**Implementačná dokumentácia k 2. úlohe do IPP 2019/2020**

Meno a priezvisko: Tomáš Ďuriš
Login: xduris05

## 1) interpret.py

### Zadanie a popis skriptu

Úlohou bolo vytvoriť skript v jazyku Python, ktorý načíta XML reprezentáciu programu, ktorá môže byť napríklad aj výstupom prvej úlohy (parse.php) zo zdrojového jazyka IPPcode20 a následne interpretuje príkazové riadky na základe vstupných parametrov a generuje výstup.

### Spôsob riešenia

Rovnako ako v prvej úlohe, tak aj tu som sa rozhodol riešiť aj rozšírenia. Ako prvé si môj skript načíta argumenty zo vstupu a následne skontroluje ich lexikálnu správnosť, spracuje ich a uloží si potrebné informácie na ďalšie použitie. Spracovanie argumentov som sa rozhodol riešiť ručne, keďže tak som si vedel najjednoduchšie ošetriť povolené kombinácie a povolené argumenty. V prípade argumentu *–help* prebehne kontrola, či bol tento argument zadaný samostatne a ak áno,vypíše sa nápoveda a program sa ukončí. Následne program pokračuje kontrolou vstupných súborov a ich otvorením. Pokiaľ všetko skončilo úspešne skript prejde na kontrolu správnosti XML a k jeho rozdeleniu na potrebné časti. V prípade nesprávnej štruktúry vstupného XML ukončíme program s návratovým kódom **31**

**Prvý beh** - skontrolujeme správnosť XML hlavičky, správnosť menoviek inštrukcií a atribútov programu, prípade odstránime nadbytočné biele znaky a overíme či je štruktúra celého XML podľa nášho zadania správna. Počas celej kontroly v prípade nájdenia nesúladu XML formátu so zadaním ukončujeme program s návratovým kódom **32**. Jednotlivé inštrukcie prechádzame zaradom riadok po riadku a vždy vykonáme kontrolu názvu, atribútov a uložíme si ich poradie (*order*), následne ak vieme, že inštrukcie sú správne zadané preusporiadame si ich podľa zadaného poradia (*order*). Už usporiadané inštrukcie znova prejdeme zaradom a podľa názvu inštrukcie (*opcode)* skontrolujeme zadaný počet argumentov s očakávaným počtom argumentov, v prípade inštrukcie **LABEL** skontrolujeme či dané návestie už nebolo zadané a ak nie, tak si ho uložíme. Argumenty prípadne usporiadame do správneho poradia a skontrolujeme ich lexikálnu a syntaktickú správnosť pomocou regulárnych výrazov. Ak všetko prebehne správne vrátime usporiadanú a overenú XML štruktúru k vykonaniu sémantickej kontroly

**Druhý beh** - sémantická kontrola prebieha taktiež po riadkoch. Získame meno inštrukcie (*opcode)*, uložíme si jeho atribúty a na základe tohoto mena vykonáme príslušnú funkciu, ktorú toto meno reprezentuje. Využívame vytvorené globálne premenné pre rámce (***GF, LF, TF***), zásobník pre volanie funkcií, dátový zásobník aj pre implementáciu **STACK** rozšírenia, slovník pre návestia aj s poradím(inštrukciou) na ktoré skočiť a premenné pre ukladanie počtu vykonaných inštrukcií a uložených premenných (rozšírenie **STATI**). Pri jednotlivých funkciách, ktoré reprezentujú syntaktickú kontrolu jednotlivých inštrukcií, prebehne prípadné "vybratie" hodnoty premennej použitím vytvorenej funkcie *symb_from_var(),* zistenie jej typu a hodnoty pomocou funkcie *check_var()*, následná sémantická kontrola a vykonanie príslušnej operácie. Výsledok, ak je treba, tak si uložíme do premennej pomocou funkcie *save_to_var().* Do premennej si vždy ukladáme už požadovanú hodnotu a typ, okrem typov string a nil, pri ktorých si musíme aj špecifikovať či sa v prípade typu string jedná o string zadaný cez inštrukciu **READ** a tým pádom sa nevyžaduje kontrola "*escape sekvencií"* alebo pri type nil, z dôvodu odlíšenia tohto typu od Neinicializovanej premennej (hodnota None). Pri sémantickej kontrole využívam mnoho pomocných funkcií, ako napríklad *get_type()* pre zistenie typu hodnoty v premennej, *interpret_escape()* pre spätné prevedenie "*escape sekvencií*" alebo *convert_to_bool()* pre prevedenie reťazcovej reprezentácii bool premennej do požadovaného tvaru. Významná funkcia je taktiež *init_var()*, ktorá vytvorí zadanú premennú v požadovanom rámci a sprístupní ju. Zvyšné funkcie sú detailnejšie popísané v zdrojovom kóde prostredníctvom komentárov.

Pri celkovej správnosti zadaného programu, sa vypíše interpretácia jeho výsledku, prípadne sa uložia zadané štatistiky a skript sa ukončí s návratovou hodnotou 0 (toto neplatí ak bola použitá inštrukcia EXIT, kde sa skript ukončí so zadaným návratovým kódom)

**Implementácia rozšírenia**

Ako som už spomínal, rozhodol som sa implementovať všetky tri zadané rozšírenia a to rozšírenie **FLOAT, STACK** a **STATI**. Pri rozšírení **FLOAT** som implementoval podporu typu *float* a to pomocou načítania cez funkciu *float.fromhex()*, pomocou ktorej aj kontrolujem správnosť zadaného čísla, pridal som podporu tohto typu pre požadované inštrukcie a jeho následný výpis v hexadecimálnej reprezentácii. Pri rozšírení **STACK** som implementoval dátový zásobník, získavanie hodnôt z neho a jednotlivé inštrukcie ktoré si dané rozšírenie vyžaduje. Rozšírenie **STATI** som implementoval podobne ako aj v parse.php, čiže formálna kontrola zadaných parametrov, uloženie potrebných informácií a výpis štatistík riadok po riadku na výstup na základe zadaného poradia. Kontrola počtu premenných a inštrukcii prebieha po každej vykonanej inštrukcii a pri počte premenných si skontrolujeme hodnotu s predchádzajúcou a uložíme tú väčšiu.

## 2) test.php

**Zadanie a popis skriptu**

Úlohou bolo vytvoriť skript v jazyku PHP, ktorú bude slúžiť pre automatické testovanie skriptov parse.php a interpret.py, prípadne ich kombinácie. Skript prejde zadaný adresár a vygeneruje **HTML** reprezentáciu správnosti oboch skriptov.

**Spôsob riešenia**

Skript začne svoj beh načítaním argumentov zo vstupu pomocou funkcie *getopt()* a prebehne ich porovnanie s poľom podporovaných argumentov. Nastavia sa predvolené parametre (predvolená cesta k súborom) a uloží sa aktuálny priečinok ( ak by parameter *–directory=súbor* nebol zadaný. Prebehne kontrola argumentov a správnosť ich kombinácie spolu s uložením potrebných informácií a ciest k zadaným súborom. Parameter *–help* rovnako ako pri interpret.py vypíše nápovedu a skript sa ukončí s návratovou hodnotou **0**. Skontroluje sa existencia súborov, vygeneruje sa **HTML** hlavička a spustí sa prechádzanie adresára ( prípade rekurzívne prechádzanie podadresárov, ak bol zadaný parameter *–recursive*)

Nájdené súbory sa uložia do poľa všetkých testov a následne sa vygenerujú prípadné chýbajúce súbory (**.in, .rc** a **.out**). V prípade generovania chýbajúcich súborov sa tieto súbory vytvoria prázdne, alebo v prípade vytvorenia .rc súboru sa do neho zapíše hodnota **0**. Následne si všetky kompletné testy zoradíme abecedne.

Ďalším krokom je už hlavný beh programu, kde pre každý test v poli všetkých testov získame jeho cestu, meno a meno adresára v ktorom sa nachádza. Následne vytvoríme dočasné súbory pre výstup z jednotlivých skriptov alebo *diff* nástroja. Získame očakávaný návratový kód a podla zadaného argumentu skriptu **test.php** (*--parse-only*, *--int-only*) vykonáme buď testovanie skriptu **parse.php** alebo testovanie skriptu **interpret.php**, prípadne ak nie je zadaný ani jeden argument vykonáme testovanie naprv **parse.php**, kde ak iný návratový kód ako **0**, tak ho porovnáme s očakávaným návratovým kódom a podľa toho vyhodnotíme úspešnosť testu, alebo ak je návratový kód **0**, tak výstup pošleme na vstup do **interpret.py** a návratový kód porovnáme s očakávaným návratovým kódom. Pokiaľ sa rovnajú a ich hodnota je **0**, skontrolujeme výstup skriptu **interpret.py** s daným .out súborom pomocou nástroja *diff* a na základe výsledku vyhodnotíme úspešnosť testu.

**HTML** reprezentáciu generujeme počas vyhodnotenia testov, vždy jeden riadok do tabuľky pre každý test. Farebne je vyznačený výsledok a v prípade chyby aj očakávaný návratový kód oproti prijatému kódu. Testy sú radené podľa priečinku a abecedne, na konci je počet úspešných testov k počtu celkových testov, pričom ak je pomer nad *50%* tak výsledok je zelený, inak je červený. Na záver sa odstránia dočasné súbory a výsledok je vypísaný na štandardný výstup