Implementační dokumentace k projektu do IPP 2017/2018
Jméno a příjmení: Matěj Mitaš
Login: xmitas02

O projektu

Vzhledem k povaze zadání projektů bylo zapotřebí si zopakovat znalosti a dovednosti nabyté v předmětu IFJ z minulého semestru. Hlavní rozdíl spočívá v pokrytí značně jiné tématiky než v onom projektu, tj. návrh a implementace vlastního interpretu daného programovací jazyka (v tomto případě *IPPcode18*, který je až na malé rozdíly (podpora typu plovoucí řadová čárka) kompatibilní s jazykem *IFJcode17*.

Ve zkratce se jedná o jazyk založený na symbolických instrukcích, jehož instrukční sada podporuje jak tříadresné tak i zásobníkové instrukce. Stejně tak je uživateli umožněno vytvářet podprogramy, instrukce *CALL* a *RETURN*. Dále disponuje řadou instrukcí na práci s řetězci.

Obecný postup návrhu

Projekt se skládá ze tří hlavních částí, které dohromady tvoří jeden celek. Z zadání plyne, že pro část *Parser* a *Test* je nutno použít programovací jazyk **PHP** a pro *Interpret* jazyk **Python**. Oba zmíněné jazyky umožnují plnohodnotně použít více druhu paradigmat, avšak vzhledem k zaměření toho předmětu bylo zvoleno objektové paradigma. Ačkoliv se jedná o jazyky s podobným zaměřením, lze mezi nimi pozorovat mnoho rozdílů. Pro uživatele nejzřejmější je pythonovský bez závorkový přístup k programování, nicméně syntaxe by pro dobrého programátora neměla představovat problém. U PHP lze pozorovat postupný nesourodý vývoj jazyka a s tím spojené obtíže např. při výběru systémové funkce pro přidání na konec pole.

Po obeznámení se s specifikami jazyků přišla na řadu dekompozice problémů a návrh abstrakce dle objektového paradigmatu. Každá část projektů má hlavní soubor (dle zadání), ve kterém je činěno pouze to nejnutnější pro chod programu (zpravidla instanciace objektu z hlavní třídy programu a invokace metody pro běh daného programu, cílem je uživateli poskytnou jednoduché rozhraní), implementace jednotlivých abstrahovaných funkčních bloků je přesunuta do samostatných souborů s předponovou *ippcode18_* pro Python a *_ippcode18_* pro PHP. Každý takový soubor představuje implementaci jedné třídy, pro zachování přehlednosti a zapouzdření kódu.

Implementace

V každé části bylo třeba číst určitý počet a kombinaci vstupních parametrů, v **PHP** zajištěno pomocí funkce *getopt*, v **Pythonu** manuálně. Vždy byla pro čtení parametrů vyhrazena jedna třída, jejímž úkolem bylo přečíst a hlavně vyhodnotit kombinace a význam parametrů. Zde byla komplexnost zvýšena požadavky rozšíření hlavně u části *Parser*, jelikož u přepínačů *–comments* a *–loc* si bylo nutno pomatovat nejen aktivaci, nýbrž i pořadí z důvodu výstupu do textového souboru. Správná funkčnost byla zajištěna pomocí konstant popisujících chování programu (BASIC, HELP, STATS, STATS_LINES, STATS_COMMMENTS).

Zároveň každý program vrací výsledek, společně na standartní výstup (potažmo v případě chyby na standartní chybový výstup s textovou zprávou pro uživatele) a návratový kód. Kódy byly zadány a jsou respektovány napříč programem.

```
Matejs-MacBook-Pro: ~/Documents/_Uni/2BITL/IPP/proj
→ cat ret_py.txt
▷ Spatne typy operandu
◈ Zadany datovy typ nepodporuje operaci zadanou instrukci

Matejs-MacBook-Pro: ~/Documents/_Uni/2BITL/IPP/proj
→
```

Interpret

```
Matejs-MacBook-Pro: ~/Documents/_Uni/2BITL/IPP/proj
→ cat ret_php.txt
Navratovy kod: 21
>Lexikalni chyba analyzatoru

Matejs-MacBook-Pro: ~/Documents/_Uni/2BITL/IPP/proj
→
```

Parser/Test

Dále se již programy značně liší, proto je tedy proberu v následující odstavcích.

Parser

V této části probíhá syntaktická a lexikální analýza. Po přečtení vstupních parametrů, je třeba inicializovat objekt **Parser**, který ze standartního vstupu přečte vstupní soubour (realizováno pomocí funkce *fgets()*, v případě volby statistiky incializuje objekt **Stats** pro uchování statistik. Samotné čtění probíhá v metodě **_parser_read_lines()** třídy **Parser**. Zde pro každý řádek vstupního souboru instancujeme **Line** s obsahem řádku. Zde proběhne primární kontrola, hlavně jestli se jedná o datový či komentářový řádek (při statistikách inkrementujeme čítač komentářů). V případě načtení prvního řádku je třeba zkontrolovat validitu manda-

torní hlavičky programu *.ippcode18*. Po úspěšné kontrole je řádek vrácen zpět a je poslán do **Instruction**, kde probíhá samotná syntaktická a lexikální analýza.

Zde je řádek pomocí regulárních výrazu uložených jako třídní atributy (**_inst_regex, _param_regex a _symb_regex**) vyhodnocen buď jako validní instrukce (podle vzorů v **_inst_blueprint** či jako nevalidní konstrukt. Výstupem této metody je položka do XML souboru na výstupu, která je uložena do datové struktury **parser_output** objektu **Gen**, který realizuje samostatné vygenerovanání výstupního XML. Každá řádek se transformuje na XML entitu **instruction** s atributem *"opcode"* a *"order"*, který obsahuje kód instrukce, resp. její pořadí v programu.

Interpret

Interepret úzce navazuje na práci parseru. Načteme vstupní argumenty, třída **Argparse**, zde je třeba dávat pozor na argument *–source*, který nám určuje vstupní soubor s v XML formátu. Je nutno opět provést parsování, avšak v zásadě stejné jako u *Parser*, proto nechám na čtenáři, aby si v kódu udělal obrázek o konkrétní implementaci.

Mnohem zajímavější je samotná interpretace kódu. Skládá se ze tří průchodů programem, v první načteme všechna návěští, v druhé zkontrolujeme skoky (undefined label error). Poté ve třídě **Interpret** běží hlavní programová smyčka v metodě **__intr_loop**. Po vytvoření Program Counteru beží tak dlouho, než narazí na poslední instrukci. Skoky v programu pomocí instrukcí *JUMP, JUMPIFEQ, JUMPIFEQ, CALL, RETURN* jsou řešeny návratovým kódem z metody **fn_execute** třídy **Function**, kdy pro instrukce neskokové inkremetuji čítač a instrukce skokové ho modifikují dle své libosti.

V samotné třídě instanci třídy **Function** je prováděno následující. Nejdříve zkontrolujeme validitu operačního kódu. Poté zkontrolujeme počet a typ předaných instrukcí. Zde bych chtěl apelovat na čtenářovu pozornost. Tato část programu je vysoce modulární, v případě nutnost zavedení nové instrukce/instrukcí není problém tuto změnu udělat. Interpret si vlastnosti instrukcí načítá ze souboru **ippcode18_blueprint.json**, kde jsou uloženy jednotlivé instrukce jako objekty s argumenty, které jsou definovany svým druhem (*VAR, SYMB, TYPE, LABEL*), u některých argumentů svým datovým typem (*INT, BOOL, STRING* a v neposlední řadě nutností obsahu hodnoty (např. můžeme přesunout do proměnné prázdný symbol s datovým typem, avšak nemůžeme indexovat řetězec prázdnou hodnotou). Po nalezení přepisu intrukce proběhne v metodě **_fn_prepare_args**, resp. **_fn_prepare_stack_args** získání a přetypování argumentů (může proběhnout až v samotné metodě realizující funkčnost dané instrukce) pomocí instance třídy **Retype**.

V poslední řadě je aktivována metoda **fn_execute**, která invokuje příslušnost metodu se signaturou reflektující počet instrukčních parametrů. Níže příklad pro **JUMPIFNEQ**:

```
def _jumpifneq(self, label_name, eq1, eq2):
```

Test

Poslední součástí projektu byl testovací skript v **PHP**. Po startu opět čteme parametry, tentokrát je jich více, zajímavé jsou *–parse-script=file* a *–int-script=file*, které nám umožnují připojit vlastní verze parseru, resp. interpretu, výchozí nastavení je spuštění pomocí souboru **parser.php** a **interpret.py** v aktuální složce. Jediným úkolem programu je buď rekurzivně či nerekurzivně prohlédnout aktuální či zadaný adresář podle parametru *–directory* a najít všechny zdrojové soubory s příponou **.src**

Samotný test se skládá až ze čtyř souborů (kromě výše zmíněného **.src** - zdrojového souboru) to jsou: **.in** - stdin, **.out** - stdout a **.rc** pro návratový kód. Zdrojový soubor je poslán do postupně do *Parseru* a *Interpretu*, kde se vyhodnotí a vrátí návratový kód, který se zkontroluje s očekávaným. To samé platí pro výstupní soubor.

Jádro implementace je třída **Tests**, po začátku programu je invokovánana její metoda **test_init**, která pomocí instance třídy **Tests_IO** a jejích metod vyhodnotí testy v daném adresáři. V cyklu procházíme výsledky a pro každý invokujeme **_test_single_test**, pro samotné vykonání testu. Nakonec pomocí **_test_render** vytvoříme čitelný a snadno použitelný HTML výstup.

Rozšíření

V *Parseru* bylo implementováno rozšíření **STATP**, v *Interpretu* **STACK** a byla učiněna příprava pro zbývající rozšíření, avšak z důvodu času již byla ponechána nedokočena. Samotné rozšíření zpravidla znamenaly určité zásahy do projektu, hlavně u **STACK** bylo třeba implementovat metodu **_fn_prepare_stack_args** pro přípravu argumentů ze zásobníku.