PROGRAMOVÉ STRUKTURY: PARADIGMATA

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

© Karel Ježek & Martin Zíma

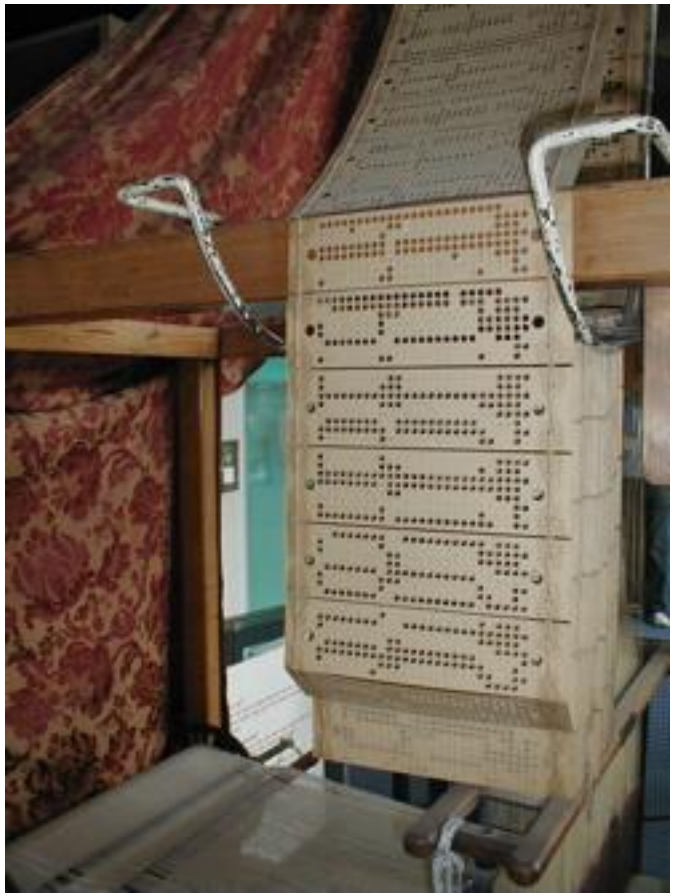
Vývoj hardware

https://upload.wikimedia.org/wikipedia/commons/b/b5/RomanAbacusRecon.jpg

- Abakus
- 2700 – 2300 př.n.l.
- Kupecké počty

- (kuličkové) počitadlo
- Sčot (RU)
- Soroban (JP)
- Suanpan (CN)

Vývoj hardware (2)

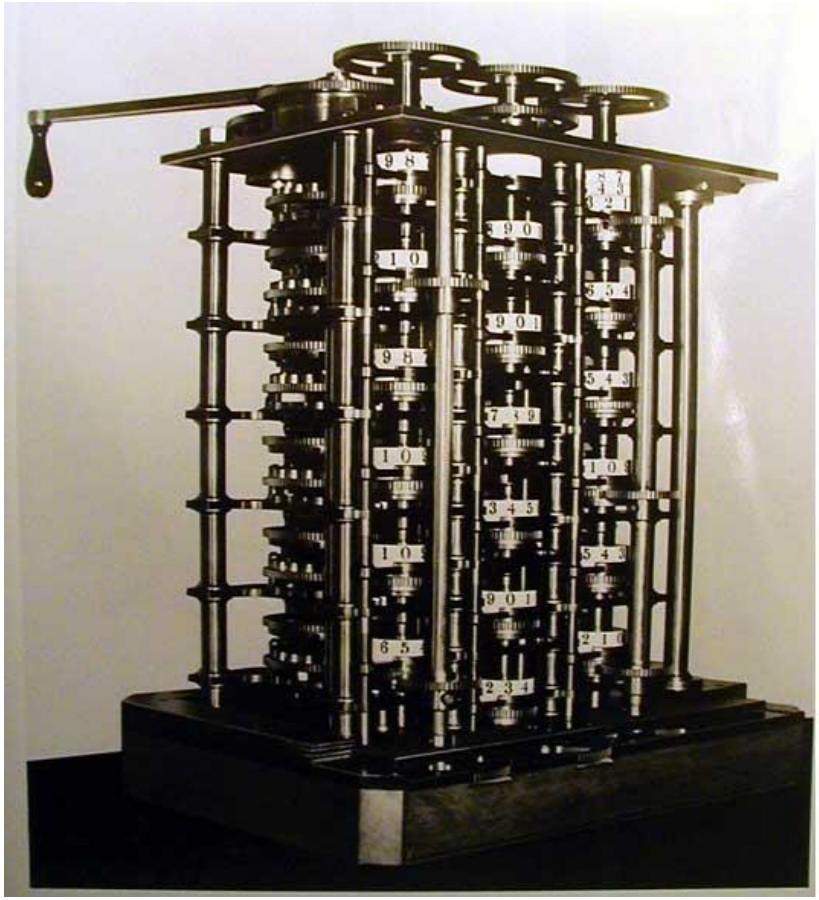


- Děrné štítky
- Kolem roku 1725
- Josef Marie Jacquard

- Programování tkalcovského stroje
- Později programování počítače

https://upload.wikimedia.org/wikipedia/commons/thumb/0/09/Jacquard.loom.cards.jpg/220px-Jacquard.loom.cards.jpg

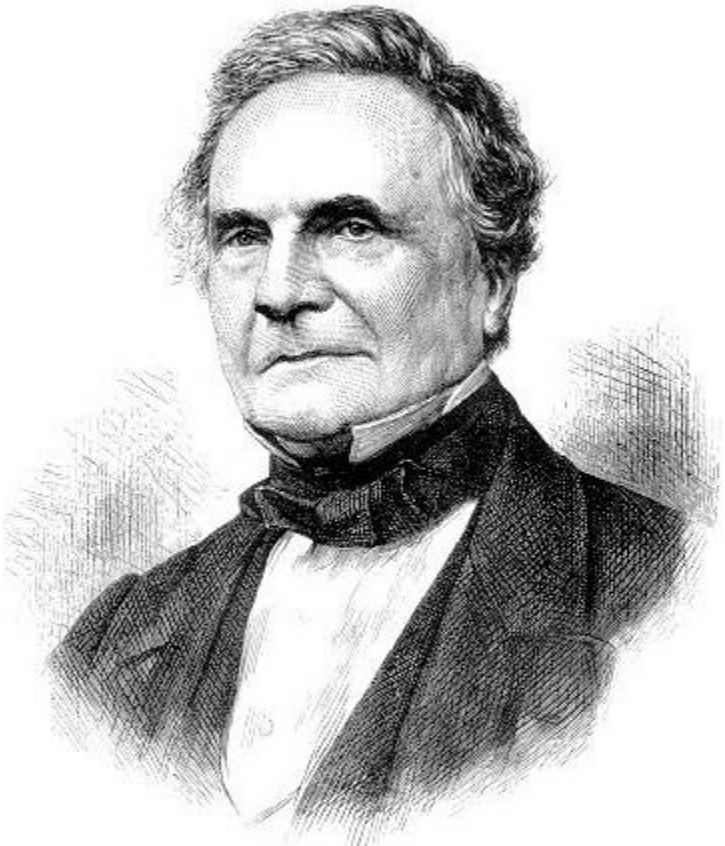
Vývoj hardware (3)



- Diferenční stroj
- 1822
- Charles Babbage
- 24 os s kolečky
- 96 koleček na ose
- Grant ve výši 7500 liber

http://data.pcworld.cz/img/article/img/3b/878a40c527ca3ac8539353867c5387.jpg

Charles Babbage



- Anglický matematik
- 1791 – 1871
- První diferenční stroj pro výpočet kvadratických rovnic (1822)
- Sestavil první tiskárnu
- Zakladatel počítačů

https://upload.wikimedia.org/wikipedia/commons/thumb/8/82/CharlesBabbage.jpg/225px-CharlesBabbage.jpg

Augusta Ada King (Byron)

https://upload.wikimedia.org/wikipedia/commons/thumb/8/87/Ada_Lovelace.jpg/220px-Ada_Lovelace.jpg

- Anglická matematička a programátorka
- 1815 – 1852
- Popsala diferenční stroj Charlese Babbage
- Napsala první program (1840)
- Hraběnka z Lovelace
- Pojmenován po ní jazyk Ada

První počítačový program (Bernoulliho čísla)

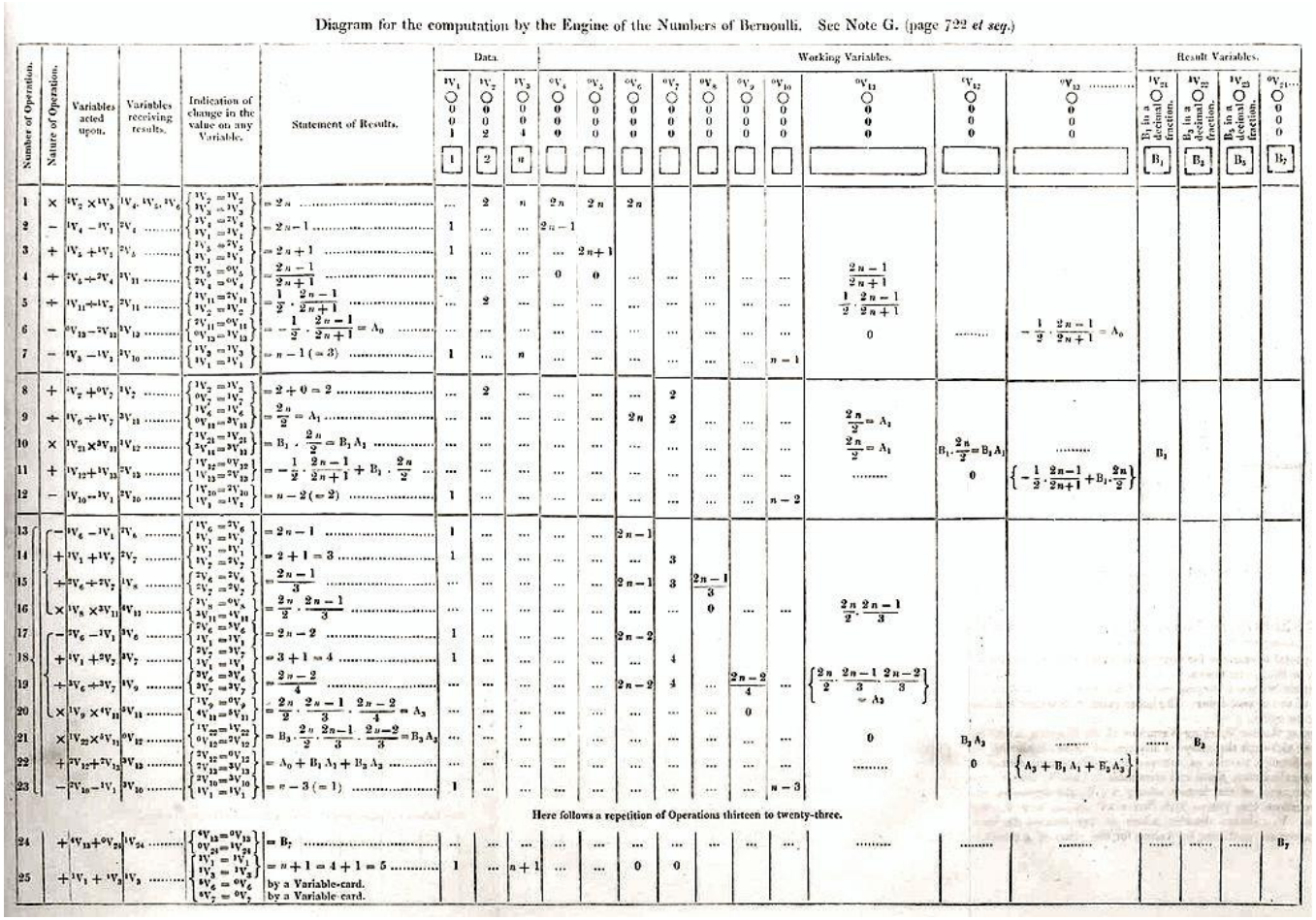


Diagram for the computation by the Engine of the Numbers of Bernoulli. See Note G. (page 722 *et seq.*)

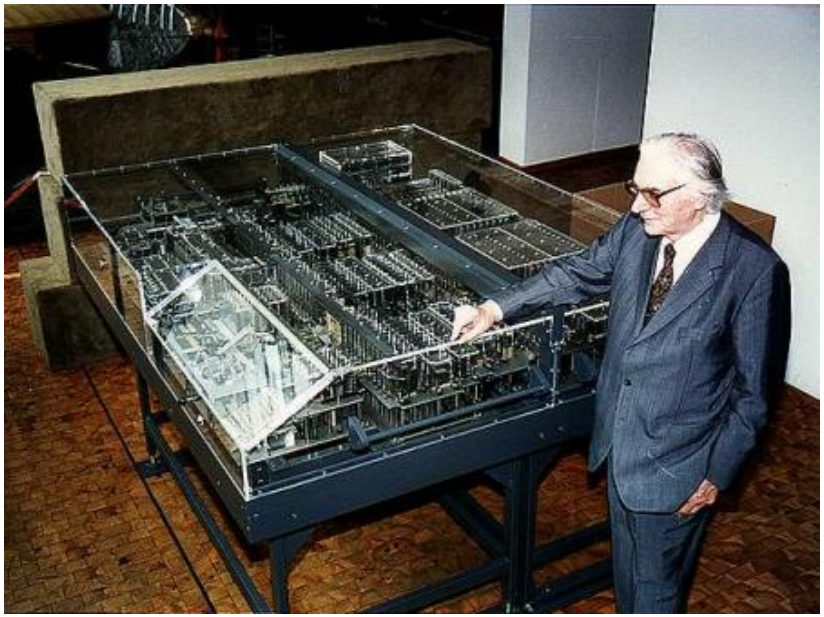
https://upload.wikimedia.org/wikipedia/commons/thumb/c/cf/Diagram_for_the_computation_of_Bernoulli_numbers.jpg/220px-Diagram_for_the_computation_of_Bernoulli_numbers.jpg

Grace Brewster Murray Hopper

http://history-computer.com/ModernComputer/Software/images/Grace_Hopper.jpg

- Americká matematička
- 1906 – 1992
- Vyvinula první kompilátor
- Zavedla slovo „bug“ pro označení počítačové chyby
- Amazing Grace

První počítače



http://i.iinfo.cz/urs/pc_01_09-120398105593836.png

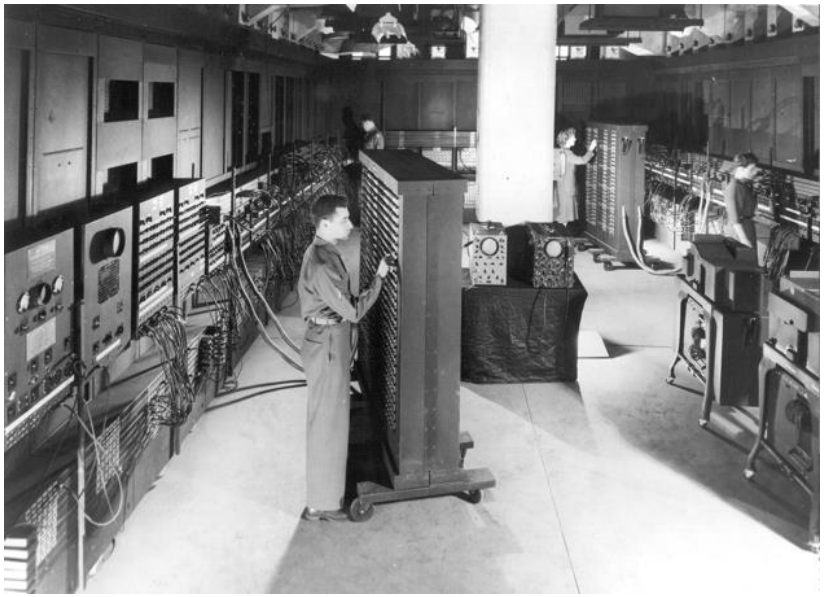
- Z1
- Konrad Zuse
- 1935-1936
- 1000 kg
- 2000 částí
- Mechanicko-elektronický počítač

První počítače (2)

http://histinf.blogs.upv.es/files/2011/12/5-300x149.jpg

- Harvard-IBM Mark I
- 1944
- Howard Aiken
- 4500 kg
- 765 000 částí
- Poháněn elektromotorem o výkonu 4 kW
- Součet dvou čísel za 0,3 s
- Součin dvou čísel za 6 s
- Výpočet funkce sin daného úhlu do 1 minuty

První počítače (3)



https://upload.wikimedia.org/wikipedia/commons/1/16/Classic_shot_of_the_ENIAC.jpg

- ENIAC
- 1946
- Tehdy 500 000 dolarů
- 27 tun, 167 m2
- Spotřeba 150 kW
- Chlazen leteckými motory
- Balistické výpočty

Historie programovacích jazyků

- Konec 40. let
  - Odklon od strojových kódů
  - Pseudokódy:
    - pseudooperace aritmetických a matematických funkcí
    - podmíněné a nepodmíněné skoky
    - auto inkrementální 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 Oriented Language), jednoduché výpočty, velká množství dat, záznamy, soubory, formátování výstupů
  - ALGOL (ALGOrithmic Language - Backus, Naur), předek všech imperativních jazyků, bloková struktura, rekurze, volání parametrů 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 (LISt Processing - McCarthy)
    - založen na teorii rekurzivních funkcí, první funkcionální jazyk, použití v UI (symbolické manipulace)
  - APL - manipulace s vektory a s maticemi
  - SNOBOL (StriNg Oriented and symBOlic Language - 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 – Vienna Definition Language),
    - uživatelsky málo přívětivý, typ reference, dynamická pole
  - SIMULA67 (Nygaard, Dahl)
    - zavádí pojem tříd, hierarchie ke strukturování dat a procedur, garbage collector, ovlivnila všechny moderní jazyky, corrutiny
  - 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, společně s vývojem Unixu
  - 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, CLISP - 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 apod.

- **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

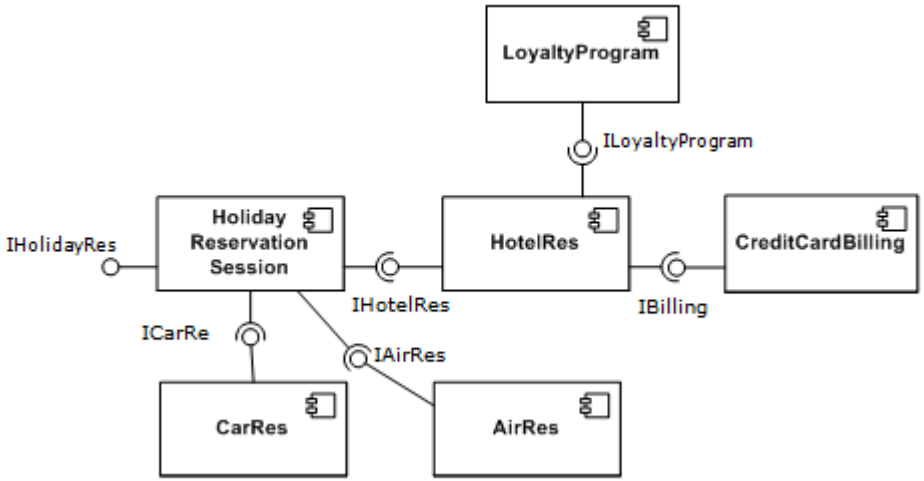
Generické programování

- 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 komunikovat s ostatními komponentami
  - jednotka nezávisle nasaditelná a verzovatelná



https://upload.wikimedia.org/wikipedia/commons/8/83/Component-based-Software-Engineering-example2.png

- 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

```
  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 and M 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
```

```
  9567
+ 1085
------
 10652
```

Logické programování

- Použití matematické logiky v programování
- 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 (PROgramming in LOGic)

```
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]
```

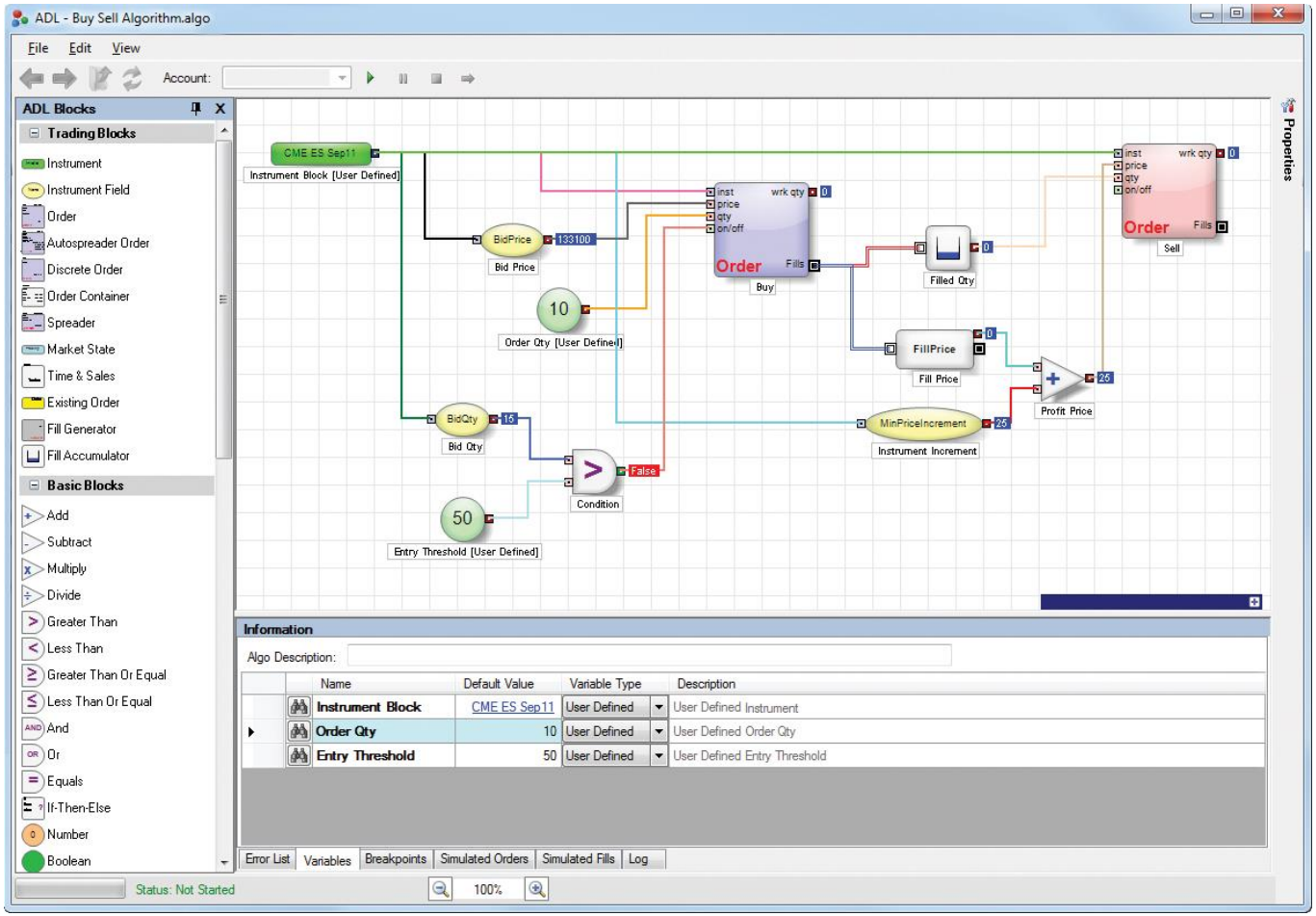
Př. event-driven verze téhož

```
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 Visual Studio (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

Vizuální programování (2)



http://blog.interfacevision.com/assets/img/posts/example_visual_language_adl.jpg

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
  - obvykle 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
  - Sdílení 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>
```

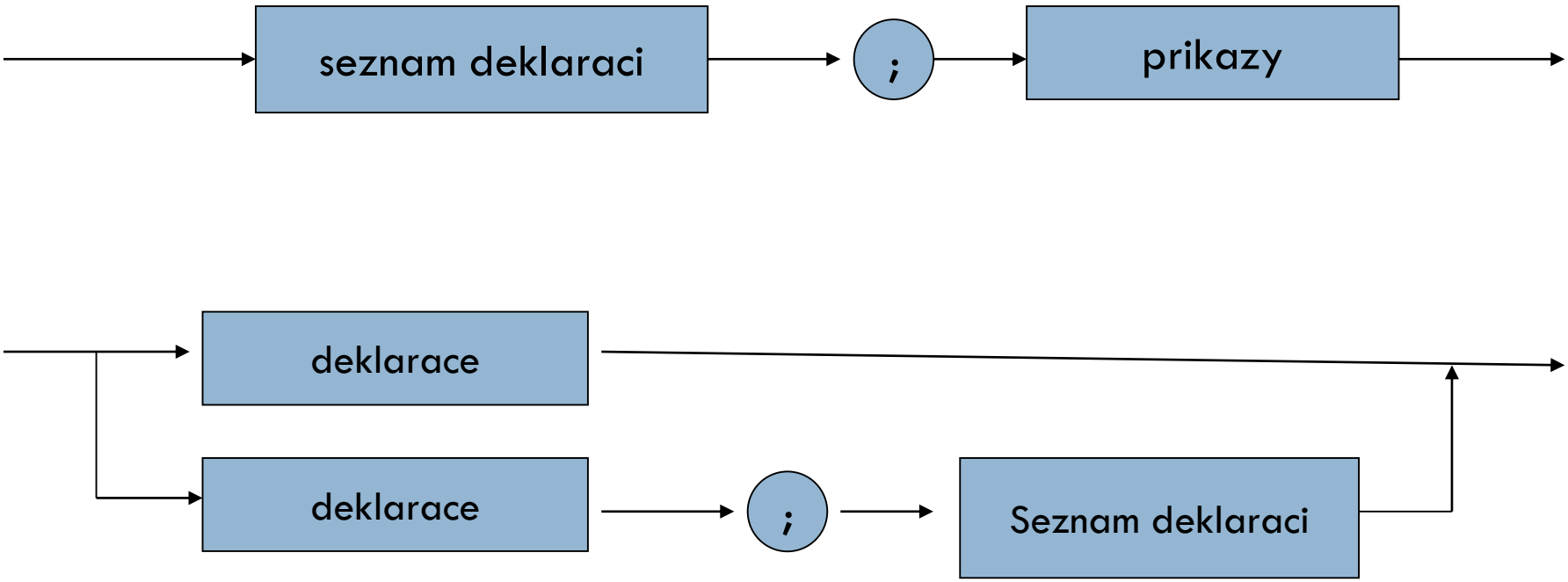
Syntax (3)

- 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

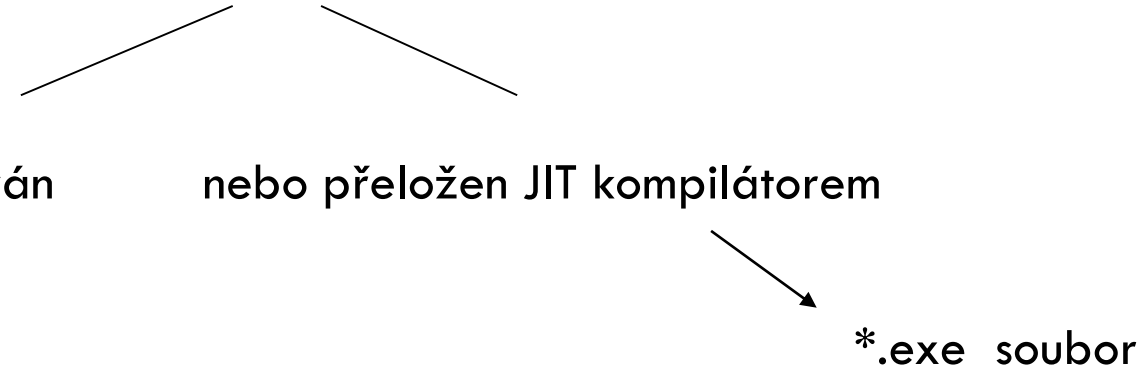
- Slovní popis nepřesný
- Formální popis:
  - Operační sémantiky
    - vyjádření významu programu posloupností přechodů mezi stavy
    - příkaz aktualizace paměti: Pokud se výraz E ve stavu s redukuje na hodnotu V potom program L:=E zaktualizuje stav s přiřazením L=V

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

  - 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 je 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