# Zadání projektu z předmětu IPP 2024/2025

Zbyněk Křivka, Ondřej Ondryáš, Radim Kocman
E-mail: krivka@fit.vut.cz

## 1 Základní charakteristika projektu

Navrhněte, implementujte a dokumentujte dva programy pro analýzu a interpretaci jednoduchého třídního čistě objektově orientovaného imperativního jazyka SOL25. K implementaci vytvořte odpovídající stručnou programovou dokumentaci. Projekt se skládá ze dvou úloh a je individuální.

První úloha se skládá z programu v jazyce Python 3.11 (viz sekce 4) spouštěného pomocí skriptu `parse.py` a dokumentace k tomuto programu (viz sekce 2.1). Druhá úloha se skládá z programu v jazyce PHP 8.4 (viz sekce 5) integrovaného do poskytnutého rámce (spouštěného pomocí `interpret.php`) a dokumentace k vaší implementaci (viz sekce 2.1 a upřesnění v sekci 5).

## 2 Požadavky a organizační informace

Kromě implementace obou programů a vytvoření dokumentace je třeba dodržet také řadu následujících formálních požadavků. Pokud některý nebude dodržen, může být projekt hodnocen nula body! Pro kontrolu alespoň některých formálních požadavků lze využít skript `is_it_ok.sh` dostupný v *Moodle* předmětu IPP.

**Termíny:**

připomínky[1] k zadání projektu do 17. února 2025;

fixace zadání projektu od 18. února 2025;

odevzdání první úlohy v úterý **18. března** 2025 do 23:59:59;

odevzdání druhé úlohy v úterý **23. dubna** 2025 do 23:59:59.

**Dodatečné informace a konzultace k projektu z IPP:**

- *E-Learning* (Moodle) předmětu IPP včetně Často kladených otázek (*FAQ*) a Souborů k projektu.
- *Fórum* v *E-Learning* (Moodle) IPP pro ak. rok 2024/2025, témata `Projekt.*`.
- Zbyněk Křivka (garant projektu): dle konzultačních slotů (viz webová stránka) nebo po dohodě e-mailem (uvádějte předmět začínající `"IPP:"`), viz `http://www.fit.vut.cz/person/krivka`. Další cvičící najdete na kartě předmětu.
- Dušan Kolář (garant předmětu; jen v závažných případech): po dohodě e-mailem (uvádějte předmět začínající `"IPP:"`), viz `http://www.fit.vut.cz/person/kolar`.

[1] Naleznete-li v zadání nějakou chybu či nejasnost, dejte prosím vědět na *Fóru* předmětu nebo e-mailem na krivka@fit.vut.cz. Validní připomínky a upozornění na chyby budou oceněny i bonusovými body.

Pokud máte jakékoliv dotazy, problémy či nejasnosti ohledně tohoto projektu, tak po přečtení *FAQ* a *Fóra* (využívejte i možnosti hledání na *Fóru*), neváhejte napsat na *Fórum* (u obecného problému, jenž se potenciálně týká i vašich kolegů) či kontaktovat garanta projektu, cvičícího (v případě individuálního problému) nebo nouzově garanta předmětu, kdy do předmětu vždy uveďte na začátek řetězec `"IPP:"`. Na problémy zjištěné v řádu hodin až jednotek dní před termínem odevzdání některé části projektu nebude brán zřetel. Začněte proto projekt řešit s dostatečným předstihem.

**Forma a způsob odevzdání:** Každá úloha se odevzdává individuálně prostřednictvím StudIS do předmětu IPP (odevzdání e-mailem je bodově postihováno) a odevzdáním stvrzujete výhradní autorství skriptů i dokumentace[2].

V aktivitě „Projekt - 1. úloha v Pythonu 3.11“ odevzdáte archiv pro první úlohu (skript `parse.py`; včetně dokumentace tohoto skriptu). Po termínu odevzdání 1. úlohy bude otevřeno odevzdání archivu v aktivitě „Projekt - 2. úloha v PHP 8.4“ pro druhou úlohu (obsah adresáře `student` v rámci *ipp-core*; včetně dokumentace).

Každá úloha bude odevzdána ve zvláštním archivu, kde budou soubory k dané úloze zkomprimovány programem ZIP, TAR+GZIP či TAR+BZIP do jediného archivu pojmenovaného `xloginj99.zip`, `xloginj99.tgz`, nebo `xloginj99.tbz`, kde `xloginj99` je váš login. Velikost každého archivu bude omezena informačním systémem (pravděpodobně na 2 MB). Archiv nesmí obsahovat speciální či binární spustitelné soubory, ani námi poskytnutý rámec *ipp-core*. Archiv bude případně obsahovat soubor `rozsireni` (viz níže). Názvy všech souborů a adresářů mohou obsahovat pouze písmena anglické abecedy, číslice, tečku, pomlčku a podtržítko (nepřibalujte skryté adresáře, typicky začínající tečkou). **Skripty úlohy (resp. u 2. úlohy obsah složky `student`) budou umístěny v kořenovém adresáři odevzdaného archivu**. Po rozbalení archivu na serveru *Merlin* (v případě 2. úlohy rozbalíme archiv do adresáře ***`student`*** našeho rámce *ipp-core*) bude možné skript(y) spustit. Archiv smí obsahovat v rozumné míře adresáře (typicky pro vaše moduly/jmenné prostory, vlastní testy a pomocné skripty nebo povolené knihovny[3], které nejsou nainstalovány na serveru *Merlin*).

**Hodnocení:** Výše základního bodového hodnocení projektu v předmětu IPP je **maximálně 20 bodů**. Navíc lze získat maximálně 5 bonusových bodů za kvalitní a podařené řešení některého z rozšíření nebo kvalitativně nadprůměrnou účast na *Fóru* projektu apod.

Hodnocení jednotlivých skriptů: `parse.py` až 7 bodů (z toho až 1 bod za dokumentaci a úpravu zdrojových kódů skriptu `parse.py`); `interpret.php` až 13 bodů (z toho návrh, dokumentace a úprava zdrojových textů skriptů až **4 body**). Dokumentace je však hodnocena maximálně 50 % ze sumy hodnocení funkčnosti dané úlohy (tedy v případě neodevzdání žádného funkčního skriptu v dané úloze bude samotná dokumentace hodnocena 0 body). Body za každou úlohu včetně její dokumentace se před vložením do StudIS zaokrouhlují na desetiny bodu.

Skripty budou spouštěny na serveru *Merlin* příkazem: *interpret skript parametry*, kde *interpret* bude `python3.11` nebo `php8.4`, *skript* a *parametry* závisí na dané úloze. Hodnocení většiny funkčnosti bude zajišťovat automatizovaný nástroj. Kvalitu dokumentace, komentářů, návrh a strukturu zdrojového kódu budou hodnotit cvičící.

Vaše skripty budou hodnoceny nezávisle na sobě, takže je možné odevzdat například jen 2. úlohu bez odevzdání 1. úlohy.

Podmínky pro opakující studenty ohledně případného uznání hodnocení loňského projektu najdete v *FAQ* na *Moodle* předmětu.

[2]Pozor na nadužívání generativních nástrojů kódu, které mohou doporučovat stejné větší úseky kódu více studentům, které bývají detekovány jako plagiátoství!

[3]Pro 1. úlohu vizte poznámku v sekci 2.2. Pro 2. úlohu budou povolené knihovny k dispozici ve složce `vendor`.

**Registrovaná rozšíření:** V případě implementace některých registrovaných rozšíření za bonusové body bude odevzdaný archiv obsahovat soubor `rozsireni`, ve kterém uvedete na každém řádku identifikátor jednoho implementovaného rozšíření[4] (řádky jsou ukončeny unixovým koncem řádku, tj. znak s dekadickou ASCII hodnotou `10`). V průběhu řešení mohou být zaregistrována nová rozšíření úlohy za bonusové body (viz *Fórum*). Nejpozději do termínu pokusného odevzdání dané úlohy můžete na *Fórum* zasílat návrhy na nová netriviální rozšíření, která byste chtěli navíc implementovat. Cvičící rozhodne o přijetí/nepřijetí rozšíření a hodnocení rozšíření dle jeho náročnosti včetně přiřazení unikátního identifikátoru. Implementovaná rozšíření neidentifikovaná v souboru `rozsireni` nebudou hodnocena.

**Pokusné odevzdání:** Pro zvýšení motivace studentů pro včasné vypracování úloh nabízíme koncept nepovinného pokusného odevzdání. Výměnou za pokusné odevzdání do uvedeného termínu (cca týden před finálním termínem) dostanete zpětnou vazbu v podobě zařazení do některého z pěti rozmezí hodnocení (0–10 %, 11–30 %, 31–50 %, 51–80 %, 81–100 %). Bude-li vaše pokusné odevzdání v prvním rozmezí hodnocení, máte možnost osobně konzultovat důvod, pokud jej neodhalíte sami. U ostatních rozmezí nebudou detailnější informace poskytovány.

Pokusné odevzdání bude relativně rychle vyhodnoceno automatickými testy a studentům zaslána orientační informace o správnosti pokusně odevzdané úlohy z hlediska části automatických testů (tj. nebude se jednat o finální hodnocení; proto nebudou sdělovány ani body či přesnější procentuální hodnocení). Využití pokusného termínu není povinné, ale jeho nevyužití může být negativně vzato v úvahu v případě reklamace hodnocení projektu.

Formální požadavky na pokusné odevzdání jsou totožné s požadavky na finální termín a odevzdání se bude provádět do speciálních aktivit „Projekt - Pokusné odevzdání 1. úlohy“ do **11. března 2025** a „Projekt - Pokusné odevzdání 2. úlohy“ do **16. dubna 2025**. Není nutné zahrnout dokumentaci, která spolu s rozšířeními pokusně vyhodnocena nebude.

## 2.1 Dokumentace

Implementační dokumentace (dále jen dokumentace) musí být stručným a uceleným průvodcem **vašeho způsobu řešení** skriptů 1., resp. 2. úlohy. Bude vytvořena ve formátu **PDF** nebo **Markdown** (viz [3]). Jakékoliv jiné formáty dokumentace než PDF či Markdown[5] (přípona `md`) budou ignorovány, což povede ke ztrátě bodů za dokumentaci. Dokumentaci je možné psát buď česky, slovensky (s diakritikou, formálně čistě), nebo anglicky (formálně čistě).

Dokumentace bude popisovat celkovou filozofii návrhu, interní reprezentaci, způsob a váš specifický postup řešení (např. řešení sporných případů nedostatečně upřesněných zadáním, konkrétní řešení rozšíření, případné využití návrhových vzorů, implementované/nedokončené vlastnosti). Dokumentaci je vhodné doplnit např. o UML diagram tříd (povinný u 2. úlohy), navržený konečný automat, pravidla vámi vytvořené gramatiky nebo popis jiných formalismů a algoritmů. Nicméně **nesmí obsahovat ani částečnou kopii zadání**.

Vysázená dokumentace 1. úlohy popisující skript `parse.py` by neměla přesáhnout 1 stranu A4 (tj. rozsah přibližně 1 až 2 normostrany). U 2. úlohy (popisující skript `interpret.php`) pak nepřesáhněte vysázené 2 strany A4 (nepočítaje vhodně velký UML diagram). Doporučení pro sazbu: 10bodové písmo Times New Roman pro text a Courier pro identifikátory a skutečně krátké úryvky zajímavého kódu (či zpětné apostrofy v Markdown); nevkládejte žádnou zvláštní úvodní stranu, obsah ani závěr. V rozumné míře je vhodné používat nadpisy první a případně druhé úrovně (12bodové a 11bodové písmo Times New Roman či ## a ### v Markdown) pro vytvoření logické struktury dokumentace.

[4] Identifikátory rozšíření jsou uvedeny u konkrétního rozšíření tučně.

[5] Bude-li přítomna dokumentace v Markdown i PDF, bude hodnocena verze v PDF, kde je jistější sazba.

Nadpis a hlavička dokumentace[6] bude na prvních třech řádcích obsahovat:

```
Implementační dokumentace k %cislo%. úloze do IPP 2024/2025
Jméno a příjmení: %jmeno_prijmeni%
Login: %xloginj99%
```

kde `%jmeno_prijmeni%` je vaše jméno a příjmení, `%xloginj99%` váš login a `%cislo%` je číslo dokumentované úlohy.

Dokumentace bude v kořenovém adresáři odevzdaného archivu a pojmenována `readme1.pdf` nebo `readme1.md` pro první úlohu a `readme2.pdf` nebo `readme2.md` pro druhou úlohu.

V rámci dokumentace bude hodnocena i funkční/objektová dekompozice a komentování zdrojových kódů. Není-li to krystalický jasné ze samotného identifikátoru funkce/modulu/třídy/parametru a případné typové anotace/nápovědy, doplňte vhodný komentář o účelu apod.

## 2.2 Programová část

Zadání projektu vyžaduje implementaci dvou skriptů[7], které mají parametry příkazové řádky a je definováno, jakým způsobem manipulují se vstupy a výstupy. Skript nesmí spouštět žádné další procesy či příkazy operačního systému. Veškerá chybová hlášení, varování a ladicí výpisy směřujte pouze na standardní chybový výstup, jinak pravděpodobně nedodržíte zadání kvůli modifikaci definovaných výstupů (ať již do externích souborů, nebo do standardního výstupu). Jestliže proběhne činnost skriptu bez chyb, vrací se návratová hodnota `0` (nula). Jestliže dojde k nějaké chybě, vrací se chybová návratová hodnota větší jak nula. Chyby mají závazné chybové návratové hodnoty:

- 10 - chybějící parametr skriptu (je-li třeba) nebo použití zakázané kombinace parametrů;
- 11 - chyba při otevírání vstupních souborů (např. neexistence, nedostatečné oprávnění);
- 12 - chyba při otevření výstupních souborů pro zápis (např. nedostatečné oprávnění, chyba při zápisu);
- 20–69 - návratové kódy chyb specifických pro jednotlivé skripty;
- 99 - interní chyba (neovlivněná integrací, vstupními soubory či parametry příkazové řádky).

Pokud zadání níže nestanoví jinak, veškeré vstupy a výstupy jsou v kódování UTF-8. Pro účely projektu z IPP musí být na serveru *Merlin* ponecháno implicitní nastavení `locale`[8], tj. `en_US.UTF-8`.

Jména hlavních skriptů jsou dána zadáním. Pomocné skripty nebo knihovny budou mít příponu dle zvyklostí v daném programovacím jazyce (`.py` pro Python 3 a `.php` pro PHP 8). Vyhodnocení skriptů bude prováděno na serveru *Merlin* s aktuálními verzemi interpretů (dne 1. 2. 2025 bylo na tomto serveru nainstalováno `python3.11`[9] verze 3.11.2 a `php8.4`[10] verze 8.4.1).

K řešení lze využít standardně předinstalované knihovny obou jazykových prostředí na serveru *Merlin*. Případné využití jiné knihovny je třeba konzultovat s garantem projektu (především z důvodu, aby se řešení projektu použitím vhodné knihovny nestalo zcela triviálním a případně byla

[6] Anglické znění nadpisu a hlavičky dokumentace najdete v *FAQ* na *Moodle* předmětu

[7] Tyto skripty jsou aplikace příkazové řádky neboli konzolové aplikace.

[8] Správné nastavení prostředí je nezbytné, aby bylo možné používat a správně zpracovávat parametry příkazové řádky v UTF-8. Pro správnou funkčnost je třeba mít na UTF-8 nastaveno i kódování znakové sady konzole (např. v programu PuTTY v kategorii *Window.Translation* nastavíte *Remote character set* na UTF-8). Pro změnu ovlivňující aktuální sezení lze využít unixový příkaz `export LC_ALL=en_US.UTF-8`.

[9] Upozornění: Na serveru *Merlin* je třeba dodržet testování příkazem `python3.11`, protože pouhým `python` nebo `python3` se spouští starší verze!

[10] Upozornění: Na serveru *Merlin* je třeba dodržet testování příkazem `php8.4`, protože pouhým `php` se spouští verze, která má omezen přístup k souborovému systému!

knihovna dostupná v rámci *ipp-core*). Seznam povolených a zakázaných knihoven je udržován aktuální na *Moodle*. Ve skriptech v jazyce PHP jsou některé funkce z bezpečnostních důvodů zakázány (např. `header`, `mail`, `popen`, `curl_exec`, `socket_*`; úplný seznam je u povolených/zakázaných knihoven na *Moodle*).

**Poznámka pro 1. úlohu (Python):** Využíváte-li nějakou z povolených knihoven, vytvořte si pro vývoj a testování vlastní *virtuální prostředí* např. pomocí `python3.11 -m venv myenv`. Po aktivaci prostředí pomocí `source myenv/bin/activate` si knihovnu nainstalujte pomocí `pip3 install`. Prostředí ani knihovnu nepřikládejte do archivu, projekty budeme spouštět ve vlastním prostředí, které bylo vytvořeno tímto způsobem (ze systémové instalace `python3.11`) a kde jsou nainstalovány všechny povolené knihovny uvedené v *Moodle*. Žádné další knihovny nebudou instalovány!

**Doporučení:** Jelikož je velká část hodnocení odvozena automatickými testy, doporučujeme i pro vývoj skriptů využívat vlastní automatické testování. K tomu je potřeba sada testů, které si sestavíte na základě správného pochopení zadání. Lze se inspirovat několika zveřejněnými testy. Kromě návratového kódu je vhodné testovat i správnost výstupu v případě úspěšného dokončení skriptu. Mezi studenty je možné testy i sdílet.

### Parametry příkazové řádky

Každý skript bude pracovat s jedním společným parametrem:

- `--help` vypíše na standardní výstup nápovědu skriptu (nenačítá žádný vstup), kterou lze převzít ze zadání (lze odstranit diakritiku, případně přeložit do angličtiny dle zvoleného jazyka dokumentace), a vrací návratovou hodnotu 0. Tento parametr nelze kombinovat s žádným dalším parametrem, jinak skript ukončete s chybou 10.

Kombinovatelné parametry skriptů jsou odděleny alespoň jedním bílým znakem a mohou být uváděny v libovolném pořadí, pokud nebude řečeno jinak. U skriptů je možné implementovat i vaše vlastní nekolizní parametry (doporučujeme konzultaci na *Fóru* nebo u garanta projektu).

Není-li řečeno jinak, tak dle konvencí unixových systémů lze uvažovat zástupné zkrácené (s jednou pomlčkou) i dlouhé parametry (se dvěma pomlčkami), které lze se zachováním sémantiky zaměňovat (tzv. alias parametry), ale testovány budou vždy dlouhé verze.

Je-li součástí parametru i soubor (např. `--source=`*`file`* nebo `--source="`*`file`*`"`) či cesta, může být tento soubor/cesta zadán/a relativní cestou[11] nebo absolutní cestou; výskyt znaku uvozovek a rovnítka ve *file* neuvažujte. Cesty/jména souborů mohou obsahovat i Unicode znaky v UTF-8.

# 3 Popis jazyka SOL25

SOL25 je interpretovaný, imperativní, netypovaný, třídní objektově orientovaný programovací jazyk s dynamickou typovou kontrolou. Je volně inspirován vlastnostmi a syntaxí jazyka Smalltalk. Podporuje definice tříd s instančními metodami a atributy, jednoduchou třídní dědičnost a polymorfní invokaci metod. Obsahuje několik vestavěných tříd (s třídními i instančními metodami), uživatel může definovat vlastní třídy. Syntaxe jazyka je jednoduchá, aby bylo možné ji snadno analyzovat a převést na dobře definovaný abstraktní syntaktický strom (AST).

[11] Relativní cesta nebude obsahovat zástupný symbol ~ (vlnka).

Všechny hodnoty (tj. výsledky výrazů a hodnoty referencované v proměnných nebo instančních atributech[12]) jsou objekty, tedy instance nějaké třídy. SOL25 využívá **referenční sémantiku**, což znamená, že typem proměnné je třída objektu, který je referencován touto proměnnou.

V jazyce je k dispozici pouze jeden druh příkazu, a to *přiřazení* výsledku výrazu do proměnné. *Výrazem* je literál, proměnná, nebo zaslání zprávy. *Zaslání zprávy* se skládá z operandu (výraz, který se vyhodnotí na objekt, který bude příjemcem zprávy) a *selektoru* zprávy s případnými argumenty této zprávy (opět výrazy). Pomocí zasílání zpráv se realizuje jak invokace metod, tak přístup k instančním atributům objektů, a představuje tak primární způsob řízení toku programu.

## 3.1 Lexikální jednotky

V SOL25 **záleží** na velikosti písmen u identifikátorů i klíčových slov (je *case-sensitive*).

- *Identifikátor* je definován jako neprázdná posloupnost číslic, písmen (malých i velkých) a znaků '`_`' začínající malým písmenem nebo znakem '`_`'. Jazyk SOL25 obsahuje navíc níže uvedená *klíčová slova*, která mají specifický význam[13], a proto se nesmějí vyskytovat jako identifikátory nových proměnných či jako selektory:

  Klíčová slova: `class`, `self`, `super`, `nil`, `true`, `false`

- *Identifikátor třídy* má stejné podmínky jako *identifikátor*, ale začíná velkým písmenem a neobsahuje '`_`'. SOL25 také obsahuje vestavěné třídy (viz sekci 3.3):

  `Object`, `Nil`, `True`, `False`, `Integer`, `String`, `Block`

  Základním a jediným nositelem typu je třída. Pro jednoduchost není třeba pracovat s třídami jako s objekty první kategorie[14].
- *Pravdivostní literál* je vyjádřen klíčovým slovem `true` pro pravdu nebo klíčovým slovem `false` pro nepravdu. Tato klíčová slova se interpretují jako instance tříd `True` a `False` (viz sekce 3.3). Jde přitom o jedináčky (*singleton*), tj. tyto třídy mají v programu jediné instance zastoupené právě klíčovými slovy `true` a `false`. Pokus o vytvoření nové instance vrací tu již existující.
- *Celočíselný literál* je tvořen neprázdnou posloupností číslic a vyjadřuje hodnotu celého čísla v desítkové soustavě s nepovinným unárním znaménkem (+ nebo -). Literál se interpretuje jako instance třídy `Integer` (klasický celočíselný rozsah implementačního jazyka interpretu).
- *Řetězcový literál* je oboustranně ohraničen jednoduchými uvozovkami ('`'`', ASCII hodnota 39) a je reprezentován instancí třídy `String`. Tvoří jej libovolný počet znaků s ASCII hodnotou větší než 31 (kromě 39 a zpětného lomítka) a zapsaných mezi uvozovkami na jednom řádku. Možný je i prázdný řetězec (`''`). Další znaky lze zapisovat pomocí escape sekvence: '`\'`', '`\n`', '`\\`' (význam je standardní). Pokud znaky za zpětným lomítkem neodpovídají žádnému z uvedených vzorů, dojde k chybě 21.
  Délka řetězce není omezena (resp. jen dostupnou velikostí haldy). Například řetězcový literál

  `'Ahoj\n\'Sve"te \\\'`'

[12] Výjimkou jsou *interní instanční atributy*, které si pro instance objektů udržuje interpret, ale přímo z jazyka nejsou dostupné.

[13] V rozšířeních mohou být použita i další klíčová slova, která ale budeme testovat pouze v případě implementace patřičného rozšíření.

[14] To znamená, že třída samotná se v základním zadání nebere jako plnohodnotný objekt, tj. není možné ji např. referencovat z proměnné. Pro účely instanciace se však může vyskytnout na místě adresáta zprávy, kterým je jinak typicky instance třídy.

reprezentuje řetězec

```
Ahoj
'Sve"te \'
```

V řetězcích se mohou vyskytovat vícebajtové znaky kódování Unicode (např. UTF-8).

- *Blokový literál* slouží pro zápis imperativního kódu. Tvoří jej *definice parametrů* a *sekvence příkazů* přiřazení uzavřená v hranatých závorkách. Obě části mohou být prázdné, ale oddělující svislítko je povinné.
  `[` *sekvence deklarací parametrů bloku* `|` *sekvence příkazů přiřazení* `]`
  Detailní popis syntaxe a sémantiky blokového literálu je v sekci 3.2.1.
- *Neplatná hodnota* je reprezentována klíčovým slovem **`nil`** (chová se obdobně jako pravdivostní literály – **`nil`** představuje jedinou instanci jedináčka **`Nil`**).
- *Selektor zprávy* slouží pro identifikaci zprávy při jejím zasílání (proloženo případnými argumenty) a pro definici jména metody reagující na zaslanou zprávu, kterou instance třídy přijímá. Skládá se z jednoho či více identifikátorů dle toho, zda je bezparametrický, nebo s parametry. Vyjma bezparametrického selektoru se bezprostředně za každým identifikátorem píše **dvojtečka**, počet dvojteček selektoru tak udává počet argumentů zprávy. Při *zasílání* zprávy s daným selektorem se za každou dvojtečkou očekává výraz pro odpovídající argument zprávy. Příklady:

```
class Main : Object {
  run  "<- definice metody - bezparametrický selektor run"
    [ |
      "zaslání zprávy 'compute:and:and:' sobě samému - selektor se dvěma arg."
      x := self compute: 3 and: 2 and: 5.
      "zaslání zprávy 'plusOne:' sobě samému - selektor s jedním arg.
       Argumentem je výsledek po zaslání zprávy 'vysl' objektu self."
      x := self plusOne: (self vysl).
      "zaslání zprávy 'asString' objektu x - bezparam. selektor"
      y := x asString.
    ]

  plusOne: "<- definice metody - selektor s jedním parametrem"
    [ :x | r := x plus: 1. ]

  compute:and:and:  "<- definice metody - selektor se třemi parametry"
    [ :x :y :z |
      "zaslání zpr. 'plus:' objektu x - selektor s jedním argumentem"
      a := x plus: y.
      "zaslání zpr. 'vysl:' sobě samému - nastaví instanční atribut 'vysl'"
      _ := self vysl: a.
      "zpráva 'vysl' se zašle sobě, výsledkem je ref. na objekt vysl;
       tomuto objektu se pak zašle zpráva 'greaterThan:' s arg. 0."
      _ := ((self vysl) greaterThan: 0)
         "výsledkem je objekt typu True nebo False, kterému se zašle zpráva
          'ifTrue:ifFalse:', argumenty jsou bezparametrické bloky"
         ifTrue:  [|u := self vysl: 1.]
         ifFalse: [|].
    ]
}
```

Povšimněte si, že identifikátor se může v selektoru opakovat. Selektor vyjadřuje pouze „rozhraní“ zprávy – předané argumenty jsou vázány na parametry bloku podle deklarovaného pořadí, ne podle identifikátoru v selektoru.

- *Blokový komentář* je ohraničen dvojitými uvozovkami (’"’, ASCII hodnota 34). Komentář je pro účely syntaxe považován za jeden bílý znak. Vnořené blokové komentáře ani jednořádkové komentáře nejsou podporovány.

## 3.2 Syntaxe a sémantika jazykových konstrukcí

Program v jazyce SOL25 se skládá ze sekvence definic tříd. Jedna z definovaných tříd musí být hlavní třída `Main` s bezparametrickou metodou `run` (příp. chyba 31). Interpret po spuštění automaticky vytvoří instanci třídy `Main` a zašle jí zprávu (zavolá metodu) `run`.

Před, za i mezi jednotlivými tokeny se může vyskytovat libovolný počet bílých znaků (mezera, tabulátor, komentář a odřádkování)[15], takže jednotlivé konstrukce jazyka SOL25 lze zapisovat na jednom či více řádcích. Za každým příkazem se píše tečka (’.’, ASCII hodnota 46). Prázdný příkaz není dovolen. Na jednom řádku lze zapsat více příkazů.

Struktura definice uživatelských tříd, syntaxe příkazu přiřazení a syntaxe výrazů včetně zaslání zprávy jsou popsány v následujících sekcích. V příloze A je dostupná **LL(1) gramatika** jazyka včetně LL tabulky, kterou můžete (ale nemusíte) při implementaci využít.

### 3.2.1 Blok

Blok je objekt, který obsahuje sekvenci příkazů, kterou je možné vyhodnotit. Literál bloku:

`[` *sekvence deklarací parametrů bloku* `|` *sekvence příkazů přiřazení* `]`

je uzavřen v hranatých závorkách. První část obsahuje seznam parametrů, který může být i prázdný. Deklarace jednoho parametru se skládá z dvojtečky bezprostředně (bez bílých znaků) následované identifikátorem. Příklad:

```
[ | a := 1. ]                  "Blok bez parametrů"
[ :x|a := x. ]                 "Blok s jedním parametrem"
[ :x :y:z | a := x plus: y. ] "Blok se třemi parametry"
```

Následuje povinné svislítko ’|’ oddělující sekvenci příkazů přiřazení. Každý příkaz je ukončen tečkou. Proměnné uvnitř bloku nejsou dopředu deklarovány a jsou vytvářeny implicitně přiřazením objektu příkazem přiřazení. Parametry bloku se chovají jako nemodifikovatelné proměnné definované na začátku bloku.

#### Příkaz přiřazení

*Příkaz přiřazení* je jediným druhem příkazu v jazyce SOL25.

*proměnná* `:=` *výraz* `.`

Všechny jeho části jsou povinné a je zakončen tečkou.

#### Proměnné a rozsah platnosti

Proměnné jazyka SOL25 jsou pouze lokální v blocích (něco jiného jsou atributy objektů, viz sekci 3.2.3). Proměnná je definována prvním přiřazením v daném bloku a není viditelná v lexikálně zanořených blocích (výjimkou je níže popsaná pseudoproměnná **`self`**, resp. **`super`**). Při pokusu o přiřazení do formálního parametru dojde k chybě 34.

[15] V příkladech v tomto zadání jsou mezery a odřádkování občas záměrně užity nekonzistentně, abyste si mohli povšimnout, kde všude bílé znaky nehrají roli.

Při pokusu o *čtení* neinicializované proměnné (např. ve výrazu) dojde k sémantické chybě 32. Vzhledem k omezenému rozsahu platnosti a viditelnosti proměnných je vyžadováno, aby se neplatná použití proměnných kontrolovala v rámci statické analýzy zdrojového kódu (viz kap. 4), zároveň však musí tuto podmínku kontrolovat vždy také interpret.

Blok tvoří tzv. *rozsah platnosti*, ve kterém jsou dostupné uvnitř bloku definované proměnné. Lokální proměnné mají tedy rozsah platnosti od *místa jejich definice* až po konec bloku, kde byly definovány. V podblocích (lexikálně zanořených blocích) nebo v blocích předaných argumentem lokální proměnné **nejsou viditelné**, takže zde můžeme definovat novou stejnojmennou proměnnou (nejde tedy o *shadowing*). V každém bloku jsou navíc k dispozici objekty **`nil`**, **`true`**, **`false`** s globální viditelností.

### Blok jako objekt

Blok je v SOL25 také objektem – instanci vestavěné třídy **`Block`** (viz také sekci 3.3). Je tedy možné mu zasílat zprávy (tj. invokovat jeho instanční metody), referencovat jej z proměnné, předávat jej jako argument ve zprávě. Příklad:

```
[|
  "Blok bez parametrů přiřazený do 'b1'"
  b1 := [ | a := String read. _ := a print. ].
  "Blok s jedním par. přiřazený do 'b2'.
   S arg., který má rozumět zprávě plus: a po provedení by vracel objekt pro x + 1."
  b2 := [ :x | _ := x plus: 1. ].
  "Blok se dvěma par. přiřazený do 'b3'.
   Při vyhodnocení očekává v 'x' libovolný objekt,
   který umí zprávu 'value:' - tedy např. jiný blok."
  b3 := [ :x:y | val := x value: y. ].
  "Provedení bezparam. bloku = poslání zprávy 'value'"
  _ := b1 value.
  "Provedení jednoparam. bloku zasláním zprávy 'value:' s argumentem 'b2'"
  c := b3 value: b2.
]
```

Literál bloku při definici není hned vykonán! Uvedením literálu bloku na nějakém místě programu vznikne instance **`Block`**. Sekvenci příkazů uvnitř bloku je pak možné provést zasláním patřičné zprávy:

- pokud je literálem bloku definovaná instanční metoda ve třídě, provede se obsah bloku zasláním zprávy s příslušným selektorem instanci této třídy (viz sekci 3.2.3);
- pokud podle literálu bloku vznikl objekt, který je referencovaný z nějaké proměnné, provede se obsah bloku zasláním zprávy **`value`** (příp. **`value:`**, **`value:value:`** atd. podle počtu parametrů bloku; zaslání zprávy se špatnou aritou vede na chybu 51) tomuto objektu.

### Výsledek bloku

Výsledkem provedení bloku je hodnota posledního vyhodnoceného výrazu v posledním provedeném příkazu bloku. Pokud žádný výraz vyhodnocen nebyl, výsledkem je **`nil`**.

```
class Main : Object {
  run [|
    a := self foo: 4.  "a = instance 14"
    b := [ :x | _ := 42. ].  "b = instance Block"
    c := b value: 16.  "c = instance 42"
    d := 'ahoj' print. "d = instance 'ahoj' - print vrací self, viz Vestavěné třídy"
  ]
```

```
  foo: [ :x |
    "s proměnnou 'u' se nijak dál nepracuje, ale výsledek zaslání
     zprávy 'plus:' bude vrácen jako výsledek volání metody 'foo'"
    u := x plus: 10.
  ]
}
```

#### Reference na příjemce zprávy `self`

Při invokaci metody je prostředí odpovídajícího bloku doplněno o předinicializovanou nemodifikovatelnou (pseudo)proměnnou **`self`**, která referencuje objekt, který zprávu přijal. Také je dostupné klíčové slovo **`super`**, které odkazuje na stejný objekt jako **`self`**, ale při jeho použití jsou zprávy směrovány do *nadtřídy* vlastní třídy příjemce (tj. aby bylo možné vynutit provedení původní metody v případě její redefinice).

Uvažujme složitější příklad níže, který demonstruje *statický rozsah platnosti* pseudoproměnné **`self`**. Pseudoproměnná **`self`** je uvnitř bloku navázána v době *definice* bloku na instanci, která je kontextovým objektem provádění metody, která tento blok obsahuje, a to i zanořeně. Blok si tuto referenci pro **`self`** pamatuje po zbytek své existence a při provádění bloku je tato reference využita (i když je blok prováděn třeba v kontextu jiného objektu).

```
class Main : Object {
  run [|
    a := A new.
    "instance bloku si zapamatuje, že self odkazuje na tuto instanci Main"
    "blok navíc tuto referenci na konci vrací"
    b := [ :arg | y := self attr: arg. z := self. ].
    "zavoláme metodu 'foo' na instanci A a předáme jí objekt 'b' typu Block"
    c := a foo: b.
    "výsledkem přiřazeným do c je instance třídy Main s instančním atributem
     attr inicializovaným na 1"
  ]
}

class A : Object {
  foo: [ :x |
    "blok předaný v x je vyhodnocen a do instance Main je jím vytvořen
     instanční atribut attr s hodnotou 1"
    u := x value: 1.
  ]
}
```

### 3.2.2 Výrazy a zasílání zpráv

Výrazem je literál, proměnná, nebo zaslání zprávy. Výsledkem výrazu je vždy objekt. Literálem může být číslo, řetězec, identifikátor třídy nebo blok. Výrazem je také proměnná včetně **`true`**, **`false`**, **`nil`**, **`self`**, **`super`**.

Nutnost/možnost užití závorek je formálně specifikovaná gramatikou v příloze A. Ve vaší implementaci se stačí držet pravidla, že každé zaslání zprávy (mimo takové, které na nejvyšší úrovni leží bezprostředně za '**`:=`**') musí být uzavřeno v závorkách. Formálněji řečeno, zaslání zprávy není asociativní a všechny zprávy (bezparametrické i s parametry) mají stejnou prioritu.

V případě klasického zaslání zprávy je na místě příjemce zprávy **výraz**, který se vyhodnotí na objekt; zpráva je pak určena selektorem. Příklad příkazu přiřazení s výrazem zaslání zprávy se dvěma

argumenty:

```
x := targetExpression selector: arg1Expression arg: arg2Expression.
```

Jednotlivými argumenty jsou opět výrazy – při zaslání zprávy se tyto výrazy nejprve vyhodnotí zleva doprava[16] a reference na výsledné objekty se použijí jako argumenty zprávy (tj. předají se do těla invokovaného bloku).

```
class Main : Object {
  run [|
    "výraz zaslání zprávy v argumentu je už nutné uzávorkovat"
    a := self attrib: (Integer from: 10).
    "vnořená zaslání zprávy, kde je výsledek užit jako cíl pro další zprávu"
    "gramatika v příloze by následující přijala i bez závorek kolem '10 asString',
     ale testovat to nebudeme"
    b := [| x := ((self attrib) asString) concatenateWith: (10 asString). ].
  ]
}
```

Vestavěné třídy nabízejí několik vestavěných *třídních metod* v roli konstruktoru (např. **`new`** nebo **`from:`**), které jsou invokovány zasláním *třídní zprávy*, kdy je příjemce vyjádřen literálem pro identifikátor třídy:

```
x := String from: 'Test\n'.
```

Není možné vytvářet vlastní (uživatelské) třídní metody. Pokud třída nerozumí zaslané třídní zprávě, vede to na statickou chybu 32.

### 3.2.3 Třídy, instanční metody a atributy

Definice třídy se skládá z klíčového slova **`class`** následovaného identifikátorem třídy, dvojtečkou a identifikátorem nadtřídy (angl. *superclass*). Uvedení nadtřídy je povinné (nemůže se tak stát, že by předkem třídy nebyl **`Object`**). Tělo třídy je ohraničeno závorkami '**`{ }`**', uvnitř těla se nachází žádná či více instančních metod. Invokací konstruktoru vzniká objekt – instance třídy. Té je možné zasílat zprávy, kterým instance *rozumí*, tj. vlastní třída nebo nadtřída metodu reagující na zprávu definuje. Pokud zaslanou zprávu třída nepodporuje (tj. neexistuje metoda nebo instanční atribut odpovídající selektoru), program končí s chybou 51, že *instance nerozuměla zprávě*.

Všechny identifikátory tříd (vestavěné i uživatelem definované) mají globální viditelnost.

#### Instanční metody

Definice instančních metod jsou v těle třídy ve tvaru:

```
selektor [ "blokový literál s odpovídajícím počtem parametrů" ]
```

Blokový literál uvedený za selektorem reprezentuje tělo metody (nesouhlasí-li *arita* selektoru a bloku, dojde k sémantické chybě 33). Při zaslání zprávy instanci nejprve dojde k vyhledání odpovídající instanční metody (dle souhlasného selektoru a stejné arity) ve vlastní třídě příjemce. V invokované metodě je navázána pseudoproměnná **`self`** na příjemce zprávy a jednotlivým parametrům jsou přiřazeny hodnoty argumentů (výrazy argumentů jsou vyhodnoceny postupně zleva doprava). Jako výsledek metody je navrácen výsledek bloku reprezentujícího tělo invokované metody.

Instanční metody jsou do třídy zděděny z jejích předků. Zděděnou instanční metodu lze v potomkovi redefinovat (angl. *override*). Při přijetí zprávy se metoda hledá nejprve ve vlastní třídě příjemce,

[16] Povšimněte si, že je použita vyhodnocovací strategie *eager evaluation*.

pak postupně v jeho předcích. Pokud instance zasílá zprávu sama sobě pomocí klíčového slova **super**, odpovídající instanční metoda je vyhledávána až v přímé nadtřídě vlastní třídy a pokračuje se případně v dalších nadtřídách.

Přetěžování (angl. *overloading*), tj. použití stejného selektoru pro více různých implementací metod, není v SOL25 podporováno.

### Instanční atributy

Uživatelsky definovaný *instanční atribut* není deklarován ve třídě[17], ale je vytvářen a inicializován programově pomocí **jednoparametrického selektoru**. Identifikátor nového instančního atributu nesmí odpovídat žádné existující instanční metodě. Přístup k hodnotě dříve definovaného instančního atributu provádíme bezparametrickou zprávou se selektorem odpovídajícím identifikátoru atributu.

```
class Main : Object {
  run [|
    "definuje a inicializuje instanční atribut 'value'"
    r := self value: 10.
    "definuje další inst. atribut 'next', incializuje hodnotou atributu 'value'"
    e := self next: (self value).
    "atribut 'value' již existuje, takže pouze modifikuje hodnotu na nil"
    t := self value: nil.
  ]
}
```

Všechny instanční atributy (vyjma interních) jsou veřejné (angl. *public*). Přístup k zatím nedefinovanému instančnímu atributu vede na interpretační chybu 51.

Především z pohledu implementace obsahují objekty také *interní instanční atributy*. Jde např. o skutečnou numerickou hodnotu objektu, který reprezentuje celé číslo, nebo o identifikátor objektu. Tyto atributy objektů jsou užity pouze pro interní účely interpretace, jsou naplněny během instanciace a uživatelsky (z kódu SOL25) nejsou přístupné ani modifikovatelné.

### Konstruktory

Za účelem vytváření instancí tříd existují v jazyce také dvě *třídní* metody, které fungují jako konstruktory. Formálně jsou tyto metody definovány na třídě **Object**, díky dědičnosti jsou pak i ve všech jejích podtřídách. Chování třídních metod není možné uživatelsky měnit, metody nelze redefinovat (angl. *override*).

**new** bezparametrická třídní metoda, která vytvoří novou instanci dle třídy příjemce a případně inicializuje interní instanční atributy implicitními hodnotami (viz sekci 3.3).

**from:** třídní metoda s jedním parametrem *obj*, která vytvoří novou instanci dané třídy a inicializaci interních instančních atributů provede referencí stejných hodnot, které obsahuje objekt *obj*. Objekt *obj* musí být stejné třídy, podtřídy nebo nadtřídy přijímající třídy (jinak chyba 53), aby měla vestavěná metoda **from:** k dispozici potřebný interní instanční atribut a mohla nakopírovat i další instanční atributy, které jsou k dispozici v *obj*.

```
class Factorial : Integer {
  factorial  "použití from: pro podtřídu třídy Integer"
  [| r := (self equalTo: 0) ifTrue: [|r := Factorial from: 1.]
                            ifFalse: [|r := (self multiplyBy:
                                       (Factorial from: (self plus: -1)
```

[17] Podobně jako v jazyce Python 3.

```
                                        )) factorial. ].
  ]
}
class Main : Object {
  run
 [| x := Factorial from: ((String read) asInteger). x := (x factorial) print. ]
}
```

Vestavěná třída **String** pak definuje další třídní metodu (konstruktor) **read**, viz níže.

## 3.3 Vestavěné třídy SOL25

Věstavěné instanční metody, které vyžadují jako argument instanci, která rozumí zprávě **value** či **value:** (např. bezparametrický blok a blok s jedním parametrem), a nedostanou ji, způsobí chybu 51.

### Třída `Object`

Instanční metody:

**identicalTo:** Testuje shodu dvou objektů, tj. že se jedná o *tentýž* objekt.

**equalTo:** Datově porovná objekt: pokud objekt nemá interní atributy, invokuje **identicalTo:**, jinak porovnává interní atributy (v potomcích je možné **equalTo:** redefinovat tak, aby vhodně porovnávala i instanční atributy).

**asString** Vrací řetězec `''` (v potomcích redefinováno rozumnější implementací).

**isNumber, isString, isBlock, isNil** Vrací **false**. V potomcích jsou redefinovány tak, že vrací **true**, pokud je příjemce instance **Integer** / **String** / **Block** / **Nil** (nebo jejich podtřídy).

### Třída `Nil : Object`

Třída **Nil** je implementována jako jedináček. Konstruktory **new** i **from:** vždy vrací stejnou instanci **Nil**, která je totožná s globálně dostupným objektem **nil**[18].

Instanční metody:

**asString** Vrací řetězec `'nil'`.

### Třída `Integer : Object`

Instance této třídy zastupují celá čísla. Odpovídající celé číslo je uloženo v interním instančním atributu. Výchozí hodnota (při použití **Integer new**) je 0.

Použitím celočíselného literálu vzniká instance třídy **Integer**, přičemž není definováno, zda jde pro konkrétní literál vždy o stejnou instanci. Je tedy možné (ale ne nutné), aby byla identita nějaké instance **Integer** nebo její podtřídy vztažená právě k internímu atributu s hodnotou – konstruktor **from:** může vrátit existující instanci. Příklad:

[18]Není tedy možné získat dvě instance **Nil**, pro které by **identicalTo:** vrátila **false**.

```
class MyInt : Integer { }
class Main : Object {
  run [|
    x := 1. y := 1. z := Integer from: 1.
    u := MyInt from: 1. w := MyInt from: 1.

    a := x equalTo: y. a := x equalTo: z.  "obojí true"
    a := x identicalTo: y.  "podle implementace"
    a := x identicalTo: z.  "podle implementace"

    a := u equalTo: x. a := u equalTo: w.  "obojí true"
    a := u identicalTo: x.  "false"
    a := u identicalTo: w.  "podle implementace"
  ]
}
```

Instanční metody:

**`equalTo:`** Porovná, zda je číselná hodnota (v interním instančním atributu) příjemce a argumentu shodná.

**`greaterThan:`**, **`plus:`**, **`minus:`**, **`multiplyBy:`** Standardní numerické operace.

**`divBy:`** Celočíselné dělení kompatibilní s implementačním jazykem interpretu[19]. Dělení nulou vede na chybu 53.

**`asString`** Vrací číslo převedené na řetězec v desítkové reprezentaci a s případným znaménkem mínus (kladné znaménko se vynechává).

**`asInteger`** Vrací sebe sama.

**`timesRepeat:`** Jako argument očekává instanci, která rozumí zprávě **`value:`**[20]. Pokud (a jen tehdy, když) je příjemce $n > 0$, blok z argumentu se provede $n$-krát. Bloku resp. argumentu se předá jako argument číslo iterace (od 1 do $n$ včetně).

### Třída `String : Object`

Instance této třídy reprezentují řetězce. Řetězec je uložen v interním instančním atributu. Výchozí hodnota (při použití **`String new`**) je prázdný řetězec `''`. Použitím řetězcového literálu vzniká instance třídy **`String`**, přičemž není definováno, zda jde pro konkrétní literál vždy o stejnou instanci.

Třídní metody:

**`read`** Načte řetězec z jednoho řádku vstupu (tj. po odřádkování včetně, které ale není součástí načteného řetězce) a vytvoří odpovídající instanci **`String`**.

Instanční metody:

**`print`** Vytiskne řetězec na výstup (bez jakýchkoliv formátovacích znaků), vrací **`self`**.

**`equalTo:`** Porovná, zda je řetězec (v interním inst. atributu) příjemce a argumentu shodný (odpovídá běžnému porovnání shody řetězců v implementačním jazyce interpretu).

[19] `https://www.php.net/manual/en/function.intdiv.php`
[20] Např. blok s jedním parametrem.

**asString** Vrací sebe sama.

**asInteger** Pokud lze příjemce jednoduše převést na celé číslo, vrací instanci **Integer**, jinak vrací **nil**.

**concatenateWith:** Vrací novou instanci **String**, která obsahuje konkatenaci příjemce a argumentu (argument musí být instance třídy **String** nebo její podtřídy, jinak vrací **nil**).

**startsWith:endsBefore:** Vrací podřetězec od indexu daného prvním argumentem (indexuje se **od 1**) po předchozí znak daným druhým argumentem. Je-li rozdíl argumentů menší či roven 0, vrací prázdný řetězec. Nejsou-li argumenty kladná nenulová celá čísla, vrací **nil**.

### Třída `Block : Object`

Instance této třídy reprezentují bloky kódu. Pokud instance vzniká podle blokového literálu, interpret zajistí, že tato instance obslouží zprávu s identifikátorem **value**, která očekává příslušný počet parametrů. Bezparametrický blok tak bude obsahovat metodu **value**, zatímco např. blok se dvěma parametry bude obsahovat metodu **value:value:**.

Další instanční metody:

**whileTrue:** Jako argument přijímá instanci, která rozumí zprávě **value** (např. bezparametrický blok), který se opakovaně provádí, dokud je příjemce vyhodnocen jako **true**.

```
"tělo nějaké metody v rámci nějaké třídy, kde se vytvoří atribut 'attr'"
x := self attr: 3.
y := [| ret := (self attr) greaterThan: 0. ] whileTrue:
       [| r := ((self attr) asString) print.
          r := self attr: ((self attr) minus: 1).].
```

### Třídy `True : Object` a `False : Object`

Třídy **True** a **False** reprezentují logické hodnoty pravdy a nepravdy. Každá z těchto tříd je implementována jako jedináček přístupný přes globálně viditelný objekt **true** a **false** (obdobně jako **Nil**).

Instanční metody:

**not** Vrací negaci logické hodnoty příjemce.

**and:** Je-li příjemce **false**, vrací **false**. Je-li příjemce **true**, vyhodnotí se argument zasláním zprávy **value** (tj. např. bezparametrický blok).

**or:** Je-li příjemce **true**, vrací **true**. Je-li příjemce **false**, vyhodnotí se argument zasláním zprávy **value** (argument je např. bezparametrický blok).

**ifTrue:ifFalse:** Vyhodnotí první argument zasláním zprávy **value**, pokud je příjemce **true**, jinak vyhodnotí druhý argument zasláním zprávy **value**.

Těmito metodami je tedy podporováno zkrácené vyhodnocování pravdivostních výrazů. Dokud argument není vyhodnocen zasláním zprávy, tak nemusí vadit, že dané zprávě nerozumí.

# 4 Analyzátor kódu v SOL25 (`parse.py`)

Skript typu filtr (`parse.py` v jazyce Python 3.11) načte ze standardního vstupu zdrojový kód v SOL25 (viz sekce 3), zkontroluje lexikální, syntaktickou a statickou sémantickou správnost kódu a vypíše na standardní výstup XML reprezentaci abstraktního syntaktického stromu programu dle specifikace v sekci 4.1.

**Tento skript bude pracovat s těmito parametry:**

- `--help` viz společný parametr všech skriptů v sekci 2.2.

**Chybové návratové kódy specifické pro analyzátor:**

- 21 - lexikální chyba ve zdrojovém kódu v SOL25;
- 22 - syntaktická chyba ve zdrojovém kódu v SOL25;
- 31 - sémantická chyba - chybějící třída `Main` či její instanční metoda `run`.
- 32 - sémantická chyba - použití nedefinované (a tedy i neinicializované) proměnné, formálního parametru, třídy, nebo třídní metody.
- 33 - sémantická chyba arity (špatná arita bloku přiřazeného k selektoru při definici instanční metody)
- 34 - sémantická chyba - kolizní proměnná (lokální proměnná koliduje s formálním parametrem bloku);

## 4.1 Popis výstupního XML formátu

Za povinnou XML hlavičkou[21] následuje kořenový element `program` (s povinným textovým atributem `language` s hodnotou SOL25 a textovým atributem `description` obsahujícím lexikálně první komentář ze zdrojového kódu včetně bílých znaků a odřádkování převedeného na ` `. Chybí-li takový komentář, atribut je vynechán.), který obsahuje pro uživatelsky definované třídy elementy `class`. Každý element `class` obsahuje dva povinné atributy: (1) `name` s identifikátorem třídy a (2) `parent` s identifikátorem nadtřídy (rodiče). Při generování některých elementů je třeba definovat pořadí v rámci společného nadelementu, což se provádí povinným textovým atributem `order`, který je vždy číslován souvisle a od jedničky. Element `class` obsahuje podelement pro každou definovanou instační metodu `method` s povinným atributem `selector` pro určení selektoru definované metody. Element metody potom obsahuje element pro blok s odpovídajícím počtem argumentů.

Blok je reprezentován elementem `block`, podelementy `parameter` pro každý parametr bloku se dvěma povinným atributy `order` a `name` pro pořadí a identifikátor parametru. Dále element `block` obsahuje podelementy pro každý příkaz sekvence příkazů a s atributem `arity` udávajícím počet očekávaných argumentů pro budoucí vyhodnocení bloku. Příkaz reprezentuje element `assign` s povinným atributem `order` pro určení pořadí příkazu v sekvenci příkazů. Příkaz zahrnuje dva povinné podelementy `var` s atributem `name` pro identifikátor cílové proměnné a podelement `expr` pro výraz pro výpočet přiřazované hodnoty.

Výraz obsahuje jeden podelement podle druhu výrazu: (1) literál (`literal`), (2) proměnná (`var`, viz výše), (3) blokový literál (`block`, viz výše) nebo (4) zaslání zprávy (`send`).

Element `literal` obsahuje dva povinné textové atributy `class` s identifikátorem vestavěné třídy (`Integer`/`String`/`Nil`/`True`/`False`) a atribut `value` reprezentující hodnotu literálu (primitivní), např.

[21] Tradiční XML hlavička včetně verze a kódování je `<?xml version="1.0" encoding="UTF-8"?>`

číslo -10, prázdný řetězec, či `true`. U literálů typu `String` a podobně při zápisu do XML atributů nepřevádějte původní escape sekvence, ale pouze pro problematické znaky v XML (<, >, &, ', ") využijte odpovídající XML entity (`<`, `>`, `&`, `&apos;`, `&quot;`). Pro vyjádření literálu identifikátoru třídy je `class="class"` a `value` obsahuje identifikátor třídy. Literál bloku je reprezentován elementem `block` (viz výše).

Zaslání zprávy reprezentuje element `send`, jehož selektor zprávy je uložen v povinném textovém atributu `selector`. Výraz pro vyhodnocení příjemce zprávy je v podelementu `expr` (viz výše) a pokud se jedná o parametrickou zprávu obsahuje element `send` ještě podelementy `arg` pro každý argument předávaný zprávě. `arg` obsahuje právě jeden podelement `expr` (viz výše) pro výraz, jehož vyhodnocením získáme skutečný argument zprávy.

Příklad úryvku jazyka SOL25

```
[ :one :two | r := Integer from: two. ]
```

a jemu odpovídající XML:

```
<block arity="2">
  <parameter name="one" order="1" />
  <parameter name="two" order="2"></parameter>
  <assign>
    <var name="r"/>
    <expr>
        <send selector="from:">
          <expr><literal class="class" value="Integer"></expr>
          <arg order="1">
            <expr><var name="two" /></expr>
          </arg>
        </send>
    </expr>
  </assign>
</block>
```

**Doporučení:** Při tvorbě analyzátoru doporučujeme využít nějaký generátor syntaktických analyzátorů a použití knihoven pro zpracování parametrů příkazové řádky, generování XML dokumentů či pro načítání XML dokumentu.

Výstupní XML bude porovnáváno s referenčními výsledky pomocí nástroje A7Soft JExamXML[22], viz [2]. Pozor, v Python příkaz `return` neslouží pro návrat chybového kódu, použijte funkci `sys.exit`.

Pokud se teprve učíte psát čitelný kód v Python 3, tak doporučujeme nastudovat [5].

## 4.2 Bonusová rozšíření

**NVP** Při návrhu a implementaci skriptu `parse.py` a pomocných skriptů bude aplikováno objektově orientované programování a využit alespoň jeden vhodný standardní návrhový vzor výjma Jedináčka (viz [4] a tipy *na Moodle*). Podmínkou hodnocení tohoto rozšíření je řádná dokumentace (proč, kde, jak, popis omezení). V případě uvedení rozšíření v dokumentaci a bez vážné snahy o implementaci může obdržet jednobodový malus (tj. -1 b). [1 b]

[22] Nastavení A7Soft JExamXML pro porovnávání XML (soubor `options`) je na Moodle.

# 5 Interpret XML reprezentace SOL25 (`interpret.php`)

Seznamte se s dodaným rámcem *ipp-core* v PHP 8.4 a s jeho **povinným** využitím navrhněte a implementujte interpret, který načte XML reprezentaci abstraktního syntaktického stromu (AST) programu v jazyce SOL25 a tento program s využitím vstupu dle parametrů příkazové řádky interpretuje a generuje výstup. S některými základními úkony pomůže dodaný rámec *ipp-core* (např. načtení standardního vstupu, výpisy, základní zpracování parametrů). Vstupní XML reprezentace AST je definována v sekci 4.1. Lze předpokládat, že vstupní XML reprezentace již nebude obsahovat chyby[23], jež měl ze zadání za úkol detekovat `parse.py`. Jelikož je rámec *ipp-core* navržen objektově orientovaně, tak požadujeme, aby i jeho využívání a rozšiřování bylo provedeno objektově orientovaně se zvážením využití vhodných návrhových vzorů (doporučených na Moodle). K objektovému návrhu a implementaci je samozřejmě povinné sepsat také stručnou a terminologicky správnou dokumentaci, jak jste rámec *ipp-core* pro splnění zadání použili/rozšířili.

Interpret navíc oproti sekci 4.1 podporuje různě naformátované značky (např. volitelné využití zkráceného zápisu značek, pokud neobsahuje značka podelement). Sémantika jednotlivých uzlů AST vychází z popisu v sekci 3. Interpretace příkazů probíhá dle atributu `order` vzestupně.

**Tento skript bude pracovat s těmito parametry:**

- `--help` viz společný parametr všech skriptů v sekci 2.2;
- `--source=`*`file`* vstupní soubor s XML reprezentací AST dle definice ze sekce 4.1 a doplnění v úvodu sekce 5;
- `--input=`*`file`* soubor se vstupy[24] pro samotnou interpretaci zadaného AST.

Alespoň jeden z parametrů (`--source` nebo `--input`) musí být vždy zadán. Pokud jeden z nich chybí, jsou chybějící data načítána ze standardního vstupu.

**Chybové návratové kódy specifické pro interpret:**

- 41 - chybný XML formát ve vstupním souboru (soubor není tzv. dobře formátovaný, angl. *well-formed*, viz [1]);
- 42 - neočekávaná struktura XML (např. špatné zanoření elementů, chybějící povinné atributy, špatné hodnoty atributu `order`, apod.) – tyto chyby nebudeme testovat;
- 88 - integrační chyba (nevalidní integrace s rámcem *ipp-core*).

Proběhne-li interpretace bez chyb, vrací se návratová hodnota `0` (nula). Chybovým případům odpovídají následující návratové kódy interpretu v případě chyby během interpretace SOL25:

- 51 - běhová chyba interpretace – příjemce nerozumí zaslané zprávě;
- 52 - běhová chyba interpretace – špatné typy argumentů zpráv;
- 53 - běhová chyba interpretace – špatná hodnota argumentu (např. dělení nulou, nekompatibilní argument pro třídní zprávu `from:`).

[23] Ale určitě by mělo dojít alespoň ke kontrole, že nedošlo k poškození formátu před předáním skriptu `interpret.php`.
[24] Vstup/vstupní soubor může být prázdný; např. neinterpretuje-li se žádný konstruktor `read`.

Vaše zdrojové kódy druhé úlohy budeme kontrolovat statickou analýzou pomocí nástroje PHPStan, kde pro zisk nějakých bodů z celé úlohy požadujeme splnění úrovně 0 a pro zisk celého 1 bodu (součást hodnocení dokumentace a kvality kódu úlohy) bez ztrát požadujeme splnění úrovně 6[25], ale všeobecně doporučujeme úroveň 9, která je v dodaném rámci nastavena jako implicitní (možnost získat až 1 bonusový bod).

Kromě statické analýzy budeme vaše zdrojové kódy analyzovat i nástrojem PHP_CodeSniffer, pro kontrolu dodržení základní štábní kultury kódu. Pro zisk nějakých bodů z celé úlohy požadujeme dodržení standardů PSR-1[26] a PSR-4[27]. Ideálně můžete dodržovat i standard PSR-12[28] (možnost získat až 0,5 bonusového bodu).

**Rámec *ipp-core*:** Pro podpoření a ilustraci vhodnosti objektového návrhu je připraven jednoduchý rámec jménem ***ipp-core***, který povinně využijete při řešení této úlohy. Rámec je napsán v PHP 8.4 a jeho aktuální verze je k dispozici na fakultním git serveru[29] nebo v adresáři `/pub/courses/ipp/ipp-core` na serveru Merlin. V zadání zmíníme jen základní vlastnosti rámce, více informací najdete v `README.md` a studiem komentovaného zdrojového kódu. Nejasnosti ohledně rámce je vhodné diskutovat na Fóru. Rámec *ipp-core* dodržuje standardy PSR-1 i PSR-12 a podporuje automatické načítání tříd (tzv. *autoloader*) dle PSR-4. Rámec je také kompatibilní se všemi úrovněmi kontroly statické analýzy nástroje PHPStan. Instalace rámce na serveru *Merlin* do vlastního aktuálního adresáře může vypadat následovně:

```
git clone https://git.fit.vutbr.cz/IPP/ipp-core.git
cd ipp-core
php8.4 composer.phar install
```

čímž se do složky `vendor` nainstalují skripty a konfigurace pro *autoloader* dle PSR-4 a dodatečné nástroje pro kontrolu kódu. *PHPStan* slouží pro statickou analýzu kódu a základní ohodnocení kvality vašeho kódu. Analýzu vašeho kódu na serveru *Merlin* podle úrovně 6 potom spustíte příkazem:

```
php8.4 vendor/bin/phpstan analyze --level=6
```

PHP_CodeSniffer slouží pro kontrolu dodržení nastavené štábní kultury kódu. Analýzu vašeho kódu na serveru *Merlin* pro splnění minimálních požadavků spustíte příkazem:

```
php8.4 vendor/bin/phpcs
```

**Veškeré úpravy provádějte a nové soubory vytvářejte pouze ve složce `student`, jejíž obsah budete jako jediný odevzdávat. Tj. i dokumentace a případný soubor `rozsireni` musí být v této složce!** Složka `core` uchovává třídy samotného rámce ze jmenného prostoru `IPP\Core` a ve složce `vendor` budou případně další knihovny třetích stran, budou-li povolené pro použití v této úloze.

Celý program využívající rámec se spouští předchystaným skriptem `interpret.php`. Vaše implementace začíná v souboru `student\Interpreter.php` a odpovídající třídě `Interpreter` ve jmenném prostoru `IPP\Student`.

Rámec *ipp-core* využijte pro načtení vstupního kódu ve formátu XML (vrací se `DOMDocument`), základní zpracování parametrů (třída `Settings`) a pro interpretaci vstupně-výstupních instrukcí využijte rozhraní `InputReader` a `OutputWriter`. K dispozici bude i skript `is_it_ok.sh` pro kontrolu

[25] `https://phpstan.org/user-guide/rule-levels`

[26] `https://www.php-fig.org/psr/psr-1/`

[27] `https://www.php-fig.org/psr/psr-4/`

[28] `https://www.php-fig.org/psr/psr-12/`

[29] První přihlášení doporučujeme provést přes webový prohlížeč na `https://git.fit.vutbr.cz/IPP`, aby došlo k automatické aktivaci loginu jako uživatelského jména místo e-mailové adresy pro SSH přístup.

základních formálních požadavků a `Makefile` s několika základními cíli. Pro výpis varování na standardní chybový výstup obsahuje `interpret.php` na začátku příkaz `ini_set('display_errors', 'stderr');`.

**Dokumentace a objektově-orientovaný (OO) návrh:** Kód bude povinně využívat rámec *ipp-core* a bude navržen objektově, což znamená, že je třeba se seznámit s terminologií OOP (min. v rozsahu přednášek), zamyslet se nad možnostmi využití OO a návrhových vzorů pro tuto úlohu, návrh stručně a jasně (správnou terminologií) popsat v dokumentaci v souladu s vaší implementací (navrhované, ale neimplementované části je třeba označit).

Vhodným způsobem promítněte OO návrh do kódu (např. volitelné otypování signatur metod a funkcí) včetně vhodného volení identifikátorů tříd, atributů a metod s ohledem na sebedokumentaci. Nestačí-li jednoslovné pojmenování pro jasné zachycení účelu, doplňte komentář, u metody popište i význam/účel parametrů, pokud jméno a typová anotace nestačí.

Povinnou součástí dokumentace je UML diagram tříd s tím, že třídy rámce lze uvádět ve zkrácené formě (tj. bez atributů a metod), ale pro přehlednost důležité vztahy i mezi třídami a rozhraními z rámce nevynechávejte. Vámi *ne*implementované části UML diagramu vhodně odlište (např. sazbou šedou barvou). Případné využití návrhových vzorů je třeba náležitě zdůvodnit a jasně zdokumentovat, jak a kterými třídami je implementován. Především návrhový vzor Jedináček je téměř vždy využit nevhodně (a navíc implementován špatně).

**Doporučení:** V případě nekompletní implementace se nejprve zaměřte na možnost definice uživatelské třídy `Main` s metodou `run` a možnost provádět příkazy v bezparametrickém bloku, dále podporujte výstupní metody a minimalisticky implementované vestavěné třídy `Object`, `String` a `Integer`.

Pro vývoj na lokálním stroji je možné využít, že rámec *ipp-core* je k dispozici jako vývojářský kontejner, který lze pohodlně používat například ve Visual Studio Code (více viz README.md v rámci *ipp-core*).

# Reference


[1] Extensible Markup Language (XML) 1.0. W3C. World Wide Web Consortium [online]. 5. vydání. 26. 11. 2008 [cit. 2020-02-03]. Dostupné z: `https://www.w3.org/TR/xml/`

[2] A7Soft JExamXML is a java-based command line XML diff tool for comparing and merging XML documents. c2018 [cit. 2020-02-03]. Dostupné z: `https://www.a7soft.com/jexamxml.html`

[3] The text/markdown Media Type. Internet Engineering Task Force (IETF). 2016 [cit. 2020-02-03]. Dostupné z: `https://tools.ietf.org/html/rfc7763`

[4] Gamma, E., a kol.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[5] van Rossum, G., a kol.: PEP 8 – Style Guide for Python Code. c2023 [cit. 2024-01-30]. Dostupné z: `https://peps.python.org/pep-0008/`

# A Gramatika

Následující LL(1) gramatika formálně definuje syntaxi programu v jazyce SOL25. Celočíselný literál je reprezentován jako $\langle int \rangle$, řetězcový literál jako $\langle str \rangle$. Dále se v ní vyskytují čtyři typy terminálů, které označují identifikátor: $\langle id \rangle$ je běžný identifikátor, $\langle id: \rangle$ je identifikátor v selektoru zakončený dvojtečkou, $\langle :id \rangle$ je identifikátor v definici parametru bloku uvozený dvojtečkou a $\langle Cid \rangle$ je identifikátor třídy. Odpovídající LL tabulka je uvedena na další stránce.

Základní struktura programu:

1. $Program \rightarrow Class\ Program$
2. $Program \rightarrow \varepsilon$
3. $Class \rightarrow$ `class` $\langle Cid \rangle$ `:` $\langle Cid \rangle$ `{` $Method$ `}`
4. $Method \rightarrow Selector\ Block\ Method$
5. $Method \rightarrow \varepsilon$

Selektory při definici metod:

6. $Selector \rightarrow \langle id \rangle$
7. $Selector \rightarrow \langle id: \rangle\ SelectorTail$
8. $SelectorTail \rightarrow \langle id: \rangle\ SelectorTail$
9. $SelectorTail \rightarrow \varepsilon$

Bloky:

10. $Block \rightarrow$ `[` $BlockPar$ `|` $BlockStat$ `]`
11. $BlockPar \rightarrow \langle :id \rangle\ BlockPar$
12. $BlockPar \rightarrow \varepsilon$
13. $BlockStat \rightarrow \langle id \rangle$ `:=` $Expr$ `.` $BlockStat$
14. $BlockStat \rightarrow \varepsilon$

Výrazy, zasílání zpráv:

15. $Expr \rightarrow ExprBase\ ExprTail$
16. $ExprTail \rightarrow \langle id \rangle$
    (bezparametrický selektor)
17. $ExprTail \rightarrow ExprSel$
18. $ExprSel \rightarrow \langle id: \rangle\ ExprBase\ ExprSel$
    (parametrický selektor)
19. $ExprSel \rightarrow \varepsilon$
20.* $ExprBase \rightarrow \langle int \rangle \mid \langle str \rangle \mid \langle id \rangle \mid \langle Cid \rangle$
21. $ExprBase \rightarrow Block$
22. $ExprBase \rightarrow$ `(` $Expr$ `)`

Povšimněte si, že pravidlo 20* ve skutečnosti představuje čtyři různá pravidla pro čtyři typy terminálů: $\langle int \rangle$, $\langle str \rangle$, $\langle id \rangle$, $\langle Cid \rangle$ (to je důležité zejm. při implementaci podle LL tabulky níže).

| | `class` | *Cid* | `:` | `{` | `}` | *id* | *id:* | `[` | `\|` | `]` | *:id* | `:=` | `.` | *int* | *str* | `(` | `)` | $ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Program | 1 | | | | | | | | | | | | | | | | | 2 |
| Class | 3 | | | | | | | | | | | | | | | | | |
| Method | | | | | 5 | 4 | 4 | | | | | | | | | | | |
| Selector | | | | | | 6 | 7 | | | | | | | | | | | |
| SelectorTail | | | | | | | 8 | 9 | | | | | | | | | | |
| Block | | | | | | | | 10 | | | | | | | | | | |
| BlockPar | | | | | | | | | 12 | | 11 | | | | | | | |
| BlockStat | | | | | | 13 | | | | 14 | | | | | | | | |
| Expr | | 15 | | | | 15 | | 15 | | | | | | 15 | 15 | 15 | | |
| ExprTail | | | | | | 16 | 17 | | | | | | 17 | | | | 17 | |
| ExprSel | | | | | | | 18 | | | | | | 19 | | | | 19 | |
| ExprBase | | 20* | | | | 20* | | 21 | | | | | | 20* | 20* | 22 | | |

**Revize zadání:**

2025-02-15: Doplnění chybějícího popisu elementu `send` v sekci 4.1. Doplněna „Poznámka pro 1. úlohu“ na straně 5 o doporučení využít virtuální prostředí Python 3.

2025-02-16: Oprava dvou syntaktických chyb v příkladech. Oprava roku v hlavičce dokumentace.