Implementační dokumentace k 2. úloze do IPP 2022/2023
Jméno a příjmení: Aleksandr Shevchenko
Login: xshevc01

1 Parsování argumentů

Úplně na začátku provádím kontrolu vstupních argumentů funkcí `parse_args`, která v případě přepínače `--help` vypíše nápovědu, a taky může skončit program návratovou hodnotou 10, pokud počet argumentů je špatný anebo **source_file** či **input_file** nejsou validní.

Pokud právě jeden z argumentů není zadán, implicitně ho nastavuji na `sys.stdin` podle zadání. Pokud nenastane žádná chyba, tato funkce vrací **source_file** a **input_file** a program běží dál.

2 Hlavní tělo programu

Po parsování argumentů následuje hlavní tělo programu, implementované ve funkci `program_runner`, která přijímá **source_file** a **input_file**.

Na začátku se provádí kontrola formátu XML vstupu (podrobněji popsáno v sekci 3). Dál pomocí funkcí `iter` z balíčku `xml.etree.ElementTree` procházím jednotlivou strukturou stromu. Ověřuji, zda XML obsahuje nutnou hlavičku a zda všechny ostatní uzly mají tag „instruction“. Do proměnné **list_of_instructions** ukládám jednotlivé instrukce, pomocí funkcí `sort_by_order` uspořáduji je podle hodnoty atributu „order“.

Dál následují dva průchody seznamem **list_of_instructions**. První najde všechna místa v kódu, označené jako `LABEL` a uloží název a pozici v kódu tohoto návěští do slovníku **array_of_labels**. Druhý průchod už zpracovává jednotlivé instrukce, a to s případným vrácením zpátky kvůli návěštím. Jelikož za běhu programu se může nastat chyba, výstup programu se vypisuje na stdout až na konci hlavního těla programu.

3 Práce s XML

Proměnná **reader** v hlavním těle programu je instancí třídy `ParserXML`. Pomocí metody `is_XML_correct` ověřuji, zda struktura XML je správná, respektive že hlavička má tag „program“ a hodnota „language“ je „IPPcode23“. První průchod instrukcemi se zabývá nejen návěštími, ale i kontrolou toho, že žádné dvě instrukce nemají stejný „order“ (v tuto chvíli instrukce už jsou seřazené vzestupně podle něj). Za tímto účelem jsem vytvořil metodu `order_checker`. V třídním atributu `instruction_order` je vždy uložená hodnota poslední ověřené instrukce, proto když se objeví instrukce se stejným „order“, program se ukončí s chybou.

4 Zpracování instrukcí

Jak již bylo zmíněno, na konci `program_runner` se provádí druhý průchod instrukcemi. Samotným zpracováním instrukcí se zabývá instance **interpret** třídy `Interpreter`. Metoda `run` této třídy vytváří instanci třídy `Instruction`, a invokuje její metodu `run_instruction`.

Pomocí funkce `getattr` a v závislosti na „opcode“ se volá příslušná metoda `run_(opcode)`. Usnadnilo to práci tím, že jsem nepotřeboval žádný seznam možných „opcode“. Pro každou instrukci je vytvořena odpovídající „run-metoda“, která provádí samotnou instrukci a vrací další pozici v kódu (obzvlášť to potřebujeme pro instrukce, pracující s návěštími).

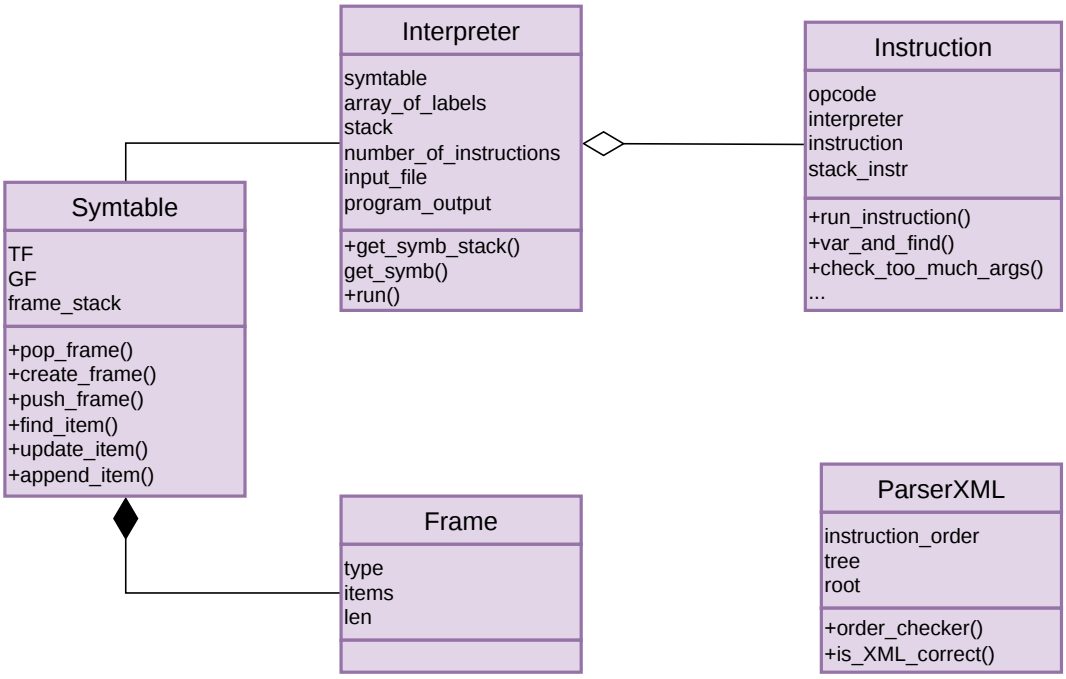
Kromě metody `run` interpreter obsahuje pomocnou metodu `get_symb`, pracující s instancí třídy `Symtable`. Ona obsahuje informace ohledně stavů jednotlivých rámců a metody pro práci s nimi: `pop_frame`, `create_frame`, `push_frame`. Kromě toho, metody `find_item`, `update_item` a `append_item` pomáhají vyhledávat a měnit obsah rámců. Samotné rámce jsou instancemi třídy `Frame`, a pamatují si jenom svůj typ, položky ve struktuře slovník a délku.

5 Pomocné funkce

Kromě již zmíněných funkcí a třídních či instančních metod mám jiné pomocné funkce. Pro ovládání chyb používám `error_message` a `exit_if_none`. Práci s řetězci usnadňují `none_to_empstr` a `converter`, pro „matching“

operandů jsou implementovány `is_var` a jí podobné funkce.

6 Diagram tříd



Tabulka symbolů obsahuje v sobě více rámců, ale jednotlivé rámce nemůžou existovat bez tabulky symbolů. Jeden interpreter zpracovává všechny instrukce, ale ty můžou existovat i bez něj. Taky by v jednom interpreteru mohlo být více tabulek symbolů, ale v mém případě to zůstalo obecnou asociací. ParserXML není nějak spojen s jinými třídami, ale přesto hraje důležitou roli v hlavním těle programu.

7 Implementovaná rozšíření

7.1 FLOAT

Největším problémem při implementaci tohoto rozšíření bylo zpracování různých typů float čísel. Jelikož můžeme načítat ze vstupu čísla ve dvou různých zápisech (dekadickém a hexadecimálním), použil jsem funkce `float`, `float.fromhex` a `float.hex`. Pro jednoduchost výsledky aritmetických operací a zápis hodnoty do proměnné provádím v hexadecimálním formátu.

Kromě instrukcí, uvedených v zadání, moje implementace podporuje práci s floating point čísly taky pro `LT`, `GT`, `EQ`, `JUMPIFEQ`, `JUMPIFNEQ` a zásobníkové varianty instrukcí.

7.2 STACK

Pro zásobníkové instrukce (kromě `PUSHS`, `POPS` a `CLEARS`) jsem neimplementoval samostatné „run-metody“. V tomto případě se volá funkce pro nezábníkovou variantu instrukce, ale nastavuje se příznak, že se má pracovat se zásobníkem. Pomocná metoda `get_symb_stack` místo `get_symb` „popuje“ symboly ze zásobníku a vrací jejich hodnotu a datový typ. V doplnění k zadání jsem implementoval instrukci `DIVS`.