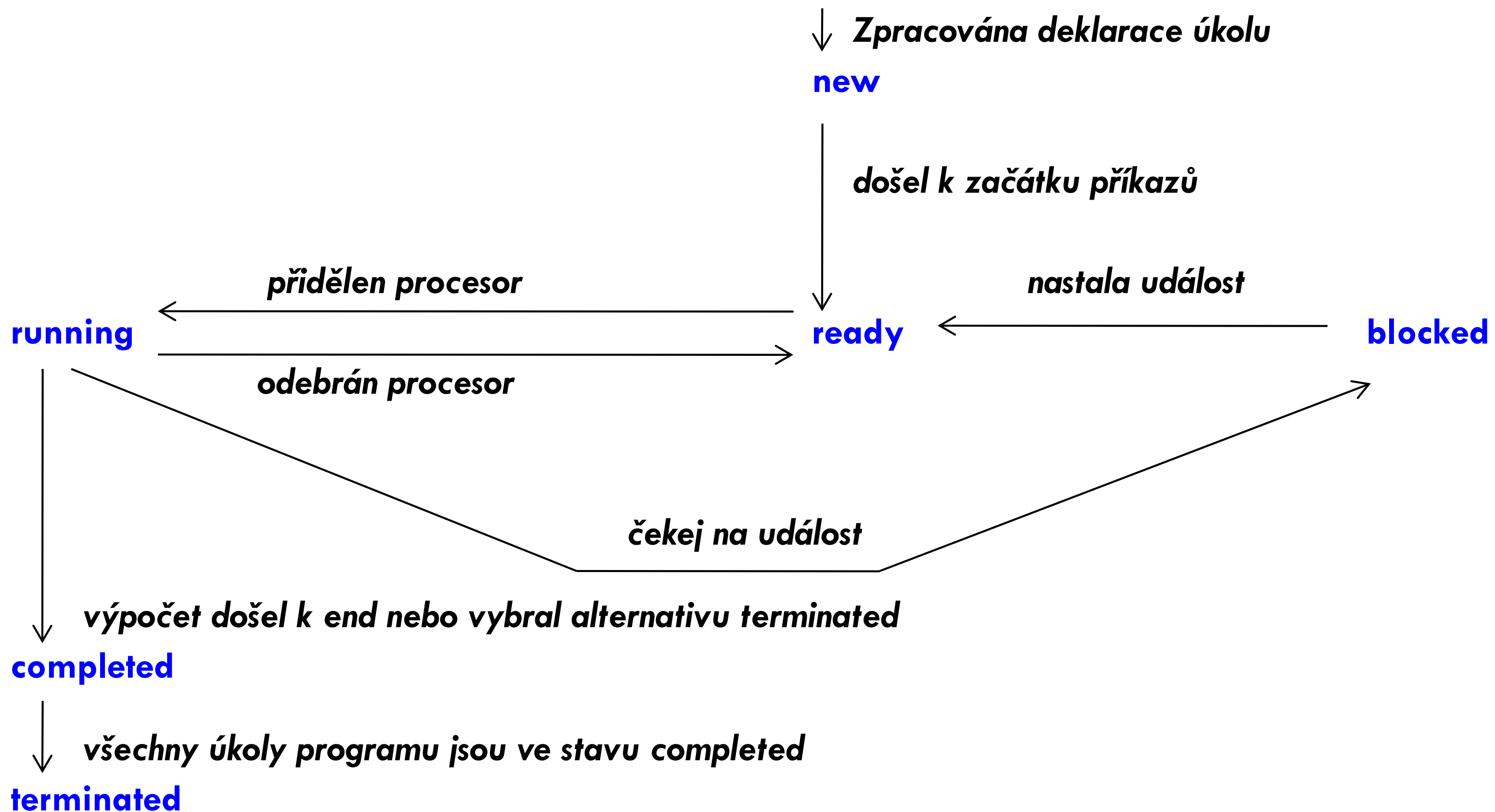
Paralelismus na úrovni podprogramů (2)

- Způsoby jejich komunikace:
  - sdílené nelokální proměnné
  - předávané parametry
  - zasílání zpráv
- Při synchronizaci musí A čekat na B (či naopak) (viz další slajd)
- Při soutěžení sdílí A s B zdroj, který není použitelný simultánně (např. sdílený čítač) a vyžaduje přístup ve vzájemném vyloučení.
- Části programu, které pracují ve vzájemném vyloučení jsou kritickými sekcemi.

Bariéra

- Synchronizační konstrukce
- Bariéra ve zdrojovém kódu skupiny vláken/procesů způsobí, že vlákna/procesy se musí zastavit na tomto místě, a nemohou pokračovat na tomto místě, dokud se ostatní vlákna k bariéře také nedostanou
- Použití – v případě, že potřebujeme výsledek z jiného vlákna abychom mohli pokračovat ve výpočtu
- V případě synchronního posílání zpráv - implicitní bariéra

Stavy úkolů (př. jazyka ADA)



Semafor

- Probereme princip synchronizačních a komunikačních prostředků (semafor, monitor, které pracují se sdílenou pamětí a zasílání zpráv, které lze použít i v distribuovaných výpočtech).
- **Semafor** = datová struktura obsahující čítač a frontu pro ukládání deskriptorů úkolů/procesů/vláken. Má dvě operace - zaber a uvolni (P a V). Je použitelný jak pro soutěžící, tak pro spolupracující úkoly.
- P a V jsou atomické operace (tzn. nepřerušitelné)
- P(semafor) /* uvažujme zatím binární */ /*zaber*/
- if semafor == 1 then semafor = 0

  else pozastav volající proces a dej ho do fronty na semafor

- V(semafor) /* také zatím binární*/ /*uvolni*/
- if fronta na semafor je prázdná then semafor = 1
- else vyber prvého z fronty a aktivuj ho

- Pokud chráníme ne 1 ale n zdrojů, nabývá čítač hodnoty 0..n
- Nebezpečnost semaforů
  - nelze kontrolovat řádnost použití → může nastat deadlock když neuvolníme

    → rychlostní závislost když nezabereme

Monitor a zasílání zpráv

- Obdobou je použití signálů

  Send(signal) --je akcí procesu 1

  Wait(signal) --je akcí procesu 2 (rozdíl proti P a V)
- **Monitor** - programový modul zapouzdřující data spolu s procedurami, které s daty pracují. Procedury mají vlastnost, že vždy jen jeden úkol/vlákno může provádět monitorovou proceduru, ostatní čekají ve frontě. (pro Javu to probereme důkladněji)
- **Zasílání zpráv** (je to synchronní způsob komunikace)
  - Při asynchronní komunikují přes vyrovnávací paměť
  - Vede k principu schůzky (rendezvous Ada)

Paralelní konstrukce v jazyce ADA

- Paralelně proveditelnou jednotkou je task

```
task  T is
  Deklarace tzv. vstupů (entry)
end T;
```

specifikační část

```
task body T is
  Lokální deklarace a příkazy
end T;
```

tělo

- Primárním prostředkem komunikace úkolů je **schůzka (rendezvous)**

Randezvous


```
task ČIŠNÍK is
       entry PIVO (X: in INTEGER);
       entry PLATIT;
                     …
end ČIŠNÍK ;


task body ČIŠNÍK is
 … --lokalni deklarace

   begin
     loop
       …--blouma u pultu
       accept PIVO (X: in INTEGER) do
       …--donese X piv
       end PIVO;
       …--sbira sklenice
       accept PLATIT do
       …--inkasuje
       end PLATIT;
      end loop;
   end ČIŠNÍK ;
```

```
task HOST1 is  --nemá zadny vstup
end HOST1;

task body HOST1 is
          …
          ČIŠNÍK.PIVO(2);  --vola vstup

          …--pije pivo
          ČIŠNÍK.PLATIT;

          …
end HOST1;
```

- Všechny úkoly se spustí současně, jakmile hlavní program dojde k begin své příkazové části (implicitní spuštění).

Randezvous (2)

- Úkoly mohou mít „vstupy“ (entry), pomocí nichž se mohou synchronizovat a realizovat rendezvous (schůzku)
- Př. Schránka pro komunikaci producenta s konzumentem. Schránka je jeden úkol, producent i konzument by byly další (zde nezapsané) úkoly

```
task SCHRANKA is
   entry PUT(X: in INTEGER);
   entry GET(X: out INTEGER);
end SCHRANKA;
task body SCHRANKA is
   V: INTEGER;
begin
   loop
          accept PUT(X: in INTEGER) do -zde čeká, dokud producent nezavolá PUT
                   V := X;
          end PUT;
          accept GET(X: out INTEGER) do -zde čeká, dokud konzument nezavolá GET
                   X := V;
          end GET;
   end loop;
end SCHRANKA; --napred do ni musí vlozit, pak ji muze vybrat
```

Randezvous (3)

- Konzument a producent jsou také úkoly a komunikují např.

  Producent: SCHRANKA.PUT(456);

  Konzument: SCHRANKA.GET(I);

- Pořadí provádění operací PUT a GET je určeno pořadím příkazů. PUT a GET se musí střídat.
- To nevyhovuje pro případ sdílené proměnné, do které je možné zapisovat a číst v libovolném pořadí, ne však současně.
- Libovolné pořadí volání vstupů dovoluje konstrukce select

```
select
        <příkaz accept neco>
        or
        <příkaz accept něco jineho>
or
        …
or
        terminate
end select;
```

Randezvous (4)

- Př. Sdílená proměnná realizovaná úkolem (dovolující libovolné pořadí zapisování a čtení)

```
task SDILENA is
   entry PUT(X: in INTEGER);
   entry GET(X: out INTEGER);
end SDILENA;
task body SDILENA is
   V: INTEGER;
begin
   loop
      select                      --dovoli alternativni provedeni
          accept PUT(X: in INTEGER) do
                    V := X;
          end PUT;
      or
          accept GET(X: out INTEGER) do
                    X := V;
          end GET;
      or
          terminate;  --umozni ukolu skoncit aniz projde koncovym end
      end select;
   end loop;
end SDILENA;
```

- nevýhodou je, že sdílená proměnná je také úkolem, takže vyžaduje režii s tím spojenou.
- Proto ADA zavedla tzv. protected proměnné a protected typy, které realizují monitor.

Paralelismus na úrovni příkazů jazyka - Occam

- Jazyk Occam je imperativní paralelní jazyk
- SEQ uvozuje sekvenční provádění

```
SEQ
   x := x + 1
      y := x * x
```

- PAR uvozuje paralelní provádění
- V konstrukcích lze kombinovat

```
WHILE next <> eof
   SEQ
       x := next
       PAR
              in  ?  Next
              out  !  x  *  x
```

Paralelismus na úrovni příkazů jazyka - Fortran

High performance Fortran

- Založen na modelu SIMD:
  - výpočet je popsán jednovláknovým programem
  - proměnné (obvykle pole) lze distribuovat mezi více procesorů
  - distribuce, přístup k proměnným a synchronizace procesorů je zabezpečena kompilátorem

Př. Takto normálně Fortran násobí dělí trojúhelníkovou matici pod hlavní diagonálou příslušným číslem na diagonále

```
REAL DIMENSION (1000, 1000) :: A
INTEGER  I, J
…
DO I = 2, N
   DO J = 1, I - 1
         A(I, J) = A(I, J) / A(I, I)
   END DO
END DO
```

- High Performance Fortran to ale umí i paralelně příkazem

```
FORALL (I = 2 : N, J = 1 : N, J .LT. I )  A(I, J) = A(I, J) / A(I, I)
```

- kterým se nahradí ty vnořené cykly
- FORALL představuje zobecněný přiřazovací příkaz (a ne smyčku)
- Distribuci výpočtu na více procesorů provede překladač.
- FORALL lze použít, pokud je zaručeno, že výsledek seriového i paralelního zpracování budou identické.

Paralelismus na úrovni programů

- Pouze celý program může v tomto případě být paralelní aktivitou.
- Je to věcí operačního systému
- Např. příkazem Unixu fork vznikne potomek, přesná kopie volajícího procesu, včetně proměnných
- Následující příklad zjednodušeně ukazuje princip na paralelním násobení matic, kde se vytvoří 10 procesů s myid 0,1,…,9.
- Procesy vyjadřují, zda jsou rodič nebo potomek pomocí návratové hodnoty fork. Na základě testu hodnoty fork() pak procesy rodič a děti mohou provádět odlišný kod.
- Po volání fork může být proces ukončen voláním exit.
- Synchronizaci procesů provádí příkaz wait, kterým rodič pozastaví svoji činnost až do ukončení svých dětí.

Paralelismus na úrovni programů (2)


```
#define SIZE 100
#define NUMPROCS 10
int a[SIZE] [SIZE], b[SIZE] [SIZE], c[SIZE] [SIZE];
void multiply(int myid)
{ int i, j, k;
  for (i = myid; i < SIZE; i+= NUMPROCS)
    for (j = 0; j < SIZE; ++j)
    { c[i][j] = 0;
      for (k = 0; k < SIZE; ++k)
        c[i][j] += a[i][k] * b[k][j];
    }
}
main()
{ int myid;
  /* prikazy vstupu a, b */
  for (myid = 0; myid < NUMPROCS; ++myid)
    if (fork() == 0)
    { multiply(myid);
      exit(0);}
  for (myid = 0; myid < NUMPROCS; ++myid)
    wait(0);
  /* prikazy vystupu c */
  return 0;
}
```

Java vlákna (threads)

Paralelně proveditelné jednotky jsou objekty s metodou run, jejíž kód může být prováděn souběžně s jinými takovými metodami a s metodou main, ta je také vláknem. Metoda run se spustí nepřímo vyvoláním start( ).
Jak definovat třídy, jejichž objekty mohou mít paralelně prováděné metody?

1. **jako podtřídy třídy Thread (je součástí java.lang balíku, potomkem Object)**
2. **implementací rozhrani Runnable**

ad 1.

```
class MyThread extends Thread   // 1. Z třídy Thread odvodíme potomka (s run metodou)
{public void run( )  { . . .}
  . . .
}
. . .
MyThread   t = new MyThread( ); // 2. Vytvoření instance této třídy potomka
. . .
```

---

ad 2.

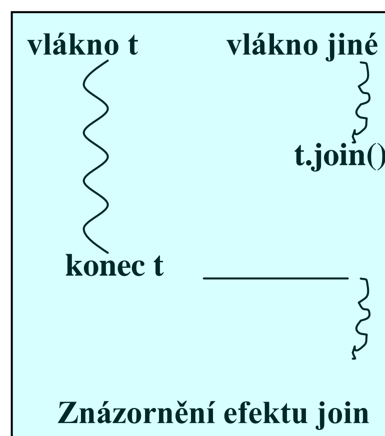
```
class MyR implements Runnable     //1. konstruujeme třídu implementující Runnable
{public void run( )  { . . .}
  . . .
}
. . .
MyR  m = new MyR( );  // 2. konstrukce objektu teto třídy (s metodou run)
Thread t = new Thread(m); //3. vytvoření vlákna na tomto objektu
//je zde použit konstruktor Thread(Runnable threadOb)
. . .
```

--

Vlákno t se spustí až provedením příkazu t.start( );

Třída Thread má řadu metod např.:

```
final void setName(String jméno) //přiřadí vláknu jméno
                                   zadání jména i Thread(Runnable jmR, String jméno)
final String getName( ) //vrací jméno vlákna
final int getPriority( )  //vrací prioritu
final void setPriority(int nováPriorita)
final boolean isAllive( ) //zjišťuje živost vlákna
final void join( ) //počkej na skončení vlákna
void run( )
static void sleep(long milisekundy)//uspání vlákna
void start( )
…
. . .
```



Znázornění efektu join

Rozhraní Runnable má jen metodu run( ).
Když uživatelova vlákna nepřepisují ostatní met. (musí přepsat jen run), upřednostňuje se runnable.

```
class MyThread implements Runnable {//př.3Vlakna implementací Runnable
  int count;
  String thrdName;

  MyThread(String name) { // objekty z myThread mohou být konstruktoru
    count = 0;           // předány s parametrem String
    thrdName = name; // řetězec sloužící jako jméno vlákna
  }

    public void run() { // vstupní bod vlákna
    System.out.println(thrdName + " startuje.");
    try {  // sleep může být přerušeno InterruptedException
      do {
        Thread.sleep(500);
        System.out.println("Ve vlaknu " + thrdName +
                           ", citac je " + count);
        count++;
      } while(count < 5);
    }
    catch(InterruptedException exc) {// nutno ošetřit přerušeni spaní
      System.out.println(thrdName + " preruseny.");
    }
    System.out.println(thrdName + " ukonceny.");
  }
}

class Vlakno {
  public static void main(String args[]) {
    System.out.println("Hlavni vlakno startuje");

    // Nejdříve konstruujeme MyThread objekt. Má metodu run(), ale
    MyThread mt = new MyThread("potomek");  // ostatní met.vlákna nemá

    // Pak konstruuj vlákno z tohoto objektu, tím mu dodáme start(),…
    Thread newThrd = new Thread(mt);

    // Až pak startujeme výpočet vlákna
    newThrd.start(); //ted běží současně metody main a run z newThrd

    do {
      System.out.print(".");
      try {
        Thread.sleep(100); //sleep je stat. metoda Thread, kvalifikovat
      }
      catch(InterruptedException exc) {//nutno ošetřit přerušeni spani
        System.out.println("Hlavni vlakno prerusene.");
      }
    } while (mt.count != 5);

    System.out.println("Konci hlavni vlakno");
  }
}  // při opakovaném spouštění se výsledky mohou lišit (rychlostní závislosti)
```

```
class MyThread extends Thread { // př. 3aVlakna   dtto, ale děděním ze Threadu
  int count;

  MyThread(String name) {
    super(name); // volá konstruktor nadtřídy a předá mu parametr jméno vlákna
    count = 0;
  }

    public void run() { // vstupní bod vlákna
    System.out.println(getName() + " startuje.");
    try {
      do {
        Thread.sleep(500); // Kvalifikace není nutná
        System.out.println("Ve vlaknu " + getName() +
                          ", citac je " + count);
        count++;
      } while(count < 5);
    }
    catch(InterruptedException exc) {//  nutno ošetřit přerušeni spaní
      System.out.println(getName() + " prerusene.");
    }
    System.out.println(getName() + " ukoncene.");
  }
}

class Vlakno {
  public static void main(String args[]) {
    System.out.println("Hlavni vlakno startuje");

    // Nejdříve konstruujeme MyThread objekt.
    MyThread mt = new MyThread("potomek"); // objekt mt je vlákno, má jméno
                                            // potomek, nemusí mít ale žádné
    // Az pak startujeme vypocet vlakna
    mt.start();

    do {
      System.out.print(".");
      try {
        Thread.sleep(100); //Kvalifikace je nutna
      }
      catch(InterruptedException exc) {//nutno ošetřit přerušeni spani
        System.out.println("Hlavni vlakno prerusene.");
      }
    } while (mt.count != 5);
    System.out.println("Konci hlavni vlakno");
  }
}
```

```
class MyThread implements Runnable { // př. 4Vlakna  Modifikace:
// vlákno se rozběhne v okamžiku jeho vytvoření
// jméno lze vláknu přiřadit až v okamžiku spuštění
  int count;
  Thread thrd; // odkaz na vlákno je uložen v proměnné thrd

  // Uvnitř konstruktoru vytváří nové vlákno konstruktorem Thread.
  MyThread(String name) {
    thrd = new Thread(this, name);// vytvoří vlákno a přiřadí jméno
    count = 0;             // Konstr.Thread lze různě parametrizovat
    thrd.start(); // startuje vlakno rovnou v konstruktoru
  }

  // Začátek exekuce vlákna
  public void run() {
    System.out.println(thrd.getName() + " startuje ");
    try {
      do {
        Thread.sleep(500);
        System.out.println("V potomkovi " + thrd.getName() +
                           ", citac je " + count);
        count++;
      } while(count < 5);
    }
    catch(InterruptedException exc) {
      System.out.println(thrd.getName() + " preruseny.");
    }
    System.out.println(thrd.getName() + " ukonceny.");
  }
}

class VlaknoLepsi {
  public static void main(String args[]) {
    System.out.println("Hlavni vlakno startuje");

    MyThread mt = new MyThread("potomek"); //v konstruktoru se i spustí

    do {
      System.out.print(".");
      try {
        Thread.sleep(100);
      }
      catch(InterruptedException exc) {
        System.out.println("Hlavni vlakno prerusene.");
      }
    } while (mt.count != 5);

    System.out.println("Hlavni vlakno konci");
  }
}    // tisky z této modifikace jsou stejné jako předchozí
```

```
class MyThread extends Thread {       // př. 5Vlakna = totéž jako 4Vlakna, ale děděním ze Threadu
  int count;
            // není třeba referenční proměnná thrd. Třída MyThread bude obsahovat instance mt
    MyThread(String name) {
    super(name); // volá konstruktor nadtřídy, předává mu jméno vlákna jako parametr
    count = 0;
    start(); // startuje v konstruktoru, jelikož odkazuje na sebe, tak není nutná kvalifikace
  }

   public void run() {  // v potomkovi ze Threadu musíme předefinovat run()
    System.out.println(getName() + " startuje");
    try {
      do {
        Thread.sleep(500);  // zde kvalifikace není nutná, protože jsme v potomkovi ze Threadu
        System.out.println("V " + getName() +
                          ", citac je " + count);
        count++;
      } while(count < 5);
    }
    catch(InterruptedException exc) {
      System.out.println(getName() + " prerusene");
    }
    System.out.println(getName() + " ukoncene");
  }
}

class DediThread { // To není potomek Threadu,  vytváří se v ní "potomek"
  public static void main(String args[]) {
    System.out.println("Hlavni vl.startuje");

    MyThread mt = new MyThread("potomek");

    do {
      System.out.print(".");
      try {
        Thread.sleep(100); // zde je nutná kvalifikace
      }
      catch(InterruptedException exc) {
        System.out.println("Hlavni vl. prerusene");
      }
    } while (mt.count != 5);

    System.out.println("Hlavni vl. konci");
  }
}
```

```
class MyThread implements Runnable {  // př. 6Vlakna spuštění více vláken s použitím Runnable
  int count;
  Thread thrd;

  MyThread(String name) {
    thrd = new Thread(this, name); // vytvoří vlákno thrd na objektu třídy MyThread
    count = 0;
    thrd.start(); // startuje vlákno thrd
  }

  public void run() {
    System.out.println(thrd.getName() + " startuje");
    try {
      do {
        Thread.sleep(500);
        System.out.println("Ve " + thrd.getName() +
                        ", citac je " + count);
        count++;
      } while(count < 3);
    }
    catch(InterruptedException exc) {
      System.out.println(thrd.getName() + " prerusene");
    }
    System.out.println(thrd.getName() + " ukoncene");
  }
}

class ViceVlaken {
  public static void main(String args[]) {
    System.out.println("Hlavni vlakno startuje");

    MyThread mt1 = new MyThread("potomek1");     // vytvoření 3 vláken a jejich spuštění
    MyThread mt2 = new MyThread("potomek2");     // v pořadí 1, 2, 3. Výpisy čítačů se při
    MyThread mt3 = new MyThread("potomek3");     // opakovaném spouštění mohou lišit

    do {
      System.out.print(".");
      try {
        Thread.sleep(100);
      }
      catch(InterruptedException exc) {
        System.out.println("Hlavni vlakno prerusene");
      }
    } while (mt1.count < 3 ||
             mt2.count < 3 ||        // zajistí, že všechna vlákna potomků již skončila
             mt3.count < 3);

    System.out.println("Hlavni vl. konci");
  }
}
```

```
class MyThread extends Thread { // př. 6a Spuštění vice vláken jako v 5, ale děděním ze Threadu
 int count;

 MyThread(String name) {
  super(name);
  count = 0;
  start(); // start
 }

 public void run() {
  System.out.println(getName() + " startuje");
  try {
   do {
    Thread.sleep(500);
    System.out.println("Ve " + getName() +
                   ", citac je " + count);
    count++;
   } while(count < 3);
  }
  catch(InterruptedException exc) {
   System.out.println(getName() + " preruseny");
  }
  System.out.println(getName() + " ukonceny");
 }
}

class ViceVlaken {
 public static void main(String args[]) {
  System.out.println("Hlavni vlakno startuje");

  MyThread mt1 = new MyThread("potomek1");
  MyThread mt2 = new MyThread("potomek2");
  MyThread mt3 = new MyThread("potomek3");

  do {
   System.out.print(".");
   try {
    Thread.sleep(100);
   }
   catch(InterruptedException exc) {
    System.out.println("Hlavni vlakno prerusene");
   }
  } while (mt1.count < 3 ||
        mt2.count < 3 ||
        mt3.count < 3);

  System.out.println("Hlavni vl. konci");
 }
}
```

Identifikace ukončení činnosti vláken

- nejčastěji k zastavení dojde doběhnutím metody run( )

- stav lze testovat metodou isAlive( ) vracející true, pokud již provedeno new a není dead
tvar: final boolean isAlive( )
Jak využít v modifikaci předchozích programů? Viz * poznámka dole

- čekáním na skončení jiného vlákna vyvoláním metody join()
tvar: final void join( ) throws InterruptedException
Např.

```
Thread t = new Thread(m); // rodič vlákna t (tj. vlákno, které vytváří t) stvořil t
t.start( ); // rodič zahaji činnost vlákna potomka tj. t
// rodič něco dělá
t.join( ); // rodič čeká na skončení t
// rodič pokračuje po skončení t
```

- existuje alternativa čekání na skončení vlákna, informující, že se čeká na jeho konec

```
Thread t = new Thread(m);
t.start( ); // zahájí činnost
// rodič něco dělá
t.interrupt( ) ; // rodič ho (tj. t) přeruší
                 // Pokud t nespí, nahodí se mu flag, který lze testovat metodami
                 // boolean interrupted(), která flag shodí, nebo
                 // boolean isInterrupted(), která flag neshodí
t.join( ); // rodič čeká na skončení t
// rodič pokračuje po skončení t
```

- existuje alternativa pro timeout: t.join(milisekundy) čeká nejvýše zadaný počet ms, pak jde dál. join(0) je nekonečné čekání jako join()

***Poznámka:**

V main př.6 změníme while na:

```
. . .
} while (mt1.thrd.isAlive() ||
         mt2.thrd.isAlive() ||
         mt3.thrd.isAlive());

    System.out.println("Main thread ending.");
  }
}
```

```
//Př. 7aVlakna   použitím join testujeme konec vláken
class MyThread extends Thread {
 int count;
 MyThread(String name) {
   super(name);
   count = 0;
   start(); // start
 }
  public void run() {
   System.out.println(getName() + " startuje");
   try {
     do {
       Thread.sleep(500);
       System.out.println("Ve " + getName() +
                   ", citac je " + count);
       count++;
     } while(count < 3);
   }
   catch(InterruptedException exc) {
     System.out.println(getName() + " preruseny");
   }
   System.out.println(getName() + " konci");
 }
}

class Join {
   public static void main(String args[]) {
   System.out.println("Hlavni vlakno startuje");

   MyThread mt1 = new MyThread("potomek1");
   MyThread mt2 = new MyThread("potomek2");
   MyThread mt3 = new MyThread("potomek3");

   try {
     mt3.join();                                    \
     System.out.println("potomek3 joined.");        |
     mt2.join();                                    |
     System.out.println("potomek2 joined.");        }  místo testování isAlive()
     mt1.join();                                    |
     System.out.println("potomek1 joined.");        /
   }
   catch(InterruptedException exc) {
     System.out.println("Hlavni vlakno prerusene");
   }
    System.out.println("Hlavni vl. konci");
 }
}
```

```
// Př. 7Vlakna
class MyThread implements Runnable { // s Runnable dtto 7, MyThread má tvar jako v př. 6
  int count;
  Thread thrd;
  MyThread(String name) {
    thrd = new Thread(this, name);
    count = 0;
    thrd.start(); // start
  }
   public void run() {
    System.out.println(thrd.getName() + " startuje");
    try {
      do {
        Thread.sleep(500);
        System.out.println("Ve " + thrd.getName() +  ", citac je " + count);
        count++;
      } while(count < 3);
    }
    catch(InterruptedException exc) {
      System.out.println(thrd.getName() + " preruseny");
    }
    System.out.println(thrd.getName() + " konci");
  }
}
class Join {
    public static void main(String args[]) {
    System.out.println("Hlavni vlakno startuje");

    MyThread mt1 = new MyThread("potomek1");
    MyThread mt2 = new MyThread("potomek2");
    MyThread mt3 = new MyThread("potomek3");

    try {
      mt1.thrd.join(); // vlákno thrd na objektu mt1
      System.out.println("potomek1 joined.");
      mt2.thrd.join();
      System.out.println("potomek2 joined.");
      mt3.thrd.join();
      System.out.println("potomek3 joined.");
    }
    catch(InterruptedException exc) {
      System.out.println("Hlavni vlakno preruseno");
    }
     System.out.println("Hlavni vl. konci");
  }
}
```

<u>Priorita vláken</u> = pravděpodobnost získání procesorového času

- **Vysoká priorita = hodně času procesoru**
- **Nízká priorita = méně času procesoru**
- **Implicitně je přidělena priorita potomkovi jako má nadřízený process**
- **Změnit lze prioritu metodou** ***setPriority***

final void setPriority(int cislo) kde číslo musí být v intervalu

Min_Priority* ≤ cislo ≥ Max_Priority*
1 .. 10

***To jsou konstanty třídy Thread**

Norm_Priority* = 5

- **Zjištění aktuální priority provedeme metodou**
  final int getPriority( )

Způsob implementace priority závisí na JVM. Ta ji nemusí také vůbec respektovat.

```
class Priority extends Thread { // Př. 8aVlakna  - s prioritami (na OS se sdílením času)
  int count;
  static boolean stop = false; //zastaví vlákno mtx, když skončí mty   } to jsou proměnné
  static String currentName;//jméno procesu, který právě běží          }         třídy Priority
  Priority(String name) {
    super(name);
    count = 0;
    currentName = name;
  }
 public void run() {
    System.out.println(getName() + " start ");
    do {
      count++;                                              // čítač iterací
      if(currentName.compareTo(getName()) != 0) { //kontrola jména vlákna v currentName
        currentName = getName();                          // s aktuálním. Při ≠ zaznamená a vypíše
        System.out.println("Ve " + currentName);   // jméno aktuálního
      }
    } while(stop == false && count < 50); // dvě možnosti ukončení běhu vlákna
    stop = true; // pokud skončilo jedno, tak pak skončí i druhé vlákno
    System.out.println("\n" + getName() + " konci");
  }
}

class Priorita {
  public static void main(String args[]) {
    Priority mt1 = new Priority("Vysoka Priorita"); // JVM to může, ale nemusí respektovat
    Priority mt2 = new Priority("Nizka Priorita");

    // nastaveni priorit
    mt1.setPriority(Thread.NORM_PRIORITY + 2); // JVM to nemusí respektovat
    mt2.setPriority(Thread.NORM_PRIORITY - 2);
    // start vláken
    mt1.start();
    mt2.start();
    try {
      mt1.join(); // main čeká na ukončení mt1, mt2
      mt2.join();
    }
    catch(InterruptedException exc) {
      System.out.println("Hlavni vlakno konci");
    }
    System.out.println("Vlakno s velkou prioritou nacitalo " +
                   mt1.count);
    System.out.println("Vlakno s malou prioritou nacitalo " +
                   mt2.count);
  }
}
```

```
class Priority implements Runnable { //Př. 8Vlakna jako 8a s Runnable
  int count;  // Každý objekt ze třídy Priority má čítač a vlákno
  Thread thrd;
  static boolean stop = false;
  static String currentName;
  Priority(String name) {
    thrd = new Thread(this, name);
    count = 0;
    currentName = name;
  }
  public void run() {
    System.out.println(thrd.getName() + " start ");
    do {
      count++;

      if(currentName.compareTo(thrd.getName()) != 0) {
        currentName = thrd.getName();
        System.out.println("V " + currentName);
      }
    } while(stop == false && count < 500);
    stop = true;
    System.out.println("\n" + thrd.getName() + " terminating.");
  }
}

class Priorita {
  public static void main(String args[]) {
    Priority mt1 = new Priority("Vysoka Priorita");
    Priority mt2 = new Priority("Nizka Priorita");
    // nastaveni priorit
    mt1.thrd.setPriority(Thread.NORM_PRIORITY + 2); // priorita 7
    mt2.thrd.setPriority(Thread.NORM_PRIORITY - 2); // priorita 3
    // start vlaken
    mt1.thrd.start();
    mt2.thrd.start();

    try {
      mt1.thrd.join();
      mt2.thrd.join();
    }
    catch(InterruptedException exc) {
      System.out.println("Hlavni vlakno preruseno");
    }

    System.out.println("Vlakno s velkou prioritou nacitalo " +
                       mt1.count);
    System.out.println("Vlakno s malou prioritou nacitalo" +
                       mt2.count);
  }
}
```

```
//Př.  81Vlakna  Ilustruje rychlostní závislosti výpočtu. Kratší/delší sleep simuluje různé rychlosti
class Konto { static int konto = 1000;}
class Koupe extends Thread {
  Koupe(String jmeno) {
   super(jmeno);
  }
  public void run()  { // vstupní bod vlákna
          System.out.println(getName() + " start.");
          int lokal;
          try {
          lokal = Konto.konto;
          System.out.println(getName() + " milenkam ");
          sleep(100);/////////////////////////////////////
          Konto.konto = lokal - 200;
          System.out.println(getName() + " ukoncene.");
          }
          catch (InterruptedException e) {}
} }
class Prodej extends Thread {
  Prodej(String jmeno) {
   super(jmeno);
  }
  public void run() { // vstupní bod vlákna
          System.out.println(getName() + " start.");
          int lokal;
          try {
          lokal = Konto.konto;
          System.out.println(getName() + " co se da ");
          sleep(200);///////////////////////////////////
          Konto.konto = lokal + 500;
          System.out.println(getName() + " ukoncene.");
          }
          catch (InterruptedException e) {}
} }
class RZ {
     public static void main (String args[])
          throws InterruptedException {
          System.out.println("Hlavni vlakno startuje");
          Koupe nakup = new Koupe("kupuji");
          Prodej prodej = new Prodej ("prodavam");
          nakup.start();
          prodej.start();
          Thread.sleep(500); // “zajistí“, že nákup i prodej skončil
          System.out.println(Konto.konto);
          System.out.println("Konci hlavni vlakno");
}   }
```

Kritické sekce

(řešení problému sdílení zdrojů formou vzájemného vyloučení současného přístupu)

1. **Metodou s označením synchronized uzamkne objekt, pro který je volána.**
   Jiná vlákna pokoušející se použít synchr. metodu uzamčeného objektu musejí čekat ve frontě, tím se zamezí interferenci vláken způsobující nekonzistentnosti paměti.
   Když proces opustí synchr. metodu, objekt se odemkne.
   Objekt může mít současně synch.i nesynch. metody a ty nevyžadují zámek => vada.
   Lze provádět nesynchronized metody i na zamknutém objektu.

   (Každý objekt Javy je vybaven zámkem, který musí vlákno vlastnit, chce-li provést synchronized metodu na objektu.)

   např.
```
class Queue {
        . . .
         public synchronized int vyber( ){ …}
        . . .
        public synchronized  void uloz(int co){ …}
        . . .
}
```

   synchronizovaný příkaz tvaru

   synchronized (výraz s hodnotou objekt) příkaz

2. **Zamkne přístup k objektu (je zadán výrazem) pro následný úsek programu. Systém musí objekt vybavit frontou pro metody, které chtějí s ním v „příkazu" pracovat.**

Komunikace mezi vlákny

(řeší situaci, kdy metoda vlákna potřebuje přístup k dočasně nepřístupnému zdroji)

- **může čekat v nějakém cyklu (neefektivní využití objektu, nad nímž pracuje)**
- **může se zřeknout kontroly nad objektem a jiným vláknům umožnit ho používat, musí jim to ale dát na vědomí**

Kooperace procesů zajišťují následující metody zděděné z třídy Object (aplikovatelné pouze na synchronized metody a příkazy):

- **wait() vlákno přejde do stavu blokovaný a uvolní zámek objektu, musí být uvnitř try bloku a má další verze:**
  final void wait( ) throws InterruptedException
  final void wait(long milisec) throws InterruptedException
  final void wait(long milisec, int nanosec) throws InterruptedException
  S nimi spolupracují metody
- **final void notify( ) oživí vlákno z čela fronty na objekt**
- **final void notifyAll( ) oživí všechna vlákna nárokující si přístup k objektu, ta pak o přístup normálně soutěží (na základě priority nebo plánovacího algoritmu JVM).**
  Mohou být volány jen z vláken, které vlastní zámek (synchronized metod a příkazů), jsou děděny z pratřídy Object.

Když je zavoláno wait(), vlákno se zablokuje a nelze ho naplánovat dokud nenastane některá z alternativ:

- **jiné vlákno nezavolá notify pro tento objekt (vlákno se tak stane runnable)**
- **„ „ „ notifyAll „ „ „ „ „**
- **„ „ „ interrupt „ „ „ „ „ a vyhodí výj.**
- **uplyne specifikovaný wait čas**

Pozn.

Konstruktor nemůže být synchronized (hlásí se syntaktická chyba). Nemělo by to ani smysl.

Jaký má efekt volání static synchronized metody? Ta je asociována s třídou. Volající vlákno zabere tedy zámek třídy (je považována také za objekt) a má pak výlučný přístup ke statickým položkám třídy. Tento zámek nesouvisí se zámky instancí této třídy.

Vlákno nemůže zabrat zámek, který vlastní již jiné vlákno, může ale zabrat opakovaně zámek který již samo vlastní (reentrantní synchronizace). Nastává, když synchrized kód vyvolá metodu, která také obsahuje synchronized kód a oba používají tentýž zámek.

Atomické akce jako např. read a write proměnných deklarovaných jako volatile (= nestálé) nevyžadují synchronizaci. Vlákno si pak nesmí tvořit jejich kopii (z optimalizačního důvodu to normálně dělat může), takže pokud hodnota proměnné neovlivňuje stav jiných proměnných včetně sebe, pak se nemusí synchronizovat. Jejich použití je časově úspornější. Balík java.util.concurrent poskytuje i metody, které jsou atomické.

Př. Semafor jako ADT v Javě

```
class Semafor {
        private int count;

        public Semafor(int initialCount) {
                count = initialCount;        // když je 1, je to binární semafor
        }

        public synchronized void cekej( ) {
                try {
                while (count <= 0 ) wait( );  // musí  být nedělitelné nad instancí semaforu
                count--;
                }
                catch (InterruptedException e) { }
        }

        public synchronized void uvolni( ) {
                count++;
                notify( );
        }
}
```

```
// Př. 82Vlakna  Odstranění rychlostních závislostí z př.81. Výsledek na času sleep nezávisí
class Konto { // instance z třídy Konto uděláme monitorem
   private int konto; // to je pamět pro vlastní konto
   public Konto(int i) { konto =i;}  // konstruktor pro založení a inicializaci konta
   public int stav() {return konto;}  // není sychronized
   public synchronized void vyber(int kolik) {
         int lokal; // pro zachovani podminek jako u RZ
         try { lokal = konto;
         Thread.sleep(100);//////////////////////////////////
         konto = lokal - kolik;
         }        catch (InterruptedException e) {}
   }
   public synchronized void vloz(int kolik) {
         int lokal;
         try { lokal = konto;
         Thread.sleep(200);/////////////////////////////////////
         konto = lokal + kolik;
         }        catch (InterruptedException e) {}
} }
class Koupe extends Thread {
   private Konto k;
   Koupe(Konto x, String jmeno) { super(jmeno); k = x; }
   public void run() { // vstupní bod vlákna
         System.out.println(getName() + " start.");
         k.vyber(200);
         System.out.println(getName() + " ukoncene.");
} }
class Prodej extends Thread {
  private Konto k;
  Prodej(Konto x, String jmeno) { super(jmeno); k = x;  }
  public void run() { // vstupní bod vlákna
         System.out.println(getName() + " start.");
         k.vloz(500);
         System.out.println(getName() + " ukoncene.");
} }
class Konta {
    public static void main (String args[]) throws InterruptedException {
         System.out.println("Hlavni vlakno startuje");
         Konto meKonto = new Konto(1000);// vytvářím konto, mohu jich udělat i více
         Koupe nakup = new Koupe(meKonto," nakupuji ");
         Prodej prodej = new Prodej (meKonto, " prodavam ");
         nakup.start();
         prodej.start();
         Thread.sleep(500);  // aby Koupe i Prodej měly čas skončit
         System.out.println(meKonto.stav());
         System.out.println("Konci hlavni vlakno");
} }
```

Rekapitulace:
Každé vlákno je instancí třídy java.lang.Thread nebo jejího potomka.

Thread má metody:

- **run()   je vždy přepsána v potomku Thread, udává činnost vlákna**
- **start()   spustí vlákno (tj. metodu run) a volající start pak pokračuje ve výpočtu. Metoda run není přímo spustitelná.**
- **yield()   odevzdání zbytku přiděleného času a zařazení do fronty na procesor**
- **sleep(milisec)   zablokování vlákna na daný čas. interval**
- **isAlive( )   běží-li, vrací true, jinak false**
- **join( )   čeká na ukončení vlákna**
- **getPriority()   zjistí prioritu**
- **setPriority()   nastaví prioritu**
- **. . . a dalších cca 20**

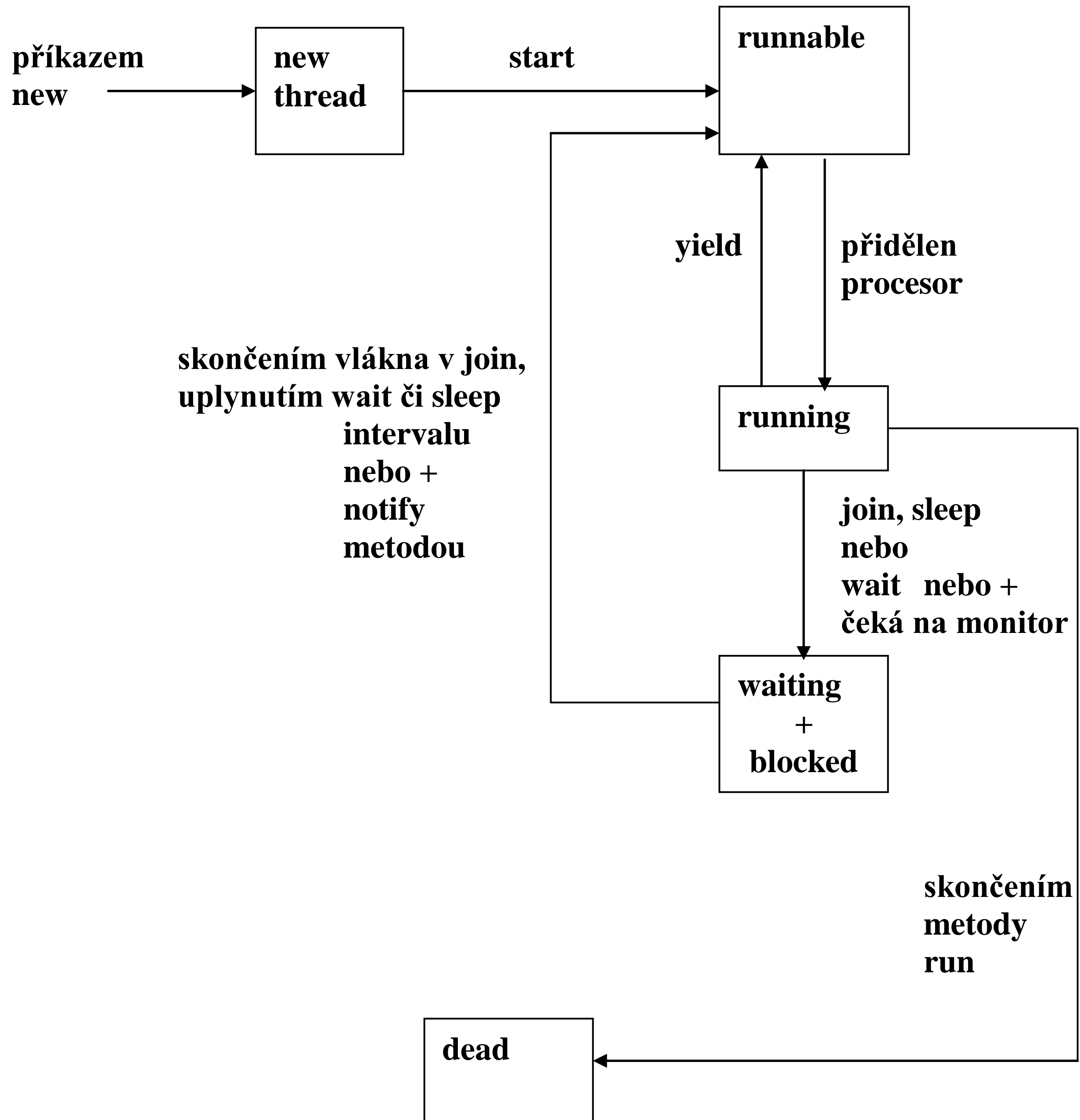
Objekt má metody, které Thread dědí

- **final void notify() oživí vlákno z čela fronty na objekt**
- **final void notifyAll() oživí všechna vlákna nárokující si přístup k objektu**
- **final void wait( ) throws InterruptedException Vlákno čeká, až jiné zavolá notify**
- **final void wait(long) čeká na notify/notifyAll nebo vypršení specifikovaného času**

stavy vláken:

- **nové   (ještě nezačalo běžet)**
- **připravené (nemá přidělený procesor)**
- **běžící   (má   „ „   „   )**
- **blokované   (čeká ve frontě na monitor)**
- **čekající (provedlo volání např.** `Object.wait` bez timeoutu, `Thread.join` bez timeoutu, či `LockSupport.park`
- **časově čekající (provedlo volání např.** `Thread.sleep`, `Object.wait` s timeoutem, `Thread.join` s timeoutem, `LockSupport.parkNanos`, `LockSupport.parkUntil`
- **mrtvé**

Plánovač vybere z fronty připravených vlákno s nejvyšší prioritou.



Obr. Přechody mezi stavy vlákna Javy (zjednodušeně)

```
 import java.io.*;     // př. 9Vlakna. Ukázka použití wait a notify. Producent ukládá do
//bufferu Queue čtená čísla, konzument z něj vybírá a vypisuje. Hlásí špatně napsané a
//chce nové. Přečtením záporného čísla program končí. Queue je monitorem. Není vláknem,
//to je důvod, proč wait, notify jsou metody z Object.

class Queue { private int [] que;     // Buffer má podobu kruhové fronty realizované polem
        private int nextIn, nextOut, filled, queSize;
        public Queue(int size) {
                que = new int [size];
                filled = 0;       // zaplněnost bufferu
                nextIn = 1;     // kam vkládat
                nextOut = 1;  // odkud vybírat
                queSize = size;
        } // konec konstruktoru

        public synchronized void deposit (int item) { // zamkne objekt
                try {
                        while (filled == queSize)
                                wait();  // odemkne objekt, když je fronta plná, a čeká
                        que [nextIn] = item;
                        nextIn = (nextIn % queSize) + 1;
                        filled++;
                        notify();  // budí vlákno konzumenta a uvolňuje monitor
                } // konec try
                catch (InterruptedException e) {
                        System.out.println("int.depos");
                }
        } // konec deposit()

        public synchronized int fetch() {
                int item = 0;
                try {
                        while (filled == 0)
                                wait();  // odemkne objekt fronta a čeká na vložení
                        item = que [nextOut];
                        nextOut = (nextOut % queSize) + 1;
                        filled--;
                        notify(); // budí vlákno producenta a uvolňuje monitor
                } // konec try
                catch(InterruptedException e) {
                        System.out.println("int.fetch");
                }
                return item;
        } // konec fetch()

} // konec třídy Queue
```

```
class Producer extends Thread {    // producent čte z klávesnice a ukládá do bufferu
        private Queue buffer;
        public Producer(Queue que) {// konstruktor producenta dostane jako param. frontu
                buffer = que;
        }
        public void run() {
           int new_item = 0; // nepřeloží se bez inicializace
                while (new_item > -1) { /* ukončíme -1 nebo
                                                záporným číslem */
                  try {//produkce
                 byte[] vstupniBuffer = new byte[20];
                 System.in.read(vstupniBuffer);
                 String s = new String(vstupniBuffer).trim(); // ořezání neviditelných znaků
                 new_item = Integer.valueOf(s).intValue(); // převedení na integer
                  }
                  catch (NumberFormatException e) {  // když číslo není správně zapsané
                          System.out.println("nebylo to dobre");
                          continue;
                  }
                 catch (IOException e) {// zachytává nepřipr. klavesnici
                          System.out.println("chyba cteni");
                  }
                  buffer.deposit(new_item);  // producent plní buffer
                }
        }
}

class Consumer extends Thread { // konzument vybírá údaj z bufferu a tiskne ho
        private Queue buffer;
        public Consumer(Queue que) {
                buffer = que;
        }
        public void run() {
                int stored_item = 0;  // chce inicializaci
                while (stored_item >  -1) { // ukončíme -1 nebo záporným číslem
                  stored_item = buffer.fetch();  // konzument vybírá buffer
                  System.out.println(stored_item);       // konzumace
                }
        }
}
```

```
public class P_C {
        public static void main(String [] args) {
                Queue buff1 = new Queue(100);
                Producer producer1 = new Producer(buff1);
                Consumer consumer1 = new Consumer(buff1);
                producer1.start();
                consumer1.start();
        }
}
```

Pozn. Ruční zápis čísel (producent) se samozřejmě stíhá hned vypisovat (konzument).

Synchronized příkazy

Na rozdíl od synchronized metod musejí specifikovat objekt, který poskytne zámek k výlučnému přístupu.

Př.

```
class Konto {
   private int konto;
   private static Object zamek = new Object( );  // vytvoříme objekt zamek, který má zámek
   public int stav() {return konto;}
   public Konto(int i){ konto =i;}
   public void vyber(int kolik) {
        int lokal; //pro zachovani podminek jako u RZ
        synchronized (zamek) {
          try { lokal = konto;                          \
          Thread.sleep(100);///////////////////////////  }   synchronizovaný příkaz
          konto = lokal - kolik;                        /
          }      catch (InterruptedException e) {}
        }
   }
   public void vloz(int kolik) {
        int lokal;
        synchronized (zamek) {
          try { lokal = konto;                          \
          Thread.sleep(300);/////////////////////////   }   synchronizovaný příkaz
          konto = lokal + kolik;                        /
          }      catch (InterruptedException e) {}
        }
} }
```

Zavržené metody starších verzí Javy!!!

final void suspend( ) pozastavení vlákna

final void resume( ) obnovení vlákna

final void stop( ) ukončení vlákna

Důvod zavržení = nebezpečné konstrukce, které snadno způsobí deadlock, když se aplikují na objekt, který je právě v monitoru.
Lze je nahradit bezpečnějšími konstrukcemi s wait a notify tak, že zavedeme např.
- bool. proměnnou se jménem susFlag inicializovanou na false,
- v metodě run suspendovaného vlákna synchronized příkaz tvaru

```
synchronized(this) {
        while (susFlag) { wait( );
        }
}
```

- příkaz suspend nahradíme voláním metody mojeSuspend tvaru

```
void mojeSuspend( ) {
        susFlag = true;
}
```

- příkaz resume nahradíme voláním metody

```
Synchronized void mojeResume( ) {
        susFlag = false;
        notify( );
}
```

Vlákna typu démon

Metodou void setDaemon(Boolean on) ze třídy Thread lze předáním jí hodnoty true určit, že vlákno bude „démonem“. To je třeba udělat ještě před spuštěním vlákna a je to natrvalo. Vlákna, která nejsou démonem jsou uživatelská.
JVM spustí jedno uživatelské vlákno (main). Ostatní vytvářená vlákna mají typ a prioritu toho vlákna –rodiče, ze kterého jsou spuštěna (pokud to nezměníme programově pomocí setPriority, či setDaemon).
JVM provádí výpočet, pokud nenastane jedna z možností

- vyvolání metody exit ze třídy Runtime, která je potomek Object
- všechna uživatelská vlákna jsou ve stavu „dead“, protože buď dokončila výpočet v run metodě nebo se mimo run dostala vyhozením výjimky.

Takže uživatelská vlákna, pokud nejsou mrtvá, **brání JVM v ukončení běhu**.
Vlákno, které je démonem, nebrání JVM skončit. Ukončí se, jakmile žádné uživatelské vlákno neběží. Používá se u aplikací prováděných na pozadí, či nepotřebujících po sobě uklízet.
boolean isDaemon( ) umožňuje testovat charakter vlákna

Skupiny vláken

- Každé vlákno je členem skupiny, což dovoluje manipulovat skupinou jako jedním objektem (např. lze všechny odstartovat jediným voláním metody start.) Skupiny lze implementovat pomocí třídy ThreadGroup z java.lang
- JVM začne výpočet vytvořením ThreadGroup main. Nespecifikuje-li se jiná, jsou všechna vytvářená vlákna členy skupiny main.
- Členství ve skupině je natrvalo
- Vlákno lze začlenit do skupiny např.

```
ThreadGroup mojeSkupina = new ThreadGroup(“jmeno“); // vytvoří skupinu
Thread mojeVlakno = new Thread(mojeSkupina, “ jmeno“); // přiřadí ke skupině

ThreadGroup jinaSkupina = myThread.getThreadGroup(); // vrací jméno skupiny,
                                                     // ke které patří myThread
```

```
// Př. 92Vlakna   Skupiny vláken a démoni
import java.io.IOException;
public class Demon implements Runnable
{
   public static void main(String[] args)
   {
      mainThread = Thread.currentThread();
      System.out.println("Hlavni zacina, skupina=" + mainThread.getThreadGroup());
      Demon d = new Demon();
      d.init();
   }

   private static Thread mainThread;

   public void init()
   {
      try
      {
         // Vytvoří novou ThreadGroup rodicGroup a jejího potomka
         ThreadGroup rodicGroup =
            new ThreadGroup("rodic ThreadGroup");

         ThreadGroup potomekGroup =
            new ThreadGroup(rodicGroup, "potomek ThreadGroup");

         // Vytvoří a odstartuje druhé vlákno
         Thread vlakno2 = new Thread(rodicGroup, this);
         System.out.println("Startuje " + vlakno2.getName() + "...");
         vlakno2.start();

         // Vytvoří a odstartuje třetí vlákno
         Thread vlakno3 = new Thread(potomekGroup, this);
         System.out.println("Startuje " + vlakno3.getName() + "...");
         vlakno3.setDaemon(true);
         vlakno3.start();

         // Vypíše počet aktivních vláken ve skupině rodicGroup
         System.out.println("Aktivnich vlaken ve skupine "
+ rodicGroup.getName() + " = " + rodicGroup.activeCount());

         System.out.println("Hlavni - mam teda skoncit (ENTER) ?");
        System.in.read();
        System.out.println("Enter.....");


         System.out.println("Hlavni konci");
        return;
```

```
    }
   catch (IOException e)
   {
      System.out.println(e);
   }
  }

  // Implementuje Runnable.run()
  public void run()
  {
   long max = 10;
   if (Thread.currentThread().getName().equals("Thread-1"))
            max *= 2;
     for (int i = 0; i < max; i++)
     {
      try {
         System.out.println(Thread.currentThread().getName() + ": " + i);
         Thread.sleep(500);
      }
        catch (InterruptedException ex)
        {
           System.out.println(ex.toString());
        }
        counter++;
     }
     System.out.println("Hlavni alive:" + mainThread.isAlive());

     System.out.println(Thread.currentThread().getName() + "skoncilo vypocet.");
  }

  private int counter = 0;
}
```

Paralelní programování s prostředky java.util.concurrent

Vestavěná primitiva Javy nestačí k pohodlné synchronizaci, protože:

- **Neumožňují couvnout po pokusu o získání zámku, který je zabrán,**
- **„ „ po vypršení času, po který je vlákno ochotno čekat na uvolnění zámku. Tj. nedovolují provést alternativní činnost.**
- **Nelze změnit sémantiku uzamčení s ohledem např. na reentrantnost, ochranu čtení versus psaní.**
- **Neřízený přístup k synchronizaci, každá metoda může použít blok synchronized na libovolný objekt**

```
synchronized ( referenceNaObjekt ) {
     // kritická sekce
}
```

- **Nelze získat zámek v jedné metodě a uvolnit ho v jiné.**

Balík java.util.concurrent poskytuje třídy a rozhraní zahrnující:

Interface Executor

```
public interface Executor {
    void execute(Runnable r);
}
```

Executor může být jednoduchý interface, dovoluje ale vytvářet systém pro plánování, řízení a exekuci množin vláken.

Paralelní kolekce implementující Queue, List, Map.

Atomické proměnné
Třídy pro bezpečnější manipulaci s proměnnými (primitivních typů i referenčních) efektivněji než pomocí synchronizace.

Synchronizační třídy (semafory, bariéry, závory (latches) a výměníky (exchangers)

Zámky
Jejich implementace dovoluje specifikovat timeout při pokusu získat zámek a dělat něco jiného, když není volný.

Nanosekundovou granularitu - Jemnější čas

Následují příklady vybraných prostředků.

//Př. 9ZLock **(Použití třídy ReentrantLock k ošetření rychlostních závislostí)**

Konstruktory má:

ReentrantLock( )
ReentrantLock(boolean fair) instance s férovým chováním při true, nepředbíhá

Metody:

int getHoldCount `()` `kolikrát drží zámek aktuální vlákno`
int getQueueLength `()` `kolik vláken chce tento zámek`
protected Thread getOwner() vrátí vlákno, které vlastní zámek, nebo null
boolean hasQueuedThread(Thread thread) čeká zadané vlákno na tento lock?
…
void lock() zabrání zámku, není-li volný, musí čekat
void unlock() uvolnění zámku
boolean tryLock() zabrání je-li volný, jinak může dělat něco jiného
boolean tryLock(long timeout, TimeUnit unit) zabrání s timeoutem
cca 20 metod

V příkladu jsou dvě verze programu na rychlostní závislosti

1. **RZRL používá lock a unlock k prostému uzamčení kritických sekcí manipulujících s kontem. Což funguje jako dřívější příklad.**
2. **RZL používá tryLock a umožňuje tím provádět náhradní činnost po dobu čekání na zámek.**

Různé rychlosti vláken lze nastavit metodou sleep.

```
// 1. VARIANTA. OŠETŘENÍ KRITICKÝCH SEKCÍ ZÁMKEM PŘI
// BEZHOTOVOSTNÍM NÁKUPU/PRODEJI
import java.util.concurrent.locks.ReentrantLock;

class Konto {
      static int konto = 1000;
      static final ReentrantLock l = new ReentrantLock();
}

class Koupe extends Thread {
  Koupe(String jmeno) {
   super(jmeno);
  }
```

```
  public void run()  { // vstupní bod vlákna
         System.out.println(getName() + " start.");
         int lokal;
         Konto.l.lock(); // zamknutí zámku ze třídy Konto
         try {
           lokal = Konto.konto;
           System.out.println(getName() + " milenkam ");
           sleep(200);//////////////////////////////////////
           Konto.konto = lokal - 200;
           System.out.println(getName() + " ukoncene.");
         }
         catch (InterruptedException e) {}
        finally {Konto.l.unlock();}  // odemknutí zámku ze třídy Konto
  }
}
class Prodej extends Thread {
  Prodej(String jmeno) {
   super(jmeno);
  }
  public void run() { // vstupní bod vlákna
         System.out.println(getName() + " start.");
      int lokal;
      Konto.l.lock();
         try {
           lokal = Konto.konto;
           System.out.println(getName() + " co se da ");
           sleep(2);////////////////////////////////////
           Konto.konto = lokal + 500;
           System.out.println(getName() + " ukoncene.");
         }
         catch (InterruptedException e) {}
         finally {Konto.l.unlock();}
  }
}
class RZRL {
    public static void main (String args[])
         throws InterruptedException {
         System.out.println("Hlavni vlakno startuje");
         Koupe nakup = new Koupe("nakupuji ");
         Prodej prodej = new Prodej ("prodavam ");
         nakup.start();
         prodej.start();
         nakup.join();
         prodej.join();
         System.out.println(Konto.konto);
         System.out.println("Konci hlavni vlakno");
    } }
```

```
// 2. VARIANTA   UMOŽNĚNÍ JINÉ ČINNOSTI, DOKUD JE LOCK ZABRÁN
import java.util.concurrent.locks.ReentrantLock;
class Konto {
      static int konto = 1000;
      static final ReentrantLock l = new ReentrantLock();
}
class Koupe extends Thread {
   Koupe(String jmeno) {
   super(jmeno);
  }

  public void run()  { // vstupní bod vlákna
         System.out.println(getName() + " start.");
         int lokal;
         boolean done = true;
         while (done) {
          if (Konto.l.tryLock()) {
                 try {
                   lokal = Konto.konto;
                   System.out.println(getName() + " milenkam ");
                   sleep(20);//////////////////////////////////////
                   Konto.konto = lokal - 200;
                   done = false;
                   System.out.println(getName() + " ukoncene.");
                 }
                 catch (InterruptedException e) {}
                 finally {Konto.l.unlock();}
          } else {System.out.println("Prozatimni cinnost 1");} // Simulujeme náhradní činnost
         }
  }
}
class Prodej extends Thread {
  Prodej(String jmeno) {
   super(jmeno);
  }
  public void run() { // vstupní bod vlákna
         System.out.println(getName() + " start.");
         int lokal;
         boolean done = true;
         while (done) {
          if (Konto.l.tryLock()) {
                 try {
                   lokal = Konto.konto;
                   System.out.println(getName() + " co se da ");
                   sleep(50);///////////////////////////////////
                   Konto.konto = lokal + 500;
                   done = false;
```

```
                    System.out.println(getName() + " ukoncene.");
                  }
                  catch (InterruptedException e) {}
                  finally {Konto.l.unlock();}
         } else {System.out.println("Zatimni cinnost 2");} // Simulujeme náhradní činnost
        }
  }
}

class RZL {
     public static void main (String args[])
          throws InterruptedException {
          System.out.println("Hlavni vlakno startuje");
          Koupe nakup = new Koupe("nakupuji ");
          Prodej prodej = new Prodej ("prodavam ");
          nakup.start();
          prodej.start();
          // nebo záměnou prodej.start(); nakup.start(); když chceme ukázat provádění
          // prozatímní činnosti 1
          nakup.join();
          prodej.join();
          System.out.println(Konto.konto);
          System.out.println("Konci hlavni vlakno");
     }
}
```

Př. 9ZSemafor (použití třídy Semaphore, která dovoluje přístup k n zdrojům)

Konstruktor má možné tvary

Semaphore(int povoleni)	**povolení udávají počet zdrojů**
Semaphore(int povoleni, boolean f)	**f určuje fér chování = FIFO obsluha je zaručena. Implicitně je f false.**

K získání povolení = přístup ke zdroji slouží metody:

void acquire( )	**pro jedno povolení**
void acquire(int povoleni)	**pro více povolení = zabrání více zdrojů**

Tyto metody blokují vlákno, dokud počet povolení = zdrojů není k dispozici, nebo dokud čekající vlákno není přerušeno vyhozením InterruptedException.

acquireUninterruptibly( )
acquireUninterruptibly(int povoleni)
Jejich vlákna jsou ale pozastavena a nepřerušitelná až do získání potřebného počtu povolení. Případné požadované přerušení se projeví až po získání povolení.

release()	**uvolní 1 povolení**
release(int povoleni)	**uvolní zadaný počet povolení**

K zabrání povolení, je-li zjištěn potřebný počet volných a nezablokování exekuce vlákna, máme boolean metody:

tryAcquire( )	**Hodnotou je true, je-li volné povolení, jinak false**
tryAcquire(int povolení)	**Hodnotou je true, je-li k dispozici postačující počet**
tryAcquire(long timeout, TimeUnit unit)	**čekají zadaný čas než to vzdají**
tryAcquire(int povolení, long timeout, TimeUnit unit)	

Př. čekání 10 sec. na jedno povolení
boolean z = tryAcquire(10, TimeUnit.SECONDS)

Př. Dražba. 100 zákazníků chce nakupovat. Cenu určují prodavači-odhadci. Jsou jen 2 (určení ceny je simulované Random), takže všem zákazníkům nestihnou odhadnout nebo je cena jednotná = méně výhodná. Za jednotnou musí kupovat ti zákazníci, kteří se nedostali k odhadci.

```
import java.util.concurrent.*;
import java.util.*;

public class SemaforTest {
  private static final int LOOP_COUNT = 100;          // 100 zákazníků
  private static final int MAX_AVAILABLE = 2;        // dva prodavači
  private final static Semaphore semaphore =
    new Semaphore(MAX_AVAILABLE, true);

  private static class Pricer {                        // určení ceny
    private static final Random random = new Random();
    public static int getGoodPrice() {                 // buď odhadcem
      int price = random.nextInt(100);
      try {
        Thread.sleep(50);                              // určit cenu trvá nějaký čas
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      }
      return price;                                    // cena určená odhadcem
    }
    public static int getBadPrice() {
      return 100;                                      // nevýhodná jednotná cena
    }
  }

  public static void main(String args[]) {
    for (int i=0; i<LOOP_COUNT; i++) {       // vytvoření a spuštění 100 vláken=zákazniků
      final int count = i;
      new Thread() {
        public void run() {
          int price;
          if (semaphore.tryAcquire()) {              // prodavač je volný, určí cenu
            try {
              price = Pricer.getGoodPrice();
            } finally {
              semaphore.release();                   // uvolnění prodavače
            }
          } else {                                   //prodavač není volný, pak náhradní řešení
            price = Pricer.getBadPrice();
          }
          System.out.println(count + ": " + price);   // tisk č. zákazníka a cena
        }
      }.start();
    }
  }
}
```

Př. 9ZSoucet (použití třídy CyclicBarrier)

Dovoluje čekání množiny vláken na sebe navzájem před pokračováním výpočtu. Nazývá se cyklická, protože může být znovu použita po uvolnění čekajících vláken.

Obvykle je použita, když úloha je rozdělena na podúlohy takové, že každá z nich může být prováděna separátně.

Má dvě podoby:
CyclicBarrier(int ucastnici) — **účastníci určují počet podúloh = vláken**
CyclicBarrier(int ucastnici, Runnable barierovaAkce) — **akce se provede po spojení všech vláken, ale před jejich další exekucí**

Má metody:
await( ) — **čeká, až všichni účastníci vyvolají await na této bariéře**
await(long timeout, TimeUnit unit) — **čeká, dokud buď všechny vyvolají await nebo nastane specifikovaný timeout**
getNumberWaiting( ) — **vrací počet čekajících na bariéru**
getParties( ) — **vrací počet účastníků procházejících touto bariérou**
isBroken( ) — **vrací true, je-li bariera porušena timeoutem, přerušením, resetem, výjimkou**
reset( ) — **resetuje barieru po brake**

V příkladu je dána celočíselná matice a chceme sečíst všechny její prvky.
V multiprocesorovém prostředí bude vhodné rozdělit ji na části, sčítat je samostatně a pak sečíst výsledky. Bariéra zabrání sečtení parciálních součtů před jejich kompletací.

```
import java.util.concurrent.*;

  public class Soucet {
    private static int matrix[][] = {
      {1, 1, 1, 1, 1},
      {2, 2, 2, 2, 2},
      {3, 3, 3, 3, 3},
      {4, 4, 4, 4, 4},
      {5, 5, 5, 5, 5}
    };
    private static int results[];
```

```
 private static class Summer extends Thread { // Jeji instance provedou částečné součty
  int row;
  CyclicBarrier barrier;                        // Vlákna této třídy pracují s bariérou

  Summer(CyclicBarrier barrier, int row) {    // To je konstruktor sumátoru
    this.barrier = barrier;
    this.row = row;
  }
  public void run() {                  // aktivita vlákna pro částečný součet
    int columns = matrix[row].length;  // řádky připouští různé počty sloupců
    int sum = 0;
    for (int i=0; i<columns; i++) {          // provedení částečného součtu řádku row
      sum += matrix[row][i];
    }
    results[row] = sum;
    System.out.println(
      "Vysledek pro radek " + row + " je : " + sum);
    try {                                    // čekání na ostatní účastníky
      barrier.await();
    } catch (InterruptedException ex) {
      ex.printStackTrace();
    } catch (BrokenBarrierException ex) {
      ex.printStackTrace();
    }
  }
}              // konec Summer = sumátoru pro částečné součty

public static void main(String args[]) {
  final int rows = matrix.length;          // tj. 5 řádků
  results = new int[rows];
  Runnable merger = new Runnable() {  // Definice bariérové akce, splynuti část. součtu
    public void run() {
      int sum = 0;                           // která sečte částečné součty
      for (int i=0; i<rows; i++) {
        sum += results[i];
      }
      System.out.println("Celkovy vysledek je " + sum);
    }
  };
  CyclicBarrier barrier = new CyclicBarrier(rows, merger); // Vytvoření bariéry
              // rows = počet účastníků, tj. řádek, merger je bariérová akce
  for (int i=0; i<rows; i++) {               // Vytvoření a spuštění vláken pro
    new Summer(barrier, i).start();          // částečné součty. Konstruktor Summeru dá
  }                                          // každému vláknu bariéru barrier a číslo řádky
  System.out.println("Cekani když není k tisku zatim zadny soucet");
 }
}
```

Př.9ZZavora Použití třídy CountDownLatch

Synchronizační prostředek, který dovoluje vláknu/vláknům čekat, až se dokončí operace v jiných vláknech.
Inicializuje se se zadaným čítačem, který je součástí konstruktoru a funguje obdobně jako počet účastníků v CyclicBarrier konstruktoru. Určuje, kolikrát musí být vyvolána metoda countDown.
Po dosažení zadaného počtu jsou všechna vlákna čekající v důsledku vyvolání metody await uvolněna k exekuci.

Metody:
void await( ) způsobí čekání volajícího vlákna až do vynulování čítače
boolean await(long timeout, TimeUnit unit) čekání se ukončí i vyčerpáním času.
void countDown( ) dekrementuje čítač a při 0 uvolní všechna čekající vlákna.
long getCount( ) vrací hodnotu čítače

String toString( ) vrací řetězec identifikující závoru a její stav (čítač).

Příklad vytvoří množinu vláken, nedovolí ale žádnému běžet, dokud nejsou všechna vlákna vytvořena.
Vlákno main čeká na závoře, až skončí všech 10 vláken.

Tento příklad by se dal řesit i jinak, třeba joinem.

```
import java.util.concurrent.*;

public class ZavoraTest {
  private static final int COUNT = 10;
  private static class Worker implements Runnable {// třída Worker má dvě závory
    CountDownLatch startLatch;
    CountDownLatch stopLatch;
    String name;
    Worker(CountDownLatch startLatch,         // konstruktor s formálními jmény závor
        CountDownLatch stopLatch, String name) {
      this.startLatch = startLatch;
      this.stopLatch = stopLatch;
      this.name = name;
    }

    public void run() {        // Metoda run třídy Worker
      try {
        startLatch.await();    //* tady se hned vlákna zastaví a poběží až po **
      } catch (InterruptedException ex) {
        ex.printStackTrace();
      }
      System.out.println("Bezi: " + name);
      stopLatch.countDown();       // dekrementace čítače závory na konci vlákna
    }
  }              // konec třídy Worker

  public static void main(String args[]) {
    CountDownLatch startSignal = new CountDownLatch(1);/ / vytvoří závoru, čítač = 1
    CountDownLatch stopSignal  = new CountDownLatch(COUNT); //čítač stopsignal = 10
    for (int i = 0; i < COUNT;  i++) {
      new Thread(
        new Worker(startSignal, stopSignal, //vytvoří 10 vláken a spustí je, ty se ale v místě *
          Integer.toString(i))).start();          // hned zastaví. Jména vláken jsou 0, 1, ..,9
    }
    System.out.println("Delej");   // Provede se po vytvoření všech 10 vláken
    startSignal.countDown();                   // startSignal závora se vynuluje  **
    try {
      stopSignal.await();      // vlákno main čeká, až všech 10 vláken skončí
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
    System.out.println("Hotovo");
  }
}
```

Př. Výměník – použití třídy Exchanger <V>

Třída Exchanger <V> umožňuje komunikaci vláken předáváním objektu V.

K předání objektů je volána metoda *exchange*, která je obousměrná, vlákna si navzájem předají data. K výměně dojde, když obě vlákna již dosáhla místo, kde volají metodu exchange.

Typickým příkladem použití je úloha producenta konzumenta. Nekomunikují ale plněním a vyprazdňováním jednoho (kruhového) bufferu, ale vymění si buffery celé (prázdný za plný a opačně).

```
import java.util.*;
import java.util.concurrent.*;

  public class VymenikTest {

    private static final int FULL = 10;       // délka bufferu
    private static final int COUNT = FULL * 12; // počet dat je 120
    private static final Random random = new Random();
    private static volatile int sum = 0;
    private static Exchanger<List<Integer>> exchanger =   // předává se
      new Exchanger<List<Integer>>();                // seznam celých čísel
    private static List<Integer> initiallyEmptyBuffer; // 2 vyměňované
    private static List<Integer> initiallyFullBuffer;        // buffery
    private static CountDownLatch stopLatch =  // závora s čítačem 2
      new CountDownLatch(2);

    private static class FillingLoop implements Runnable { // to je Producent
      public void run() {
        List<Integer> currentBuffer = initiallyEmptyBuffer;
        try {
          for (int i = 0; i < COUNT; i++) {
            if (currentBuffer == null)
              break; // stop na null
            Integer item = random.nextInt(100); // producent generuje náhodná čísla
            System.out.println("Produkovano: " + item);
            currentBuffer.add(item); // add, remove, isEmpty jsou v java.util.concurrent
            if (currentBuffer.size() == FULL)  // je-li plný, volej
              currentBuffer =                          // exchange
                exchanger.exchange(currentBuffer);
          }
        } catch (InterruptedException ex) {
          System.out.println("Vada exchange na strane producenta");
        }
```

```
      stopLatch.countDown();                    // dekrementuje čítač závory
    }
  }

  private static class EmptyingLoop implements Runnable { // to je Konzument
    public void run() {
      List<Integer> currentBuffer = initiallyFullBuffer;
      try {
        for (int i = 0; i < COUNT; i++) {
          if (currentBuffer == null)
            break; // stop na null
          Integer item = currentBuffer.remove(0);
          System.out.println("Konzumovano " + item);
          sum += item.intValue(); // aby konzument něco dělal, sčítá konzumované položky
          if (currentBuffer.isEmpty()) { // volej exchange při prázdném
            currentBuffer =
              exchanger.exchange(currentBuffer);
          }
        }
      } catch (InterruptedException ex) {
        System.out.println("Vada exchange u konzumenta");
      }
      stopLatch.countDown();                    // dekrementuje čítač závory
    }
  }

  public static void main(String args[]) {
    initiallyEmptyBuffer = new ArrayList<Integer>(); // vytvoř buffery
    initiallyFullBuffer = new ArrayList<Integer>(FULL);
    for (int i=0; i < FULL; i++) {              // naplnění bufferu
      initiallyFullBuffer.add(random.nextInt(100));
    }
    new Thread(new FillingLoop()).start();       // vytvoř a spusť producenta
    new Thread(new EmptyingLoop()).start();   // vytvoř a spusť konzumenta
    try {
      stopLatch.await();                // čekání na závoře
    } catch (InterruptedException ex) {
        ex.printStackTrace();
    }
    System.out.println("Soucet vsech polozek je " + sum);
  }
}
```

Skriptovací jazyky

Použití tradičních jazyků

•Pro vytváření programů „z gruntu“
•Obvyklé je použití v týmu
•Velké, řádně specifikované a výkonnostně vyspělé aplikace

} mělo by to tak být, ale jako vždy skutečnost je trochu jiná

Použití skriptovacích jazyků

•K vytváření aplikací z předpřipravených komponent
•K řízení aplikací, které mají programovatelný interface
•Ke psaní programů, je-li rychlost vývoje důležitější než efektivita výpočtu
•Jsou nástrojem i pro neprofesionální programátory

Vznik 70 léta Unix „shell script“´= sekvence příkazů čtených ze souboru a prováděných jako by z klávesnice ostatní to převzali (AVK script, Perl script, …)

Skript = textový soubor určený k přímému provedení (k interpretaci)

Vlastnosti

•Integrovaný překlad a výpočet
•Nízká režie a snadné použití (např. impl.deklarace)
•Zduřelá fukčnost ve specif. oblastech (např. řetězce a regul. výrazy)
•Není důležitá efektivita provedení (často se spustí je 1x)
•Nepřítomnost sekvence překlad-sestavení-zavedení

Tradiční použití

•Administrace systémů (sekvence shell příkazů ze souboru)
•Vzdálené řízení aplikací (dávkové jazyky)

Nové použití

•Visual scripting = konstrukce grafického interface z kolekce vizuálních objektů (buttons, text, canvas, back. a foregr. colours, …). (Visual Basic, Perl)
•Lepení vizuálních komponent (např spreadsheet do textového procesoru)
•Dynamické Web stránky = dynamické řízení vzhledu Web stránky a jednoduchá interakce s uživatelem, která je interpretována prohlížečem
•Dynamicky generované HTML = část nebo celá HTML je generována skriptem na serveru. Používá se ke konstrukci stránek, jejichž obsah se vyhledává z databáze.

Prostředky Web skriptů: VBScript, JavaScript, JScript, Python, Perl, …

Python - vlastnosti

Název Monty Python's Flying Circus

Všeobecné vlastnosti

Základní vlastností je snadnost použití

Je stručný

Je rozšiřitelný (časově náročné úseky lze vytvořit v C a vložit)

Lze přilinkovat interpret Pythonu k aplikaci napsané v C

Má vysokoúrovňové datové typy (asociativní pole, seznamy), dynamické typování

Mnoho standardních modulů (pro práci se soubory, systémem, GUI, URL, reg.výrazy, …)

Strukturu programu určuje odsazování

Proměnné se nedeklarují, má dynamické typy

Python - spouštění

Spuštění

- **V příkazovém okně zápisem *python* (musí být cesta na python.exe a pro nalezení programových modulu naplněna systémová proměnná PYTHONPATH. Implicitne vidi do direktory kde je spuštěn). Např.**

cmd → cd na direktory se soubory .py → c:\sw\python25\python.exe → import P1fibo → P1fibo.fib(5) Ukončení Pythonu exit() nebo Ctrl-Z a odřádkovat

interaktivní zadávání příkazů lze provádět ihned za promt >>> tzv.interaktivní mód

- **Nebo příkazový mód c:\sw\python25>skript.py**
- **Alternativou je použití oken IDLE (Python GUI) po jeho spuštění se objeví oknoPythonShell ve tvaru:**

```
Python 2.5 (r25:51908, Sep 19 2006, 09:52:17) [MSC v.1310 32 bit (Intel)] on win32
Type "copyright", "credits" or "license()" for more information.
    ****************************************************************
        ...
    ****************************************************************
 IDLE 1.2
>>>
```

V jeho záhlaví vybereme File → NewWindow → vznikne okno Untitlet

V záhlaví Untitled vybereme File→Open a otevřeme např P1fibo.py soubor, pak Run→RunModule

>>>fib(5)

V oknech lze editovat, před spuštěním ale ulož.

Všimněte si rozdílu spuštění modulu P1fibo - definuje funkce, které musím vyvolat
P2cita – je přímospustitelný kód >>>import P2cita.py

Python - čísla

Interaktivní použití připomíná komfortní kalkulačku s proměnnými, typy a funkcemi. Napr.

```
>>>print "Monty Python's " + " Flying Circus"   # pro řetězce lze použít " i  ' ale v párech
Monty Python's Flying Circus
>>> print 'Ahoj' + 1    #  má silný typový systém
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
>>> print 15/2          #celočíselné
7
>>> print 15/2.0        #reálné,                pro získání zbytku po dělení je operátor %
7.5
>>> vyska = sirka = 10.5 ** 2   #násobné přiřazení, umocňování
>>> plocha = vyska*sirka        #jména jsou case sensitive
>>> plocha*(5.5+12)
212713.59375
>>> print "Soucet cisla %d a cisla %d je: %d" % (vyska, sirka, vyska+sirka) #formátování
     najdete v textu
Soucet cisla 110 a cisla 110 je: 220
>>> Imag = (1 - 1J)*-1j                              # umí komplexní čísla
>>> Imag
(-1-1j)
```

Python - operátory

Python automaticky převádí celé číslo na něco, čemu se říká *velké celé číslo* (*long integer*; nezaměňujte s typem long v jazycích C a C++, kdy se číslo ukládá na 32 bitech). Jde o specifickou vlastnost jazyka, která umožňuje pracovat s celými čísly o prakticky neomezené velikosti. Platíme za to cenou mnohem pomalejšího zpracování.

```
>>> 123456789123456789 * 123456789
15241578765432099750190521L
```

```
>>> x += 1 #je použitelný zkratkový zápis výrazů M += N, M -= N, M *= N, M /= N, M %= N
```

Relační operátory ==, >, <, != nebo <>, <=, >=

Booleovské hodnoty True, False

Booleovské operace and, or, not

Artmetické operátory zahrnují mocnění

```
>>> 2**2**3            pozor, je pravoasociativní
256
>>> (2**2)**3
64
```

Python - sekvence

Sekvence jsou uspořádané množiny prvků

Zahrnují 8bitové řetězce, unikódové řetězce, seznamy a n-tice

Všechny sekvence S sdílí společnou množinu funkcí:

- len(S) počet prvků
- max(S) největší prvek
- min(S) nejmenší prvek
- x in S test přítomnosti x v S
- x not in S test nepřítomnosti
- S1 + S2 konkatenace dvou sekvencí
- tuple(s) konverze členů S na n-tici
- list(S) konverze S na seznam
- S*n nová sekvence tvořená n kopiemi S
- S[i] i-tý prvek S, počítá se od 0
- S[i:j] část sekvence S začínající i-tým prvkem až do j-tého bez něj

Pár př.

t = tuple('100') ⇒ t = ('1', '0', '0')

l = list('100') ⇒ l = ['1', '0', '0']

Pozn. Příklady lze do pythonu překopírovat i rovnou z PowerPointu

Python - řetězce

Řetězce

```
>>> print 'vlevo '+ ('a vlevo' * 3)              #operátor  '+' zřetězuje  a ' * ' opakuje
 vlevo a vlevo a vlevo a vlevo
>>> s1 = 'ko '                                   #má řetězcové proměnné
>>> s2 = u'dak '                                  # u je to v Unicode
>>> print s1*3 + s2
ko ko ko dak
>>>s1=str(20)                   # pro převod jakéhokoliv objektu na řetězec
>>> s1
'20'
>>> unicode('\x51\x52\x53')     # pro převod na Unicode řetězce hexad. znaků
u'QRS'
>>> s = 'CSc'
>>>print s.find('Sc')                             # vyhledání v textu
1                                                   najde začátek podřetězce
>>>
print s.replace('CSc', 'PhD')    # náhrada textu
'PhD'
>>>'co to tady je'.split()       # split vrací seznam řetězců, může mít parametr=vypouštěný řetěz
['co', 'to', 'tady', 'je']
```

A mnoho dalších funkcí

Python - řetězce

```
>>> s1 = 'to je moje chyba‘        #má řezy a výřezy řetězců, čísluje od 0
>>> s1[1:3]
'o ‘                          je to o a mezera
>>> s1[3:]                    # od třetího dál
'je moje chyba'
>>> s1[:3]
'to ‘                         t, o a mezera
>>> s1[3]
'j'
>>> s1[0] = ‘6’                         #je chyba, 'str' object does not support item assignment
>>> len(s1)                             # vrací délku řetězce
16
```

#Řetězce přes více řádek píšeme mezi ““““ ““““

```
>>>print"""ssssssssssssssssssssssssjjjjjjjjjjjjjjjjjjjjjjjj
jjjjjjjjjjjjjjjjjjjjjjjj"""
ssssssssssssssssssssssssjjjjjjjjjjjjjjjjjjjjjjjj
jjjjjjjjjjjjjjjjjjjjjjjj
>>>
```

Python – kolekce – seznamy

kolekce v Pythonu nemusí mít prvky téhož typu, zahrnují seznamy, n-tice, slovníky, pole, množiny

Konstruktorem seznamů jsou [ ]

```
>>> seznam = []
>>> jinySeznam = [1, 2, 3]
>>> print jinySeznam
[1, 2, 3]
>>> print jinySeznam[2]          # čísluje od 0
3
>>> jinySeznam[2] = 7            # oproti řetězci je seznam modifikovatelný (mutable)
>>> print jinySeznam             # což znamená, že může měnit své prvky
[1, 2, 7]
 >>> print jinySeznam[-1]        # Záporné indexy indexují od konce a  stejně tak je to i u řetězců
7


>>> r='12345'
>>> r[-2]
'4'
>>>
```

Python – kolekce - seznamy

append je operátor (metoda), která připojuje svůj argument na konec seznamu

```
>>> seznam.append("neco")            #pokračujeme s hodnotami z předchozího slidu
>>> print seznam
['neco']
>>> seznam.append(jinySeznam) #appendovat lze libovolny objekt
>>> print seznam
['neco', [1, 2, 7]]
>>> print seznam[1][2] #v seznamech lze indexovat
7
>>> adresy = [['Mana', 'Karlov 5', 'Plzen', '377231456'],
      ['Pepina', 'V ZOO 42', 'Praha', '222222222']]
>>> adresy[1][2]
'Praha'
>>> del seznam[1]  #odstraneni indexem zadaneho prvku
>>> print seznam
['neco']
>>> seznam.append("5")
print seznam
['neco', 5]
```

Python – kolekce - seznamy

Spojení seznamů provedeme operátorem +

```
>>> novySeznam = seznam + jinySeznam
>>> print novySeznam
['neco', 1, 2, 7]
>>> print [1,3,5,7].index(5)      #lze zjistit index prvku
 2
>>> len(adresy)      #lze zjistit delku seznamu
2
```

Další metody pro práci se seznamy

extend(L) přidá na konec prvky seznamu L ; dtto s[len(s):] = L

insert(i, x) vloží prvek i na pozici x

remove(x) odstraní první výskyt prvku x v seznamu

pop(i) odstraní prvek na pozici i a vrátí jeho hodnotu

pop() odstraní poslední prvek a vrátí jeho hodnotu

count(x) vrátí počet prvků s hodnotou x

sort() seřadí prvky dle velikosti (modifikuje seznam), výsledek nevrací

reverse() obrátí ho, nevrací výsledek

```
>>> L=['co', 'to', 'tady', 'je']
>>> L.reverse()
>>> L
['je', 'tady', 'to', 'co']
```

Python – kolekce - seznamy

```
>>> s = [1,2,3]
>>> s
[1, 2, 3]
>>> s[len(s):] = [9,8,7]   # části s od indexu: včetně nahradí zadaným seznamem
>>> s
[1, 2, 3, 9, 8, 7]
>>> s[1:3]  # lze dělat řezy a výřezy v seznamech jako v řetězcích
[2,3]
>>> s = [6,3.4,'a']
>>> s.sort()
>>> s
[3.3999999999999999, 6, 'a']
>>> s = [1,2,3,4,5]
>>> s.pop(1)
2
>>> s.pop(-1)
5
>>> s.pop()
4
>>> s=[1.1,2.2,3.3,4.4,5.5]
>>> del s[2]
>>> s
[1.1000000000000001, 2.2000000000000002, 4.4000000000000004, 5.5]
```

Python – kolekce - množina

Množiny nepatří mezi sekvence protože nejsou uspořádané. Jsou ale založeny na seznamech, prázdná množina je prázdný seznam

```
>>> import sets       # Set je v modulu sets, od verze 2.5 je součástí jazyka jako set
>>> M=sets.Set()
>>> N=sets.Set(['a',2,3])         # Set provede převedení seznamu na množinu
>>> O=sets.Set([1,2,3,4])
>>> U= N.union(O)
>>> U
Set(['a', 1, 2, 3, 4])
>>> U.intersection(N)
Set(['a', 2, 3])
>>> U.intersection(N) == N
True
>>> sets.Set([2,3]).issubset(N)             # test je-li podmnožinou N.   [2,3] se musí konvertovat
True
>>> U.issuperset(N)
True
>>> 2 in O
True
```

Python – kolekce – N-tice

N-tice Pythonu je posloupnost hodnot uzavřená do ( ), lze s ní nakládat jako s celkem.

Po vytvoření nelze n-tici měnit (oproti seznamům jsou „immutable“)

Na jejich prvky lze odkazovat indexem

```
>>> ntice = (1, 2, 3, 4, 5)
>>> print ntice[1]
 2
>>> ntice1 = (1, 2, 3)
>>> ntice2 = ntice1 + (4,)      # čárka způsobí, že se zápis chápe jako n-tice a ne jako číslo
>>> print ntice2
(1, 2, 3, 4)
>>> 2 in (1,2,3)
True
>>> len((1,2,3))
3
>>> max((1,2,3,4,5,(2,2)))
(2, 2)
>>> min((1,2,3,4,5))
1
```

Python – kolekce – slovníky (dictionary)

Jsou to asociativní pole. Položky jsou zpřístupněny pomocí klíčů immutable typu, hodnotou může být libov. typ

Realizují se hash tabulkami. Konstruktorem jsou { }.

Konstruktorem je také dict([dvojice,dvojice, …])

```
>>> dct = {}              # na pocatku třeba mame prazdnou tabulku
>>> dct['bubu'] = ‘klicem je retezec a hodnota je take retezec’ # do ni muzeme vkladat
>>> dct['integer'] ='Celé číslo‘
>>> print dct['bubu']
klicem je retezec a hodnota je take retezec
>>> dct[3]=44.5
>>> print dct[3]
44.5
>>>adresy = {           # naplnme oba sloupce slovníku
          'Mana' : ['Karlova 5', 'Plzen', 377123456],
          'Blanka' : ['Za vsi', 'Praha', 222222222] }
>>> print adresy ['Mana']
['Karlova 5', 'Plzen', 377123456]
>>> mesto = 1
>>> print adresy['Blanka'] [mesto]          # můžeme indexovat v části hodnota (ta je seznamem)
Praha
>>>
```

Python – kolekce – slovníky (dictionary)

```
>>>adresy.get('Mana')              # zjistí hodnotovou část
['Karlova 5', 'Plzen', 377123456]
>>> adresy.has_key('Blanka')
True
>>> adresy.values()
[['Karlova 5', 'Plzen', 377123456], ['Za vsi', 'Praha', 222222222]]
>>> dct.update(adresy)             #spoji slovniky dct a adresy
>>>a =dct.copy()                   #nakopiruje dct do a, stačí napsat a = dct
>>>len(a)
4
>>> del adresy['Mana']             #odstraneni polozky s danym klicem
>>> print adresy.keys()            #tiskne seznam vsech klicu
['Blanka']
>>> adresy.clear()                 #vyprazdni slovnik adresy
>>> adresy.items ()                #navraci seznam vsech polozek slovniku adresy
[]
>>> dct.items()
[('bubu', 'klicem je retezec a hodnota je take retezec'), ('integer', 'Cel\xe9 \xe8\xedslo'), (3, 44.5), ('Blanka', ['Za
    vsi', 'Praha', 222222222]), ('Mana', ['Karlova 5', 'Plzen', 377123456])]
```

Python – kolekce – slovníky (dictionary)

Operace na slovnících

len(D)	vrací počet položek slovníku D
D[k]	vrací hodnotu s klíčem k, neexistuje-li vyvolá exception
D[k] = v	dosadí do položky s klíčem k hodnotu v
del D[k]	odstraní položku s klíčem k, neexistuje-li vyvolá exception
D.has_key(k)	true, když tam klíč k je
D.items()	seznam (klíč, hodnota) z D
D.keys()	seznam všech klíčů z D
D.values()	seznam všech hodnot z D v pořadí jako D.keys()
D.update(E)	spojí slovníky E a D, při stejných klíčích použije hodnoty z E
D.get(k, x)	vrací hodnotu D[k] , neexistuje-li, vrací x nebo None při neuvedení
D.setdefault(k [, x])	-----------------“-----------------“----------------, a dosadí ho do D[k]
D.iteritems()	vrací iterátor nad dvojicemi (klíč, hodnota) D
D.iterkeys()	-------“-------------- klíči D
D.itervalues	-------“-------------- hodnotami D
D.popitem()	vrátí a odstraní z D libovolnou položku. Při prázdném D vyvolá exception
x in D	True je-li x klíčem v D
x not in D	obráceně

Python – kolekce – slovníky (dictionary)

Př.

```
>>> D= {1:'jedna', 2:'dva', 3:'tri', 4:'ctyri'}
>>> D.popitem()
(1, 'jedna')
>>> for l in D.itervalues(): print l

dva
tri
Ctyri
>>> for l in D.iterkeys(): print l

2
3
4
>>>for l in D.iteritems(): print l

(2, 'dva')
(3, 'tri')
(4, 'ctyri')
```

Python – zásobník

Není tam explicitně typ zásobník, ale snadno se realizuje seznamem

Pomocí append() přidáme na vrchol

a pomocí pop() odebereme z vrcholu

```
>>> stack = []
>>> stack.append(1)
>>> stack.append('dva')
>>> stack.append(3)
>>> stack.pop()
3
>>> print stack.pop()
dva
>>>
```

Python - fronta

Explicitně není, ale snadno se realizuje zásobníkem

Pomocí append() přidáme na konec fronty

a pomocí pop(0) odebereme z čela fronty

```
>>> queue = ['prvni', 'druhy', 'treti']
>>> queue.append("posledni")
>>> vylezlo = queue.pop(0)
>>> print vylezlo
prvni
>>>
```

Python - pole

Existuje modul **array.**

Jeho prvky mohou být jen základních typů (znakové, integer, real)

Používá se málo, nahrazuje se vestavěným typem **seznam.**

Python -cykly

for in konstrukcí lze cyklovat přes vše co lze indexovat. Pozor na mezery

```
>>> for i in range(1, 20):
        print "%d x 120 = %d" % (i, i*120)          #co to tiskne? 20 radku tvaru 1 x 120 = 120
>>> for znak in ' retezec ': print znak
>>> for slovo in ('jedna', ‘dva', 'dva', ‘tri', ‘nic'): print slovo
>>> for prvek in ['jedna', 2, 'tri']: print prvek
```

Nechť seznam obsahuje odkazy na objekty 1,2,3,4, při jeho změně tam lze dát jiná čísla

```
mujSeznam = [1, 2, 3, 4]
for index in range(len(mujSeznam)):
   mujSeznam[index] += 1
   print mujSeznam              #tiskne [2, 2, 3, 4] pak [2, 3, 3, 4], pak [2, 3, 4, 4] a [2, 3, 4, 5],
```

Enumerate zajistí, že při každém průběhu máme dvojici index, hodnota

```
mujSeznam = [9, 7, 5, 3]
for index, hodnota in enumerate(mujSeznam):
     mujSeznam[index] = hodnota + 1
     print mujSeznam                              #tiskne stejny efekt jako predchozi
```

Řezy a výřezy jsou také dovoleny

```
for x in mujSeznam[2:] : print x                  #tiskne 6 a pak 4
```

Python - cykly

```
j = 1
while j <= 12:                              odděluje řídící řetězec od výrazů
   print "%d x 12 = %d" % (j, j*12)         # d=c.cislo, f=desetinne, e=s exponentem, s=retezec
   j = j + 1
else: print 'konec'     # else cast je nepovinna
```

Při vnořování pozor na správné odsazování

```
for nasobitel in range(2, 13):
     for j in range(1, 13):
            print "%d x %d = %d" % (j, nasobitel, j*nasobitel)
```

break jako v C přeruší nejblíže obepínající cyklus

continue pokračuje sdalší iterací cyklu

Cyklus while i for může mít else část, která se provede, jeli cyklus skončen (nesmí ale skončit break)

větvení

```
if j > 10:
    print "Toto se mozna vytiskne"
elif j == 10:                    #těch elif částí může být libovilně
    print "Toto se pak nevytiskne"
else:                            #else část je nepovinná
    print "Toto se mozna nevytiskne"

seznam = [1, 2, 3, 0, 4, 5, 0]          #program nic.py
index = 0
while index < len(seznam):
   if seznam[index] == 0:
      del seznam[index]
   else:
      index += 1
print seznam
```

Prázdný příkaz

```
pass
```

Python – precedence operátorů

x **y mocnina

-x, ~x minus, jedničkový komplement

*, /, % krát, děleno, modulo

+, -

x<<y, x>>y posun o y bitů vlevo, vpravo

x&y and po bitech

x^y exklusivni or po bitech

x|y or po bitech

<, <=, ==, >=, !=, x in y, x not in y, x is y, x is not y porovnání , testy členství a identity

not x booleovské not

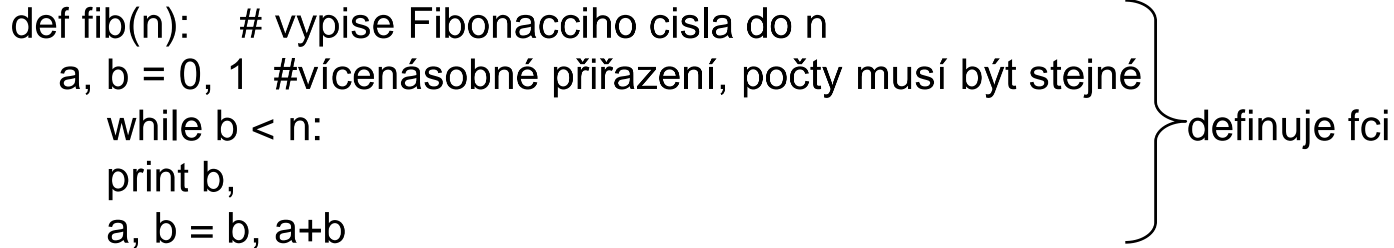
x and y booleovské and

x or y booleovské or

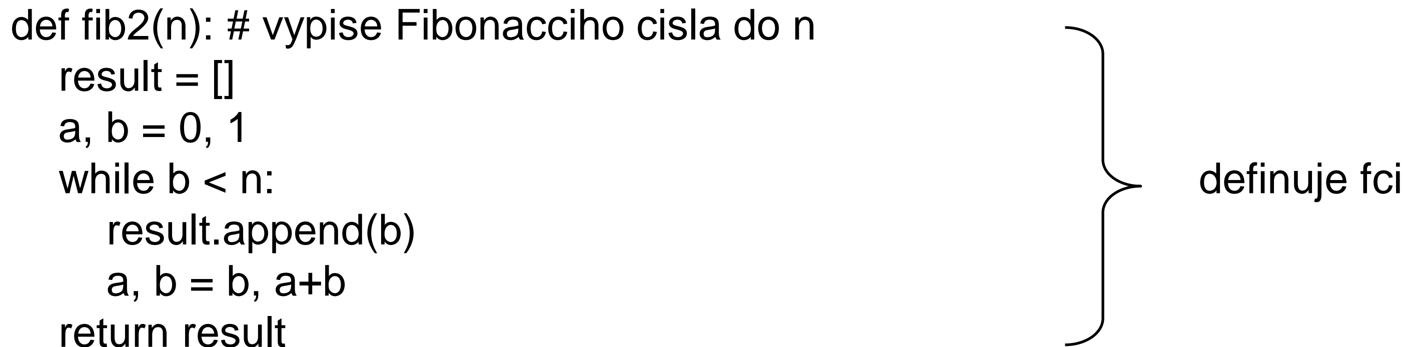
Python – funkcionální programování

Definice funkcí

Fibonacciho čísla – modul   - soubor P1fibo.py




```
def fib(n):    # vypise Fibonacciho cisla do n
    a, b = 0, 1  #vícenásobné přiřazení, počty musí být stejné
        while b < n:
        print b,
        a, b = b, a+b
```




```
def fib2(n): # vypise Fibonacciho cisla do n
    result = []
    a, b = 0, 1
    while b < n:
        result.append(b)
        a, b = b, a+b
    return result
```

```
>>> def nic():
      pass  # to je prazdny prikaz

>>> print nic()  #Funkce, které nemají return mají hodnotu None
None
```

Python – funkcionální programování

Definice fcí jsou sdružovány do modulů (modules)

Moduly představují samostatné prostory jmen.

Interpret příkazem import shellu nebo spuštěním v IDLE se modul zavede

```
>>> import P1fibo                # objekty z modulu se ale musi psat    P1fibo.fib(3)
```

nebo

```
>>> from P1fibo import fib       # objekt fib se jiz nemusi kvalifikovat
```

nebo

```
>>> from P1fibo import *         # objekty z modulu se nemusi kvalifikovat
```

nebo

```
>>> from P1fibo import fib2, fib
>>> fib(9)
1 1 2 3 5 8
```

Moduly mohou obsahovat libovolný kód, nejen definice fcí

```
>>> import P2cita
```

Všechny proveditelné příkazy se při zavlečení příkazem import ihned provedou

Python – funkcionální programování

Python obsahuje i funkcionály (funkce s argumentem typu funkce, více v LISPu)

map(funkce, posloupnost) Tento funkcionál aplikuje pythonovskou funkci **funkce** na každý z členů posloupnosti **posloupnost** (t.j seznam, n-tice, řetězec) . Př.

```
>>> list = [1,2,3,4,5]            # lze overit kopirovanim po radcich do IIDLE
>>> def cube (x):
	return x*x*x

>>> L = map(cube, list)
>>> print L
[1, 8, 27, 64, 125]
```

filter(funkce, posloupnost) vybírá všechny prvky posloupnosti **posloupnost**, pro které **funkce** vrací hodnotu True.

```
>>> def lichost(x): return (x%2 !=0)

>>> print filter(lichost, L)
[1, 27, 125]
```

Python – funkcionální programování

reduce(funkce, posloupnost) redukuje **posloupnost** na jedinou hodnotu tím, že na její prvky aplikuje danou binární **funkci**.

```
>>> def secti(x,y):return x+y

>>> print reduce (secti,L)
225
>>> L
[1, 8, 27, 64, 125]
```

Python – funkcionální programování

Lambda výrazy definují anonymní fce (tj. bez jména, převzato z LISPu) zápisem:

lambda <seznam parametrů> : < výraz >

Př.

```
>>> L=[1,2,3,4,5,6]
>>> print map(lambda x,y,z: x+y+z, L,L,L)
[3, 6, 9, 12, 15, 18]
>>> mojefce1 = lambda x,y:  y*x
>>> mojefce2 = lambda x: x+1
>>> print mojefce1(mojefce2(5),6)
36
```

Generátory seznamů - pro vytváření nových seznamů

[<výraz> for <proměnná> in <kolekce> if <podmínka>]

Můžeme vyjádřit ekvivalentním zápisem:

```
L = []
for proměnná in kolekce:
    if podmínka:
        L.append(výraz)
```

Např.

```
>>> [n for n in range(10) if n % 2 == 0 ]  #generuje sudá čísla do 10. % je operator modulo
[0, 2, 4, 6, 8]
>>> [n * n for n in range(5)]     #if část je nepovinná,
[0, 1, 4, 9, 16]
>>> hodnoty = [1, 13, 25, 7]
>>> [x for x in hodnoty if x < 10]   #v casti kolekce muze byt samozrejme i explicitni seznam
[1, 7]
```

Výpis jmen proměnných a fcí platných v daném modulu provede fce dir(jménomodulu)
„ všech „ „ provedeme fcí dir()

Python obsahuje moduly pro práci s:

- Řetězci
- Databázemi
- Datumy
- Numerikou
- Internetem
- Značkovacími jazyky
- Formáty souborů
- Kryptováním
- Adresářemi a soubory
- Komprimováním
- Persistencí
- Generickými službami
- OS službami
- Meziprocesovou komunikací a síťováním
- Internetovými protokoly
- Multimediálními službami
- GUI
- Multijazykovými prostředky
- A další

sys Umožňuje interakci se systémem Python:

- exit() — ukončení běhu programu
- argv — seznam argumentů z příkazového řádku
- path — seznam cest prohledávaných při práci s moduly
- platform – identifikuje platformu
- modules slovník již zavedených modulů
- A další

os Umožňuje interakci s operačním systémem:

- name — zkratka charakterizující používaný operační systém; užitečná při psaní přenositelných programů
- system — provedení příkazu systému
- mkdir — vytvoření adresáře

  Př.
  ```
  >>>chdir("c:\sw")
  >>>getcwd()
  'c:\\sw'
  ```
- getcwd — zjistí současný pracovní adresář (z anglického **get c**urrent **w**orking **d**irectory)
- A další

re Umožňuje manipulaci s řetězci předepsanou regulárními výrazy, jaké se používají v systému Unix:

- search — hledej vzorek kdekoliv v řetězci
- match — hledej pouze od začátku řetězce
- findall — nalezne všechny výskyty vzorku v řetězci
- split — rozděl na podřetězce, které jsou odděleny zadaným vzorkem
- sub, subn — náhrada řetězců
- A další

math **Zpřístupňuje řadu matematických funkcí:**

- sin, cos, atd. — trigonometrické funkce
- log, log10 — přirozený a dekadický logaritmus
- ceil, floor — zaokrouhlení na celé číslo nahoru a dolů
- pi, e — konstanty
- …

time **Funkce pro práci s časem a datem:**

- time — vrací současný čas (vyjádřený v sekundách)
- gmtime — převod času v sekundách na UTC (tj. na čas v univerzálních časových souřadnicích — známější pod zkratkou GMT z anglického Greenwich Mean Time, tedy greenwichský [grinidžský] čas)
- localtime — převod do lokálního času (tj. posunutého vůči UTC o celé hodiny)
- mktime — opačná operace k localtime
- sleep — pozastaví běh programu na zadaný počet sekund
- …

random **Generátory náhodných čísel :**

- randint — generování náhodného čísla mezi dvěmi hranicemi (včetně)
- sample — generování náhodného podseznamu z jiného seznamu
- seed — počáteční nastavení klíče pro generování čísel
- …

Python – vstupy a výstupy, souborové objekty

f = open(jméno [, mod [, bufsize]])	# Otevření souboru, [ ] jsou metasymboly
	# mod je r = read, w = write; bufsize = velikost buferu, moc se neužívá
f.read()	# přečte celý soubor a vrátí ho jako řetězec
f.read(n)	# přečte dalších n znaků, když jich zbývá méně – tak jich vrátí méně
f.readline()	# vrátí další řádku f nebo prázdný řetězec, když je f dočteno
f.readlines()	# přečte všechny řádky f a vrátí je jako seznam řetězců včetně ukončení
f.write(s)	# zapíše řetězec s do souboru f
f.writelines(L)	# zapíše seznam řetězců L do souboru f
f.seek(p [, w])	# změní pozici: při w=0 na p, při w=1 jde o p dopředu, při w=2 na p odzadu
	[, w] je nepovinné
f.truncate([p])	# vymaže obsah souboru za pozicí p
f.tell()	# vrací současnou pozici v souboru
f.flush()	# vyprázdní buffer zkompletováním všech transakcí s f
f.isatty()	# predikát – je tento soubor terminál?
f.close()	# uzavře soubor f

Python – vstupy a výstupy

Př P3citacSlov.py Spustit Idle, File- Open- P3citacSlov.py, Run- Run Module

```
def pocetSlov(s):
    seznam = s.split()  # rozdeli retezec s na jednotliva slova a vrati jejich seznam
    return len(seznam)  # vrátíme počet prvků seznamu

vstup = open("jezek.txt", "r")  # otevre soubor pro cteni, alias pro open je file
                                # k otevření pro zápis použij argument “w”
                                # k otevření binárních použij argument “wb”, “rb”
                                # k otevření pro update použij +, např. “r+b”,  “r+”
                                # k otevření pro append  použij argument “a”
celkem = 0  # vytvoříme a vynulujeme proměnnou
for radek in vstup.readlines():  # vrátí seznam řádků textu, readlines(p) omezí na p bytů
     # soubor.read() přečte celý soubor a dodá ho jako string, možno i read(pbytů)
     # metoda readline() přečte 1 řádek končící \n
    celkem = celkem + pocetSlov(radek) # sečti počty za každý řádek
print "Soubor ma %d slov." % celkem  # tisk dle ridiciho retezce

vstup.close()                           # zavreni souboru
```

Následuje př.P31citacSloviZnaku.py . Lze spustit **a) Po spusteni v Idle na vyzvu napsat: jezek.txt**

Nebo b) v CMD C:\sw\Python25>Python d:\PGS\Python\P31citacSloviZnaku.py d:\PGS\Python\jezek.txt

Take b) v CMD D:\PGS\Python> C:\sw\Python25>Python P31citacSloviZnaku.py Python\jezek.txt

argument 0 argument 1

```
# -*- coding: cp1250 -*-
import sys
# jmeno souboru si vyžádáme od uživatele,
if len(sys.argv) != 2:
    jmenoSouboru = raw_input("Zadejte jmenosouboru: ") #vestavena fce pro cteni retezce
else:   # else ho zadame z prikazoveho radku tj. Moznost b)
    jmenoSouboru = sys.argv[1] #je to 2.argument prikazu vyvolani Pythonu
vstup = open(jmenoSouboru, "r")
# vytvoříme a znulujeme příslušné proměnné.
slov = 0
radku = 0
znaku = 0
for radek in vstup.readlines():
    radku = radku + 1
    # Řádek rozložíme na slova a spočítáme je.
    seznamSlov = radek.split()
    slov = slov + len(seznamSlov)
    # Počet znaků určíme z délky původního řádku,
znaku = znaku + len(radek)
print "%s ma %d radku, %d slov a %d znaku" % (jmenoSouboru, radku, slov, znaku)
vstup.close()
```

Python – vstupy a výstupy

Způsob vyvolání ze shellu Pythonu:

```
>>> import P31citacSloviZnaku
Zadejte jmeno souboru: d:\pgs\Python\jezek.txt
```

Nebo z příkazového řádku DOS

```
        c:\Python25> d:\PGS\Python\ P31citacSloviZnaku.py  d:\pgs\Python\jezek.txt
dtto je c:\Python25> python d:\PGS\Python\ P31citacSloviZnaku.py  d:\pgs\Python\jezek.txt
```

Nebo spustíme z IDLE

Je-li čtený soubor v pracovním directory IDL, stačí

Zadejte jmeno souboru: jezek.txt

Další př. I/O možnosti:

-číst řádky lze i cyklování přes objekt typu soubor: for line in file: #napr. for line in vstup
print line #   print line

-zapisovat lze i metodou write(řetězec) na řetězec lze vše převést metodou str(něco)

-print vst.tell()   tiskne pozici ve file

-vst.seek(10,1) změní aktuální pozici v souboru otevřeném v vst o 10 dál

-modul pickle má metody pro konverzi objektů na řetězce a opačně

pickle.dump(objekt, file_otevřeno_pro_zápis) zakleje objekt do souboru, který lze poslat po síti, nebo uložit v paměti jako perzistentní objekt

objekt = pickle.load(file_pro_čtení) vytvoří ze souboru opět objekt

Python - skripty

Skript je skupina po sobě jdoucích příkazů uložená do souboru. Původně krátký program pro příkazový interpret OS

Z důvodu strukturování je lépe vytvořit v souboru řídící fci a tu v tom souboru hned vyvolat

Skripty se obvykle spouští z příkazového řádku (častější v Unixu než ve Windows)

```
#Pr. skript1.py ,
# spust ve Windows: c:\sw\python25>python d:pgs\python\skript1.py,
# nebo kratsi zapis, daš-li skript do direktory  s python.exe, staci …>python skript1.py
# definice fce main.            Může mít i jiné jméno.
def main():
    print 'bezi skript 1'
    jmeno = raw_input('jak se jmenujes? : ')
    print 'Ahoj ', jmeno

# vyvolani fce main
main()
raw_input('skonci stiskem ENTER')
                                                    # Ctrl C zastavi beh skriptu
```

Python - skripty

Z příkazového řádku lze předávat argumenty. Ty jsou ulozeny jako seznam v systémové proměnně sys.argv

```
# Pr. skript2.py
import sys
def main():
    print "pri spusteni pridane argumenty, ulozi se do sys.argv"
    print sys.argv

main()
```

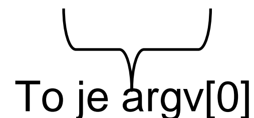
Spustíme zápisem v cmd okně, **analogický způsob s argumenty již v IDLE nelze provest**

C:\sw\python25>python skript2.py ar1 ar2 ar3

Vypise se :

pri spusteni pridane argumenty, ulozi se do sys.argv

[‘skript2.py’, ‘ar1’, ‘ar2’, ‘ar3’]



To je argv[0]

Python - skripty

Skript může akceptovat z příkazového řádku i přepínače (volby), stejně jako argumenty.
K tomu je vhodné použít modul *getopt* , který obsahuje podporu pro synt. analýzu řetězce přepínačů, které chce skript rozpoznávat a argumentů. Funkce getopt vrací dva seznamy. První tvoří n-tice nalezených přepínačů ve tvaru (přepínač, jeho parametr) druhý vrací normální argumenty.

```
# skript3.py
import getopt, sys
                                   přiřadí od argv[1];    mustr pro 4 optios
def main():
   (volby, argumenty) = getopt.getopt(sys.argv[1:], 'f:vx:y:')    # je-li dvojtecka za znakem,
                                                                  #tak prepinac vyzaduje parametr
   print 'volby:', volby
   print 'argumenty:', argumenty

main()
```

dvojice přepínačů a jejich parametrů
pojmenování dovolí volné pořadí při spouštění

Spuštění

…python skript3.py –x100 –v –y50 –f nejakySoub ar1 ar2

argumenty

Python - skripty

Podporu pro zpracování řádků ze vstupních souborů dává modul *fileinput* . Z proměnné sys.argv načte argumenty příkazového řádku a použije je jako jména vstupních souborů, které po řádcích zpracuje

```
# skript4.py              vypouští řádky začínající komentářem a tiskne počet radek
import fileinput
def main():                          #čte řádky souborů daných argv[1], argv[2], …
    for radek in fileinput.input():
        if fileinput.isfirstline():      #pozna zacatek souboru
            print 'soubor %s' % fileinput.filename()
        if radek[:1] != '#':             #znak s indexem [0]
            print radek
   print 'pocet radek ', fileinput.lineno()     #lineno() je pocet radek za vsechny soubory
                                  # další fce viz modul fileinput
main()
```

Spuštění např.

```
C:\sw\python25>python skript4.py skript3.py skript2.py
                              argument1                    argument2
```

Python - skripty

Spouštět skripty ve Windows lze i dalšími způsoby:

-Přesuneme se v MSDOS okně do adresáře se skripty (napr. D:\pgs\python a v příkazovém řádku napíšeme např.

c:\sw\python25>python.exe skript1.py

-Na soubor se skriptem klepneme prav tl., vybereme Odeslat na plochu (vytvořit zástupce).

Na vzniklou ikonu klepneme prav. tl., vybereme vlastnosti a v nich lze nastavit adresář,

doplnit argumenty a zavést klávesovou zkratku stiskem ctrl + písmeno. Tou lze pak skript spouštět

Pokud má skript parametry, lze je uvést v poli Cíl

Lze spustit i poklepáním na ikonu.

-Spustit lze také Start – Spustit a do dialogového okna zapsat např.

python skript1.py

program se ale skončí zavřením okna (zůstane otevřené při uvedení přepínače -i)

Python - skripty

Přesměrování vstupů / výstupů lze provést z příkazového řádku

Např.

c:\sw\python25>python skript2.py   <skript1.py   >out.txt

Skript2.py bude číst ze souboru skript1.py a zapisovat do souboru out.txt

Čtení a zápis lze provádět rovněž s použitím modulu **sys**

Př. skript5.py

```
# skript5.py
import string, sys
def main():
   sys.stdout.write(string.replace(sys.stdin.read(), # řetězec daný 1.argumentem bude nahrazen
                                sys.argv[1], sys.argv[2] ))   # řetězcem daným druhým argumentem
main()
```

Normálně spuštěn by pracoval s klávesnicí a obrazovkou. Spustíme ho s přesměrováním

c:\sw\python25>python skript5.py main hlavni <skript1.py   >nic.txt

Python - skripty

Krátkým skriptům postačuje jediná funkce.

U rozsáhlých skriptů je vhodné oddělit řídící funkci main od dalších funkcí

Př skript6.py

Provádí překlad dvouciferných čísel do slovního tvaru.

Řídící fce main( ) volá fci **prekladDo99** se zadaným argumentem (viz %)

Skript voláme z příkazového řádku např.

… python skript6.py 23

Python - skripty

```
import sys
do9 = {'0':'', '1':'one', '2':'two', '3':'three', '4':'four' }                #atd.
od10do19 = {'0':'ten', '1':'eleven', '2':'twelve', '3':'thirteen'}            #atd
od20do90= {'2':'twenty', '3':'thirty', '4':'fourty', '5':'fifty'}             #atd

def prekladDo99(cifernyTvar):
    if cifernyTvar == '0': return('zero')
    if len(cifernyTvar) > 2:
        return "vice nez dvouciferne cislo"
    cifernyTvar = '0' + cifernyTvar
    decades, units = cifernyTvar[-2], cifernyTvar[-1]
    if decades == '0': return do9[units]
    elif decades =='1': return od10do19[units]
    else: return od20do90[decades]+' '+do9[units]

def main():
    print prekladDo99(sys.argv[1])

main()
```

Python - skripty

Skripty lze použít jako moduly, chceme-li jejich kód spustit v jiném skriptu nebo modulu.

Tuto možnost zajistí podmíněné volání řídící fce main( )

```
if  __name__  == '__main__' :        #__name__ je atributem obsahujicim jmeno fce
    main( )
else
    #  případný inicializační kód modulu
```

Bude-li soubor volán jako skript, bude mít proměnná __name__ hodnotu __main__,

Je-li soubor importován jako modul do jiného modulu, obsahuje proměnná

__name__ název souboru

Použití ukazuje př.skript7.py a skriptJakoModul.py (viz%)

Také naopak, modul může být upraven tak, aby mohl být spuštěn jako skript. K tomu stačí když v proměnné __name__ zajistí, že obsahuje hodnotu '__main__'

Python - skripty

Př.skript7.py

```
. . .
# az sem to je stejne se skript6.py
def main():
    print prekladDo99(sys.argv[1])

if __name__ == '__main__':
    main()
else: print __name__, 'je zaveden jako modul'
```

Vyvolat   c:\sw\python25>python skript7.py 12

Př skriptJakoModul.py

```
import skript7
#delej si co chces
c = '12'
print skript7.prekladDo99(c)
```

Python třídy a objekty

class C(R0, R1, …) : **#** dovoluje vice rodicu
    B_i **#** blok proveden jednou při zpracování definice.
        # pomocí přiřazování vytvoří proměnné třídy a zpřístupni je *třída***.***proměnná*
    def m_0 (self, …) : #metoda
        B_0 #blok
    def m_1 (self, …) :
        B_1
    …

Jméno konstruktoru je vždy __init__(parametry), v potomkovi se musí explicitně volat
V konstruktoru potomka je nutné explicitně volat konstruktor jeho rodiče příkazem

rodič.__init__(self, případné další parametry)

Proměnné instance jsou vytvářeny přiřazovacím příkazem uvnitř konstruktoru
Privátní metody a proměnné instancí tříd jsou pojmenovány **__jméno**
a jsou použitelné jen uvnitř třídy.

Probíhá-li výpočet uvnitř metody třídy, má přístup do
    lokálního prostoru jmen = argumenty a proměnné deklarované v metodě
    globálního prostoru jmen = funkce a proměnné deklarované na úrovni modulu
    vestavěného prostoru jmen = vestavěné funkce a výjimky

Python třídy a objekty

Př.P4Obrazce (Spustit bud v Idle nebo v d:\pgs\python>c:\sw\Python25\python a pak >>>import P4Obrazce

```
class Ctverec:
    def __init__(self, strana):     # __konstruktor__,   explicitne uvadeny  self  = konvence pro this
        self.strana = strana        # přiřazení = i definici lokální proměnné každé instance
    def vypocitejPlochu(self):
        return self.strana**2

class Kruh:
    def __init__(self, polomer):
        self.polomer = polomer
    def vypocitejPlochu(self):
        import math
        return math.pi*(self.polomer**2)

class Obdelnik2x3:  # kdyz nema konstruktor zadny parametr, nemusi se uvest
    def vypocitejPlochu(self):
        return 2*3

seznam = [Kruh(8), Ctverec(2.5), Kruh(3), Obdelnik2x3()]

for tvar in seznam:
    print "Plocha je: ", tvar.vypocitejPlochu() # tady se jiz self neuvádí, není zde ani definované
```

Python třídy a objekty

Dědit lze i od rodiče z jiného modulu

```
class Potomek(modul.Rodic):
        <příkaz1>
        . . .
         <příkazN>
```

Příkazy jsou nejčastěji definicemi metod. Lze definovat vnořené třídy

Metody jsou implicitně dynamické (virtuální) a mohou překrýt metodu rodiče

Statickou lze metodu udělat i po definici jejím zasláním metodě *staticmethod*

```
jménometody = staticmethod(jménometody)
```

Podobné jsou metody třídy, nemají také self parametr

```
jménometody = classmethod(jménometody)
```

Lze definovat destruktor __del__, ten se provede (dělá úklidové akce), když je objekt rušen, např. výstup z rozsahu platnosti objektu.

Python třídy a objekty

Má násobnou dědičnost

```
class Potomek(Rodic1, Rodic2, …RodicM):
        <příkaz1>
        . . .
         <příkazN>
```

Řešení problému násobného dědění

Není-li atribut v Potomkovi, hledá se v Rodiči1, pak v Rodiči Rodiče1, …v Rodiči2…

Tj. do hloubky a pak z leva do prava .

Př. P42.py

Python třídy a objekty

```
class Otec:
     def __init__(self):
       self.oci = 'zelene'
       self.usi = 'velke'
       self.ruce = 'sikovne'
    def popis(self):
       print self.oci, self.usi, self.ruce

class Matka:
    def __init__(self):
       self.oci = 'modre'
       self.nos = 'maly'
       self.nohy = 'dlouhe'
    def popis(self):
       print self.oci, self.nos, self.nohy

class Potomek(Matka, Otec):
    def __init__(self):
       Otec.__init__(self)        # 1.
       Matka.__init__(self)       # 2.
    def popis(self):
       print self.oci, self.usi, self.ruce, self.nos, self.nohy

petr = Potomek()
petr.popis()
```

Python třídy a objekty

Spustit bud v Idle nebo v d:\pgs\python>c:\sw\Python25\python a pak >>>import P42

Co vypíše?

modre velke sikovne maly dlouhe

Při změně na

```
class Potomek(Matka, Otec):
    def __init__(self):
        Matka.__init__(self)
        Otec.__init__(self)
```

Vypíše

zelene velke sikovne maly dlouhe

Př P41perzistence.py

Ilustruje zcela obecnou možnost vytváření perzistentních objektů = uložení objektu do souboru (to funguje v každém jazyce)

Python perzistentní objekty

```
class A:                                # příklad P41perzistence.py
    def __init__(self, x, y):
        self.x = x
        self.y = y
    def save(self, fn):                 # uložení objektu do souboru
        f = open(fn, "w")
        f.write(str(self.x) + '\n') # převeď na řetězec
        f.write(str(self.y) + '\n')
        return f     # do stejného souboru budou své hodnoty
                     # připisovat objekty odvozených tříd
    def restore(self, fn):              #  obnovení objektu ze souboru
        f = open(fn)
        self.x = int(f.readline()) # převeď zpět na původní typ
        self.y = int(f.readline())
        return f
a = A(1, 2)                             # Vytvorime instance.
a.save('a.txt').close()                 # Ulozime instance.
newA = A(5, 6)                          # Obnovime instance.
newA.restore('a.txt').close()
print "A: ", newA.x, newA.y             # Podivej se na disk, je tam soubor a.txt a v nem 1, 2
```

Python třídy a objekty

Python má i destruktor

```
def __del__(self):
    <příkazy>  # píkazy se provedou když je objekt odstraňován z paměti
               # to nastane samovolně, když čítač odkazů na objekt je 0 (objekt má čítač),
               # takže destruktor se moc nepoužívá (oproti C++ není tak důležitý)
```

Př.P43Destruktor.py

Po dokončení fce *zkus( )* bude počet odkazů na objekt *logickeJmenoSouboru* nula, takže překladač automaticky zavolá definovaný destruktor

Python třídy a objekty

Př. P43Destruktor.py

```
class UkazkaDestruktoru:
    def __init__(self, soubor):             #vytvori soubor, otevre ho a zapise do nej
        self.file = open(soubor, 'w')
        self.file.write('tohle zapisuji do souboru\n')
    def write(self, retezec):
        self.file.write(retezec)
    def __del__(self):  # destruktor, write overime ze se provedl
        self.write( "__del__ se provedlo")

def zkus():
    logickeJmenoSouboru = UkazkaDestruktoru('pomocnySoubor')
    logickeJmenoSouboru.write('tohle take zapisuji do souboru\n')
    # zde objekt logickeJmenoSouboru prestane existovat

zkus()
```

Po spuštění se podívej na soubor pomocnySoubor

Python třídy a objekty

Privátní atributy
Omezená podpora formou: *__jméno je textově nahrazeno __classname__jméno*
a tímto jménem je atribut nadále přístupný

Prázdné třídy poslouží jako typ záznam např.

```
class Record:
    pass

petr = Record( )   # vytvoří prázdný záznam o Petrovi

# jednotlivé položky instance není nutné deklarovat předem, lze to provést dodatečně
petr.jmeno = 'Petr Veliky'
petr.vek = 39
petr.plat= 40000
```

Python – výjimky jsou také třídy

Obecný tvar pythonských výjimek:

```
try:
    # ošetřované příkazy
except TypVyjimky:
     # zpracování výjimky
except TypVyjimky:   # při neuvedení TypVyjimky zachytí se všechny dosud nechycené
     # zpracování výjimky
. . .
else:
     # činnost, když nennastane výjimka (nepovinné)

finally:
     # činnost prováděná ať výjimka nastane či nenastane
```

Vyhození výjimky způsobíme příkazem   raise JmenoVyjimky # = instance výjimky

Možné formy: raise TřídaTypuException, instance
raise instance   je zkrácením raise instance._class_, instance
samotné raise lze užít jen uvnitř except a vyhodí posledně nastalou výj.

Python – výjimky jsou také třídy

Př. P44vyjimky.py   Čte řádek souboru, třetí údaj je považován za číslo. Je-i 122 způsobí dělení 0 výjimku, při všech jiných vadách zachytává nespecifikovanou výjimku

```
print 'Start programu.'
try:
    data = file(‘data.txt')
    print 'Soubor s daty byl otevren.'
    try:
        hodnota = int(data.readline().split()[2])  #  radek tvaru slovo slovo cislo …
        print 'Hodnota je %d.' % (hodnota/(122-hodnota))
    except ZeroDivisionError:
        print 'Byla nactena hodnota 122.'
    except: print "Stalo se neco neocekavaneho."
finally:
    data.close()
    print 'Soubor s daty byl uzavren.'
print 'Konec programu.'
```

Python – výjimky jsou také třídy

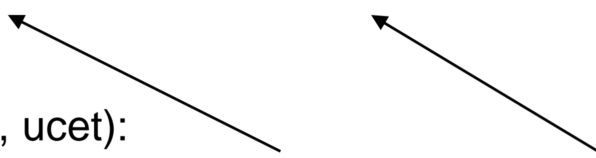
Uživatelem definované výjimky jsou instance tříd odvozených z třídy Exception.
Hierarchie zpracování děděných výjimek je jako v Javě

Př.P45.py

```
class NejakaError(Exception):
    pass
try:
    raise NejakaError , ' Informace co se deje \n‘ #zpusobime vyjimku
except NejakaError:
    print u'Narazili jsme na chybu při zpracování.'  #zpracovani vyjimky
```

Př.P46.py Přetažení konta

```
# -*- coding: cp1250 -*-
class ChybaZustatku(Exception):
    hodnota = 'Nelze vybrat. Na vašem účtu je jen %5.2f korun.'
class Ucet:
    def __init__(self, pocatecniVklad):
        self.stav = pocatecniVklad
        print 'Mate založen účet s vkladem  %5.2f korun.' % self.stav
    def vlozit(self, castka):
            self.stav = self.stav + castka
    def vybrat(self, castka):
        if self.stav >= castka:
            self.stav = self.stav - castka
        else:
            raise ChybaZustatku, ChybaZustatku.hodnota % self.stav
    def zustatek(self):
        return self.stav
    def prevod(self, castka, ucet):
        try:
            self.vybrat(castka)
            ucet.vlozit(castka)
        except ChybaZustatku, e:
            print str(e)

mujUcet = Ucet(300)
mujUcet.vybrat(200)
mujUcet.vybrat(200)            # lze vyzkouset ruzne castky
```



Python – událostmi řízené programování a GUI

Vlastnosti:

- **Program po spuštění čeká v nekonečné smyčce na výskyty událostí**
- **Při výskytu události provede odpovídající akci a nadále čeká ve smyčce**
- **Skončí až nastane konec indikující událost**

Události může generovat OS (obvyklé u programů s GUI), nebo vnější čidla

Předvedeme na freewarovém multiplatformním systému Tk (Tkinter pro Python)

Př. P5udalosti.py

Program zachycující události stisknutí klávesy, dokud nenastane ukončující událost (stisk mezery). Na stisk klávesy program reaguje výpisem kódu klávesy

Program používá modul Tkinter s prostředky pro GUI

1. Vytvoříme třídu KlavesovaAplikace pro naši aplikaci
2. Tato třída obsahuje metody pro zpracování událostí
3. Součástí konstruktoru třídy je vytvoření GUI okna pro výpis kódu klávesy
4. Vytvoříme instanci třídy
5. Této instanci zašleme zprávu *mainloop*

!!! Ale pozor, spouštěj ho z příkazového řádku OS, … c:\sw\Python25>python ne z IDLE, protože IDLE samo využívá Tkinter

>>> import P5udalosti

Nebo poklepem na ikonu souboru P5udalosti.py

```
from Tkinter import *
class KlavesovaAplikace(Frame):           # Vytvori GUI
   def __init__(self):
      Frame.__init__(self)        #vytvori ramec do ktereho budou vkladany dalsi prvky
      self.txtBox = Text(self)    #vytvori a vlozi do ramce prvek pro praci s radky textu
      self.txtBox.bind("<space>", self.zpracujUdalostUkonceni) #navaze mezeru na udalost
      self.txtBox.pack()          #pack je manazer umisteni. Vlozi prvek textovy box do jeho rodice
      self.pack()                 #az ted se textovy box zviditelni
      self.txtBox.bind("<Key>", self.zpracujStiskKlavesy) #navaze stisk klavesy na zpracujStiskKl…
   def zpracujUdalostUkonceni(self, udalost):
      import sys
      sys.exit()
   def zpracujStiskKlavesy(self, udalost):            # metoda zpracovani udalosti
      str = "%d\n" % udalost.keycode      #str nabyde hodnotu kodu klanesy
      self.txtBox.insert(END, str)        #vlozi text za posledni znak
      if udalost.keycode == 88:           # to je x
         self.txtBox.insert(END, 'cteme x') #vlozi text za posledni znak
      return "break"
mojeAplikace = KlavesovaAplikace()      # vytvoreni instance
mojeAplikace.mainloop()                 #spustime cekaci smycku
```

Python – událostmi řízené programování a GUI

Některé prvky GUI Tkinter, lze vyzkoušet interaktivně

>>> from Tkinter import * naimportuje jména ovládacích prvků

>>> top = Tk( ) vytvoří **widget** (**ovládací prvek**) na nejvyšší úrovni, ostatní budou jeho potomky. Je to okno, do něj budeme další prvky přidávat

>>> dir(top) vypíše všechna jména = atributy objektu top třídy Tk

>>> F = Frame(top) vytvoří widget rámec (frame), který je potomkem top. Do něj budou umisťovány ostatní ovládací prvky. Objekt z Frame má již atribut pack

Funguje i F=Frame() t.j. primo bez zavedeni top

>>> F.pack( ) aktivuje packer, protoze je ramec zatim prazdny, zdrcne ho na listu

>>> lPozdrav = Label(F, text = 'everybody out') vytvoří objekt třídy Label jako potomka F, jeho atribut text má iniciovanou hodnotu. Lze udat i barvu a typ písma …

>>> lPozdrav.pack( ) pakuje lPozdrav do rámce, ted se objevi v okenku

>>> lPozdrav.configure(text = 'vsichni ven') metoda configure dovoluje změnit vlastnosti objektu takze zde zameni text za novy

>>> lPozdrav['text'] = "everybody out" menime-li jen jednu vlastnost, tak je tohle kratší

Python – událostmi řízené programování a GUI

>>> F.master.title('Ahoj') dovoluje nastavit titulek okna metodou title pro widget na vrcholu hierarchie = objekt top

>>> bQuit = Button(F, text= 'Konec', command = F.quit) vytvoří tlačítko s nápisem Konec, které je spojeno s příkazem F.quit. Předáváme jméno metody quit

>>> bQuit.pack( ) zajistí zviditelnění tlačítka

>>> top.mainloop( ) odstartuje provádění smyčky. Činnost teď řídí Tkinter. Zmizely **>>>** a objeví se až po stisku tlačítka Konec a pak lze provest >>> quit()

Př. P51

Je skriptem, který to dělá.

Musí se spustit z příkazové řádky OS import P51 nebo poklepem na ikonu

Vzniklé okno čeká ve smyčce na stisk Konec …

Python – událostmi řízené programování a GUI

```
from Tkinter import *         #naimportuje jména ovládacích prvků

# Vytvorime okno.
top = Tk()                    # vytvoří widget (ovládací prvek) na nejvyšší úrovni
F = Frame(top)                # vytvoří widget rámec (frame), který je potomkem top
F.pack()                      # pakuje rámec na lištu

# Pridame ovladaci prvky.
# vytvoří objekt třídy Label jako potomka F
lPozdrav = Label(F, text="everybody out")
lPozdrav.pack()               # pakuje lPozdrav do rámce
F.master.title('Nazev')       # nastavi titulek okenka
# vytvoří tlačítko s nápisem Konec, barvy, specifikuje příkaz asociovaný s uvolň. tlačítka
bQuit = Button(F, text="Konec", fg=‘white’, bg=‘blue’, command=F.quit)
bQuit.pack()                  # zajistí zviditelnění tlačítka (umístí ho do rámce)
# Spustime smycku udalosti.
top.mainloop()
quit()                        # stisk Konec ukonci vypocet
```

Př. P52.py reakce na tlacitka a na souradnice klicku

```
# -*- coding: cp1250 -*-        #jsou tam ceska pismena
from Tkinter import *
vrchol = Tk()                   #vytvori ovladaci prvrk na nejvyssi urovni

def odezva(e):                  # definuje reakci na udalost odezva
  print 'klik na', e.x, e.y     # tiskne souradnice x, y
def klik():                     #definuje odezvu na stisk tlacitka
   print u"Stiskl jsi tlačítko!"

f = Frame(vrchol, width=200, height=300)        #parametry urci velikost ramce
# pružněji spoji ramec f s levym tlacitkem mysi a odezvou metoda bind
f.bind('<Button-1>', odezva)  #<Button-1> je leve, <Button-2> je prave tlacitko
f.pack()
for i in range(4):
   tlacitko=Button(text=u"Já jsem tlačítko", command=klik)
   tlacitko.pack()

vrchol.mainloop()
quit()   #ukonci beh Pythonu po zavreni okna
```

Př.53 Okno se zapisovacím polem, horkou klávesou a tlačítky mazání a konec

```
# -*- coding: cp1250 -*-
from Tkinter import *

# Nejdříve vytvoříme funkci pro ošetření události.
def vymazat():
  eTxt.delete(0, END)    # metoda maze text od nulteho znaku do konce

def eHotKey(u):
  vymazat() # zavedeme pro mazani i hot key

# Vytvoříme hierarchicky nejvyšší okno a rámeček.
hlavni = Tk()
F = Frame(hlavni)
F.pack()
```

```
# Nyní vytvoříme rámeček s polem pro vstup textu.
fVstup = Frame(F, border=20) # velikost okna je s parametrem 20
eTxt = Entry(fVstup) # prvek tridy Entry je pro zadavani jednoradkoveho textu
eTxt.bind('<Control-m>', eHotKey) # navazani ctrl-m na mazani
fVstup.pack()
eTxt.pack()

# Nakonec vytvoříme rámečky s tlačítky.
# Pro zviditelnění je vmáčknutý = SUNKEN
fTlacitka = Frame(F, relief=SUNKEN, border=1)
bVymazat = Button(fTlacitka, text="Vymaz text", command=vymazat)
bVymazat.pack(side=RIGHT, padx=5, pady=2) #5 a2 jsou vycpavky (mista) mezi prvky
bKonec = Button(fTlacitka, text="Konec", command=F.quit)
bKonec.pack(side=LEFT, padx=5, pady=2)
fTlacitka.pack(side=TOP, expand=True)

# Nyní spustíme čekací smyčku
F.mainloop()
quit()
```

Python – událostmi řízené programování a GUI

Př.P54.py OO přístup ke GUI aplikacím

Celá aplikace se zapouzdří do třídy buď tak, že

1) Odvodíme třídu aplikace od tkinter třídy Frame (užívá dědičnost) nebo
2) Uložíme referenci na hierarchicky nejvyšší okno do členské proměnné (užívá kompozici)

Použijeme postup 2 pro konstrukci s polem typu Entry, a tlačítky Vymaz a Konec v OO podobě.

- Do konstruktoru aplikace dáme jednotlivé části GUI.
- Referenci na prvek typu Frame přiřadíme do self.hlavniOkno, čímž zajistíme metodám třídy přístup k prvku typu Frame
- Ostatní prvky, ke kterým přistupujeme přiřadíme členským proměnným instance z Frame
- Funkce pro zpracování událostí se stanou metodami třídy aplikace, takže mohou přistupovat k datovým členům aplikace pomocí reference self.

```
# -*- coding: cp1250 -*-
from Tkinter import *

class AplikaceVymazat:
    def __init__(self, rodic=0):
        self.hlavniOkno = Frame(rodic,width=200,height=100)
        self.hlavniOkno.pack_propagate(0) #aby platila zadana vyska, sirka a ne implicitni
        # Vytvoříme widget třídy Entry
        self.vstup = Entry(self.hlavniOkno)
        self.vstup.insert(0, "Pocatecni text")
        self.vstup.pack(fill=X) #prvek vstup zabira ve smeru X cele mozne misto
```

```
# Nyní přidáme dvě tlačítka v ramecku a použijeme efekt drážky.
    fTlacitka = Frame(self.hlavniOkno, border=2, relief=GROOVE) #drazkovany relief
    bVymazat = Button(fTlacitka, text="Vymazat",
                      width=8, height=1, command=self.vymazatText)
    bKonec = Button(fTlacitka, text="Konec",
                    width=8, height=1, command=self.hlavniOkno.quit)#velikost tlacitka
    bVymazat.pack(side=LEFT, padx=15, pady=1) # urcuji stranu a vnejsi vzdalenosti
    bKonec.pack(side=RIGHT, padx=15, pady=1)
    fTlacitka.pack(fill=X)   #vyplneni tlacitek ve smeru X
    self.hlavniOkno.pack()

    # Nastavíme nadpis okna.
    self.hlavniOkno.master.title("Vymazat")

  def vymazatText(self):
    self.vstup.delete(0, END)        #od 0 do konce vymazat

aplikace = AplikaceVymazat()
aplikace.hlavniOkno.mainloop()
quit()
```

Běžné Tk prvky:

Tlačítko	**Button**
Plátno	**Canvas**
Zaškrtávací tlačítko	**Checkbutton**
Jednořádkový vstup	**Entry**
Rámeček	**Frame**
Více řádek textu	**Listbox**
Nálepka	**Label**
Rolovací menu	**Menu**
Tlačítko výběru	**Menubutton**
Radiové tlačítko	**Radiobutton**
Posuvný ovladač	**Scale**
Posuvný ukazatel	**Scrollbar**
Textový editor	**Text**

Python a databáze

Nainstalovat pyodbc takto: **(pyodbc = object database connectivity)**

- **Ovládací panely**
  - **Nástroje pro správu**
    - **Datové ODBC zdroje**
      - **Přidat**
        - » **Vybrat databázi (driver do MS Access)**
        - » **Dokončit**
        - » **Název zdroje dat (třeba Lidé, je to název propojení)**
        - » **Vybrat**
        - » **Na C:/ je v MS Access vytvořená databáze1.mdb, tak ji vyber**
        - » **OK**
      - **Objeví se v Správce zdrojů dat ODBC**
      - **OK**
- **Zavřít okno**

Python a databáze

Relační databázový model: tabulky, atributy, n-tice (záznamy)

Tabulka Studenti

kod	Jmeno	Prijmeni	Vek	Škola	Prumer
12	Karel	Dadak	23	VUT	2,01
07	Jana	Tlusta	19	CVUT	1,95
28	Josef	Hora	20	ZCU	1,75
…	…	…	…	…	…

Primární klíč = jeden nebo více atributů, sloužící k jednoznačné identifikaci záznamu. Např. jméno a příjmení, nebo rodné číslo, apod.

Databázi chápejme jako soustavu tabulek navzájem propojených přes společné atributy

Python a databáze

Jazyk SQL:

SELECT	výběr hodnot specifikovaných atributů
FROM	určení tabulek ze kterých se dělá select nebo delete
INNER JOIN	spojení záznamů z více tabulek k získání jediné výsl. relace
WHERE	určení podmínek, které musí splňovat vybírané záznamy
GROUP BY	určení podmínek pro seskupování vybíraných záznamů
ORDER BY	určení podmínek dle kterých se uspořádají vybírané záznamy
INSERT	**INTO** tabulku (atributy) **VALUES** (hodnoty)
UPDATE	tabulku **SET** atribut **=** hodnota, atd **WHERE** podminka
DELETE	**FROM** tabulka **WHERE** podminka

Python a databáze

Př.

select Jmeno, Prijmeni from Studenti where Vek > 23

select Jmeno, Prijmeni, Prumer from Studenti where Vek > 23 order by Prumer, Vek

select * from Studenti

insert into Studenti (Jmeno, Prijmeni, Vek, Skola, Prumer)
values (‘Jan’, ‘Novy’, 25, ‘CVUT’, 3,1)

delete from Studenti where Vek>39 and Prumer>3

update Studenti set Skola=‘VUT’, Vek= 12 where Jmeno=‘Gustav’ and Prijmeni=‘Klaus’

Často je třeba vyhledat podmnožinu kartézského součinu více tabulek
Např tabulek Studenti a Skoly, kde Skoly vyjadřuje relaci Škola a Město kde sídlí

Select IDcislo, prumer from Studenti inner join Skoly on Studenti.Skola = Skoly.Skola
where Skola.Mesto = Praha order by IDcislo

Python a databáze

DB-API zajistí propojení s databází, vytvoří a zviditelní objekt **kurzor** dovolující provádět operace na databázi (selekty, inserty, updaty, delety). V objektu kurzor jsou interně uloženy výsledky dotazu

K výběru řádků výsledku dotazu v podobě objektu lze použít metody:

fetchone() vrací n-tici = další řádek výsledku uloženého v kurzoru

fetchmany(n) vrací n řádků, které jsou na řadě ve výsledku uloženém v kurzoru

fetchall() vrací všechny řádky výsledku.

Současně dochází k posouvání kurzoru

Python používá SQL jako vnořený jazyk, doplní ho na úplný jazyk.

commit příkaz k zakončení transakce (zapsání změn do databáze)

close uzavření kurzoru, uzavření spojeni

Python a databáze

Spustit Python, třeba Idle

```
>>> import pyodbc
>>> c=pyodbc.connect("DSN=Lide")          #vytvori spojeni c se zdrojem dat (byl pojmenovan Lide)
>>> cu=c.cursor()                         #vytvori kurzor cu
>>> cu.execute("select Jmeno, Prijmeni, Vek, Skola from Studenti")
<pyodbc.Cursor object at 0x012C32A0>
>>> for row in cu: print row.Jmeno, row.Prijmeni,row.Skola
```

Vypíše se výsledek a kurzor cu se dostane na konec (musí se znovu nastavit příkazem cu.execute)

```
>>> cu.execute('select sum(vek),prumer from Studenti group by prumer')
<pyodbc.Cursor object at 0x00DA33E0>
>>> for r in cu: print r[0],r[1]
```

Vypíše dvojice (suma veku, prumer) tj. ti se stejným průměrem budou agregováni na 1 řádek

```
>>> cu.execute("update Studenti set Skola ='TUO' where Skola = 'VSB'")
číslo                          #vypisuje počet zasažených řádků
>>> cu.execute("delete from Studenti where Skola = 'TUO'")
Číslo                          #vypisuje počet zasažených řádků
>>> cu.execute("insert into Studenti (Jmeno, Prijmeni, Vek, Skola, Prumer) values ('Krystof',
    'Kolumbus', 450, 'Zivota', 1.1)")
1
>>> c.commit()                 #musí se udelat, není zde autocommit
>>> c.close()                           #uzavre spojeni
```

Python a databáze

Výběr dat z více tabulek – inner join

Tabulka Studenti viz dříve

Tabulka Skoly se sloupci

Škola	**Město**
ZCU	Plzen
KU	Praha
CVUT	Praha
VUT	Brno
VSB	Ostrava

Dotaz na studenty z Plzně

```
>>> cu.execute("select Prijmeni, Vek from Studenti inner join Skoly on
    Skoly.Skola=Studenti.Skola where Skoly.Mesto='Plzen'")

>>> for row in cu: print row.Prijmeni,row.Vek
```

```
>>> cu.execute("select * from Studenti where Vek>10")
<pyodbc.Cursor object at 0x011CB110>
>>> vsechnaPole = cu.description            #metoda vybere zahlavi vysledku
>>> for pole in vsechnaPole: print pole[0] #pole[0]= jmeno sloupce zahlavi

kod
Vek
Jmeno
Prijmeni
Skola
Prumer
>>> for pole in vsechnaPole: print pole[1] # pole[1] =typ sloupce zahlavi

<type 'int'>
<type 'float'>
<type 'str'>
<type 'str'>
<type 'str'>
<type 'float'>
```

```
>>> vsechnyZaznamy = cu.fetchall() #vyber  vsech zaznamu z kurzoru
>>> for zaznam in vsechnyZaznamy:
…           print "\n"
…           for polozku in zaznam:
…                      print polozku
…
1 12.0 Karel Dadak VUT 2.00999999046
2 9.0   Jana  Tlusta CVUT 1.95000000021
3 23.0 Josef Hora ZCU 1.45000004768
4 56.0 Krystof Kolumbus KU 1.29999995232
5 31.0 Josef Druhy ZCU 2.04999995232
>>> for row in cu: print row.Prijmeni,row.Vek
    //nic se nevytiskne, protože kurzor se provedením fetchall dostal az na konec
>>> cu.execute("select * from Studenti where Vek>10")    nově ho naplníme
>>> jedenZaznam = cu.fetchone()
>>> print jedenZaznam[3]
Dadak
>>> jedenZaznam = cu.fetchone()
>>> print jedenZaznam[3]
Hora
```

Pozor, kurzor dává interní reprezentaci objektu, pro výběr hodnot nutno použít index

```
for zaznam in vsechnyZaznamy: print zaznam

<pyodbc.Row object at 0x00A5A368>
<pyodbc.Row object at 0x00A5A4E8>
<pyodbc.Row object at 0x00A5A770>
<pyodbc.Row object at 0x00A5A140>
>>> for zaznam in vsechnyZaznamy: print zaznam[2],

Karel Josef Krystof Josef
>>> for zaznam in vsechnyZaznamy: print zaznam[-1],

2.00999999046 1.45000004768 1.29999995232 2.04999995232
```

Python a XML

Obsah

- XML
- Validace – DTD a XSD
- Práce s XML - SAX a DOM
- Python a XML
- Tvorba XML bez použití knihoven
- Knihovna PyXML – SAX
- Knihovna PyXML – DOM
- Knihovna LXML – validace DTD a XSD

XML

- eXtensible Markup Language („rozšiřitelný značkovací jazyk“)
- Vyvinut a standardizován W3C
- Výměna dat mezi aplikacemi
- Publikování dokumentů – popisuje obsah ne vzhled (styly)
- Styly – vzhled CSS, transformace XSL
- DTD, XSD – definují jaké značky, atributy, typy bude XML obsahovat
- Parser zkontroluje, zda XML odpovídá definici

Syntaxe XML

- XML dokument je text, Unicode – zpravidla UTF-8
- „well-formed“ = správně strukturovaný
  - Jeden kořenový element
  - Neprázdné elementy musí být ohraničeny startovací a ukončovací značkou (<ovoce>Jablko</ovoce>)
  - Prázdné elementy mohou být označeny tagem „prázdný element“ (<ovoce/>)
  - Všechny hodnoty atributů musí být uzavřeny v uvozovkách – jednoduchých (') nebo dvojitých ("), ale jednoduchá uvozovka musí být uzavřena jednoduchou a dvojitá dvojitou. Opačný pár uvozovek může být použit uvnitř hodnot
  - Elementy mohou být vnořeny, ale nemohou se překrývat; to znamená, že každý (ne kořenový) element musí být celý obsažen v jiném elementu
- Příklad jidlo.xml

DTD

- Document Type Definition
- jazyk pro popis struktury XML případně SGML dokumentu
- Omezuje množinu připustných dokumentů spadajících do daného typu nebo třídy
- DTD tak například vymezuje jazyky HTML a XHTML.
- Struktura třídy nebo typu dokumentu je v DTD popsána pomocí popisu jednotlivých elementů a atributů. Popisuje jak mohou být značky navzájem uspořádány a vnořeny. Vymezuje atributy pro každou značku a typ těchto atributů.
- Připojení ke XML: `<!DOCTYPE kořen SYSTEM "soubor.dtd">`
- Příklad DTD
- DTD je poměrně starý a málo expresivní jazyk. Jeho další nevýhoda je, že DTD samotný není XML soubor.

XSD

- XML Schema Definition
- Popisuje strukturu XML dokumentu
- Definuje:
  - místa v dokumentu, na kterých se mohou vyskytovat různé elementy
  - Atributy
  - které elementy jsou potomky jiných elementů
  - pořadí elementů
  - počty elementů
  - zda element může být prázdný, nebo zda musí obsahovat text
  - datové typy elementů a jejich atributů
  - standardní hodnoty elementů a atributů
- Příklad XSD

Aplikace XML

- XHTML – Nástupce jazyka HTML.
- RDF – Resource Description Framework, specifikace, která umožňuje popsat metadata, např. obsah a anotace HTML stránky.
- RSS – je rodina XML formátů, sloužící pro čtení novinek na webových stránkách
- SMIL – Synchronized Multimedia Integration Language, popisuje multimedia pomocí XML.
- MathML – Mathematical Markup Language je značkovací jazyk pro popis matematických vzorců a symbolů pro použití na webu.
- SVG – Scalable Vector Graphics je jazyk pro popis dvourozměrné vektorové grafiky, statické i dynamické (animace).
- DocBook – Sada definic dokumentů a stylů pro publikační činnost
- Jabber – Protokol pro Instant messaging
- SOAP – Protokol pro komunikaci mezi Webovými službami
- Office Open XML, OpenDocument – Souborový formát určený pro ukládání a výměnu dokumentů vytvořených kancelářskými aplikacemi

Verze XML

- Aktuální verze XML je 1.1 (od 16. srpna 2006)
- První verze XML 1.0
- Obě verze se liší v požadavcích na použité znaky v názvech elementů, atributů atd.
  - Verze 1.0 dovolovala pouze užívání znaků platných ve verzi Unicode 2.0, která obsahuje většinu světových písem, ale neobsahuje později přidané sady jako je Mongolština a podobně.
  - Verze XML 1.1 zakazuje pouze řídící znaky, což znamená, že mohou být použity jakékoli jiné znaky.
- Je třeba poznamenat, že omezení ve verzi 1.0 se vztahuje pouze na názvy elementů a atributů. Jinak obě verze dovolují v obsahu dokumentu jakékoli znaky. Verze 1.1 je tedy nutná, pokud potřebujeme psát názvy elementů v jazyku, který byl přidán do Unicode později.

Související technologie

- Jmenné prostory v XML - Umožňují kombinovat značkování podle různých standardů v jednom dokumentu (příklad `tabulky.xml`)
- XSLT - Transformace dokumentu v XML na jiný, odvozený dokument - v XML, HTML nebo textový.
- XQuery - Dotazy nad daty v XML.

Práce s XML - SAX

- SAX – Simple API for XML
  - Sériový přístup ke XML
  - Proudové zpracování, při kterém se dokument rozdělí na jednotlivé jeho části
  - Pak se volají jednotlivé události, které ohlašují nalezení konkrétní části
  - Způsob jejich zpracování je na programátorovi
  - Vhodné pokud se čte celý obsah souboru
  - Nízké paměťové nároky, vysoká rychlost, nelze zapisovat

Práce s XML - DOM

- Document Object Model
- Objektově orientovaná reprezentace XML
- Umožňuje modifikaci obsahu, struktury
- DOM umožňuje přístup k dokumentu jako ke stromu
- Celý XML dokument v paměti (náročné na paměť)
- Vhodné tam, kde přistupujeme k elementům náhodně

Tvorba XML bez použití knihoven

- Příklad txt_to_xml.py

XML knihovny Pythonu

- PyXML
  - http.//sourceforge.net/projects/pyxml
- LXML
  - http://pypi.python.org/pypi/lxml

PyXML – SAX

- Parser čte XML dokument a pro každou dílčí část vyvolá událost
- My musíme napsat obsluhu události (handler)
- Import z knihovny:
  - `from xml.sax import handler, make_parser`
- Vytvoření parseru:
  - `parser = make_parser()`
- Parsování:
  - `parser.parse(infile)`
- Vytvoření třídy handleru:
  - `Class MySaxHandler(handler.ContentHandler)`
  - Uvnitř např. metody `startElement`
- Nastavení handleru:
  - `parser.setContentHandler(handler)`
- Zjištění well-formed: Příklad `sax_ver.py`
- Práce s elementy: Příklad `sax_elem.py`
- Práce s atributy: Příklad `sax_elem.py`

PyXML – DOM

- Všechny objekty v paměti, stromová struktura
- Uzly stromu – nody
- Typy nodů
  - Node – základní prvek a předek dalších druhů nodů
  - Document – počáteční uzel
  - Attr – atribut
  - Element – element
  - Text – obsah elementu
- Knihovny pro práci s DOM – **minidom**, ElementTree,…
- Parsování: `doc = minidom.parse(inFile)`
- Kořen stromu: `rootNode = doc.documentElement`
- Zjištění určitých potomků: `ovoce = rootNode.getElementsByTagName("ovoce")`
- Příklady: `dom_ver.py, dom_elem.py, dom_atr, dom_add.py`

LXML – validace DTD

- Import z knihovny:
  - `from lxml import etree`
- Postup validace:
  - `doc = etree.parse(xmlName)`
  - `dtd = etree.DTD(dtdfile)`
  - `if dtd.validate(doc) == True: ...`
- Příklad: `val_dtd.py`

LXML – validace XSD

- Postup validace:
  - `doc = etree.parse(xmlName)`
  - `xmlschema_doc = etree.parse(xsdfile)`
  - `xmlschema = etree.XMLSchema(xmlschema_doc)`
  - `if xmlschema.validate(doc) == True:`

    `...`
- Příklad `val_xsd.py`

Python a Web

Webové frameworky (1)

- XIST, HTMLTags, HTMLgen, HyperText
  - Tvorba statických stránek
  - Stránka je programově sestavena, lze ji uložit na disk
  - Lze snadno a rychle vygenerovat stránky
  - Použití spíše jako výstup programu (export sady stránek)
- Django
  - Oblíbená rozsáhlá knihovna, rychlý vývoj
  - Tvorba relačních webů – admin rozhraní, uživatelská rozhranní s různými právy

Webové frameworky (2)

- TurboGears
  - Velká knihovna
  - Tvorba webů založených na databázi (obsahuje např. knihovnu SQLObject – mapování objektů do relačních databází)
- Zope
  - Tvorba serverů se správou objemných systémů
  - O-o skriptovací skriptovací jazyk
  - Obsahuje ZODB – databází objektů
  - Nová verze Zope 3

Webové frameworky (3)

- Další
  - CherryPy
  - Pylons
  - web.py
  - QP, Gizmo
  - …
- web2py (dříve Gluon)
  - Vývoj přes webové rozhranní
  - Využívá řízení psané v pythonu
  - Vícejazyčná podpora
  - Logování chyb

XIST

- Rozšířený HTML/XML generator
- http://www.livinglogic.de/Python/xist/
- Předchůdci: HTMLgen, HyperText
- Stromová tvorba HTML, parsing HTML transformace
- Příklad tvorba HTML

Logické programování – použití: um. Int., databáze, sém.web

PROLOG (programming in Logic)

Program popisuje "*svět*" Prologu (nevyjadřuje tok řízení = vývojový diagram výpočtu)

tvoří jej

databáze faktů a pravidel (tzv. klauzulí), které mají tvar:

fakta: predikát(arg1, arg2, ...argN).

argumenty mohou být konstanty nebo proměnné

pravidla: hlava :- tělo. „:-“ čteme „jestliže“

hlavou je predikát s případnými argumenty, tělem je posloupnost faktů příp. vázaných konjunkcí „,“ nebo disjunkcí „;”

predikát(argumenty) :- fakta.

cíle: ?- predikát(arg1, arg2, ...argN).

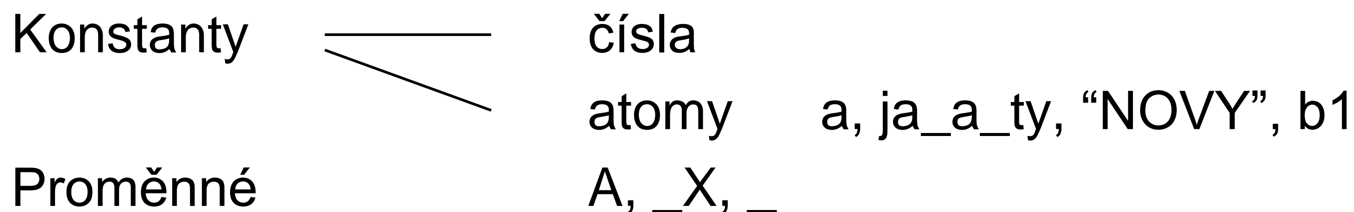
Zápisem cíle spustíme výpočet, výsledkem jsou hodnoty proměnných cíle

elementy programu jsou

konstanty, proměnné, struktury

společně se nazývají termy

Př.



Konstanty — čísla

atomy a, ja_a_ty, “NOVY”, b1

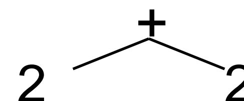
Proměnné A, _X, _

začínají velkým písmenem nebo podtržítkem. _ je tzv.anonymní proměnná, její hodnota nás nezajímá a nevypisuje se

Struktury muz(jan),

cte(student(jan,novy),kniha(A,Nazev))

2+2 je totéž jako +(2, 2)



Mohou být: 1)tvořené operátory a operandy

2)tvořené seznamy – výklad později

Př. dum.pro

Jméno fce tzv funktor. ma, dům, dvere jsou atomy

argumenty

```
ma(dum, dvere).
ma(dum, okno).
ma(dvere, klika).
rozbite(dvere).
```

fakta

```
?-ma(dum, dvere).
?-ma(Neco, klika).
?-ma(dum, Neco).
?-ma(X, Y), rozbite(Y).
```

Cíle =dotazy

Odpovi yes
Odpovi Něco = dvere
Odpovi Něco = dvere
Odpovi X=dům, Y=dvere

Postup spuštění:
V interaktivnim režimu spustit Amzi Prolog, pak Listener, pak Start. Vypíše se prompt ?- Pak zavedeme program pomocí: Listener, Consult a z okna Otevřít vybereme 1dum.pro Tím Prolog zná co platí v jeho světě a můžeme se ptát na cíle. Po odpovědi odřádkujeme Při opravě programu a jeho uložení zavádíme jej pomocí Listener, Reconsult, které přepíše původní databázi faktů a pravidel. Consult by přidalo nový program za starý

Pravidla

cil(argumenty):-logicka formule z podcílů.

Matematický zápis pravidla by byl $p \leftarrow q1 \,\&\, q2 \,\&\, \ldots \,\&\, qn$.

Př. rodina.pro

```
rodic(eva, petr). rodic(eva, jan).             %eva je rodicem petra a jana
rodic(jindrich, jan). zena(eva). muz(petr).% vlastni jmena zacinaji malym
muz(jan). muz(jindrich).                        %pismenem, jsou to konstanty
matka(M,D):-rodic(M,D), zena(M).
otec(O,D):- rodic(O,D), muz(O).
```

Význam pravidel:

M je matkou D, jestliže M je rodičem D a zároveň M je žena.

O je otcem D, jestliže O je rodičem D a zároveň O je mužem.

Jak definovat rodinné vztahy? Zkuste si to

deda(D, V) :- rodic(D, X), rodic(X, V).

Predikáty se odlišují dle jména i arity

otec(O,D) je jiný predikát než otec(O)

Prvý je vlastně binární relací mezi dvěma osobami, druhý je unární relací říkající pouze zda nějaká osoba je otcem

Anonymní proměnná = nechceme znát její hodnotu

Může jich být více v klauzuli a navzájem nesouvisí

Označuje se podtržítkem _

Př.

Definice otce = někdo, kdo je rodičem a mužem a nezajímá nás kdo jsou děti

otec(O):- rodic(O,_), muz(O).

Def.syna zkuste sami

Databázi lze i průběžně doplňovat

Rekurze

Definice českých panovníků – přemyslovců

Premyslovec je premysl_orac

Premyslovec je syn premyslovce

Zapsáno prologovsky:

```
premyslovec(premysl_orac).
premyslovec(P):-syn(P,R), premyslovec(R).
```

Alternativně to lze vyjádřit jedinou klauzulí: Přemyslovec P je buď přemysl-oráč, nebo je to syn nejakeho R, který je přemyslovcem.

```
premyslovec(P):-
    (P=premysl_orac) ; (syn(P,R),premyslovec(R)).
```

Pokud nám odpověď nestačí, můžeme ji odmítnout zápisem středníku

Prolog pak zkusí nalézt jiné řešení

1. Jak vytvořit dotaz na jména všech přemyslovců?
2. Jak definovat potomka po meči?
3. Jak definovat příbuznost osob X a Y po meči?
4. Jak definovat příbuznost osob X a Y

Ad1 Předpokládejme, že v databázi prologu máme fakta o potomcích tvaru syn(nezamysl, kresomysl) atd. Pak se lze dotazovat

```
?-premyslovec(P).
P= … ;            pokud na odpoěď reagujeme ;  bude se hledat jiné řešení.
P= … ;
…
no                až se vyčerpají všechny možnosti, bude odpověď no.
```

Všechna řešení lze nalézt pohodlněji jediným složeným dotazem

```
?-premyslovec(P), write(P), fail.
```

který váže konjukcí tři podcíle: nalezeni přemyslovce P, výpis P, a vždy nesplněného predikátu fail, který způsobí hledání jiného řešení, které samozřejmě zase skončí fail, ale vypíše nám další jméno.

Princip rezoluce aneb jak Prolog hledá řešení

Předpokládejme pravidla a , b tvaru

a :- a_1, a_2, … , a_n.

b :- b_1, b_2, … , b_m.

Nechť b_i je a

Pak rezolucí je

b :- b_1, b_2, … b_{i-1}, a_1, a_2, … , a_n , b_{i+1} , … , b_m.

Tzn. když se při plnění cílů z těla pravidla b narazí na zpracování cíle b_i alias a, začnou se zpracovávat cíle těla pravidla a.

Jednotlivé cíle jsou predikáty, které Prolog porovnává s klauzulemi v jeho databázi. Proces porovnávání, když dopadne úspěšně, se nazývá **unifikací**

Unifikace

- porovná-li se volná proměnná s konstantou, naváže se na tuto konstantu,
- porovnají-li se dvě volné (neinstalované) proměnné, stanou se synonymy,
- porovná-li se volná proměnná s termem, naváže se na tento term,
- porovnají-li se termy, které nejsou volnými proměnnými, musí být pro úspěšné porovnání stejné.

Př. unifikace v dotazu ?- X = Y, Y = a. Pak X i Y má hodnotu a

Pozor, operátorem „=„ unifikujeme, nepřiřazujeme, pro přiřazení slouží operátor „is“

Př.3nsd.pro největšího společného dělitele

1.Největší společný dělitel A a A je A

2.Největší společný dělitel A a B je NSD jestliže

při A větším než B platí: A1 je A-B a největší společný dělitel A1 a B je NSD

3. při A menším než B platí B1 je B-A a největší společný dělitel B1 a A je NSD

```
nsd(A,A,A).                                  % 1
nsd(A,B,NSD) : - A>B,                        % 2
                  A1 is A-B,
                  nsd(A1,B,NSD).
nsd(A,B,NSD) : - A<B,                        % 3
                  B1 is B-A,
                  nsd(B1,A,NSD).
```

Po zkonzultování souboru s programem spustíme výpočet např.

?-nsd(16,12,X).

Př.4fakt.pro Výpočet faktoriálu

Faktoriál 1 je 1.

Faktoriál N je F, jestliže platí, že nějaké M má hodnotu N-1 a současně faktoriál M je G a současně F má hodnotu G*N

```
fakt(1,1).
fakt(N,F) :- M is N-1,
                       fakt(M,G),
                        F is G * N.
```

Výpočet spustíme dotazem např.

```
?-fakt(3,X).
```

Pokud bychom výsledek odmítli vznikne chyba přeplnění stacku. Objasněte proč?

Zásady při plnění cílů

- Dotaz může být složen z několika cílů.
- Při konjunkci cílů jsou cíle plněny postupně zleva.
- Pro každý cíl je při jeho plnění prohledávána databáze od začátku.
- Při úspěšném porovnání klauzule s cílem je její místo v databázi označeno ukazatelem. Každý z cílů má vlastní ukazatel.
- Při úspěšném porovnání cíle s hlavou pravidla, pokračuje výpočet plněním cílů zadaných tělem pravidla.
- Cíl je splněn, je-li úspěšně porovnán s faktem databáze, nebo s hlavou pravidla databáze a jsou splněny podcíle těla pravidla.
- Není-li během exekuce některý cíl splněn ani po prohlédnutí celé databáze, je aktivován mechanismus návratu.
- Splněním jednotlivých cílů dotazu je splněn globální cíl a systém vypíše hodnoty proměnných zadaných v dotazu.
- Zjistí-li se při výpočtu, že globální cíl nelze splnit, je výsledkem no.

Mechanismus návratu

- exekuce se vrací k předchozímu splněnému cíli, zruší se instalace (navázání) proměnných a pokouší se opětovně splnit tento předchozí cíl prohledáváním databáze dále od ukazatele pro tento cíl,
- splní-li se opětovně tento cíl, pokračuje se plněním dalšího, (předtím nesplněného) vpravo stojícího cíle,
- nesplní-li se předchozí cíl, vrací se výpočet ještě více zpět (na vlevo stojící cíl).

Shrnutí základních principů

- **<u>Program</u> specifikujeme množinou klauzulí. Klauzule mají podobu faktů, pravidel a dotazu. Prolog zná pouze to, co je definované programem.**
- **<u>Fakt</u> je jméno relace a argumenty (objekty) v daném uspořádání. Uspořádání je důležité.**
- **<u>Pravidlo</u> vyjadřuje vztahy, které platí jsou-li splněny podmínky z těla (cíle). Hlavu tvoří vždy jen jeden predikát.**
- **<u>Dotaz</u> může tvořit jeden nebo více cílů. Cíle mohou obsahovat proměnné i konstanty, Prolog najde tolik odpovědí kolik je požadováno (pokud existují).**
- **<u>Proměnná</u> je v klauzuli obecně kvantifikována. Její platnost je omezena na klauzuli.**

- **<u>Definice predikátu</u> je posloupnost klauzulí pro jednu relaci. Predikát může určovat vztah, databázovou relaci, typ, vlastnost, funkci. Jméno predikátu musí být atomem.**
- **<u>Plnění cíle</u> provádí Prolog pro nový cíl prohledáváním databáze od začátku, při opakovaném pokusu prohledáváním od naposled použité klauzule.**
- **<u>Rekurzivní definice predikátu</u> musí obsahovat ukončovací podmínku.**
- **<u>Typ termu</u> je rozpoznatelný syntaxí. Atomy a čísla jsou konstanty. Atomy a proměnné jsou jednoduchými termy. Anonymní proměnná představuje neznámý objekt, který nás nezajímá. Struktury jsou složené typy dat. Pravidlo je strukturou s funktorem :-**
- **<u>Funktor</u> je určen jménem a aritou**

Unifikace termů podrobněji

Dva termy jsou úspěšně porovnány (lze také říci, že si jsou podobné), pokud

- **jsou totožné nebo**
- **proměnné v termech lze navázat na objekty tak, že po navázání proměnných jsou tyto termy totožné.**

Př. datum(D, M, 2009) datum(X, 12, R)

jsou unifikovatelné: D je X, M je 12, 2009 je R

datum(D, M, 2009) datum(X, 12, 2004)

nejsou

bod(X, Y, Z) datum(D, M, 2003)

nejsou

datum(D, M, 2009) datum(X, 12)

nejsou

- **Prolog vybere vždy nejobecnější možnost porovnání**
- **Porovnání vyjadřuje operátor =**

Při porovnání proměnné se strukturou je nutné vyloučit případ, kdy se tato proměnná vyskytuje ve výrazu

Př.?- X = f(X). %neproveditelné porovnání

Způsobí   navázání X na f(X)

→ na f(X)

→ na f(X) …

Způsobí to stack overflow

! ! Aritmetické výrazy jsou termy ! !

?- X = +(2, 2).   Bude odpověď

X = 2 + 2

?- X = 2 + 2 .   Bude také odpověď

X = 2 + 2

Protože + je jménem struktury a 2 , 2 jsou argumenty

Pro vyhodnocení nutno použít predikát is

?- X is +(2, 2).

X = 4

?- X is 2 + 2 .

X = 4

Seznamy

Seznam je rekurzivní datová struktura tvaru:

[e1, e2, …, en], kde ei jsou elementy seznamu.

Elementy seznamů jsou často opět seznamy

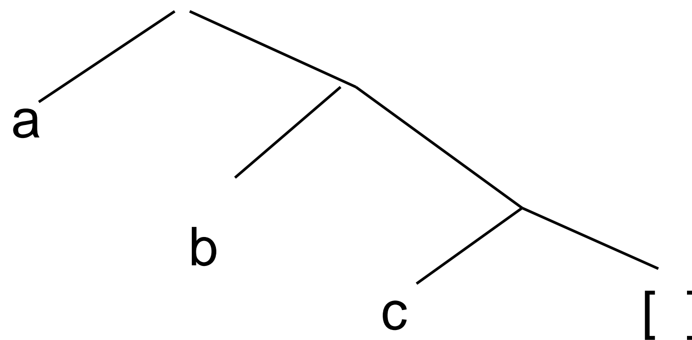
Svislítkem lze seznam rozdělit na prvý element (také se nazývá hlava) a zbytek seznamu

[Hlava | Zbytek]

[ ] to je prázdný seznam

Př. [a, b, c] dtto [a | [b, c]] dtto [a, b | [c]] dtto [a, b, c | [ ]]

Graficky to zachycuje obr. Všimněte si, že zbytek seznamu je opět seznam (případně prázdný seznam)



Příklady, jaké budou výsledky porovnání

[X] = [a, b, c, d]	?
no	
[X | Y] = [a, b, c, d]	?
yes	X=a, Y=[b,c,d]
[X, Y | Z] = [a, b, c, d]	?
yes	X=a, Y=b, Z =[c,d]
[X] = [[a, b, c, d]]	?
yes	X= [a, b, c, d]
[X | Y] = [[a, [b, c]], d]	?
yes	X=[a, [b, c]], Y=[d]
[X | Y] = [ ]	?
no	
[X | Y] = [d]	?
yes	X=d, Y= [ ]

Predikáty pro práci se seznamy

Zjištění, zda v nejvyšší úrovni seznamu existuje prvek dělá predikát

member(Prvek,Seznam)

Slovně lze jeho činnost vyjádřit

member platí, 1. je-li prvek na začátku seznamu, 2. jinak member platí pokud prvek je ve zbytku seznamu

```
member(X,[X|_]).                          %1.
member(X,[_|Y]) :- member(X,Y).           %2.
?-member(a, [b,a,c,[d,a]]).
yes
?-member(a, [b,c,[d,a]]).
no
```

Tento predikát je mezi standardními, nemusíme ho programovat

Nalezení posledního prvku seznamu

last(Seznam, Prvek)

1. Je-li v seznamu jen jeden prvek, tak je tím posledním,
2. jinak je to poslední prvek ze zbytku seznamu

```
last([X],X).                    %1.
last([_|T],X) :- last(T,X).     %2.

?- last([a,[b,c]],X).
X = [b,c]
```

Odstranění prvku ze seznamu

delete(Původníseznam, Výslednýseznam, Prvek)

```
delete([X|T],T,X).
delete([Y|T],[Y|T1],X) :- delete(T,T1,X).
```

Zkusme formulovat slovně

Je-li prvek prvním v seznamu,je výsledkem zbytek, jinak je výsledkem seznam se stejným prvním prvkem, ale se zbytkem, v němž je vynechán uvažovaný prvek

Dotazovat se můžeme např.

?- delete([a,b,a],L,a).

L = [b,a] ;	středníkem jsme odmítli řešení, hledá jiné
L = [a,b] ;	znovu jsme odmítli, hledá jiné
no	další řešení již neexistuje

Přidání seznamu k seznamu

append(Seznam1,Seznam2,Výsledek)

```
append([ ],X,X).
append([A|B],X,[A|C]):-append(B,X,C).
```

Formulujme predikát slovně: Když přidáme k prázdnému seznamu seznam X, výsledkem bude seznam X. Když přidáme k seznamu (jehož prvý prvek je A a jeho zbytek je B) seznam X, bude výsledný seznam mít prvý prvek A a jeho zbytkem bude seznam C, který vznikne přidáním k seznamu B seznam X.

```
?- append([a,b],[c],X).
X = [a,b,c]          a pokud ted odradkujeme (tj. odpoved nam staci), odpovi
yes
?-
```

Další pozoruhodnosti append

append([ ],X,X).
append([A|B],X,[A|C]):-append(B,X,C).

Zeptáme se, jaké dva seznamy musíme appendnout, aby vzniklo [a,b,c]
Prolog nám je najde a pokud budeme chtít, najde nám všechny možnosti

```
?- append(X,Y,[a,b,c]).
X = []
Y = [a,b,c] ;
X = [a]
Y = [b,c] ;
X = [a,b]
Y = [c] ;
X = [a,b,c]
Y = [] ;
no
?-
```

Námi definovaný append je obousměrný (lze zaměnit co je vstup a co výstup)

Argumenty funkcí (tj. prologovské proměnné) mohou být bound (vázané) nebo free (volné). Zkrácené je označme b, f

Příklady :

bbb **?-append([a, b], [c], [a, b, c]).**
yes

bbf **?-append([a, b], [c], S3).**
S3 = [a,b,c] ;
no

bfb **?-append([a, b], S2, [a,b,c,d]).**
S2 = [c,d] ;
no
?-

bff **?-append([a, b], S2, S3).**
S2 = H159
S3 = [a,b | H159] ;
no

fbb **?-append(S1, [c, d], [a,b,c,d]).**
S1 = [a,b] ;
no

Formulujte příklady dotazů slovně!

fbf ?-append(S1, [c, d], S3).

S1 = [ ]

S3 = [c,d] ;

S1 = [H277] H s číslem je interní proměnná Prologu

S3 = [H277,c,d] ;

S1 = [H277,H303]

S3 = [H277,H303,c,d] ;

atd

ffb ?-append(S1, S2, [c, d]).

S1 = []

S2 = [c,d] ;

S1 = [c]

S2 = [d] ;

S1 = [c,d]

S2 = [] ;

no

fff ?-append(S1, S2, S3).

```
S1 = []
S2 = H253
S3 = H253 ;
S1 = [H323]
S2 = H253
S3 = [H323 | H253] ;
S1 = [H323,H349]
S2 = H253
S3 = [H323,H349 | H253] ;
  atd
```

Vícesměrnost dalších predikátů

Predikát member je také vícesměrný

member(X,[X|_]).

member(X,[_|Y]) :- member(X,Y).

?- member(X,[a,[b,c],d]). %Formulujte slovně!

X = a ;

X = [b,c] ;

X = d ;

no

?-

Vícesměrnost dalších predikátů

Také delete je vícesměrný predikát

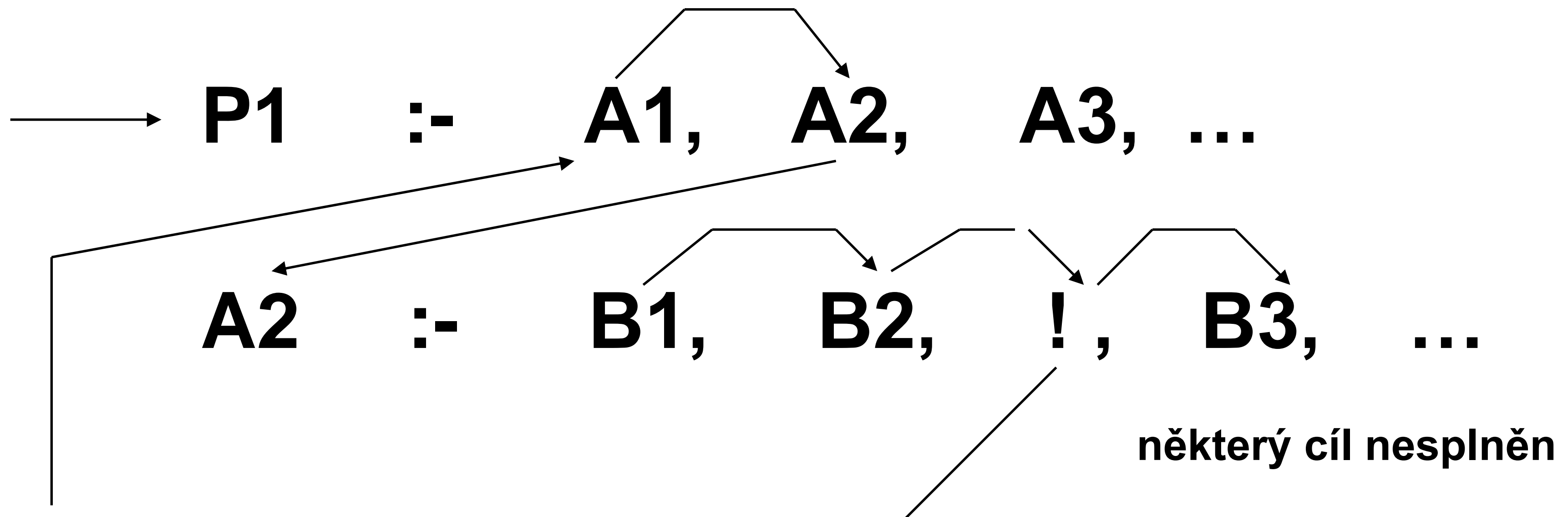
```
delete([X|T],T,X).
delete([Y|T],[Y|T1],X) :- delete(T,T1,X).

?- delete(X,[a,b],c).                %Formulujte slovně!
X = [c,a,b] ;
X = [a,c,b] ;
X = [a,b,c] ;
no
?-
```

Seznamy znaků jsou řetězce. Řetězce se uzavírají do řetězcových závorek

```
?- [X,Y|Z]="abcd".

X = 97
Y = 98
Z = [99,100]
yes
?-
```

Ovlivnění mechanismu návratu

Mechanismus navracení lze ovlivnit tzv Predikátem řezu označený jako ! Ten způsobí při nesplnění některé cíle za ním přeskok až na nové plnění cíle A1, tj způsobí nesplnění cíle A2.



Samotný ! je vždy splněn

Predikát řezu

- Použijeme jej, když chceme zabránit hledání jiné alternativy
- Odřízne další provádění cílů z pravidla ve kterém je uveden
- Je bezprostředně splnitelným cílem. Projeví se pouze, když má přes něj dojít k návratu
- Změní mechanismus návratu tím, že znepřístupní ukazatele vlevo od něj ležících cílů (přesune je na konec Db)

Př. použití řezu

```
fakt(N,1) :-N=0,!.     % ! Zabrání výpočtu fakt pro záporný argument při odmítnutí výsledku
fakt(N,F) :- M is N-1,
             fakt(M,G),
             F is G * N.
?-fakt(1,X).
X=1;
no
```

Př.Hanoiské věže

Jsou dány 3 trny A, B, C.

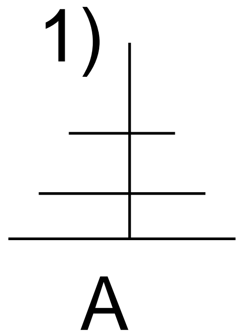
Na A je navléknuto N kotoučů s různými průměry tak, že vytváří tvar pagody (menší je nad větším).

Přemístěte kotouče z A na C s využitím B jako pomocného trnu tak, aby stále platilo, že nikdy není položen větší kotouč na menší

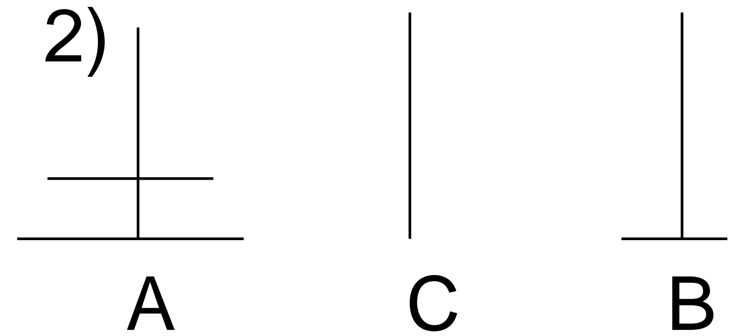
Následující obrázek ukazuje postup přemístění pro 3 kotouče

Př.Hanoiské věže

1)

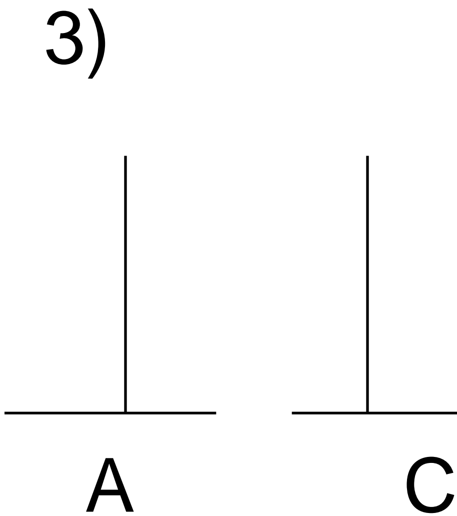


2)

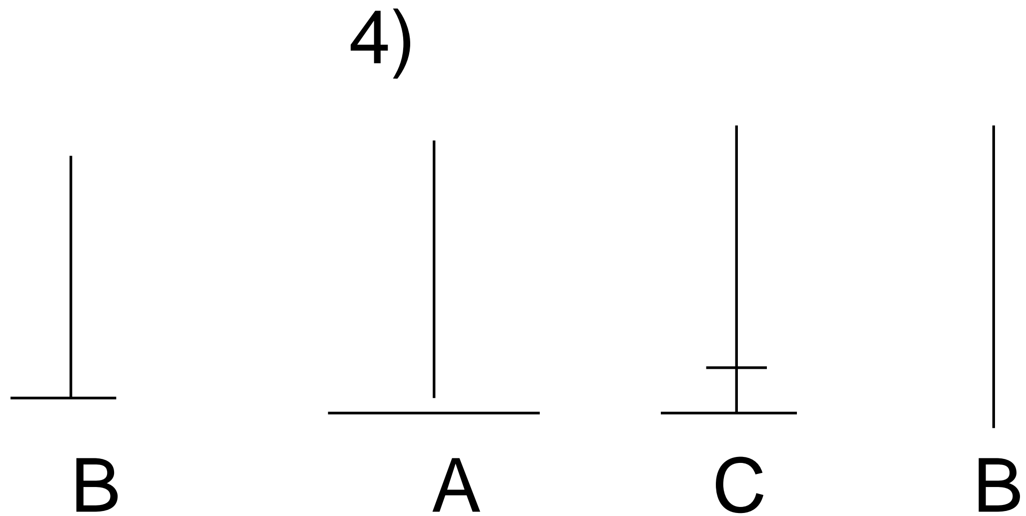


A C B A C B

3)



4)



A C B A C B

Př.Hanoiské věže (6Hanoi.pro)

```
hanoi(N) :- presun(N,levy,stred,pravy), !.
presun(0,_,_,_) :- !.
presun(N,A,B,C) :-    %presun z A na B za pouziti pomocného C

                 M is N-1,
           presun(M,A,C,B),
           inform(A,B),
           presun(M,C,B,A).
inform(A,B) :- write([presun,disk,z,A,na,B]),
           nl.
```

Spustíme např.

?-hanoi(5). nebo ?-veze. Což způsobí výpis výzvy, přečtení počtu kotoučů, výpočet přesunů a to se opakuje až do zadání záporného počtu kotoučů. Je to i příklad, jak udělat cyklus, repeat nedělá nic, je ale libovolněkrát splnitelný, read a write nejsou při návratu znovusplnitelné (vazbu na X nelze rozvázat), fail způsobí nesplnění a tedy návrat (až k repeat)

Standardní predikáty jazyka Prolog

Zahrnují skupiny predikátů

- I/O operace
- Řídící predikáty a testy
- Predikáty pro práci s aritmetickými výrazy
- Predikáty pro manipulaci s databází
- Predikáty pro práci se strukturami

Jazyk Prolog je vlastně tvořen resolučním mechanismem a skupinou standardních predikátů, z nichž si většinu ukážeme, ale nemusíte si všechny pamatovat.

I/O operace

write(X) zápis termu do výstupního proudu

nl odřádkování

tab(X) výstup X mezer

read(X) čtení termu ze vstupního proudu

Řídící predikáty a testy

true	vždy splněný cíl
fail	vždy nesplněný cíl
var(X)	splněno, je-li X volnou proměnnou
nonvar(X)	splněno, neplatí-li var(X)
atom(X)	splněno, je-li X instalováno na atom
integer(X)	splněno, je-li X instalováno na integer
atomic(X)	splněno, je-li X instalováno na atom nebo integer
not(X)	X musí být interpretovatelné jako cíl. Uspěje, pokud X není splněn
call(X)	X musí být interpretovatelné jako cíl. Uspěje, pokud X je splněn
halt	ukončí výpočet
X = Y	pokus o porovnání X s Y
X \= Y	opak =
X == Y	striktní rovnost
X \== Y	úspěšně splněn, neplatí-li ==
!	změna mechanismu návratu
repeat	nekonečněkrát splnitelný cíl
X , Y	konjunkce cílů
X ; Y	disjunkce cílů

Predikáty pro práci s aritmetickými výrazy

X is E E musí být aritm. výraz, který se vyhodnotí a porovná s X

E1 + E2 při instalovaných argumentech (pod. –, *, /, mod)

E1 > E2 při instalovaných argumentech (pod. >=,<,=<,\=,=)

E1 =:= E2 uspěje, jsou-li si hodnoty E1, E2 rovny

E1 =\= E2 uspěje, nejsou-li si hodnoty E1, E2 rovny

Predikáty k manipulaci s databází a klauzulemi

To je již jen pro informaci

listing(X) výpis všech klauzulí na jejichž jméno je X instalováno

listing výpis celého programu

clause(X, Y) porovnání X a Y s hlavou a s tělem klauzule

asserta(X) přidání klauzule instalované na X na začátek databáze

assertz(X) totéž, ale přidává se na konec databáze

retract(X) odstranění prvého výskytu klauzule X z databáze

findall(X,Y,Z) všechny výskyty termu X v databázi, které splňují cíl Y jsou vloženy do seznamu Z

Závody gymnastek –91závod.pro

Typická úloha ze sobotní přílohy novin

Urcete jmena vitezek disciplin z techto informaci:

1) Dvorakova ani Sobotkova nevyhraly preskok ani bradla.
2) V preskoku nezvitezila Vera ani Ludmila.
3) Sobotkova se nejmenuje ani Vera ani Jirina a kamaradi
4) s Beckovou
5) Junkova neni ani Monika ani Jirina
6) Vera nevyhrala na bradlech, Monika nevyhrala v prostnych.
7) Jednou z disciplin byla kladina.
8)V kazde ze ctyr disciplin zvitezila jina zavodnice.

Závody gymnastek –91závod.pro

Čísla jednotlivých podmínek v zadání jsou v komentářích

```
vitpres(X,Y):-pojmen(X,Y),          %pozitivní cíl = důležité
    not((Y=sobotkova; Y=dvorakova)),                    %1
    not((X=vera; X=ludmila)).                           %2
vitbrad(X,Y):-pojmen(X,Y),
    not((Y=sobotkova; Y=dvorakova)),                    %1
    not(X=vera).                                        %6
vitpros(X,Y):-pojmen(X,Y),
     not(X=monika).                                     %6
vitklad(X,Y):-pojmen(X,Y).                              %7
```

Závody gymnastek –91závod.pro

```
pojmen(X,junkova):-jmeno(X),
        X\=monika, X\=jirina.                     %5
pojmen(X,sobotkova):-jmeno(X),
        X\==vera, X\==jirina.                     %3
pojmen(X,beckova):- jmeno(X).                     %4
pojmen(X,dvorakova):- jmeno(X).
jmeno(monika).                                    jsou
jmeno(jirina).                                    taková
jmeno(vera).                                      jména
jmeno(ludmila).
ruzne(A1,A2,A3,A4):-
   A1\=A2,A1\=A3,A1\=A4,A2\=A3,A2\=A4,A3\=A4.      %8
```

Závody gymnastek –91závod.pro

```
vitez(X1,Y1,X2,Y2,X3,Y3,X4,Y4):-
                    vitpres(X1,Y1), vitbrad(X2,Y2),
                    vitpros(X3,Y3), vitklad(X4,Y4),
                    ruzne(X1,X2,X3,X4), ruzne(Y1,Y2,Y3,Y4). %8

go:-    vitez(JPR,PPR,JBR,PBR,JPROS,  %tim se spusti
        PPROS,JKLAD,PKLAD),
        write(JPR),write(PPR),write(JBR),write(PBR),nl,
        write(JPROS),write(PPROS),
        write(JKLAD),write(PKLAD).
```

Závody gymnastek (permutacemi) –92závod1.pro

vypiseme seznam jmen a seznam prijmeni vitezu soutezi v poradi: vitezka Bradel, vitezka Kladiny, vitezka Preskoku, vitezka Prostnych

```
perm([],[]).
perm(L,[X|P]) :- del(X,L,L1), perm(L1,P).
del(X,[X|T],T).
del(X,[Y|T],[Y|T1]) :- del(X,T,T1).

%hleda poradove cislo N, prvku E, v zadanem seznamu
poradi(E,[E|T],1) :-!.
poradi(E,[X|T],N) :- poradi(E,T,M), N is M+1.
```

Závody gymnastek (permutacemi) –závod1.pro

```
%permutuj Jmena a Prijmeni (ta jsem popsal jen pocatecnimi pismeny)
vitez :- perm([j,l,m,v],J), perm([b,d,j,s],P),
    J=[J1,J2,J3,J4], P=[P1,P2,P3,P4],
% sob se nejmenuje ver    sob se nejmenuje jir
    poradi(s,P,N1),poradi(v,J,N2), N1\=N2, poradi(j,J,N3), N1\=N3,
%jun neni mon    jun neni jir
    poradi(j,P,N4),poradi(m,J,N5), N4\=N5, N4\=N3,
%bradla nevyhrala vera ani dvorak. ani sobotk
    J1\=v, P1\=d, P1\=s,
%preskok nevyhrala vera ani lud. ani dvorak, ani sobotk
    J3\=v, J3\=l, P3\=d, P3\=s,
%prostna nevyhrala monika
    J4\=m,
    write(J),write(P).
```

Funkcionální programování úvod

Probereme základy jazyka LISP. Plný popis jazyka najdete např.na http://www-2.cs.cmu.edu/afs/cs/project/ai-repository/ai/html/cltl/cltl2.html

- **Imperativní jazyky jsou založeny na von Neumann architektuře**
  - **primárním kriteriem je efektivita výpočtu**
  - **Modelem je Turingův stroj**
  - **Základní konstrukcí je příkaz**
  - **Příkazy mění stavový prostor programu**
- **Funkcionální jazyky jsou založeny na matematických funkcích**
  - **Program definuje funkci**
  - **Výsledkem je funkční hodnota**
  - **Modelem je lambda kalkul (Church 30tá léta)**
  - **Základní konstrukcí je výraz (definuje algoritmus i vstup)**
  - **Výrazy jsou:**
    - **Čisté = nemění stavový prostor programu**
    - **S vedlejším efektem = mění stavový prostor**

Funkcionální programování úvod

Vlastnosti čistých výrazů:

- **Hodnota výsledku nezávisí na pořadí vyhodnocování (tzv.Church-Roserova vlastnost)**
- **Výraz lze vyhodnocovat paralelně , např ve výrazu (x*2+3)/(fce(y)*x) lze pak současně vyhodnococat dělence i dělitele. Pokud ale fce(y) bude mít vedlejší efekt a změní hodnotu x, nebude to čistý výraz a závorky paralelně vyhodnocovat nelze.**
- **nahrazení podvýrazu jeho hodnotou je nezávislé na výrazu, ve kterém je uskutečněno (tzv. referenční transparentnost)**

  -vyhodnocení nezpůsobuje vedlejší efekty

  -operandy operace jsou zřejmé ze zápisu výrazu

  -výsledky operace jsou zřejmé ze zápisu výrazu

Funkcionální programování- úvod

Př. nalezení největšího čísla funkcionálně vyjádřeno

a) Ze dvou čísel.

označme symbolem <u>def</u> definici fce

max2(X, Y) <u>def</u> jestliže X > Y pak X jinak Y

b) Ze čtyř

max4(U, V, X, Y) <u>def</u> max2(max2(U, V), max2(X,Y))

c) Z n čísel

max(N) <u>def</u> jestliže délka(N) = 2
pak max2(prvý-z(N), druhý-z(N))
jinak max2(prvý-z(N), max(zbytek(N)))

Prostředky funkcionálního programování jsou:

- **Kompozice složitějších funkcí z jednodušších**
- **rekurze**

Funkcionální programování- úvod

Def.: Matematická fce je zobrazení prvků jedné množiny, nazývané definiční obor fce do druhé množiny, zvané obor hodnot

- **Funkcionální program je tvořen výrazem E.**
- **Výraz E je redukován pomocí přepisovacích pravidel**
- **Proces redukce se opakuje až nelze dále redukovat**
- **Tím získaný výraz se nazývá normální formou výrazu E a je výstupem funkcionálního programu**

Př. Aritmetický výraz E = (4 +7+ 10) * (5 – 2)=(11+10)*(5-2) = 20*(5-2) =20*3=60

s přepis. pravidly určenými tabulkami pro +, *, …

Smysl výrazu je redukcemi zachován = vlastnost referenční transparentnosti je vlastností vyhodnocování funkcionálního programu

Funkcionální programování- úvod

- **Church-Roserova věta:** Získání normální formy je nezávislé na pořadí vyhodnocování subvýrazů

- Funkcionální program sestává z definic funkcí (algoritmu) a aplikace funkcí na argumenty (vstupní data).
- Aparát pro popis funkcí je tzv **lambda kalkul**

  používá operaci aplikace fce F na argumenty A, psáno FA

  „ „ abstrakce ve tvaru λ (x) M [x]

  definuje fci (zobrazení) x → M [x]

Př. λ (x) x * x * x Tak to píší matematici, v programu je to uzávorkováno

forma = výraz

definuje bezejmennou fci x * x * x lambda výrazem

Funkcionální programování- úvod

- Lambda výrazy popisují bezejmenné fce
- „ „ „ jsou aplikovány na parametry např.

(λ (x) x * x * x) 5 = 5 * 5 * 5 = 125

popis funkce argumenty

aplikace (vyvolání) funkce

- Ve funkcionálním zápisu je zvykem používat prefixovou notaci ~ funkční notaci ve tvaru funktor(argumenty)

př. ((lambda (x) (* x x)) 5)

((lambda (y) ((lambda (x) (+ (* x x) y)) 2)) 3)

→ 7 přesvědčíme se v Lispu?

Funkcionální programování- úvod

V předchozím příkladu výraz ((lambda (x) (+ (* x x) y)) 2)) obsahuje vázanou proměnnou x na 2 a volnou proměnnou y. Ta je pak ve výrazu ((lambda (y) ((lambda (x) (+ (* x x) y)) 2)) 3) **vázána na 3**

V LISPu lze zapisovat také

((lambda (y x) (+ (* x x) y)) 2 3) dá to také 7 ?

((lambda (y x) (+ (* x x) y)) 3 2)

Ta upovídanost s lambda má důvod v přesném určení pořadí jaký skutečný argument odpovídá jakému formálnímu.

Pořadí vyhodnocování argumentů lze provádět:

- **Všechny se vyhodnotí před aplikací fce = eager (dychtivé) evaluation**
- **Argument se vyhodnotí těsně před jeho použitím v aplikaci fce = lazy (líné) evaluation**

Pochopit význam pojmu

- **Volná proměnná**
- **Vázaná proměnná**

Funkcionální programování- LISP

- Vývoj, verze: Maclisp, Franclisp, Scheme, Commonlisp, Autolisp
- Použití:
  - UI (exp.sys., symb.manipulace, robotika, stroj.vidění,přiroz.jazyk)
  - Návrh VLSI
  - CAD
- Základní vlastnost: **<u>Vše je seznamem</u>** (program i data)

Ovládání CLISPu

Po spuštění se objeví prompt [cislo radku]>

Interaktivně lze zadávat příkazy např (+ 2 3). LISP ukončíte zápisem (exit).

Uděláte-li chybu, přejdete do debuggeru, v něm lze zkoušet vyhodnocení, navíc zápisem Help se vypíší další možnosti, např Abort vás vrátí o 1 úroveň z debuggerů zpět (při chybování v debug. se dostanete do dalšího debug. vyšší úrovně). Program se ale většinou tahá ze souboru

K zatažení souboru s funkcemi použijte např.

(LOAD “d:\\moje\\pgs\\neco.lsp”) když se dobře zavede, odpovi T

Funkcionální programování- LISP

Datové typy:

-atomy: -čísla (celá i reálná)

-znaky

-řetězce "to je řetězec"

S - výrazy -symboly (T, NIL, ostatní)

k označování proměnných

a funkcí. T je pravda, NIL nepravda

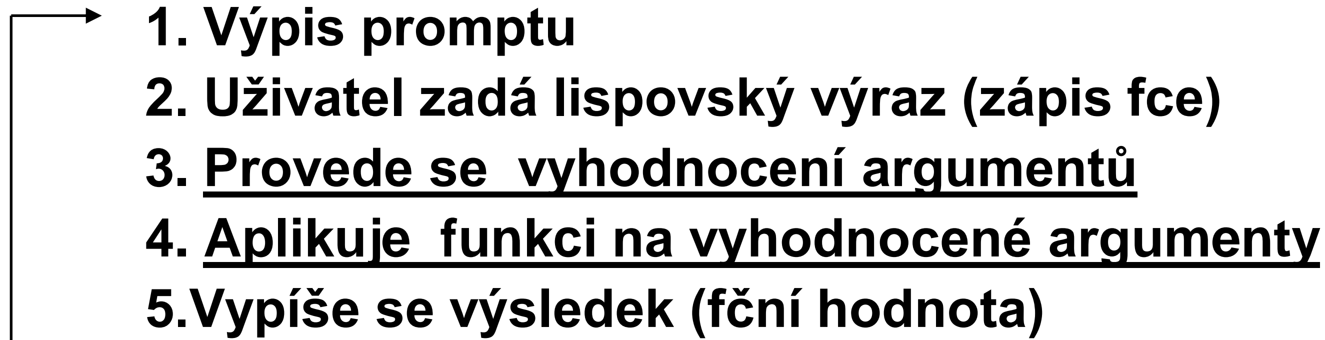
-seznamy (el1 el2 ...elN), NIL je prázdný seznam jako ()

- Seznamy mohou být jednoduché a vnořované
  např (sqrt (+ 3 8.1)) je seznam použitelný k vyvolání fce
- Velká a malá písmena se nerozlišují v CommonLispu

Funkcionální programování- LISP

Postup vyhodnocování (často interpretační):

Cyklus prováděný funkcí eval:



- **Pár příkladů s aritmet. funkcemi spustíme v Lispu**

(+ 1111111111111111 2222222222222222)

má nekonečnou aritmetiku

(sqrt (* 4 pi)   zná matem fce a pi

2*2   způsobí chybu

- **Při chybě se přejde do nové úrovně interpretu**
- **V chybovém stavu lze napsat help ¬**
- **Abort = návrat o úroveň výše**

Funkcionální programování- LISP

- Funkce pracují s určitými typy hodnot
- Typování je prostředkem zvyšujícím spolehlivost programů
- Typová kontrola:

  -statická znáte z Javy

  -dynamická (Lisp, Prolog, Python) – nevyžadují deklaraci typů argumentů a funkčních hodnot

Zajímavosti -komplexní čísla (* #c(2 2) #c(1.1 2.1)) dá výsledek #c(-1.9999998 6.3999996)

-operátory jsou n-ární (+ 1 2 3 4 5) dá výsledek 15

-nekonečná aritmetika pro integer

Funkcionální programování- LISP

Elementární funkce (teoreticky lze s nimi vystačit pro zápis jakéhokoliv algoritmu, je to ale stejně neohrabané jako Turingův stroj)

- **CAR alias FIRST** — **selektor-vyběr prvého prvku**
- **CDR alias REST** — **selektor-výběr zbytku seznamu (čti kúdr)**
- **CONS** — **konstruktor- vytvoří dvojici z agumentů**
- **ATOM** — **test zda argument je atomický**
- **EQUAL** — **test rovnosti argumentů**

Ostatní fce lze odvodit z elementárních

 např. test prázdného seznamu funkcí NULL

  (NULL X) je stejné jako (EQUAL X NIL)

Lisp mu předloženou formuli vyhodnocuje

(prvouČástPovažujeZaFunkci dáleNásledujíJejíArgumenty)

Argumenty se LISP snaží vyhodnotit, což mnohdy nechceme

Jak zabránit vyhodnocování – je na tgo fce QUOTE zkracovaná apostrofem

Př. (FIRST ‘(REST (1 2 3))) (FIRST (REST ‘(1 2 3)))

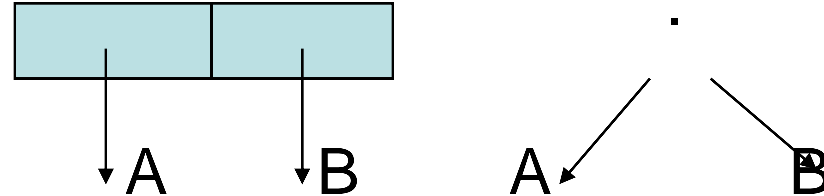
 dá hodnotu REST dá hodnotu 2

Funkcionální programování- LISP

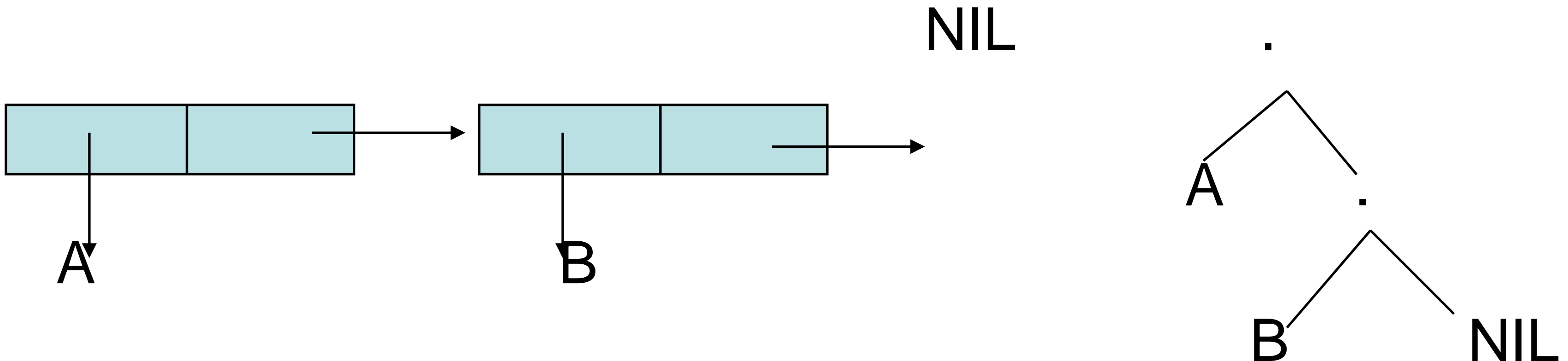
Zobrazení, jak CONS vytváří lispovské buňky – tzv tečka dvojice

(CONS ‘a ‘b) → (A . B)   cons dvou atomů je tečka dvojice

Lispovská buňka



(CONS ‘a ‘(b))   cons s druhým argumentem seznam je seznam



Funkcionální programování- LISP

Převod tečka notace do seznamové notace:

- **Vyskytuje-li se „. „ před „(„ lze vynechat „. „ i „(„ i odpovídající „)„**
- **Při výskytu „. NIL„ lze „. NIL„ vynechat**

Př. ‘((A . Nil) . Nil) je seznam ((A)) !!je třeba psát mezera tečka mezera

‘(a . Nil) je seznam (A)

‘((a . Nil) . ((b . (c . Nil)) . Nil)) je seznam ((A) (B C))

Seznam je takový S-výraz, který má na konci NIL

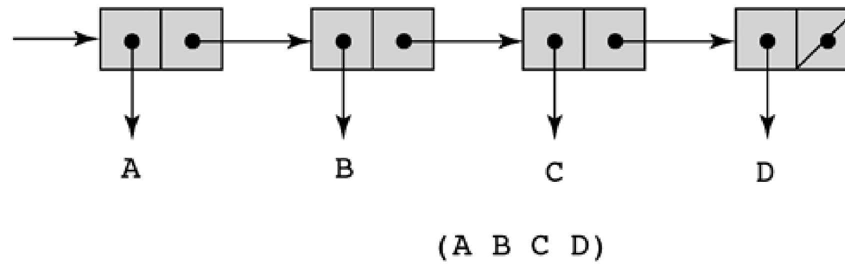
Forma (také formule) je vyhodnotitelný výraz

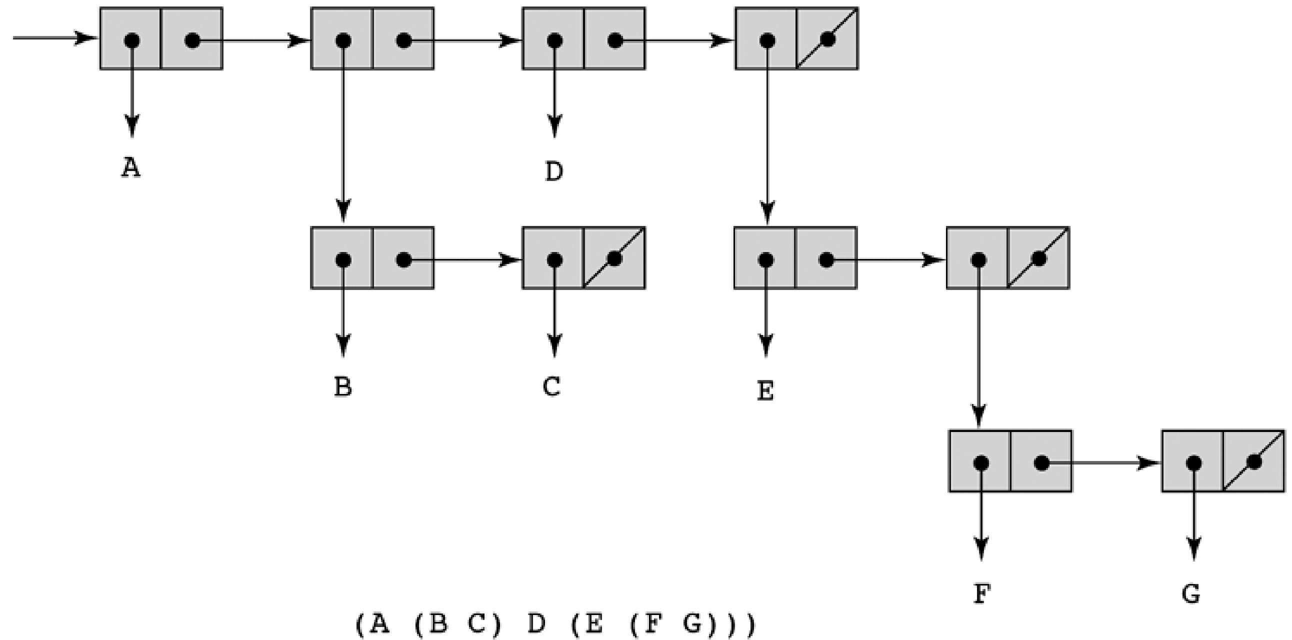
K řádnému programování v Lispu a jeho derivátech je potřebný editor hlídající párování závorek, který k freewarové verzi nemám

Funkcionální programování- LISP

Interní reprezentace

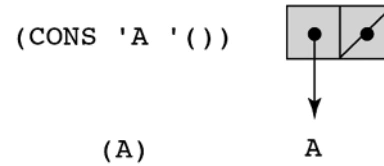
seznamů

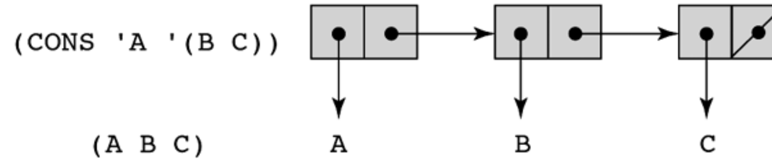


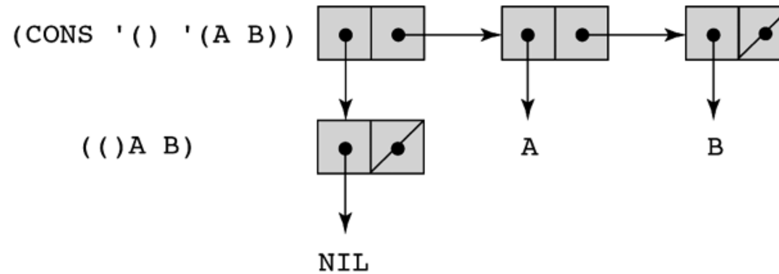


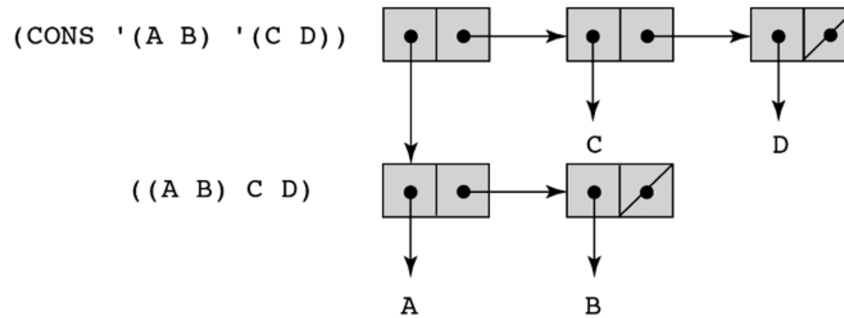
Funkcionální programování- LISP

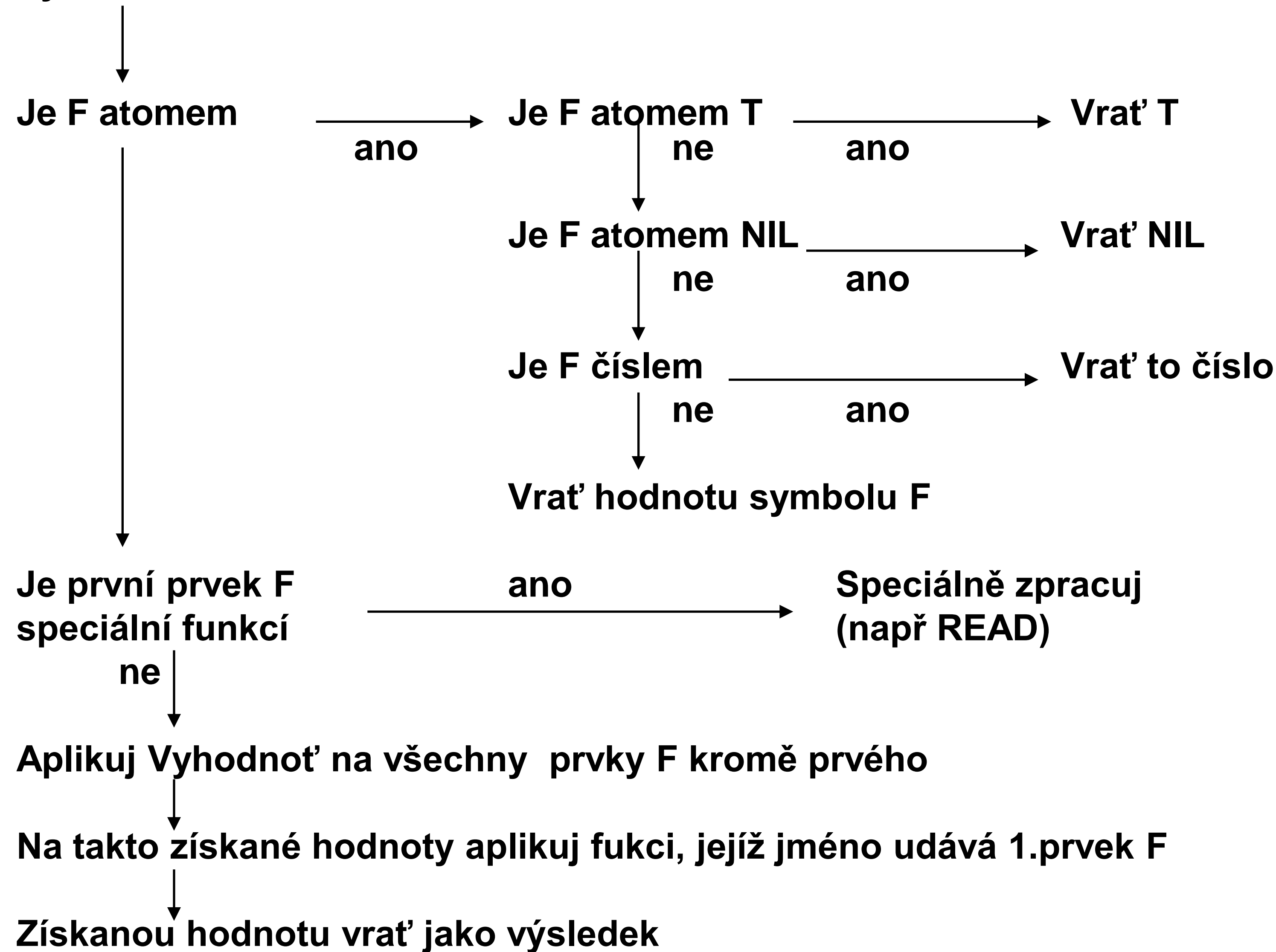
Výsledky fce CONS











Obr. Schéma lispovského vyhodnocování

Kromě CONS jsou další konstruktory APPEND a LIST

APPEND:: (seznam x seznam … x seznam) → seznam

Vytvoří seznam z argumentů – vynechá jejich vnější závorky

Př. (APPEND ‘(A B) ‘(B A)) dá (A B B A)

(APPEND ‘(A) ‘(B A)) dá (A B A)

Common Lisp připouští libovolně argumentů Append

(APPEND NIL ‘(A) NIL) dá (A)

(APPEND () ‘(A) ()) dá také (A)

(APPEND) dá NIL

(APPEND ‘(A)) dá (A)

(APPEND ‘((B A)) ‘(A) ‘(B A)) dá ((B A) A B A)

výsledky (A B B A) (A B A)

výsledky (A) (A) nil (A) ((B A) A B A)

Konstruktory APPEND a LIST

LIST:: (seznam x seznam x … x seznam) → seznam

Vytvoří seznam ze zadaných argumentů

Př.(LIST ‘A ‘(A B) ‘C) →

(A (A B) C)

(LIST ‘A) →

(A)

(LIST) →

nil

(LIST ‘(X (Y Z)) ‘(X Y)) →

((X (Y Z)) (X Y))

(APPEND ‘(X (Y Z)) ‘(X Y)) →

(X (Y Z) X Y)

(CONS ‘(X (Y Z)) ‘(X Y)) →

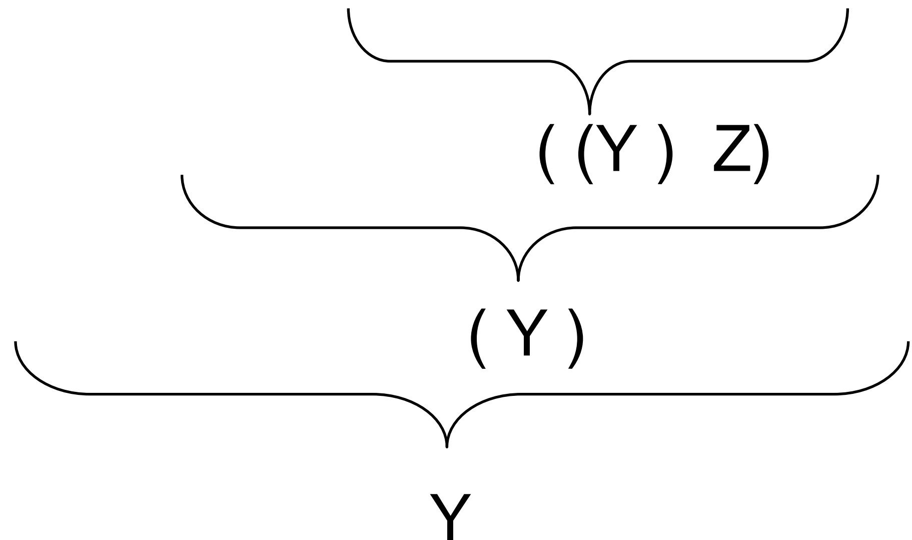
((X (Y Z)) X Y)

Selektory CAR a CDR lze vnořovat zkráceně

CxxR **CAAR, CADR, CDAR, CDDR**

CxxxR **CAAAR, …**

Př. (CAADR ‘(X (Y Z)) = (CAR (CAR (CDR ‘(X ((Y) Z)))))



(CADAR ‘ ((1 2 3) (4 5)))

2

LISP - Testy a větvení

ATOM	je argument atomický?	(atom 3) dá T
NULL	„ „ prázdný seznam?	
NUMBERP	„ „ číslem?	
SYMBOLP	„ „ symbolem?	
LISTP	„ „ seznamem?	

Př. (NULL NIL) → (NULL ( )) →

(LISTP ( )) → (LISTP NIL) →

(SYMBOLP NIL) → (SYMBOLP ( )) →

všechny mají hodnotu T, protože () i nil jsou prázdný seznam

Př.

(NUMBERP ‘(22))	(NUMBERP ’22)	(NUMBERP 22)
NIL	T	T

Př.

(SYMBOLP ‘A) je T

Když ale do A přiřadíme (SETQ a 2) bude (SYMBOLP ‘A) dávat NIL

LISP - Testy a větvení

Je tam několik testů rovnosti, nemusíte si je pamatovat, berte to jako poznámku

=	jsou hodnoty argumentů (může jich být více) stejná čísla?
EQ	„ „ „ stejné atomy?
EQUAL	„ „ „ stejné s-výrazy? Se stejnou hodnotou
>	„ „ „ v sestupném pořadí?
>=	„ „ „ v nestoupajícím pořadí?
<	<= je obdobné

Př. (EQ 3.0 3) dá NIL (= 3.0 3) dá T

(EQ ‘(A) ‘(A)) dá NIL (EQ ‘A ‘A) dá T

(EQUAL ‘(A) ‘(A)) i (EQUAL (+ 2 3 3) (CAR ‘(8))

dají T protože jejich argumenty vypisují stejnou hodnotu A či 8

(< 2 3 5 (* 4 4) 20) → T neboť posloupnost je vzestupná

(= 1.0 1 1.00 (- 3 1 1.00)) → T je to divné, ale bere je jako stejná

LISP - Testy a větvení

AND, OR mají libovolně argumentů, NOT má jen jeden

Všechny hodnoty různé od NIL považují za pravdivé

Argumenty vyhodnocují zleva doprava. Hodnotou fce je hodnota naposledy vyhodnoceného argumentu.

Používá zkrácený výpočet argumentů. Tzn pokud při AND vyhodnotí argument jako NIL, další již nevyhodnocuje. Obdobně pro OR vyhodnotí-li argument jako T, další nevyhodnocuje

Př. (AND (NULL NIL) (ATOM 5) (+ 4 3)) dá 7

(OR NIL (= 2 (CAR ‘(2)) (+ 1 1) (- 3.0 1.0) 0) 9) dá 9

IF forma

(IF podm then-část else-část)

Větvení se potřebuje k vytváření uživatelových fcí

Př. ((lambda (x) (if (< x 0) (* x x (- x)) (* x x x))) -5) dá 125

LISP - Testy a větvení

COND je spec. fcí s proměnným počtem argumentů

```
(COND (podm1  forma11  forma12  …  forma1n)
      (podm2  forma21  forma22  …  forma2m)
              . . .
      (podmk  formak1  formak2  …  formako) )
```

Postupně vyhodnocuje podmínky, dokud nenarazí na prvou, která je pravdivá. Pak vyhodnotí formy patřící k pravdivé podmínce. Hodnotou COND je hodnota poslední z vyhodnocených forem. Při nesplnění žádné z podm, není hodnota COND definována (u Common Lispu).

Pseudozápis pomocí IF:

```
COND  if podm1 then
              { forma11  forma12  …  forma1n}
      else
      if podm2 then
              { forma21  forma22  …  forma2m}
      else
              . . .
      if podmk then
              { formak1  formak2  …  formako}
      else NIL
```

LISP - Přiřazování

Přiřazení je operace, která pojmenuje hodnotu a uloží ji do paměti

- **Je ústupkem od čistě funkcionálního stylu**
- **Může zefektivnit a zpřehlednit i funkcionální program**
- **Mění vnitřní stav výpočtu (vedlejší efekt přiřazení)**

Zahrnuje funkce pro: -I/O,

-pojmenování uživ. fcí

-pojm. hodnot symbolů (SET, SETQ)

(SETQ jméno-symbolu argument)

vrátí hodnotu argumentu a naváže hodnotu argumentu na nevyhodnocené jméno symbolu

SET je obdobná, ale vyhodnotí i jméno symbolu

LISP - Přiřazování

Př.

```
(SETQ X 1) → 1
(SETQ X  (+ 1 X) → 2
X → 2
>(SETQ LETADLO ‘BOING)
BOING
>LETADLO
BOING
>(SETQ BOING ‘JUMBO)
JUMBO
>(SETQ LETADLO BOING)
JUMBO
>LETADLO
JUMBO
>(SET LETADLO ‘AEROBUS)
AEROBUS
>LETADLO
JUMBO
> JUMBO
AEROBUS
```

LISP – Definice funkcí

Definujeme zápisem: (DEFUN jméno-fce (argumenty) tělo-fce)

- **Přiřadí jménu-fce lambda výraz definovaný tělem-fce, tj. (LAMBDA (argumenty) tělo-fce). Vytvoří funkční vazbu symbolu jméno-fce**

- **Argumenty jsou ve fci lokální**
- **DEFUN nevyhodnocuje své argumenty**
- **Hodnotou formy DEFUN je nevyhodnocené jméno-fce,**
- **Tělo-fce je posloupností forem, nejčastěji jen jedna. Při vyvolání fce se všechny vyhodnotí. Funkční hodnotou je hodnota poslední z forem**

Př

(defun fce(x) (+ x x) (* x x)) odpoví fce a vytvoří funkční vazbu pro FCE

Když ji pak vyvoláme např. (fce 5) odpoví nám 25

LISP – Definice funkcí

>(DEFUN max2 (x y) (IF (> x y) x y))

max2

(max2 10 20)

20

>(DEFUN max4 (x y u v) (max2 (max2 x y) (max2 u v)))

max4

>(max4 5 9 12 1)

Interaktivní psaní lispovských programů způsobuje postupnou demenci vzhledem k závorkám ⇒ lepší možnost load ze souboru

(LOAD "jmeno-souboru")

(LOAD "D:\\PGS\\LISP\\soubor.lsp")

LISP – Definice funkcí

Př. 1max.lsp

```
(defun max2(x y) (if (> x y) x y))
(defun ma(n) (if
;;;        (equal (list-length n) 2)  list-lenth je standardní fce
;;;        naprogramujeme si ji sami pojmenovanou delka
             (equal (delka n) 2)
               (max2 (car n) (car (cdr n)))
               (max2 (car n) (ma (cdr n)))
            )
)
(defun delka(n)
    (if (equal n nil)
              0
              (+ 1 (delka (cdr n))) ))

;;;        vyvolání např  (ma '(1 8 3 5))
```

LISP – Definice funkcí

Př. 2sude-poradi.lsp

```
 ;;;vybira ze seznamu x prvky sude v poradi
(defun sude (x)
   (cond
    ((not (null (cdr x)))
          (cons(car (cdr x))(sude (cdr (cdr x)))))
    (t nil)
))
```

Význam zápisu:

Pokud má x více než jeden prvek, dej do výsledku druhý prvek (tj car z cdr x) s výsledkem rekurzivně vyvolané fce sude s argumentem, kterým je zbytek ze zbytku x (tj cdr z cdr x, tj část x od tretího prvku dál). Pokud má seznam x jen jeden prvek, je výsledkem prázdný seznam

LISP – Definice funkcí

Př.3NSD-Fakt.lsp

```
(defun nsd (x y)
    (cond ((zerop (- x y)) y)        ;;; je-li rozdil x y nula, vysledek je y
          ((> y x) (nsd x (- y x))) ;;je-li y vetsi x vyvolej nsd x a y-x
          (t (nsd y (- x y)))       ;;jinak vyvolej nsd y a x-y
))

(defun fakt (x)
    (cond   ((= x  0) 1)           ;;faktorial nuly je jedna
            (T (* x  (fakt (- x 1)))) ;;jinak je x krat faktorial x-1
))
```

LISP – Definice funkcí

P5 4AppendMember.lsp

Redefinice append a member musíme explicitně povolit. Po load hlasi, že funkce je zamknutá. Pokud odpovíme :c ignorujeme zamknutí makra a funkce se předefinuje

```
(DEFUN APPEND (X  Y)   ;;;pro dva argumenty
    (IF (NULL  X)
                  Y          ;;je-li X prazdny je vysledkem Y
                  (CONS  (CAR  X) (APPEND (CDR  X)  Y))
) )    ;;jinak je vysledkem seznam zacinajici X a za nim APPEND…

(DEFUN MEMBER  (X  S)  ;;; je jiz take mezi standardnimi
    (COND         ((NULL  S)  NIL)
          ((EQUAL  X  (CAR  S))  T)       ;;; standardni vraci S místo T
          (T  (MEMBER  X  (CDR  S)))
) )
```

Př volání (MEMBER ‘X ‘(A B X Z)) tato dá T standardní dá (X Z)

LISP – Další standardní funkce

Ad aritmetické

(- 10 1 2 3 4) → 0

(/ 100 5 4 3) → 5/3

Ad operace na seznamech

(LIST-LENGTH ‘(1 2 3 4 5)) → 5

Výběr posledního prvku, je také ve standardních

```
(DEFUN LAST (S)
    (COND ((NULL (CDR S))
                              S
                              (LAST (CDR  S))
))
(LAST ‘(1  3  2  8  (4  5))) → ((4  5))
```

LISP – Další standardní funkce (pro informaci)

Ad vstupy a výstupy

(OPEN soubor :DIRECTION směr) otevře soubor a spojí ho s novým proudem, který vrátí jako hodnotu.

Hodnotou je jmeno proudu (= souboru)

Např.

(SETQ S (OPEN “d:\\moje\\data.txt” :direction :output)) ;;; :input

Standardně je vstup z klávesnice, výstup obrazovka

(CLOSE proud) zapíše hodnoty na disk a uzavře daný proud (přepne na standardní)

Např. (CLOSE S)

(READ proud)

(PRINT a proud)

Př.5Average.lsp Výpočet průměrné hodnoty

```
(defun sum(x)
    (cond ((null x) 0)
          ((atom x) x)
          (t (+ (car x) (sum (cdr x))))))
(defun count (x) ;;; je take mezi standardnimi, takze povolit předef    :c
    (cond ((null x) 0)
          ((atom x) 1)
          (t (+ 1 (count (cdr x))))))

(defun avrg ()          ;;;hlavni program je posloupnost forem
    (print "napis seznam cisel")
    (setq x (read))
    (setq avg (/ (sum x) (count x)))
    (princ "prumer je ")
    (print avg))    ;;je-li real a prvky jsou celociselne, vypise zlomkem
```

Př. 6hanoi.lsp * výstup příkazem format tvaru:

(FORMAT	**cíl**		**řídící řetězec**	**argumenty)**
	na obrazovku	t	~%	odřádkování
	netiskne ale vrátí	nil	~a	řetězcový argument
	Na soubor	proud	~s	symbolický výraz
			~d	desítkové číslo

```
(DEFUN hanoi (n from to aux)
    (COND   ((= n 1) (move from to))
          (T         (hanoi (- n 1) from aux to)
                     (move from to)
                     (hanoi (- n 1) aux to from)
)))
(DEFUN move (from to)
    (format T  "~%move the disc from ~a to ~a." from to)
)
```

LISP – Další standardní funkce

Ad funkce pro řízení výpočtu

(WHEN test formy) ; je-li test T je hodnotou hodnota poslední formy

(DOLIST (prom seznam) forma) ; váže prom na prvky až do vyčerpání seznamu a vyhodnocuje formu

Př. (DOLIST (x '(a b c)) (print x)) → a b c

(LOOP formy) ;opakovaně vyhodnocuje formy až se provede forma return

Př. (SETQ a 4) → 4

(LOOP (SETQ a (+ a 1)) (WHEN (> a 7) (return a))) → 8

(DO ((var1 init1 step1) … (varn initn stepn)) ;;inicializace

(testkonce forma1 forma2 ...formam)

formy-prováděné-při-každé-iteraci)

Př. 7fibo.lsp (N-tý člen = člen N-1 + člen N-2)

```
(defun fibon(N)
    (cond
          ((equal N 0) 0)   ;;;trivialni pripad
          ((equal N 1) 1)   ;;;  “       “
          ((equal N 2) 1)   ;;;  “       “
          (T (foo (- N 2)))
))

(defun  foo(N)
    (setq F1 1)    ;;; clen n-1
    (setq F2 0)    ;;; clen n-2
    (loop
          (setq F (+ F1 F2))          ;;; clen n
          (setq F2 F1)                ;;;  novy clen n-2
          (setq F1 F)                 ;;; clen n-1
          (setq N (- N 1))
          (when (equal N 0) (return F))
))
```

Př 8DOpriklad.lsp

Co se tiskne?

Promenna1 init1 Promenna2 init2

```
(DO (( x  1 (+  x  1))  (y  10  (*  y  0.5))) ;soucasna inicializace
```

Test konce step1 step1

```
     ((>  x  4)  y)
     (print  y)
     (print ‘pocitam)
)
```

koncova forma

formy provadene pri iteracich

10

POCITAM

5.0

POCITAM

2.5

POCITAM

1.25

POCITAM

0.625

Př.8NTA.lsp nalezení pořadím zadaného členu seznamu

```
(setq v "vysledek je ")
(defun nta (S x)
    (do ((i 1 (+ i 1)))
        ((= i x) (princ v) (car S)) ;test konce a vysledna forma
        (setq S (cdr S))
    ))
(defun delej () (nta (read) (read)))
```

Dá se také zapsat elegantně neefektivně = rekurzivě

```
(defun nty (S x)
    (cond ((= x 0) (car S))            ; pocitame poradi od 0
              (T (nty (cdr S) (- x 1)))
    ))
```

LISP – Další standardní funkce (pro informaci)

(EVAL a) vyhodnotí výsledek vyhodnocení argumentu

Př.

>(EVAL (LIST ‘REST (LIST ‘QUOTE ‘(2 3 4))))

(3 4)

(QUOTE ‘(2 3 4)

(REST ‘(2 3 4))

(EVAL ‘(REST ‘(2 3 4)))

>(EVAL (READ))

(+ 3 2) *to napiseme pro READ*

5

>(EVAL (CONS ‘+ ‘(2 3 5)))

10

(SET ‘A 2)

(EVAL (FIRST ‘(A B)))

2

LISP – rozsah platnosti proměnných

```
(DEFUN co-vraci (Z)
    (LIST (FIRST Z) (posledni-prvek))
)
(DEFUN  posledni-prvek ( );;ta fce pracuje s nelokalnim Z, ale co to bude?
    (FIRST (LAST  Z))
)
>(SETQ  Z  ‘(1  2  3  4))
(1  2  3  4)
>(co-vraci ‘(A  B  C  D))
```

u dynamickeho

u statickeho

(A 4) u statickeho rozsahu platnosti, platnost jmen je dána lexikálním tvarem programu-CLISP

(A D) u dynamickeho rozsahu platnosti, platnost jmen je dána vnořením určeným exekucí volání funkcí -GCLISP

Shrnutí zásad

- **Lisp pracuje se symbolickými daty.**
- **Dovoluje funkcionální i procedurální programování.**
- **Funkce a data Lispu jsou symbolickými výrazy.**
- **CONS a NIL jsou konstruktory, FIRST a REST jsou selektory, NULL testuje prázdný seznam, ATOM, NUMBERP, SYMBOLP, LISTP testují typ dat, =, EQ, EQUAL, testují rovnost, <, >, … testují pořadí**
- **SETQ, SET přiřazují symbolům globální hodnoty**
- **DEFUN definuje funkci, parametry jsou v ní lokální.**
- **COND umožňuje výběr alternativy.**
- **AND, OR, NOT jsou logické funkce.**
- **Proud je zdrojem nebo konzumentem dat. OPEN jej otevře, CLOSE jej zruší.**
- **PRINT, PRIN1, PRINC TERPRI zajišťují výstup.**
- **READ zabezpečuje vstup.**
- **EVAL způsobí explicitní vyhodnocení.**
- **Zápisem funkcí a jejich kombinací vytváříme formy (vyhodnotitelné výrazy).**
- **Lambda výraz je nepojmenovanou funkcí**
- **V Lispu má program stejný syntaktický tvar jako data.**

Máte-li chuť, zkuste vyřešit

Co to pocita? – 91Co.lsp

```
(DEFUN  co1 (list)
    (IF (NULL list)  ( )
                        (CONS (CAR list) (co2 (CDR list)))
))

(DEFUN  co2 (list)
    (IF (NULL list)  ( )
                        (co1  (CDR list ))
))
```

LISP – schéma rekurzivního výpočtu (pro informaci)

S jednoduchým testem

```
(DEFUN fce (parametry)
   (COND     (test-konce    koncová-hodnota);;primit.příp.
             (test    rekurzivní-volání) ;;redukce úlohy
))
```

S násobným testem

```
(DEFUN fce (parametry)
   (COND     (test-konce1            koncová-hodnota1)
             (test-konce2            koncová-hodnota2)
                      . . .
             (test-rekurze rekurzivní-volání)
                      . . .
))
```

Př.92rekurze.lsp

```
;;odstrani vyskyty prvku e v nejvyssi urovni seznamu S
(DEFUN delete (e  S)
   (COND        ((NULL S) NIL)
         ((EQUAL e (CAR S)) (delete e (CDR S)))
         (T (CONS (CAR S) (delete e (CDR S)))) ))

;;zjisti maximalni hloubku vnoreni seznamu;; MAX je stand. fce
(DEFUN max_hloubka (S)
   (COND        ((NULL S) 0)
         ((ATOM (CAR S)) (MAX 1 (max_hloubka (CDR S))))
         (T (MAX (+ 1 (max_hloubka (CAR S)))
                (max_hloubka (CDR S)) )) ));;nasobna redukce

;;najde prvek s nejvetsi hodnotou ve vnorovanem seznamu
(DEFUN max-prvek (S)
   (COND        ((ATOM S) S)
         ((NULL (CDR S)) (max-prvek (CAR S)))
         (T (MAX (max-prvek (CAR S)) ;;nasobna redukce
                (max-prvek (CDR S)) ))  ))
```

LISP - Funkcionály

Funkce, jejichž argumentem je funkce nebo vrací funkci jako svoji hodnotu. Vytváří programová schémata, použitelná pro různé aplikace. (Higher order functions) ***Pamatujte si alespoň ten pojem***

(pro informaci)

Př. *pro každý prvek s seznamu S proveď f(s)*

to je schéma

Programové schéma pro zobrazení

f : (s1, s2, . . ., sn) → (f (s1), f (s2), . . ., f (sn))

```
(DEFUN zobrazeni (S)
    (COND       ((NULL S) NIL)
                (T (CONS       (transformuj (FIRST S))
                               (zobrazeni (REST S)) ))
))
```

LISP – Funkcionály (pro informaci)

Programové schéma filtru

```
(DEFUN filtruj (S)
    (COND       ((NULL S) NIL)
                ((test-prvku (FIRST S))
                        (CONS (FIRST S) (filtruj (REST S))) )
                (T  (filtruj (REST S)))  ))
```

Programové schéma nalezení prvého prvku splňujícího predikát

```
(DEFUN najdi-prvek (S)
    (COND       ((NULL S) NIL)
                ((test-prvku (FIRST S))  (FIRST S))
                (T (najdi-prvek (REST S))) ))
```

Programové schéma pro zjištění zda všechny prvky splňují predikát

```
(DEFUN zjisti-všechny (S)
    (COND       ((NULL S) T)
                ((test-prvku (FIRST S) (zjisti-všechny (REST S)))
                (T NIL) ))
```

LISP – Funkcionály (pro informaci)

- Při použití schéma nahradíme název funkce i jméno uvnitř použité funkce skutečnými jmény.
- Abychom mohli v těle definice funkce použít argument v roli funkce, je třeba informovat LISP, že takový parametr musí vyhodnotit pro získání popisu funkce.

?Př.? (pro informaci)

Schéma aplikace funkce na každý prvek seznamu

```
(DEFUN aplikuj-funkci-na-S (funkce  S)
    (COND        ((NULL  S) NIL)                    ?LISP
                 (T (CONS         (funkce (FIRST S))
                                  (aplikuj-funkci-na-S  funkce (REST S)) )     )
```

-FUNCALL je funkcionál, aplikuje funkci na argumenty

```
(DEFUN aplikuj-funkci-na-S (funkce  S)
    (COND        ((NULL  S) NIL)
                 (T (CONS         (funcall funkce (FIRST S))
                                  (aplikuj-funkci-na-S  funkce (REST S)) )     )
```

?tak to nejde, chtěl by vyhodnotit car jako proměnnou

```
(aplikuj-funkci-na-S    car    '((a b) (c d)) )
(aplikuj-funkci-na-S    'car    '((a b) (c d)) )   zabráníme vyhodnocení car
(a  c)                            a to pak bude vysledek
```

Porovnání klasických konstrukcí – jména

Jména (identifikátory)

-max. délka (Fortran= 6, Fortran90, ANSI C = 31, Cobol 30, C++ neom., ale omezeno implementací; ADA, Java = neom.)

-case sensitivity (C++,C, Java ano, ostatní ne). Nevýhodou je zhoršení čitelnosti = jména vypadají stejně , ale mají různý význam.

Speciální slova

–Klíčová slova = v určitém kontextu mají speciální význam

–Předdefinovaná slova = identifikátory speciálního významu, které lze předefinovat (např vše z balíku java.lang – String, Object, System…)

–Rezervovaná slova = nemohou být použita jako uživatelem definovaná jména (např.abstract, boolean, break, …, if, …, while)

Proměnné = abstrakce paměťových míst

Formálně = 6tice atributů

(jméno, adresa, hodnota, typ, doba existence, rozsah platnosti)

Způsob deklarace: explicitní / implicitní

Porovnání klasických konstrukcí – jména

- Jméno – nemají je všechny proměnné
- Adresa – místo v paměti (během doby výpočtu či místa v programu se může měnit)
- Aliasy – dvě proměnné sdílí ve stejné době stejné místo
  - Pointery
  - Referenční proměnné
  - Variantní záznamy (Pascal)
  - Uniony (C, C++)
  - Fortran (EQUIVALENCE)
  - Parametry podprogramů
- Typ – určuje množinu hodnot a operací
- Hodnota – obsah přiděleného místa v paměti

- L hodnota = adresa proměnné
- R hodnota = hodnota proměnné
- Binding = vazba proměnné k atributu

Porovnání klasických konstrukcí – jména a typy

Kategorie proměnných podle vazby s typem a s paměťovým místem

- **Statická vazba** (jména s typem / s adresou)
  navázání se provede před dobou výpočtu a po celou exekuci se nemění
  Vazba s typem určena buď explicitní deklarací nebo implicitní deklarací
- **Dynamická vazba** (jména s typem / s adresou)
  nastane během výpočtu nebo se může při exekuci měnit
  - Dynamická vazba s typem
    specifikována přiřazováním (např. Lisp)
    výhoda – flexibilita (např. generické jednotky)
    nevýhoda- vysoké náklady + obtížná detekce chyb při překladu
  - Vazba s pamětí (nastane alokací z volné paměti, končí dealokací)
    doba existence proměnné (lifetime) je čas, po který je vázána na určité paměťové místo.

Porovnání klasických konstrukcí – jména a typy

Kategorie proměnných podle doby existence (lifetime)

- Statické = navázání na paměť před exekucí a nemění se po celou exekuci

  Fortran 77, C static

  výhody: efektivní – přímé adresování, podpr. senzitivní na historii

  nevýhody: bez rekurze
- Dynamické

  V zásobníku = přidělení paměti při exekuci zpracování deklarací. Pro skalární proměnnou jsou kromě adresy přiděleny atributy staticky (lokální prom. C, Pascalu).

  výhody: rekurze, nevýhody: režie s alokací/dealokací, ztrácí historickou informaci, neefektivní přístup na proměnné (nepřímé adresy)

  Explicitní na haldě = přidělení / uvolnění direktivou v programu během výpočtu. Zpřístupnění pointery nebo odkazy (objekty ovládané new/delete v C++, objekty Javy)

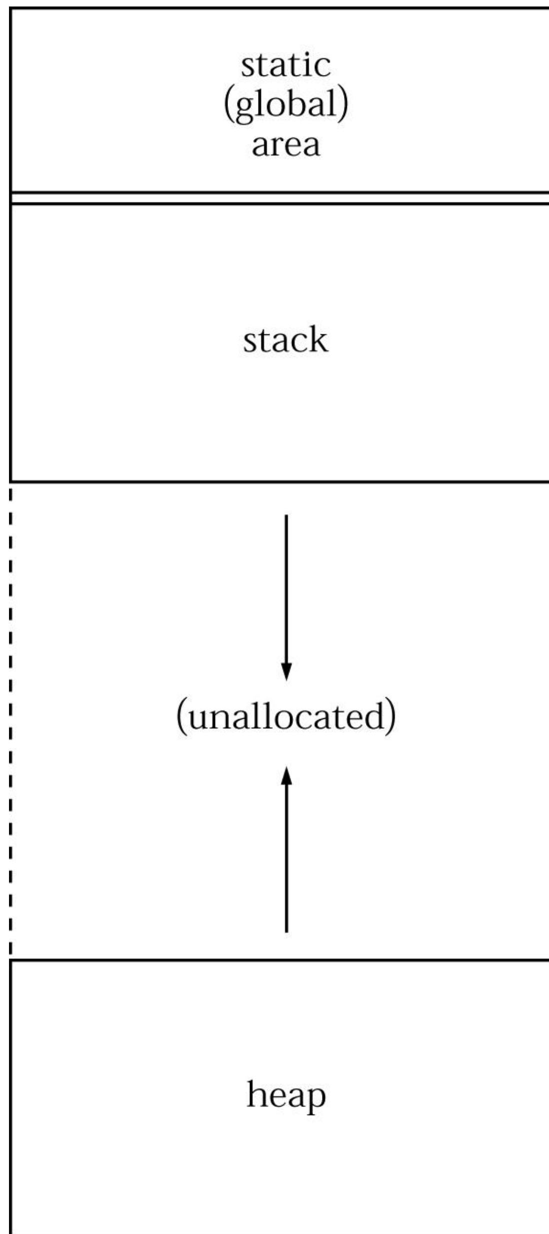
  výhody: umožňují plně dynamické přidělování paměti,

  nevýhody: neefektivní a nespolehlivé (zejm. při slabším typovém systému)

  Implicitní přidělování na haldě = alokace/dealokace způsobena přiřazením

  Výhody: flexibilita, nevýhody: neefektivní – všechny atributy jsou dynamické, špatná detekce chyb

Většina jazyků používá kombinace – násl. obr.ukazuje rozdělení pam.prostoru


static
(global)
area
stack
(unallocated)
heap

Porovnání klasických konstrukcí – typy

Typová kontrola je aktivita zabezpečující, že operandy operátorů jsou kompatibilních typů

Kompatibilní typy jsou takové, které jsou buď legální pro daný operátor, nebo jazyk dovoluje implicitní konverzi pomocí překladačem generovaných instrukcí na legální typ (automatická konverze = anglicky coercion)

Při statické vazbě s typem je možná statická typová kontrola

Při dynamické vazbě s typem je nutná dynamická typová kontrola.

Programovací jazyk má silný typový systém, pokud typová kontrola odhalí veškeré typové chyby

Problémy s typovou konverzí:

a) Při zužování **R → I** (možná neproveditelnost rounding/truncation),

b) Při rozšiřování **I → R** (možná ztráta přesnosti)

Porovnání klasických konstrukcí – typy

Konkrétní jazyky (které mají/nemají silný typový systém):

- Fortran77

  nemá z důvodů parametrů, příkazu Equivalence

- Pascal

  není plně silný protože dovoluje variantní záznamy

- C, C++

  nemá z důvodů lze obejít typovou kontrolu parametrů, uniony nejsou kontrolovány

- ADA, Java

  téměř jsou –

Pravidla pro **coerci** (implicitně prováděnou konverzi) výrazně oslabují silný typový systém

Porovnání klasických konstrukcí – typy

Kompatibilita typů se určuje na základě:

Jmenné kompatibility – dvě proměnné jsou kompatibilních typů, pokud jsou uvedeny v téže deklaraci, nebo v deklaracích používajících stejného jména typu

dobře implementovatelná, silně restriktivní

Strukturální kompatibility – dvě proměnné jsou kompatibilní mají-li jejich typy identickou strukturu

flexibilnější, hůře implementovatelné

Pascal a C (kromě záznamů) používá strukturální,

ADA, Java - jmennou

Porovnání klasických konstrukcí – jména a typy

<u>Rozsah platnosti</u> (scope) proměnné je částí programového textu, ve kterém je proměnná viditelná. Pravidla viditelnosti určují, jak jsou jména asociována s proměnnými

<u>Rozsah existence</u> (lifetime) je čas, po který je proměnná vázána na určité paměťové místo

Statický (lexikální) rozsah platnosti

- Určen programovým textem
- K určení asociace jméno – proměnná je třeba nalézt deklaraci
- Vyhledávání: nejprve lokální deklarace, pak globálnější rozsahová jednotka, pak ještě globálnější. . . Uplatní se pokud jazyk dovolí vnořování prog.jednotek
- Proměnné mohou být zakryty (slepé skvrny)
- C++, ADA, Java dovolují i přístup k zakrytým proměnným (Třída.proměnná)
- Prostředkem k vytváření rozsahových jednotek jsou bloky

Dynamický rozsah platnosti

- Založen na posloupnosti volání programových jednotek (namísto hlediska statického tvaru programového textu, řídí se průchodem výpočtu programem)
- Proměnné jsou propojeny s deklaracemi řetězcem vyvolaných podprogramů

Porovnání klasických konstrukcí – jména a typy

```
Př. MAIN
     deklarace x
     SUB 1
          deklarace x
                    …
          call SUB 2
          …
     END SUB 1
     SUB 2
          …
          odkaz na x  // je x  z MAIN nebo  ze SUB1 ?
          …
     END SUB 2
     …
     CALL  SUB 1
     …
   END MAIN
```

Porovnání klasických konstrukcí – jména a typy

Rozsah platnosti (scope) a rozsah existence (lifetime) jsou různé pojmy. Jméno může existovat a přitom být nepřístupné.

Referenční prostředí jsou jména všech proměnných viditelných v daném místě programu

V jazycích se statickým rozsahem platnosti jsou referenčním prostředím jména lokálních proměnných a nezakrytých proměnných obklopujících jednotek

V jazycích s dynamickým rozsahem platnosti jsou referenčním prostředím jména lokálních proměnných a nezakrytých proměnných aktivních jednotek

Porovnání klasických konstrukcí – jména a typy

```
public class Scope
  { public static int x = 20;
    public static void f()
    { System.out.println(x); }
    public static void main(String[] args)
    { int x = 30;
      f();
    }
  }
```

Java používá statický scope, takže tiskne . . .20

Pokud by používala dynamický, pak tiskne . . .30

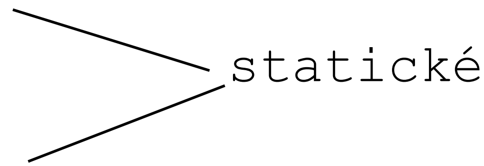
Dynamický používá originální LISP, VBScript, Javascript, Perl (starší verze)

Porovnání klasických konstrukcí – jména a typy

Konstanty (mají fixní hodnotu po dobu trvání jejich existence v programu, nemají atribut adresa = na jejich umístění nelze v programu odkazovat) :

- Určené v době překladu– př.Javy:

```
static final int zero = 0;
```



- Určené v době zavádění programu

```
static final Date now = new Date();
```

- Dynamické konstanty: -v C# konstanty definované readonly

  -v Javě: každé non-static final přiřazení v konstruktoru.

  -v C: #include <stdio.h>

```
const int i = 10;     // i je statická urč.při překladu
const int j = 20 * 20 + i;// j  „      -------     „
int f(int p) {
    const int k = p + 1; // k je dynamická
    return k + I + j;
        }
```

- Literály = konstanty, které nemají jméno
- Manifestová konstanta = jméno pro literál

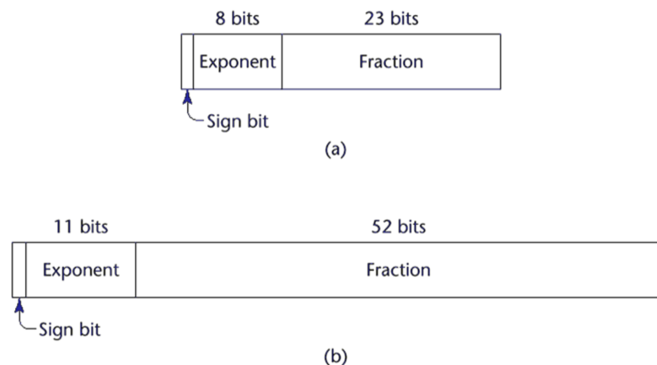
Porovnání klasických konstrukcí – typy

Typ: Definuje kolekci datových objektů a operací na nich proveditelných

- primitivní = jejich definice nevyužívá jiných typů
- složené

Integer – reflektují harwareové možnosti počítače, aproximace celých čísel

Floating Point – obvykle reflektují hardware, aproximace reálných čísel (např. ADA type delka is digits 12 range 0.0 ..300000.0;) – !!!pozor na konverze!!! v jazycích pro vědecké výpočty zaváděn min. ve dvou formách



Decimal -pracují s přesným počtem cifer (finanční aplikace)

Boolean –obvykle implementovány bytově, lze i bitově. V C je nahražen ***int*** *0 / nenula*

Character –kódování ASCII (128 znaků), UNICODE (16 bitů) Java, Python i C#

Porovnání klasických konstrukcí – typy

Ordinální (zobrazitelné=přečíslitelné do integer). Patří sem:
- primitivní mimo float
- definované uživatelem (pro čitelnost a spolehlivost programu). Zahrnují:
  - **vyjmenované typy** =uživatel vyjmenuje posloupnost hodnot typu,
    Implementují se jako seznam pojmenovaných integer konstant,
    např Pascal, ADA, C++
    type BARVA = (BILA, ZLUTA, CERVENA, CERNA) ;
    C# př. enum dny {pon, ut, str, ctvr, pat, sob, ned};
    Java je má od 1.5 př. public enum Barva (BILA,ZLUTA,CERVENA,CERNA);
    V nejjednodušší podobě lze chápat také jako seznam integer
    Realizovány ale jako typ třída Enum. Možnost konstruktorů, metod, …
    ale uživatel nemůže vytvářet potomky Enum
    enum je klíčové slovo.
  - **typ interval** =souvislá část ordinálního typu. Implementují se jako
    typ jejich rodiče (např. type RYCHLOST = 1 .. 5)

Výhody:ordinálních typů jsou čitelnost, bezpečnost

Porovnání klasických konstrukcí – typy

String – hodnotou je sekvence znaků

- Pascal, C, C++ = neprimitivní typ, pole znaků
- Java má typ, String class – hodnotou jsou konstantní řetězce,
  StringBuffer class – lze měnit hodnoty a indexovat, je podobné jako znaková pole
- ADA, Fortran90, Basic, Snobol = spíše primitivní typ, množství operací
- délka řetězců:
  - statická (Fortran90, ADA, Cobol, String class Javy), efektivní implementace
  - limitovaná dynamická (C, C++ indikují konec znakem null)
  - dynamická (Snobol4, Perl,Python), časově náročná implementace

Porovnání klasických konstrukcí – typy

Array – agregát homogenních prvků, identifikovatelných pozicí relativní k prvému prvku

V jazycích odlišnosti:

jaké mohou být typy indexů ?

C, Fortran, Java celočíselné, ADA,Pascal ordinální

? Způsob alokace

1. Statická pole (= pevné délky)
   ukládaná do statické oblasti paměti (Fortran77), globální pole Pascalu, C.
   Meze indexů jsou konstantní
2. Statická pole ukládaná do zásobníku
   (Pascal lokální, C lokální mimo static)
3. Dynamická v zásobníku . (ADA)
   (= délku určují hodnoty proměnných). Flexibilní
4. Dynamická na haldě (Fortran90, Java, Perl).

Porovnání klasických konstrukcí – typy

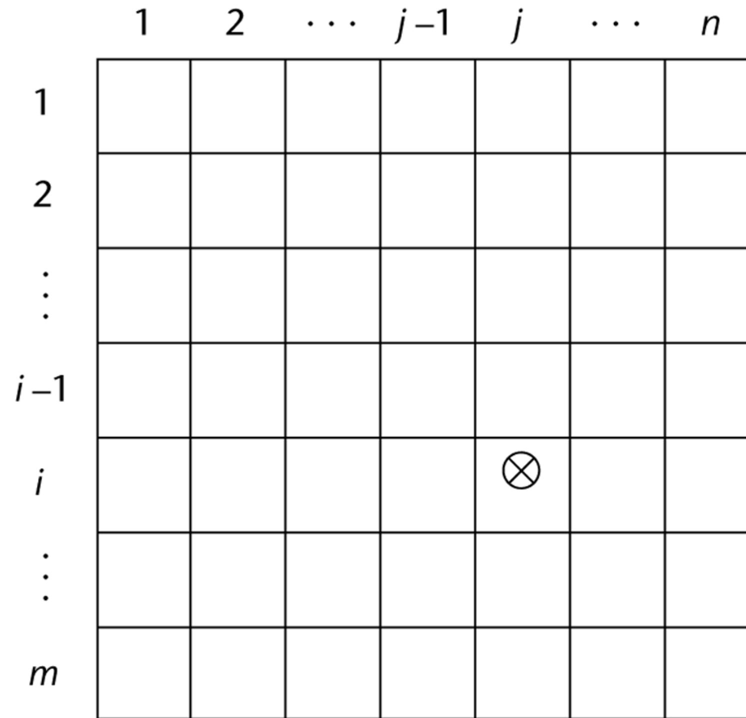
Přístupová fce pro jednorozměrné pole má tvar

location(vector[k]) = address (vector[lower_bound])

+ ((k-lower_bound) * element_size)

Přístupová fce pro vícerozměrná pole (řazení po sloupcích / řádcích)

location (a[i,j]) = address of a [row_lb,col_lb] + (((i - row_lb) * n) + (j - col_lb)) * element_size // lb = lower bound



Porovnání klasických konstrukcí – typy

Co o poli potřebuje vědět překladač je tzv. deskriptor pole

jednorozměrné
Array
Element type
Index type
Index lower bound
Index upper bound
Address

vícerozměrné
Multidimensioned array
Element type
Index type
Number of dimensions
Index range 1
⋮
Index range *n*
Address

Porovnání klasických konstrukcí – typy

Asociativní pole

Perl,Python mají asociativní pole = neuspořádaná kolekce dvojic (klíč, hodnota), nemají indexy

- **Př. Perl: Jména začínají `%;` literály jsou odděleny závorkami**

```
%cao_temps = ("Mon" => 77, "Tue" => 79, "Wed" =>
  65, …);
```

- **Zpřístupnění je pomocí slož.závorek s klíčem**

```
$cao_temps{"Wed"} = 83;
```

  – **Prvky lze odstranit pomocí `delete`**

```
delete $cao_temps{"Tue"};
```

Porovnání klasických konstrukcí – a typy

Record –(záznam) možně heterogenní agregát datových prvků, které jsou zpřístupněny jménem (kartezský součin v prostoru typů položek)
odkazování na položky OF notací Cobol, ostatní „.“ notací
Př C **struct {int i; char ch;} v1,v2,v3;**

Operace -přiřazení (pro identické typy), inicializace, porovnání
Uniony – typy, jejichž proměnné mohou obsahovat v různých okamžicích výpočtu hodnoty různých typů.

Př. C **union u_type {int i; char ch;} v1,v2,v3; /* free union- nekontroluje typ*/**

Př.Pascal

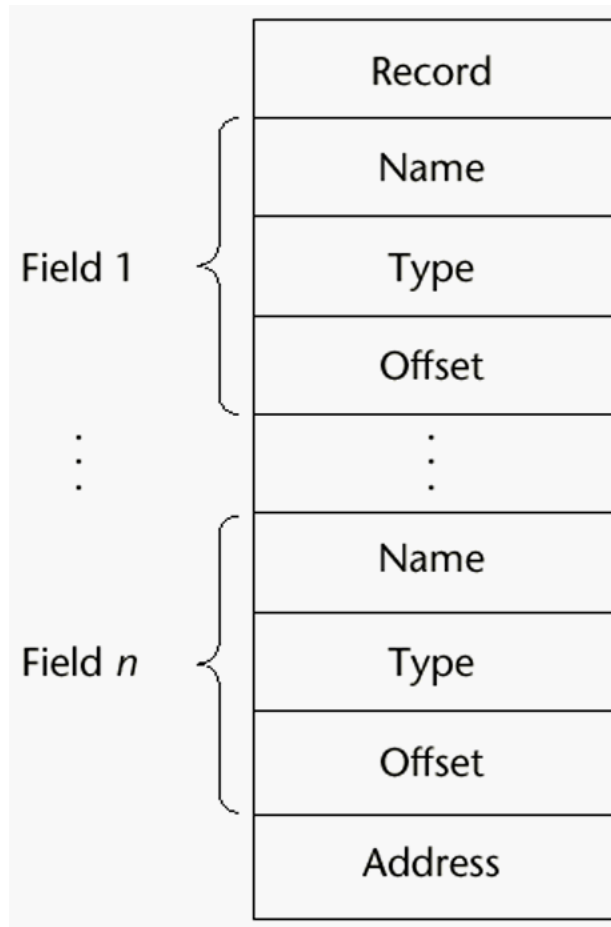
```
type R = record
        ...
        case RV : boolean of            /*discriminated union*/
                false : (i : integer);
                true : (ch : char)
        end;
var V : R; ...
V.RV := false; V.i := 2; V.RV := true; write(V.ch);
```

řádně se přeloží a vypíše nesmysl. %

Porovnání klasických konstrukcí – typy

Přístup k prvkům záznamu je mnohem rychlejší než k prvku pole

$$\text{Location (record.field_i)} = \text{address (record.field_1)} + \sum_{i=1}^{n-1} \text{offset_i}$$



Porovnání klasických konstrukcí – typy

Set – typ, jehož proměnné mohou mít hodnotu neuspořádané kolekce hodnot ordinálního typu

Zavedeny v Pascalu

*type NejakyTyp = Set of OrdinalniTyp (*třeba char*);*

Java, Python má třídu pro set operace, Ada a C je nevedou

Implementace – bitovými řetězci, používají logické operace

Vyšší efektivita než pole ale menší pružnost (omezovaný počet prvků)

Porovnání klasických konstrukcí – typy

Pointer - typ nabývající hodnot paměťového místa a nil (někde null)

Použití pro: -nepřímé adresování normálních proměnných

-dynamické přidělování paměti

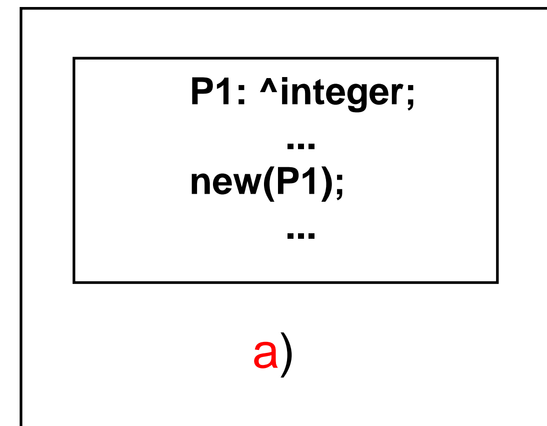
Operace s ukazateli: -přiřazení, dereference

Špatnost ukazatelů = **dangling** (neurčená hodnota) **pointers** a **ztracené** proměnné

Pascal – má jen pointery na proměnné z heapu alokované pomocí **new** a uvolňované **dispose**

- dereference tvaru jménopointeru^.položka

new(P1); P2 := P1; dispose(P1); P2 je teď dangling pointer



Problém ztracených proměnných

new(P1); .... new(P1); /* b)

V místě a) i b) je ztracená paměť

- implementace dispose znemožňující dangling není možná

Porovnání klasických konstrukcí – typy

C, C++ * je operátor dereference

& je operátor produkující adresu proměnné

`j = *ptr` přiřadí j hodnotu umístěnou v `ptr`

Pointer bývá položkou záznamu, pak tvar: *p.položka, nebo p → polozka

Pointerová aritmetika pointer + index

```
float stuff[100];
float *p; // p je ukazatel na float
p = stuff;
```

`*(p+5)` je ekvivalentní `stuff[5]` nebo `p[5]`

`*(p+i)` je ekvivalentní `stuff[i]` nebo `p[i]`

Pointer může ukazovat na funkci (umožňuje přenášet fce jako parametry)

Dangling a ztraceným pointerům nelze nijak zabránit

Porovnání klasických konstrukcí – typy

Nebezpečnost pointerů v C:

```
main()
{ int x, *p;
    x = 10;
    *p = x;          /* p =&x  teprve tohle je správně*/
    return 0;
}
```

Pointer je neinicializovaný, hodnota x se přiřadí do neznáma

```
main()
{ int x,*p;
    x = 10; p=x;            /* p =&x  tohle je správně*/
    printf(“%d”,*p);
    return 0;
}
```

Pointer je neinicializovaný, tiskne neznámou hodnotu

Porovnání klasických konstrukcí – typy

ADA – pouze proměnné z haldy (access type)

– dereference (pointer . jméno_položky)

– dangling významně potlačeno (automatické uvolňování místa na haldě, jakmile se výpočet dostane mimo rozsah platnosti ukazatele)

Hoare prohlásil: “The introduction of pointers into high level languages has been a step backward” (flexibility ↔ safety)

Java:

– nemá pointery

– má referenční proměnné (ukazují na objekty místo do paměti)

– referenčním proměnným může být přiřazen odkaz na různé instance třídy. Instance Java tříd jsou dealokovány implicitně ⇒ dangling reference nemůže vzniknout.

– Paměť haldy je uvolněna garbage collectorem, poté co systém detekuje, že již není používána.

C#:

- má pointry tvaru *referent-type * identifikátor* type může být i void

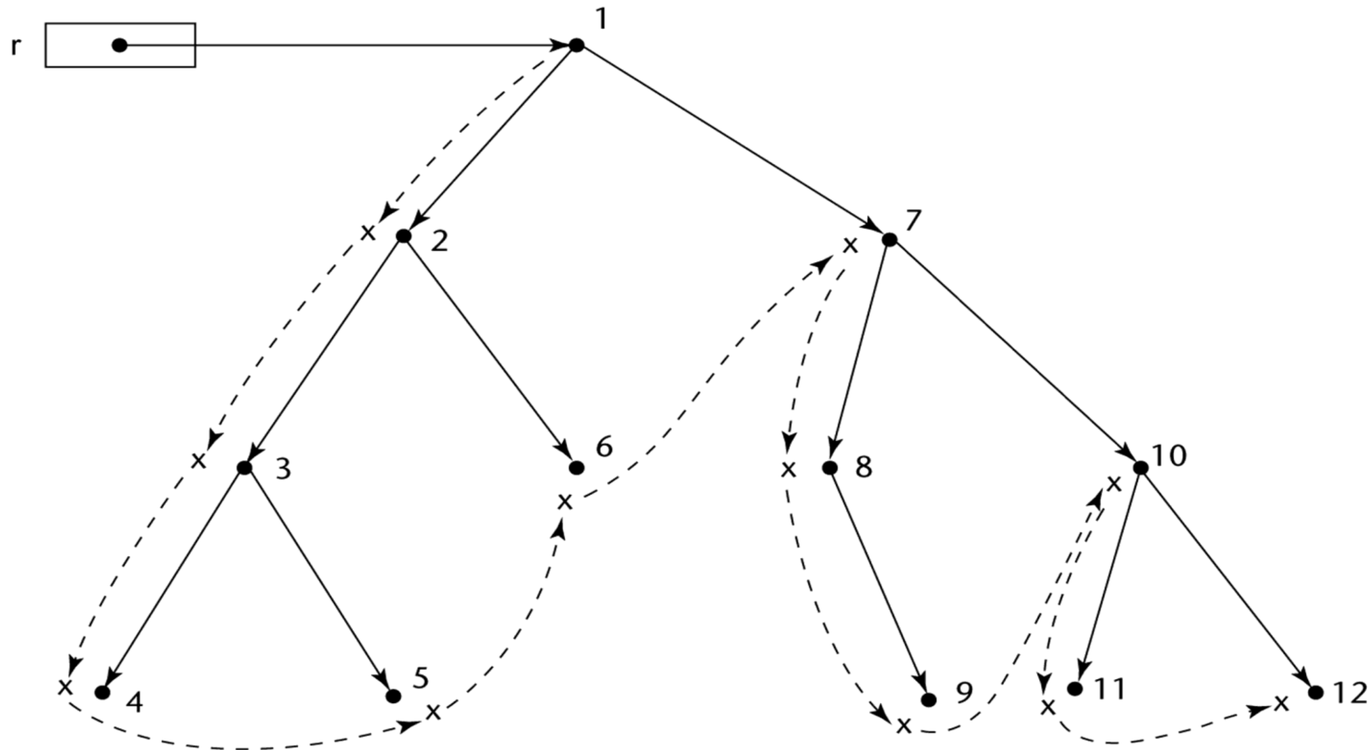
- metody pracující s pointerem musí mít modifikátor *unsafe*

- referenční proměnné má také

Porovnání klasických konstrukcí – **Manažování haldy**:

-Čítačem odkazů (každá buňka vybavena čítačem, když 0⇒ji vrátí do volných)

-Garbage collectorem (když nemá, prohledá všechny buňky a haldu zdrcne)



Průběh označování označování paměťových míst čističem

Porovnání klasických konstrukcí – výrazy a příkazy

Výrazy - aritmetické
-logické

Ovlivnění vyhodnocení:

1. Precedence operátorů?
2. Asociativita operátorů?
3. Arita operátorů?
4. Pořadí vyhodnocení operandů?
5. Je omezován vedlejší efekt na operandech?
   X=f(&i)+(i=i+2); je dovoleno v C
6. Je dovoleno přetěžování operátorů?
7. Je alternativnost zápisu (Java, C) C=C+1; C+=1; C++; ++C mají tentýž efekt, což neprospívá čitelnosti

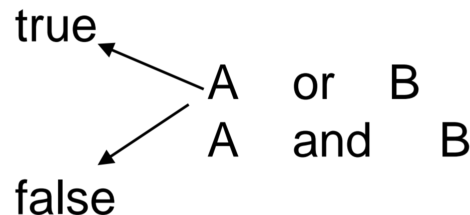
Porovnání klasických konstrukcí – výrazy a příkazy

```
#include <stdio.h>
int f(int *a);

int main()
{int x,z;
 int y=2; int i=3;
 /*C, C++, Java přiřazení produkuje výslednou hodnotu použitelnou jako operand*/
 x = (i=y+i) + f(&i);   /* ?pořadi vyhodnoceni operandů jazyky neurčují*/
 printf("%d\n",i);  printf("%d\n",x);

 y=2; i=3;
 z = f(&i) + (i=y+i);   /* ?pořadi vyhodnoceni a tedy výsledek se mohou lišit*/
 printf("%d\n",z); printf("%d\n",i);
 getchar();
 return 0;
}              /*BC vyhodnocuje nejdříve fci, ale MicrosoftC vyhodnocuje zprava doleva*/

int f(int *i)
{int x;
 *i = *i * *i;
 return *i;
}
```

Porovnání klasických konstrukcí – výrazy a příkazy

! ! u C, C++ na záměnu if (x = y) … /*je přiřazením ale produkuje log. hodnotu*/
se zápisem **if (x == y) …**
Proto C#, Java dovolují za **if** pouze logický výraz

Logické výrazy nabízí možnost zkráceného vyhodnocení



ADA

if A and then B then S1 else S2 end if;
if A or else B then S1 else S2 end if;

zkrácené:

if A and then B then S1 else S2 end if;
if A or else B then S1 else S2 end if;

Java

např. pro **(i=1;j=2;k=3;):**

if (i == 2 && ++j == 3) k=4; ?jaký výsledek i=1, j=2, k=3

Porovnání klasických konstrukcí – výrazy a příkazy

Využití podmíněných výrazů

v přiřazovacích příkazech C jazyků, Javy k = (j == 0) ? j+1 : j-1 ;

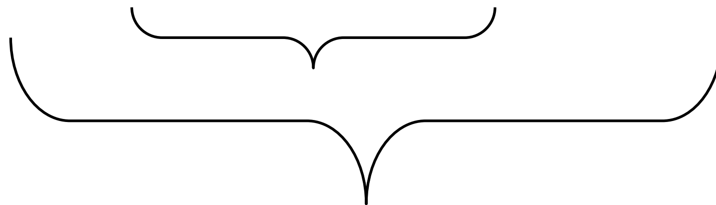
(pozn. v C jazycích, Javě produkuje přiřazení hodnotu, může být ve výrazu.)

v řídících příkazech

neúplný a úplný podmíněný příkaz

problém „dangling else“ při vnořovaném „if“

if x=0 then if y=0 then z:=1 else z:=2;



řešení: Pascal, Java – else patří k nejbližšímu nespárovanému if

ADA – párování if … end if

Porovnání klasických konstrukcí – výrazy a příkazy

Příkaz vícenásobného selektoru - přepínač

Pascal, ADA:

```
case expression of
    constant_list : statement1;
          ....
    constant_listn : statementn;
end;
```

Alternativa else / others. „Návěští“ ordinálního typu

C jazyky, Java

```
switch (expression) {
    case constant_expr1 : statements;
          ....
    case constant_exprn : statementsn;
    default : statements
}
```

Alternativy u **C, C++, Java** separuje „break“, u **C#** také goto.

„Návěští“ je výraz typu **int, short, char, byte** u **C#** i **enum, string**.

Porovnání klasických konstrukcí – výrazy a příkazy

<u>Cykly</u>

loop … end loop; primitivní tvar

while *podmínka* do …; cyklus logicky řízený pretestem Pascal,ADA
while (*podmínka*) *příkaz*; Cjazyky, Java

repeat … until *podmínka*; cyklus logicky řízený posttestem Pascal
do { příkazy ; } while (podmínka); Cjazyky, Java

for … ; s parametrem cyklu, krokem,iniciální a koncovou hodnotou

Pascal:

for *variable* := *init* **to** *final* **do** *statement*;

po skončení cyklu není hodnota *variable* definována

ADA obdobně, ale *variable* je implic. deklarovanou proměnnou cyklu, vně neexistuje
Java vyžaduje explicitní deklaraci parametru cyklu

Porovnání klasických konstrukcí – výrazy a příkazy

C++, C#, Java

```
        for (int count = 0; count < fin; count++) { ...     } ;
                  1.                  2.          4.          3.
```

Co charakterizuje cykly:

- Jakého typu mohou být parametr a meze cyklu?
- Kolikrát se vyhodnocují meze a krok?
- Kdy je prováděna kontrola ukončení cyklu?
- Lze uvnitř cyklu přiřadit hodnotu parametru cyklu?
- Jaká je hodnota parametru po skončení cyklu?
- Je přípustné skočit do cyklu?
- Je přípustné vyskočit z cyklu?

Porovnání klasických konstrukcí – výrazy a příkazy

Rozporný příkaz skoku

Nevýhody - znepřehledňuje program
- je nebezpečný
- znemožňuje formální verifikaci programu

Výhody - snadno implementovatelný
- efektivně implementovatelný

Formy návěští:

číslo: Pascal

číslo Fortran

identifikátor: Cjazyky

<<identifikátor>> ADA

proměnná PL/1

Java nemá skok GOTO, částečně jej nahrazuje break, ten ukončí blok s nav

```
nav1:   {
              nav2:   {         break nav2;
                      }
        }
```

Porovnání klasických konstrukcí – výrazy a příkazy

Zásada: Používej skoků co nejméně

Př. Katastrofický důsledek snahy po obecnosti (zavest promennou navesti v PL1)

```
B1: BEGIN; DCL L LABEL;
    ...
B2: BEGIN; DCL X,Y FLOAT;
          ...
L1: Y=Y+X;
          ...
L=L1;
END;
    ...
GOTO L; --zde existuje promenna L, ale hodnota L1 jiz neexistuje, jsme mimo B1
END;
```

Porovnání klasických konstrukcí – podprogramy

Procedury a Funkce jsou nejstarší formou abstrakce – abstrakce procesů

(Java a C# nemají klasické funkce, ale metody mohou mít libovolný typ)

Základní charakteristiky:

- Podprogram má jeden vstupní bod
- Volající je během exekuce volaného podprogramu pozastaven
- Po skončení běhu podprogramu se výpočet vrací do místa, kde byl podprogram vyvolán

Pojmy:

- Definice podprogramu
- Záhlaví podprogramu
- Tělo podprogramu
- Formální parametry
- Skutečné parametry
- Korespondence formálních a skutečných parametrů
  - jmenná vyvolání: jménopp(jménoformálního jménoskutečného, …
  - poziční vyvolání: jménopp(jménoskutečného, jménoskutečného…
- Default (předběžné) hodnoty parametrů

Porovnání klasických konstrukcí – podprogramy

Kritéria hodnocení podprogramů:

- Způsob předávání parametrů?
- Možnost typové kontroly parametrů ?
- Jsou lokální proměnné umisťovány staticky nebo dynamicky?
- Jaké je platné prostředí pro předávané parametry, které jsou typu podprogram ?
- Je povoleno vnořování podprogramů ?
- Mohou být podprogramy přetíženy (různé podprogramy mají stejné jméno) ?
- Mohou být podprogramy generické ?
- Je dovolena separátní kompilace podprogramů ?

Ad umístění lokálních proměnných:

-dynamicky v zásobníku

umožní rekurzivní volání a úsporu paměti

potřebuje čas pro alokaci a uvolnění, nezachová historii, musí adresovat nepřímo

(Pascal, Ada výhradně dynamicky, Java, C většinou)

-dynamicky na haldě (Smalltalk)

-staticky, pak opačné vlastnosti (Fortran90 většinou, C static lokální proměnné)

Porovnání klasických konstrukcí – podprogramy

Lokální data podprogramů jsou spolu s dalšími údaji uloženy v AKTIVAČNÍM ZÁZNAMU

Místo pro lokální proměnné Místo pro předávané parametry (Místo pro funkční hodnotu u funkcí) Návratová adresa Informace o uspořádání aktivačních záznamů Místo pro dočasné proměnné při vyhodnocování výrazů

Aktivační záznamy většiny jazyků jsou umístěny v zásobníku.

- Umožňuje vnořování rozsahových jednotek
- Umožňuje rekurzivní vyvolání

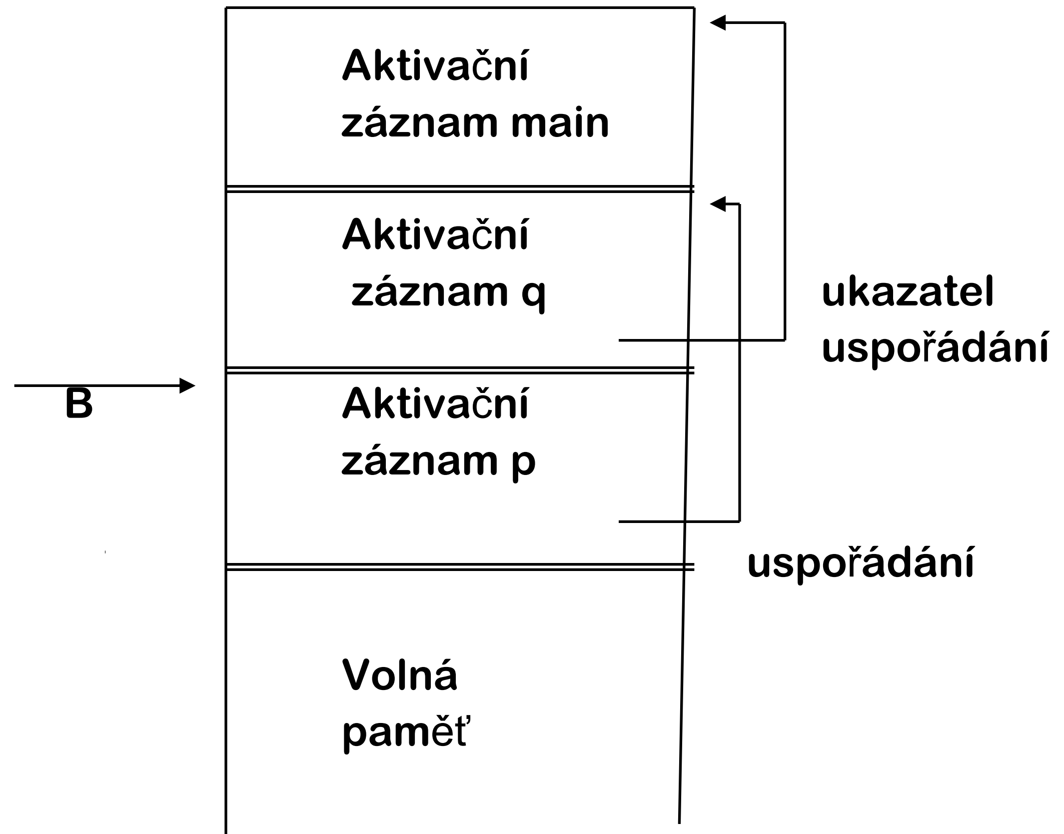
-Aktuální AZ je přístupný prostřednictvím ukazatele (nazvěme ho B) na jeho bázi

-Po skončení rozsahové jednotky je její AZ odstraněn ze zásobníku dle ukazatele uspořádání AZ

Porovnání klasických konstrukcí – podprogramy

Př. v jazyce C

```
int x;
void p( int y)
{ int i = x;
  char c; ...
}
void q ( int a)
{ int x;
  p(1);
    ukazatel
}
main()
{ q(2);
  return 0;
}
```



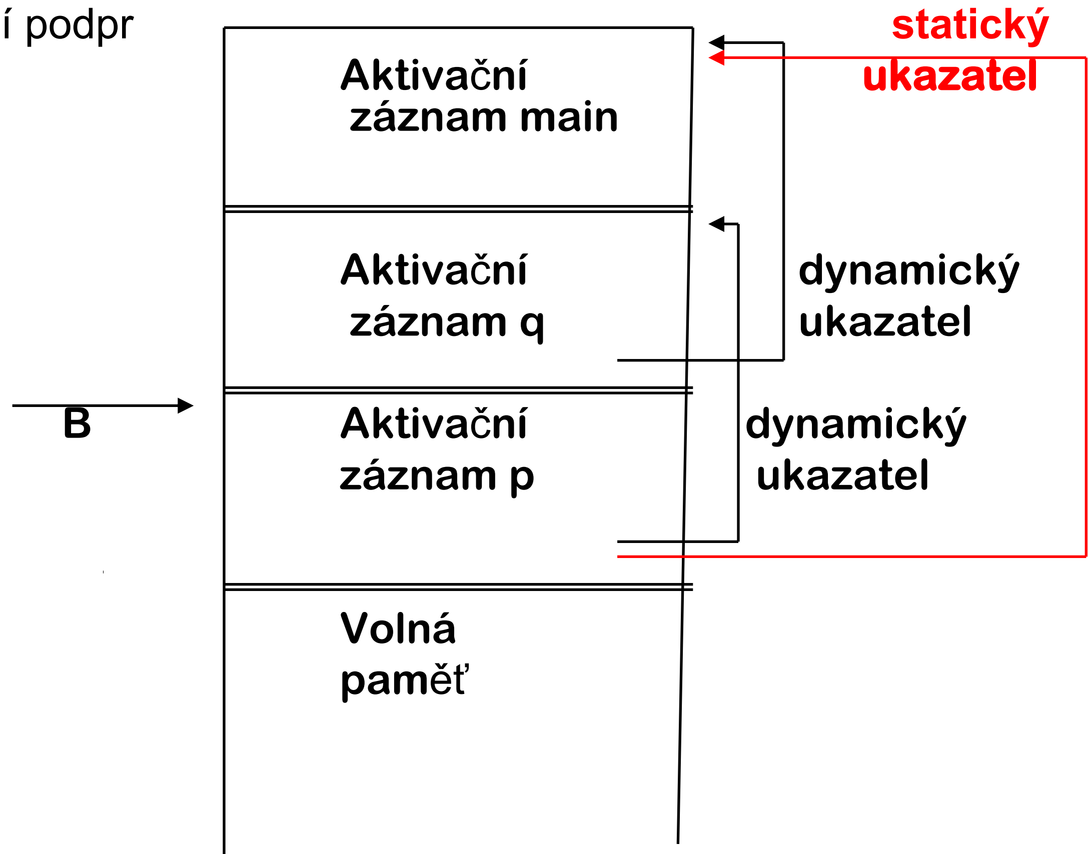
Jazyky se statickým rozsahem platnosti proměnných a vnořováním podprogramů vyžadují dva typy ukazatelů uspořádání (řetězců ukazatelů):

1. **(dynamický) Na rušení AZ opuštěných rozsahových jednotek (viz výše)**
2. **(statický) pro přístup k nelokálním proměnným**

Porovnání klasických konstrukcí – podprogramy

Př. Uvažujme možnost vnořování podpr

```
main() {
    int x;
void p( int y)
{ int i = x;
  char c; ...
}
void q ( int a)
{ int x;
  p(1);
}

  q(2);
  return 0;
}
```



K přistupování na proměnnou x z funkce p je nutno použít statický ukazatel

V C, C++ má statický řetěz délku 1, není proto nutný. V Adě, Pascalu může nabývat libovolné délky

Porovnání klasických konstrukcí – podprogramy

- Použití řetězce dynamických ukazatelů k přístupu k nelokálním proměnným způsobí, že nelokální proměnné budou zpřístupněny podle dynamické úrovně AZ
- Použití řetězce statických ukazatelů způsobí, že nelokální proměnné budou zpřístupněny podle lexikálního tvaru programu
- Cjazyky, Java mají buď globální (static) proměnné, které jsou přístupné přímo, nebo lokální patřící aktuálnímu objektu / vrcholovému AZ, které jsou přístupné přes „this“ pointer
- Jazyky s vnořovanými podprogramy při odkazu na proměnnou, která je o n úrovní globálnější než-li aktuálně prováděný podprogram, musí sestoupit do příslušného AZ o n úrovní statického řetězce.
- Úroveň vnoření L rozsahových jednotek, potřebnou velikost AZ a offset F proměnných v AZ vůči jeho počátku zaznamenává překladač. (L, F) je dvojice, která reprezentuje adresu proměnné.

Porovnání klasických konstrukcí – podprogramy

Způsoby předávání parametrů:

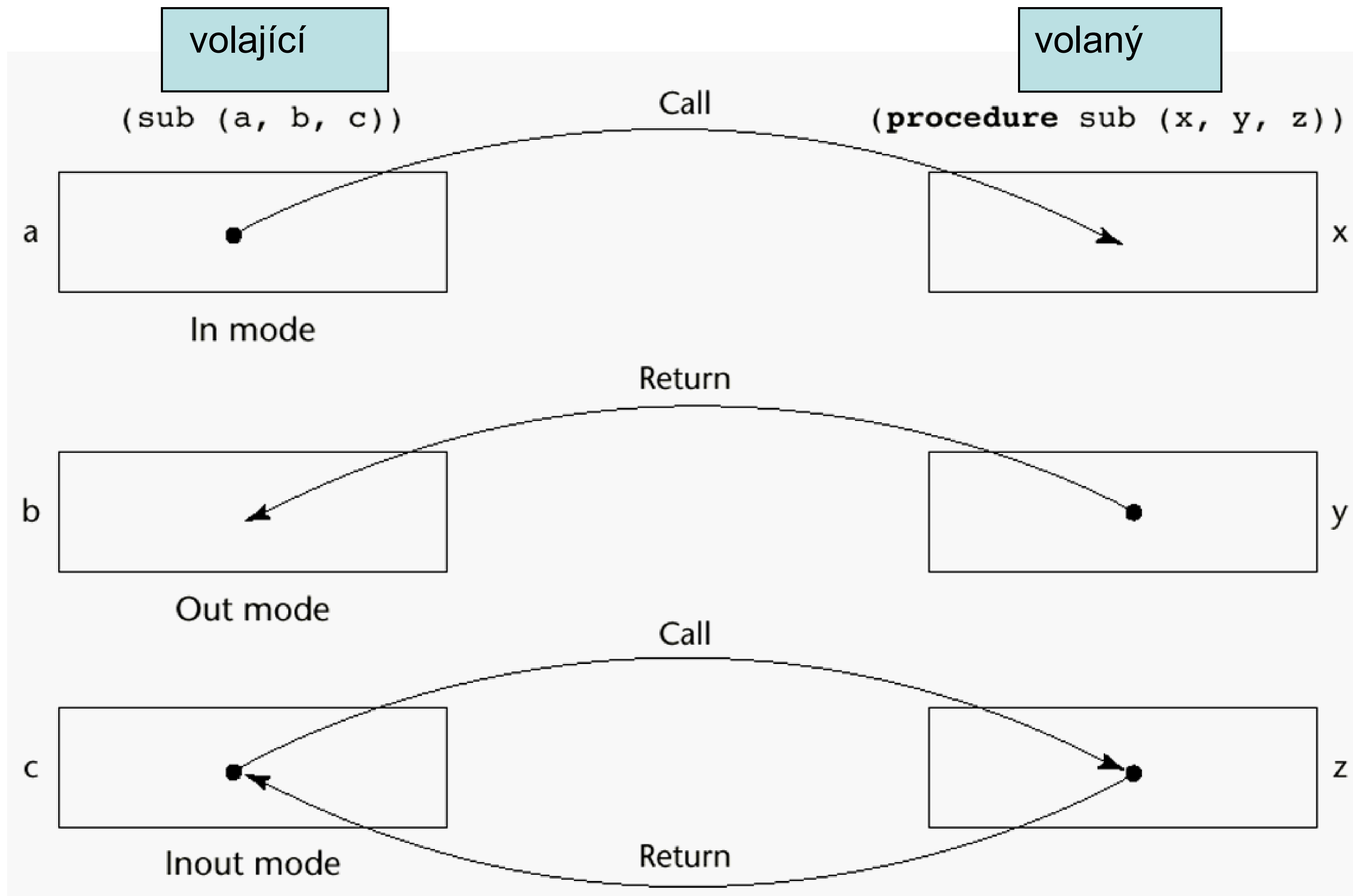
- **Hodnotou** (in mode), obvykle předáním hodnoty do parametru (lokální proměnné) podprogramu (Java). Vyžaduje více paměti, zdržuje přesouváním
- Výsledkem (out mode), do místa volání je předána při návratu z podprogramu lokální hodnota. Vyžaduje dodatečné místo i čas na přesun
- Hodnotou výsledkem (in out mode), kopíruje do podprogramu i při návratu do místa volání. Stejné nevýhody jako předešlé
- **Odkazem** (in out mode), předá se přístupová cesta. Předání je rychlé, nepotřebuje další paměť. Parametr se musí adresovat nepřímo, může způsobit synonyma.

```
podprogram Sub( a ,  b) ;
. . .
Sub( x , x) ;            . . .
```

- Jménem (in out mode), simuluje textovou substituci formálního parametru skutečným. Neefektivní implementace, umožňuje neprůhledné triky
- Předání vícerozměrného pole – je-li podprogram separátně překládán, potřebuje znát velikost pole. C, C++ má pole polí ukládané po řádcích, Údaje pro mapovací fci požadují zadání počtu sloupců např. void fce (int matice [ ] [10]) {…} v definici funkce. Výsledná nepružnost vede k preferenci použití pointerů na pole.

  Java má jednorozměrná pole s prvky opět pole. Každý objekt pole dědí length atribut. Lze proto deklarovat pružně float fce (float matice [ ] [ ]) { …} Podobně ADA

Porovnání klasických konstrukcí – podprogramy



Porovnání klasických konstrukcí – podprogramy

- Podprogramy jako parametry, C, C++ dovolují předávat jen pointery na funkce, Ada nedovoluje vůbec

```
sub1 {
        sub2 {
        }
        sub3 {
                call sub4 (sub2)
        }
        sub4 (subformalni)
                call subformalni
        }
        call sub3
}
```

Jaké je výpočtové prostředí sub2 po jeho vyvolání v sub4 ? Existuje více možností

1. Mělká vazba – platné je prostředí volajícího podprogramu (sub4)
2. Hluboká vazba – platí prostředí, kde je definován volaný podprogram /sub1)
3. Ad hoc vazba – platí prostředí příkazu volání, který předává podprogram jako parametr (sub3)

Blokově strukturrované jazyky používají 2, SNOBOL užívá 1, 3 se neužívá

Porovnání klasických konstrukcí – podprogramy

<u>Přetěžovaný podprogram</u> je takový, který má stejné jméno s jiným podprogramem a existuje ve stejném prostředí platnosti (C++, Ada, Java). Poskytuje **<u>Ad hoc polymorfismus</u>**

C++, ADA dovolují i přetěžování operátorů

Př.ADA

```
function "*"(A, B: INT_VECTOR_TYPE) return INTEGER is
    S: INTEGER := 0;
begin
    for I in A'RANGE loop
            S:= S + A(I) * B(I);
    end loop;
    return S;
end  "*";
```

Př.C++

```
int  operator *(const vector &a, const vector &b);  //tzv. function prototype
```

Porovnání klasických konstrukcí – podprogramy

Generické podprogramy dovolují pracovat s parametry různých typů. Poskytují tzv **parametrický polymorfismus**

C++ obecný tvar:

template<class parameters> definice funkce, která může obsahovat class parametry

Example:

```
template <class Typf>
Typf max(Typf first, Typf second) {
return first > second ? first : second;
}
```

Instalace je možná pro libovolný typ, který má gefinováno > , např. integer

```
int max(int first, int second)
 {return first > second ? first : second;}    1)
```

1) efect je následující

C++ template funkce je instalována implicitně když buď je použita v příkazu vyvolání nebo je použita s & operatorem

```
 int a, b, c;
char d, e, f;
...
c = max(a, b);
```

OOP

- **<u>Objektový polymorfismus</u>** v jazycích Java, Object Pascal a C++
- Objektové konstrukce v programovacích jazycích
- Jak jsou dědičnost a polymorfismus implementovány v překladači
- Příklady využití polymorfismu
- Násobná dědičnost a rozhraní

Př. 1JavaZvirata v OOPOstatni

```
class Animal {
    String  type ;
    Animal() {                      type = "animal  ";}
    String sounds() {               return "not known";}
    void prnt() {                   System.out.println(type + sounds());}
}
class Dog extends Animal {
    Dog() {                         type = "dog   "; }
    String sounds() {               return "haf";   }
}
class Cat extends Animal {
    Cat() {             type = "cat   ";   }
    String sounds() {        return "miau";   }
}
 public class Anim {
  public static void main(String [] args) {
   Animal notknown = new Animal();
   Dog filipes = new Dog();
   Cat tom = new Cat();
   tom.prnt();
   filipes.prnt();
   notknown.prnt();
} } //konec. Co se tiskne? Dle očekávání kočka mnouka, pes steka
```

Objektové prostředky Pascalu

Př.2PascalZvířata Obdobný příklad v Pascalu

```
program Zvirata;
{$APPTYPE CONSOLE}
uses  SysUtils;
type
    Uk_Zvire = ^Zvire;
    Zvire = object
            Druh    : String;
            procedure Inicializuj;
            function Zvuky: String;
            procedure Tisk;
           end;
    Uk_Pes = ^Pes;
    Pes =   object(Zvire)
            procedure Inicializuj;
            function Zvuky: String;
           end;
    Uk_Kocka = ^Kocka;
    Kocka = object(Zvire)
            procedure Inicializuj;
            function Zvuky: String;
           end;
```

```
{-----------Implementace metod--------------------------}
   procedure Zvire.Inicializuj;
   begin  Druh := 'Zvire  '
   end;
   function Zvire.Zvuky: String;
   begin  Zvuky := 'nezname'
   end;
   procedure Zvire.Tisk;
   begin  writeln(Druh, Zvuky);
   end;

   procedure Pes.Inicializuj;
   begin  Druh := 'Pes   '
   end;
   function Pes.Zvuky: String;
   begin  Zvuky := 'steka'
   end;

   procedure Kocka.Inicializuj;
   begin  Druh := 'Kocka   '
   end;
   function Kocka.Zvuky: String;
   begin  Zvuky := 'mnouka'
   end;
```

```
{----------------Deklarace objektu--------------------}
    var
      U1, U2, U3: Uk_Zvire;
      Nezname: Zvire;
      Micka: Kocka;
      Filipes: Pes;

{---------------Hlavni program-------------------------}
begin
    Filipes.Inicializuj;   Filipes.Tisk;    {  !!!???   }
    new(U1); U1^.Inicializuj; U1^.Tisk;
    Micka.Inicializuj; writeln(Micka.Druh, Micka.Zvuky);
    readln;
end.
```

Konec Př.1ObjectPascalZvířata. Co se tiskne? Pes zde vydává neznámé zvuky

Lze zařídit stejné chování i Java programu?

Umí se stejně chovat i Java? Co se tiskne?

```
class Animal {      //program AnimS.java
    String  type ;
    Animal() {                              type = "animal  ";}
    static String sounds() {                return "not known";}
    void prnt() {       System.out.println(type + sounds());}
}
class Dog extends Animal {
    Dog() {                                 type = "dog   "; }
    static String sounds() {                return "haf";   }
}
class Cat extends Animal {
    Cat() {                                 type = "cat   ";   }
    static String sounds() {                return "miau";   }
}
 public class Anim {
  public static void main(String [] args) {
   Animal notknown = new Animal();
   Dog filipes = new Dog();
   Cat tom = new Cat();
   tom.prnt();
   filipes.prnt();
   notknown.prnt();            //vsechno se ozyva not known
} }
```

```
class Animal { //                    program AnimF.java ?co to ted udela?
   String  type ;
   Animal() {                        type = "animal  ";}
   final String sounds() {          return "not known";}
   void prnt() {          System.out.println(type + sounds());}
}
class Dog extends Animal {
   Dog() {                type = "dog    "; }
   final String sounds() {                        return "haf";   }
}
class Cat extends Animal {
   Cat() {                type = "cat    ";   }
   final String sounds() {          return "miau";   }
}
 public class Anim {
  public static void main(String [] args) {
   Animal notknown = new Animal();
   Dog filipes = new Dog();
   Cat tom = new Cat();
   tom.prnt();
   filipes.prnt();
   notknown.prnt();
} }          //zase vsichni jsou not known
```

```
class Animal { //       soubor AnimF.java   ?co to ted udela?
    String  type ;
    Animal() {                              type = "animal  ";}
   private final String sounds() {          return "not known";}
    void prnt() {          System.out.println(type + sounds());}
}
class Dog extends Animal {
    Dog() {                type = "dog   "; }
 private final String sounds() {            return "haf";   }
}
class Cat extends Animal {
    Cat() {                type = "cat   ";   }
 private final String sounds() {       return "miau";   }
}
 public class Anim {
  public static void main(String [] args) {
   Animal notknown = new Animal();
   Dog filipes = new Dog();
   Cat tom = new Cat();
   tom.prnt();
   filipes.prnt();
   notknown.prnt();
} }            //opet vsichni davaji zvuky not known
```

Na dalším obrázku máte kombinace private, static, final a výsledné chování

```
class Animal {
  public String  type ;
  public Animal() {
     type = "animal  "; }
//static                                  //ne   ne    ano   ne    ne    ne    ano
//private                                 //ne   ne    ne    ano   ano   ano   ano
//final                                   //ne   ne    ne    ne    ne    ano   ne
  String sounds() {
   return "not known";
  }
  public void prnt() {
   System.out.println(type + sounds());
} }
class Dog extends Animal {
  public Dog() {
     type = "dog   ";
  }
//static                                  //ne   ne    ano   ne    ne    ne    ano
//private                                 //ne   ne    ne    ano   ne    ano   ano
//final                                   //ne   ano   ne    ne    ne    ano   ne
   String sounds() {return "haf"; }
}
public class AnimX {
 public static void main(String [] args) {
  Animal notknown = new Animal();
  Dog filipes = new Dog();
  filipes.prnt();
  notknown.prnt();
```

Správně štěká (první dva sloupce)

neštěká (ostatní sloupce)

Ostaní kombinace hlásí chybu

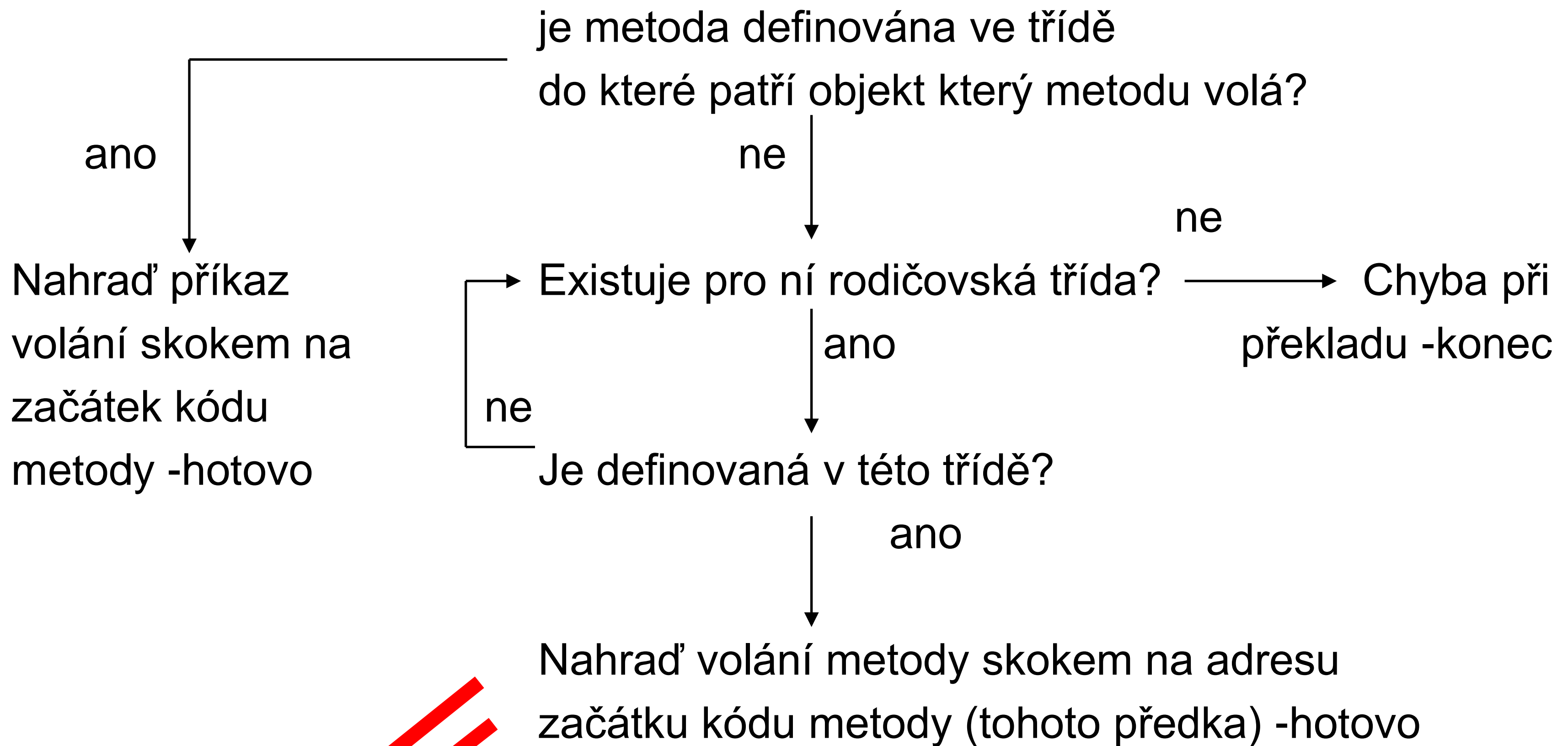
Př.Zvirata1x Správné řešení aby pes stekal v Borland Pascal

```
Zvire =  object
          Druh    : String;
          constructor Inicializuj;                {!!! Inicializuj musí být konstruktor}
          function Zvuky: String; virtual;        {!!! Zvuky musí být virtual}
          procedure Tisk;
         end;
. . .
Pes =   object(Zvire)
          constructor Inicializuj;                {!!!}
          function Zvuky: String; virtual;        {!!!}
        end;
. . .
Filipes.Inicializuj;
 Filipes.Tisk;                          { !!!??? }

 new(U1);
 U1^.Inicializuj;
 U1^.Tisk; . . .
```

řešení Object Pascalem (Pascal z Delphi)

```
Př.Zvirata1
Zvire =  class
           Druh    : String;
           constructor Inicializuj;                  {!!!}
           function Zvuky: String; virtual;          {!!!}
           procedure Tisk;
          end;
Pes =   class(Zvire)
           constructor Inicializuj;                  {!!!}
           function Zvuky: String; override;        {!!!u potomka je overide místo virtual}
         end;
. . .
Filipes := Pes.Inicializuj;
 Filipes.Tisk;                                       { !!!??? }

 new(U1);
 U1^:= Zvire.Inicializuj;
 U1^.Tisk;

 Micka := Kocka.Inicializuj;
 writeln(Micka.Druh, Micka.Zvuky);  . . .
```

Dědičnost a statická (brzká) vazba =proč pes neštěká



Obsahuje-li tato metoda volání další metody, je tato další metoda metodou předka, i když potomek má metodu, která ji překrývá – viz Zvirata

Dědičnost a dynamická (pozdní) vazba= pak pes štěká

- Realizovaná pomocí virtuálních metod
- Při překladu se vytváří pro každou třídu tzv. datový segment, obsahující:
  - údaj o velikosti instance a datových složkách
  - údaj o předkovi třídy
  - ukazatele na tabulku metod s pozdní vazbou (**V**irtual **M**ethod **T**able)
- Před prvým voláním virtuální metody musí být provedena (příp. implicitně) speciální inicializační metoda – constructor (v Javě přímo stvoří objekt)
- Constructor vytvoří spojení (při běhu programu) mezi instancí volající konstruktor a VMT. Součástí instance je místo pro ukazatel na VMT třídy, ke které instance patří. Constructor také v případech kdy je objekt na haldě ho přímo vytvoří (přidělí mu místo -tzv. Class Instance Record). Objekty tvořené klasickou deklarací (bez new …) jsou umístěny v RunTime zásobníku, alokaci jejich CIR tam zajistí překladač.
- Volání virtuální metody je realizováno nepřímým skokem přes VMT
- Pokud není znám typ instance (objektu) při překladu (viz případ kdy ukazatel na objekt typu předka lze použít k odkazu na objekt typu potomka), umožní VMT polymorfní chování tzv **objektový polymorfismus**

Dědičnost a dynamická (pozdní) vazba

Př CPP zápis.

class A { public: void f( ); virtual void g( ); double x, y; } ;

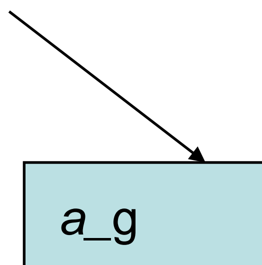
class B: public A { public: void f( ); virtual void h( ); **void g( )**; double z; };

<u>Alokované místo (CIR) objektu *a* třídy A</u>

Místo pro x

místo pro y

VMT ukazatel



VMT pro a

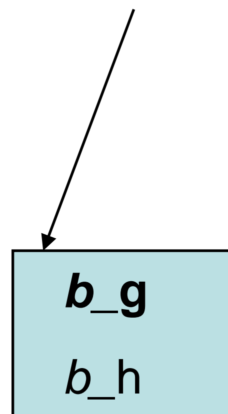
<u>Alokované místo (CIR) objektu *b* třídy B</u>

Místo pro x

místo pro y

místo pro z

VMT ukazatel



VMT pro b

Pozn.: a_f , b_f jsou statické, skok na jejich začátek se zařídí při překladu

Dědičnost a dynamická (pozdní) vazba

Př CPP zápis.   Zde g z třídy A není ve třídě B překrytá, takže objekt b dědí a_g

class A { public: void f( ); virtual void g( ); double x, y; } ;

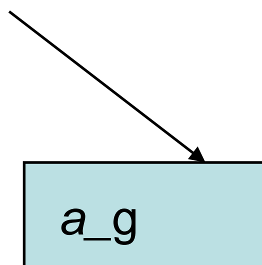
class B: public A { public: void f( ); virtual void h( ); double z; };

<u>Alokované místo objektu *a* třídy A</u>

Místo pro x

místo pro y

VMT ukazatel



VMT pro a

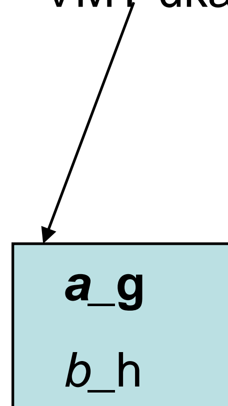
<u>Alokované místo objektu *b* třídy B</u>

Místo pro x

místo pro y

místo pro z

VMT ukazatel



VMT pro b

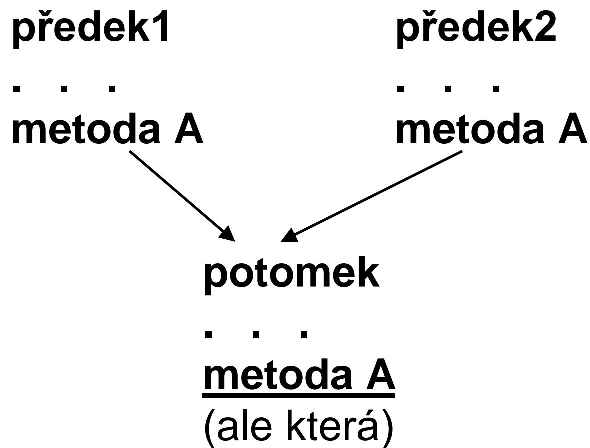
OOP konstrukce C++

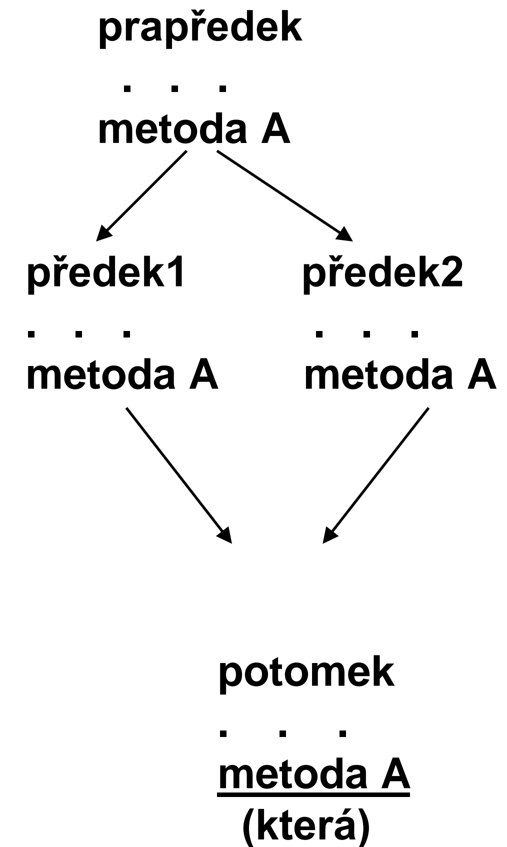
přístup v rodiči	modifikace přístupu v potomkovi public	private
public	public	private
protected	protected	private
private	nepřístupný	nepřístupný

Př.

```
CLASS Předek1 {  PUBLIC: int p11;
                 PROTECTED: int p12;
                 PRIVATE: int p13;
};
CLASS Předek2 {  PUBLIC: int p21;
                 PROTECTED: int p22;
                 PRIVATE: int p23;
};
CLASS Potomek: PUBLIC Předek1, PRIVATE Předek2; {
          PUBLIC: int pot1;
          PROTECTED: int pot2;
          PRIVATE: int pot3;
};
// Jaké datové elementy má Potomek? Public pot1, p11. Protected pot2, p12. Private
   pot3, p21, p22
```

OOP konstrukce C++ (problém násobné dědičnosti)





Řešení:

- **napsat plným jménem (jméno předka :: jméno)**
- **virtual předek1, virtual předek2** } **slovem virtual dáme informace, že se má uplatnit jen jeden**

Objektové vlastnosti C#   (pro informaci)

- Používá **class** i **struct**
- Pro dědění při definici tříd používá CPP syntax

  public class NovaTrida : RodicovskaTrida { . . . }
- V podtřídě lze nahradit metodu zděděnou od rodiče zápisem

  new definiceMetody;

  ta pak zakryje děděnou metodu stejného jména.

  Metodu z rodiče lze ale přesto volat pomocí zápisu např.

  base.vykresli( );
- Dynamická vazba je u metody rodičovské třídy povinně označena

  virtual

  a u metod odvozených tříd povinně označena

  override

  (převzato z Objekt Pascalu)

Př.

Objektové vlastnosti C#   (pro informaci)

```
public class Obrazec {
    public virtual void Vykresli ( ) { . . . }
    . . .
}
public class Kruh : Obrazec {
    public override  void Vykresli ( ) { . . . }
    . . .
}
public class Ctverec : Obrazec {
    public override void Vykresli ( ) { . . . }
    . . .
}
```

- **Má abstraktní metody, např, abstract public void Vykresli( );**

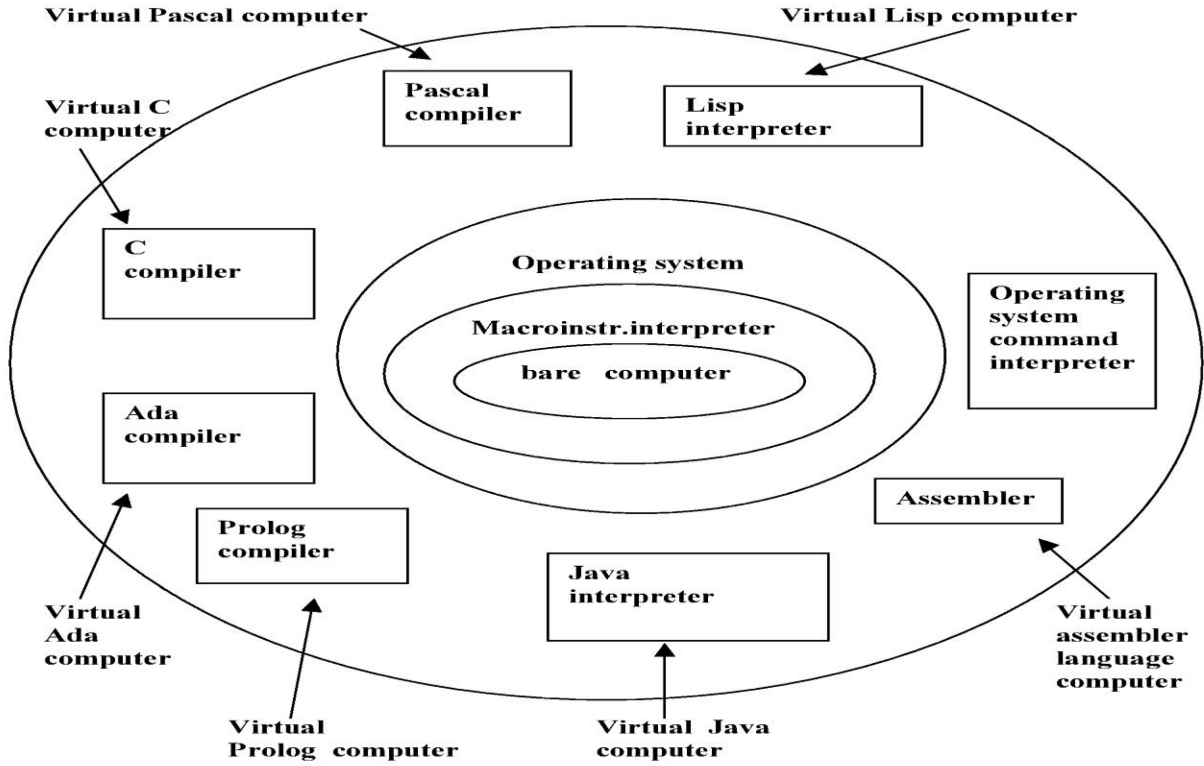
  ty pak musí být implementovány ve všech potomcích a třídy s abstract metodou musí být označeny abstract
- **Kořenem všech tříd je Object jako u Javy**
- **Nestatické vnořené třídy nejsou zavedeny**

Virtuální počítač

Hladiny

- Uživatelský program
- Překladač programovacího jazyka
- Operační systém
- Interpret makroinstrukcí
- Procesor

Virtual computer



Překladač

Z formálního hlediska je překladač zobrazení

Kompilátor : Zdrojový jazyk → Cílový jazyk

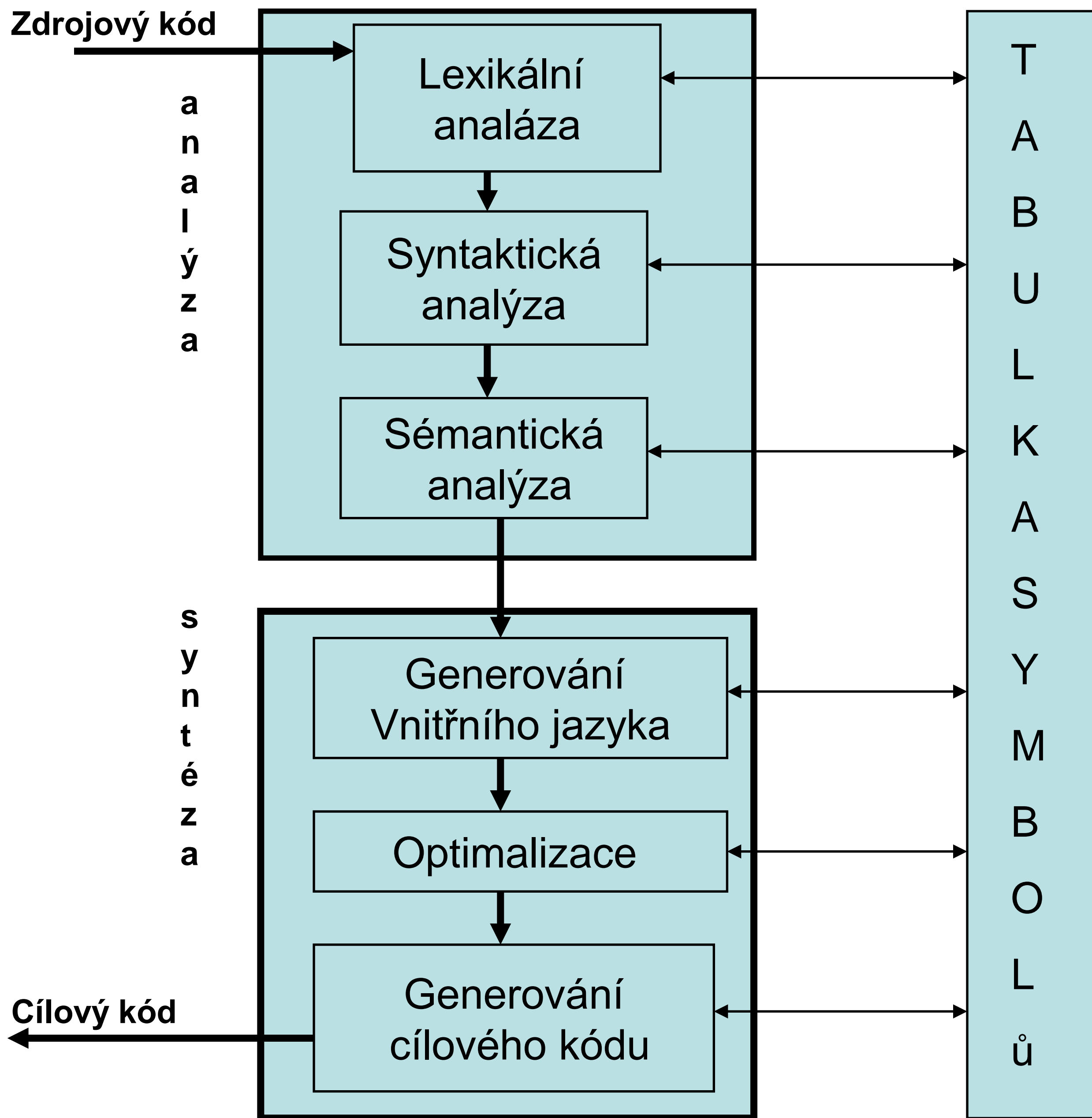
Jeho hlavní části tvoří

Analytická část:

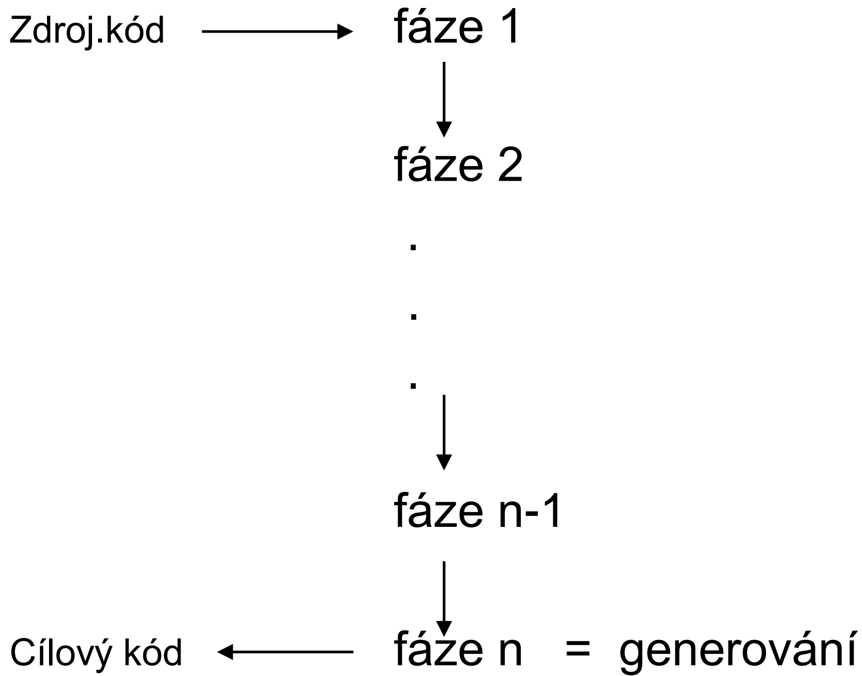
- Lexikální analýza –rozpoznává symboly jazyka
- Syntaktická a sémantická analýza –rozpoznává a kontroluje strukturu a kontextové závislosti programu

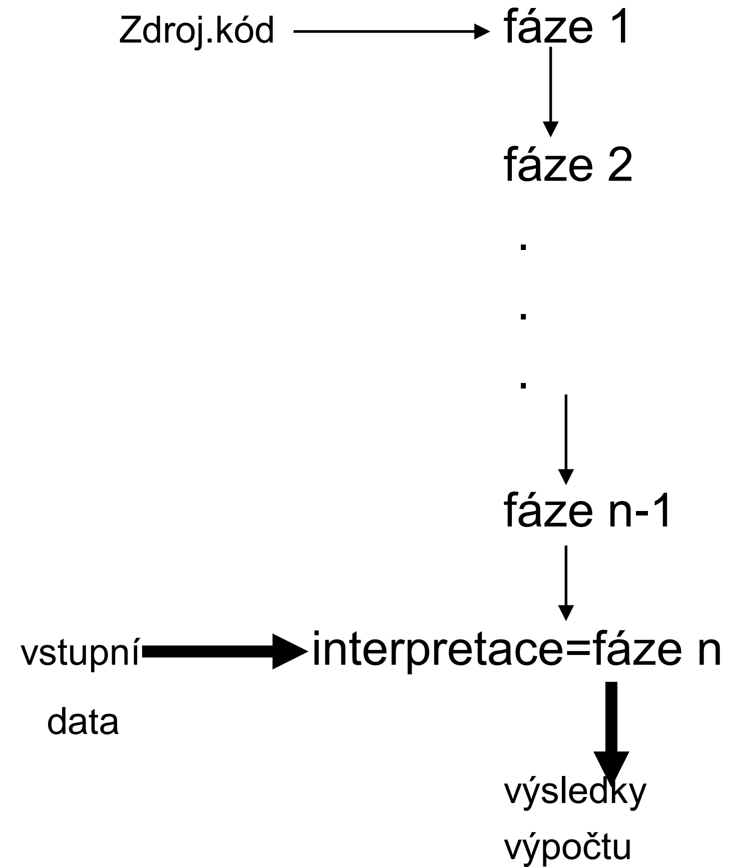
Syntetická část:

- Sémantické zpracování –převádí program do vnitřního jazyka překladače
- Generování cílového kódu -generuje cílový kod z vnitřního jazyka


Zdrojový kód
analýza
Lexikální analáza
Syntaktická analýza
Sémantická analýza
syntéza
Generování Vnitřního jazyka
Optimalizace
Generování cílového kódu
Cílový kód
TABULKA SYMBOLů

Kompilátor | Interpret





Interpret

V čistě jen interpretační podobě se nepoužívá (neefektivní). Od kompilátoru se liší poslední částí

Analytická část:

- Lexikální analýza
- Syntaktická a sémantická analýza

Syntetická část:

- Sémantické zpracování
- Interpretace

Lexikální analýza

1. Zakódování lexikálních elementů (do číselné podoby)
2. Vynechávání komentářů

Výhody: pevná délka symbolů pro další fáze zpracování

Např.
```
{ a = b * 3 + 5 ; /*  poznamka */
  c = a + 1 ;
}
```

při kódování:	ident	konst	+	*	=	{	}	;
na čísla	1	2	3	4	5	6	7	8

převede na: 1 adr a, 5, 1 adr b, 4, 2 3, 3, 2 5, 8, 1 adr c, 5, 1 adr a, 8, 7

Tj. vnitřní jazyk lexikálního analyzátoru (mezijazyk)

Lexikální analýza

Tvary lexikálních symbolů jsou popsatelné regulárními gramatikami:

<symbol> → <identifikátor> |
<číslo> |
class | if | while | . . . |
+ | - | * | / | . . . |
(|) | { | } | . . . |
=+ | ++ | == | . . .

<identifikátor> → <identifikátor> a | <identifikátor> b | . . .
<identifikátor>0 | . . . <identifikátor>9 |
a | b | . . .

Lexikální analyzátor je konečným automatem

Syntaktický analyzátor

- Zjišťuje strukturu překládaného textu (syntaktický strom)
- Je založen na bezkontextových gramatikách (dovolují popisovat vnořované závorkové struktury)
- Vytváří derivační strom

<složený příkaz> → { <seznam příkazů> } (1)

<seznam příkazů> → <příkaz> ; <seznam příkazů> | (2)

<příkaz> ; (3)

<příkaz> → <proměnná> = <výraz> (4)

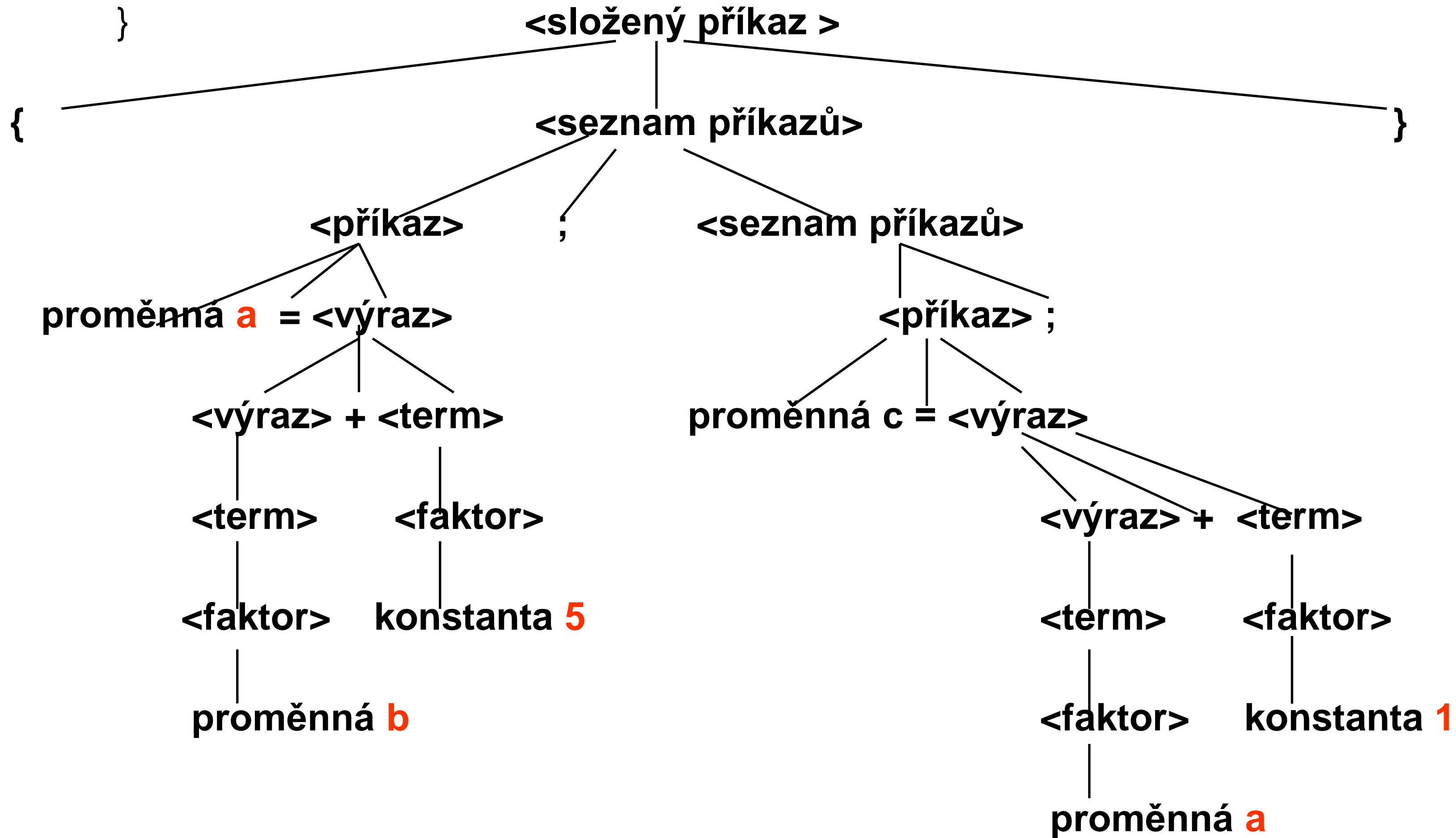
<výraz> → <výraz> + <term> | <výraz> - <term> | <term> (5)(6)(7)

<term> → <term> * <faktor> | <term> / <faktor> | <faktor> (8)(9)(10)

<faktor> → (<výraz>) | proměnná | konstanta (11)(12)(13)

Syntaktický analyzátor

Např. { a = b + 5 ; /* poznamka */
c = a + 1 ;
}



Syntaktický analyzátor

```
{ a = b + 5 ; /*  poznamka */
  c = a + 1 ;
}
```

Struktura je zachycena SA jako posloupnost použití gramatických pravidel při odvození tvaru programu z počátečního symbolu gramatiky

Možnosti:

- Levá derivace 1,2,4,5,7,10,12,10,13, . . .

  (rozepisuje vždy nejlevější neterminální symbol)
- Pravá derivace 1,2,3,4,5,10, . . .

  (rozepisuje vždy nejpravější neterminální symbol)

Sémantické zpracování

Souběžně či následně s rozpoznáváním syntaktické struktury jsou vyvolávány sémantické akce (sdružené s každým z gramatických pravidel), které převádí program do vnitřního jazyka překladače.

Formy vnitřních jazyků:

- **a. Postfixová notace (operátory následují za operandy)**
- **b. Prefixová notace**
- **c. Víceadresové instrukce**

Např. a = (b + c) * (a + c) ;

	a.	b.	c.
1	LOA a	ST	PLUS b c pom1
2	LOV b	LOA a	PLUS a c pom2
3	LOV c	MUL	MUL pom1 pom2 pom3
4	PLUS	PLUS	ST a pom3
5	LOV a	LOV b	
6	LOV c	LOV c	
7	PLUS	PLUS	
8	MUL	LOV a	
9	ST	LOV c	

Optimalizace

PLUS b c pom1

PLUS a c pom2

MUL pom1 pom2 pom3

ST a pom3

Redukce počtu pomocných proměnných, eliminace nadbytečných přesunů mezi registry, optimalizace cyklů

PLUS b c pom1

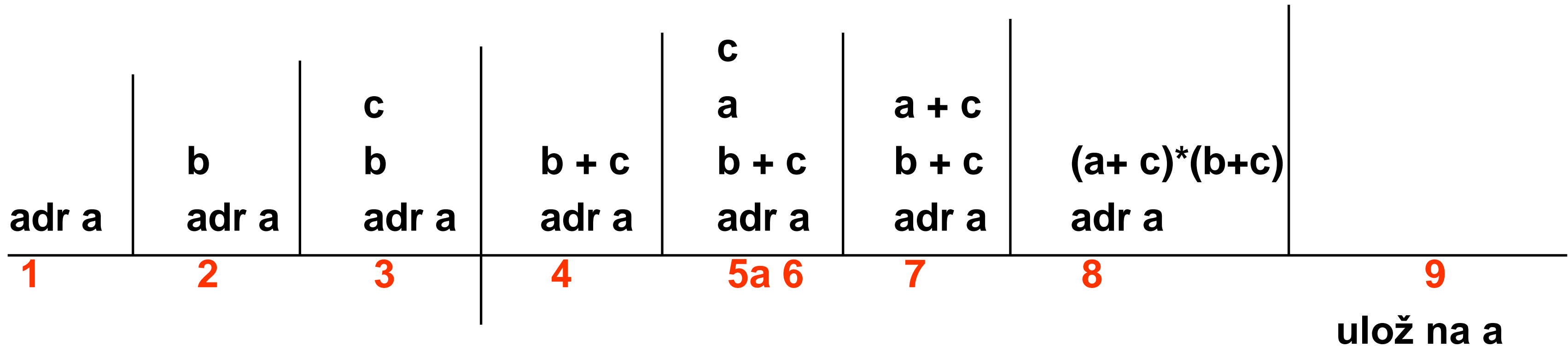
PLUS a c pom2

MUL pom1 pom2 pom1

ST a pom1

Interpretace

LOA a	dej adresu operandu a na vrchol zásobníku
LOV b	dej obsah operandu b na vrchol zásobníku
LOV c	dej obsah operandu c na vrchol zásobníku
PLUS	sečti vrchol a podvrchol, výsledek vlož do zásobníku
LOV a	dej obsah operandu a na vrchol zásobníku
LOV c	dej obsah operandu c na vrchol zásobníku
PLUS	sečti vrchol a podvrchol, výsledek vlož do zásobníku
MUL	vynásob vrchol a podvrchol, výsledek vlož do zásobníku
ST	ulož hodnotu z vrcholu zásobníku na adresu uloženou pod vrcholem



Generování

- Logicky jednoduché (expanze makroinstrukcí vnitřního jazyka)
- Prakticky komplikované (respektování instrukčních možností procesoru)

```
PLUS  b     c     pom1
PLUS  a     c     pom2
MUL   pom1  pom2  pom1
ST    a     pom1
```

vnitřní jazyk víceadresových instrukcí

```
MOV  b,  R0
ADD  c,  R0
STO  R0, p1
MOV  a,  R0
ADD  c,  R0
MUL  p1, R0
STO  R0, a
```

přeložený strojový kód