- **CHARAKTERIZUJTE PROCEDURÁLNÍ PROGRAMOVÁNÍ**

*- Charakterizace dle **paradigmat** (tedy dle způsobů formulace problémů, prostředků jejich zpracování, řešení, atd.)*
- Procedurální = Imperativní. Členění kódu do procedur – podprogramů, jež vykonávají určitou dílčí činnost.
Java, Ada95, C++

- **„ OO „**

- k popisu používá **objekty** s určitými vlastnostmi a činnostmi, které se nad těmito vlastnosmi dají vykonávat. mezi objekty se definují různé vztahy → 3 zákl. kameny OOP – **zapouzdřenost** (encapsulation), **dědičnost** (inheritance) a **polymorfismus** (polymorfism)

- **„ FUNKCIONÁLNÍ „**

- založeno na **matemat. fcích**:
- program definuje funkci, jejž výsledkem je fční hodnota
- model = **lambda kalkuk** (Lisp)
- základní konstrukce = **výraz**

- **„ DEKLARATIVNÍ „**

- předek **constraint progr**. a **aplikativního programování** (a to je předkem funkcionálního a logického), založeno na **deklarování** (tedy ne algoritmizace)

- **„ APLIKATIVNÍ „**

- deklarativní, člení se na funkcionální a logické

- **„ LOGICKÉ „**

- založeno na základě **deklarace faktů** (predikátů) a **relací** (vztahů), často rekurzivně definovaných, využívající logických formulací k vyjádření vzájemných relací. (např. Prolog)

- **„ SOUBĚŽNÉ „**

- rozdělejuje se na **paralelní** a **distribuované**

- **„ VIZUÁLNÍ „**

- nejčastěji programování ve Windows, kde základem je **vizuální návrh** a **řešení událostním řízení**m pro většinu logiky aplikace (např. Visual Editor pro JAVU na Eclipse)

- **JAKÁ JSOU GLOBÁLNÍ KRITERIA NA PROGRAMOVACÍ JAZYK**

1) **spolehlivost** (typová kontrola, zpracování výjimečných situací
2) **efektivita** (překladu a výpočtu)
3) **strojová nezávislost**
4) **čitelnost a vyjadřovací schopnosti** (jednoduchost, ortogonalita)
5) **řádně definovaná syntax a sémantika** (strojová a humánní čitelnost (× zapisov.))
6) **úplnost v Turingově smyslu** (tj. schopnost popsat lib. algoritmus)

- **JAKÁ HLEDISKA OVLIVŇUJÍ SPOLEHLIVOST PROGRAMOVACÍHO JAZYKA**

- Existence resp. způsob implementace **typové kontroly** a zpracování **výjimečných situací**

- **„ „ EFEKTIVITU „**

- Efektivita **překladu** a **výpočtu** programu

- **„ „ ČITELNOST A VYJADŘOVACÍ SCHOPNOSTI**

1) **jednoduchost** (kontra příkad v Céčku: C++, C+=1, ++C, C=C+1)
2) **ortogonalita** (Existuje malá množina primitivních konstrukcí, z níž lze vytvářet další konstrukce → všechny kombinace jsou legální (kontra příklad v Céčku – struktury mohou být funkční hodnotou, pole ne))
3) **strukturované příkazy**
4) **strukturované datové typy**
5) **podpora abstrakčních prostředků**

- **DEFINUJTE POJMY SYNTAX, SÉMANTIKA PROGRAMOVACÍHO JAZYKA**

syntax = forma či struktura výrazů, příkazů a programových jednotek
sémantika = význam výrazů, příkazů a programových jednotek

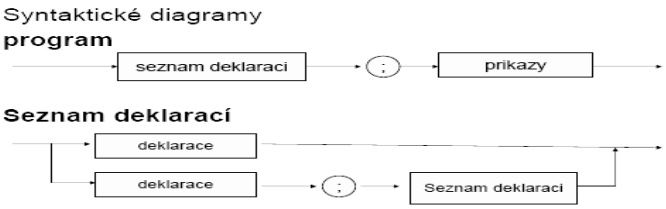
- **DEFINUJTE POJEM ÚPLNOSTI JAZYKA V TURINGOVĚ SMYSLU**

Programový jazyk je úplný v Turingově smyslu, jestliže je **schopný popsat libovolný výpočet** (algoritmus). K tomu stačí celočíselná aritmetika, celočíselné proměnné spolu se sekvenčně prováděnými přikazy zahrnující přiřazení a cyklus (while)

- **ZAPIŠTE V BNF TVAR PŘÍKAZU … (NAPŘ. IF)**

<if> → <podmínka>;<tělo> |
<podmínka>;<tělo>;<else-větev>
(BNF = Backus Naurova forma, bezkontextová gramatika)

- **ZAPIŠTE SYNTAKTICKÝM DIAGRAMEM TVAR PŘÍKAZU …**



- **OBJASNĚTE POJMY STATICKÁ A DYNAMICKÁ SÉMANTIKA**

Statická sémantika má neměnný význam výrazů, příkazů a programových jednotek (definice proměnných, typování),
dynamická ano a podle toho jak může být význam změněn, rozlišujeme **3 dynamické sémantiky**: operační, axiomatické a denotační

- **JAKÉ DRUHY CHYB V PROGRAMU ROZLIŠUJEME**

1) **Lexikální** (např. nedovolený znak)
2) **Syntaktické** (např. chyba ve struktuře)
3) **Statické sémantiky** (např. nedefinovaná proměnná)
4) **Dynamické sémantiky** (např. dělení 0)
5) **Logické** (chyba v algoritmu)

- **JAKÉ DRUHY CHYB A VE KTERÉ FÁZI JE SCHOPEN NALÉZT PŘEKLADAČ**

ve **fázy překladu** najde chyby lexikální a syntaktické
a ve **fázy před výpočtem** může najít chyby statické sémantiky

- **DEFINUJTE SLOVNĚ TERMY JAZYKA PROLOG**

- termy jsou **konstanty**, **proměnné** a **struktury** (tj. elementy programu)

- **CO JSOU ATOMY PROLOGU**

= nečíselné konstanty, např. `x`, `my_vsichni`, `"USMEVAVY"`, `fav06`

- **OBJASNĚTE POJEM ANONYMNÍ PROMĚNNÁ PROLOGU A JEJÍ VLASTNOSTI**

to je taková proměnná, jejíž **hodnotu nechceme znát**. Zapisujeme jí pomocí znaku podtržítka "_". Její **vlastnosti** jsou:
1) v klauzulu jich může být více, **2)** navzájem nesouvisí, **3)** zpřehledňuje programy

- **POPIŠTE PRINCIP REZOLUCE PROLOGU**

dojceme k výsledku (k cíly) při **rozvinutí podcílů** (viz způsob plnění cílů), např.:
a :- a1, a2,…, an
b :- b1, b2,…,bn
nechť bi=a, pak rezolucí je: b :- b1, b2,… bi-1, a1, a2,…, an, bi+1,…, bm

- **POPIŠTE PRINCIP UNIFIKACE PROLOGU**

- porovnává-li se **volná proměnná** s **konstantou**, naváže se na tuto konstantu,
- porovnávají-li se **dvě volné** (neinstalované) **proměnné**, stanou se synonymy,
- porovnává-li se **volná proměnná s termem**, naváže se na tento term
- porovnávají-li se **termy, které nejsou voln. prom.**, pro úsp. porovn. musí být stejné

- **POPIŠTE ZPŮSOB PLNĚNÍ CÍLŮ V PROLOGU**

- 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.

- **JAK PROBÍHÁ NÁVRAT PŘI NESPLNĚNÍ CÍLE V PROLOGU**

- aktivuje se **mechanismus návratu**, který vypadá takto:

Exekuce se vrací k předchozímu splněnému cíli, zruší se instalace proměnných a pokouší se opětovně splnit tento 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 opětovně zpět.

- **VYSVĚTLETE MECHANISMUS PŮSOBENÍ PREDIKÁTU ŘEZU**

- Predikát řezu **odřízne další provádní cílu** z hlavy pravidla
- Změní mechanismus návratu tím, že **znepřístupní ukazatele vlevo do něj ležících cílů** (přesune je na konec databáze)

- **ČÍM SE ODLIŠUJE ČERVENÝ A ZELENÝ ŘEZ PROLOGU**

- **Zelený řez** nemění deklarativní smysl (znemožní návrat, který by stejně skončil neúspěchem)
- **Červený řez** mění deklar. smysl (znemožní návrta, který by našel jiné řešení)

- **JAKÝ ÚČINEK MÁ PREDIKÁT REPEAT**

- nekonečněkrát splnitelný cíl

- **„ FAIL**

- vždy nesplněný cíl → pokud je podcílem, způsobí hledání jiné alternativy

- **„ ASSERTA**

- přidá klauzule na začátek databáze

- **PŘÍKLAD – POROVNÁVÁNÍ TERMŮ**
- **PŘÍKLAD – PRÁCE SE SEZNAMY**
- **CHARAKTERIZUJTE „ČISTÉ VÝRAZY“**

- čisté výrazy **nemění** stavový prostor programu

- **POPIŠTE CHURCH–ROSEROVU VLASTNOST VÝRAZŮ**

- Hodnota výsledku **nezávisí** na pořadí vyhodnocování

- **DEFINUJTE S–VÝRAZY LISPU**

1) **atomy** (čísla, znaky, řetězce, symboly T NIL,…) k označení proměnných a fcí
2) **seznamy** (el1 el2 … eln), NIL

- **POPIŠTE ZÁKLADNÍ CYKLUS VYHODNOCOVÁNÍ LIPOVSKÉHO PROGRAMU**

1) výpis promptu, **2)** uživatel zadá lispovský výraz, **3)** provede se vyhodnocení argumentů, **4)** aplikuje funkci na vyhodnocené argumenty, **5)** vypíše se výsledek (fční hodnota)

- **JAKÉ JSOU ELEMENTÁRNÍ FUNKCE LISPU A JEJICH SÉMANTIKA**

- **`car`** `(first)` , **`cdr`** `(rest)` , **`cons`**, **`atom`**, **`equal`**

- **POPIŠTE SÉMANTIKU LISPOVSKÉ FUNKCE COND**

- `COND` (jehož synonimum je `if`) umožňuje vykonání těla v **závislosti** na testovací části; umožňuje výběr alternativy

- **POPIŠTE TVAR A ÚČINEK LISPOVSKÉ FUNKCE DEFUN**

- *`DEFUN` přiřadí jménu-fce lambda výraz definovaný tělem-fce, tj.* `(LAMBDA (argumenty) tělo-fce)`. Vytvoří funkční vazbu symbolu **jméno-funkce**

- **DEFINUJTE TVAR A VYUŽITÍ LAMBDA VÝRAZŮ**

- **využití** – popisují bezejmenné fce
- **definice tvaru** (rekurzivní) – seznam, jehož prvním elementem je lambda výraz reprezentující funkci, jež se zavolá s argumenty, které jsou výsledkem vyhodnocených následných elemtů lambda tvaru

- **CO TO JSOU FUNKCIONÁLY, POPIŠTE NĚKTERÝ (PŘÍKLADY)**

= fce, jejichž argumentem je fce nebo vrací fci jako svoji hodnotu. Vytváří programová schémata, použitelná pro různé aplikace (Higher order functions)
např.:
MAPCAR (aplikuje fci na prvky seznamů, které jsou dalšími argumenty, z výsledků tvoří seznam)
`(MAPCAR (FUNCTION +) '(1 2 3 4) '(1 2 3))`
FIND-IF (nalezne prvý prvek seznamu, vyhovující predikátu)
`(FIND-IF #'SYMBOLP '(3 (a) b 1 v))`

- **JAKÉ KONSTRUKCE PRO IMPLEMENTACI ADT V PROGRAMOVACÍCH JAZYCÍCH ZNÁTE**

- **ADA** – `package` (Package specification + Package body)
- **C++, Java** - `class`

- **JAKÉ JSOU ZÁKLADNÍ VLASTNOSTI ADT**

1) definice typu a operací nad ním je obsažena v jedné syntaktické jednotce
2) reprezentace objektů tohoto typu je ukryta před programovými jednotkami, které jej využívají

- **JAKÉ JSOU VÝHODY POUŽITÍ ADT**

- díky vlastnosti **1**: lepší organizace programu a modifikovatelnost programu, separátní překlad
- díky vlastnosti **2**: spolehlivější (důsledek ukrytí dat), nezávislost uživatele na konkrétní implementaci ADT

- **POPIŠTE ODLIŠNOSTI PACKAGE V JAZYCE ADA A JAVA**

- **package v jazyku Ad**a má obdobnou funkci Javovské třídy (resp. něco jako rozhraní + třída). **Package v Javě** je balík zastřešující třídy, rozhraní, adaptéry apod. poskytující ucelené nástroje, slouží jako namespaces v C++ (neboli 2. úroveň řízení rozsahu platnosti v Javě), v souborovém systému je java package adresářem.

- **POROVNEJTE VLASTNOSTI KONSTRUKTORŮ JAZYKŮ C++ A JAVA**

stejné: inicializace členských dat, mohou mít parametry, implicitní volání při vytváření instance (explicitní je možné taktéž)
odlišné: v C++ může pouze alokovat paměť (nevytváří objekt)

- **POPIŠTE MECHANISMUS NAMESPACE V C++**

…explicitní určení rozsahu platnosti jmem pro vyloučení kolizí ze separátně překládananých knihoven v hlavičkovém souboru je specifikován prostor zápisem

```
namespace JmenoProstoru.
```

… klient pak použije klauzuli:

```
using namespace JmenoProstoru
```

- **POPIŠTE MOŽNOST PŘÍSTUPU K METODÁM A PROMĚNNÝM JAVY V ZÁVISLOSTI NA MODIFIKÁTORU PŘÍSTUPU**

- **public** – přístup odkudkoliv
- **implicitní** (= neuveden, friendly) – přístup v tomtéž balíku, v téže třídě a v potomcích
- **protected** – jako implicitní, navíc je přístupný i v podtřídách různých balíků
- **private** – přístupný pouze v téže třídě

- **VYSVĚTLETE POJEM PARAMETRICKÝ POLYMORFISMUS A UVEĎTE JEHO JAZYKOVÉ KONSTRUKCE**

= násobné využití operace nad různými typy (vytvoření šablony)
ADA – **generické moduly** - uvedení klíčového slovo `generic` a následně generických parametrů, které mohou být: numerické, disktrétní, privátní nebo obecné typy
C++ – **generické třídy** (popisují obecný algoritmus) - při vytváření jejich objektu bude určen konkrétní typ jako parametr; obecný formát konstrukce:

```
template <class Ttyp1, Ttyp2,…> class jmeno-tridy {…}
```

JAVA – **generické typy** (od 1.5 omezené parametrizování)

- **POPIŠTE ZÁKLADNÍ RYSY GENERICKÝCH TŘÍD C++ A GENERICKÝCH MODULŮ ADY**

- *viz předchozí*
- **ADA** – určení generic. ve specicikační části package
- **C++** - popisují obecný algoritmus, konkrétní typ dat bude určen jako parametr při vytváření objektu, instaluje generické třídy v době překladu

- **JAKÉ TYPY MOHOU (NEMOHOU) BÝT GENERICKÝM PARAMETREM JAVY**

- Mohou to být pouze **třídy**, tj. např. kontejnery jako ArrayList a LinkedList.
- Např. rozhraní to být nemůže.

- **UVEĎTE ROZDÍL MEZI PARAMETRIZOVANÝMI METODAMI, PŘETÍŽENÝMI METODAMI A METODAMI S PARAMETRY JAZYKA JAVA.**

- **Parametrizované metody** = generické metody. Mají typové parametry, které informují překladač o skutečných parametrecha a návratové hodnotě při volání metody
- **přetížené metody** = metody se stejným jménem, ale různými parametry (i různé typy parametrů)
- **metody s parametry** – obecně metody, jenž mají nějaký vstup realizovaný pomocí parametrů (metody bez parametrů, zvláště pak statické, by se mohli brát jako procedury). Mohou být jak přetížené, tak parametrizované.