

Jméno a příjmení: Jiří Juřica  
Login: xjuric29

Skripty jsem se snažil strukturovat do logických částí pomocí oddělených souborů, proto se i tento skript skládá z hlavní části `parse.php` a následně souborů ve složce `parse_lib` `scanner.php`, `syntax.php` a `others.php`. V `parse.php` probíhá ošetření argumentů a případný zápis statistik z rozšíření do souboru a jinak už jen volá fci `syntaxer()` z `syntax.php`, který řídí celé zpracování.

[illegible]

## Interpret.py

```

classDiagram
    class Interpreter {
        - inst: list
        - vars: set
        - instructionList: list
        # flowControl: FlowControl
        # stack: list
        # frames: Frames
        # logger: Logger
        - argument_parsed()
        + start()
        - set_labels()
        - stat()
    }
    class Logger {
        + verbose: bool
        + print(message: str)
    }
    class SimpleType {
        # value: None
        + get_value() instoolstr
    }
    class Instruction {
        # instructionRequiredTypes: list
        # operands: list
        # frames: Frames
        # flowControl: FlowControl
        # check_operands()
        # get_checked_value(entity): instoolstr
    }
    class XML {
        - instructionList: list
        - logger: Logger
        - stack: list
        - frames: Frames
        - xmlRoot: xml.etree.ElementTree.Element
        + get_instruction_list() list
        + convert_argtype str, value str
    }
    class Frames {
        - globalFrame: dict
        - frameStack: list
        - tmpFrame: dict
        + create_frame()
        + push_frame()
        + pop_frame()
        + get_frame() dict
        + get_local_frame() dict
        + get_var_count() int
    }
    class FlowControl {
        + instructionCounter: int
        + positionBack: list
        + labelDict: dict
    }
    class Const {
        - replace_escape_seq(value str)
    }
    class Label {
    }
    class Type {
    }
    class Var {
        - frame: str
        - name: str
        - frames: Frames
        + get_frame() str
        + get_name() str
        + define()
        + set_value()
        + get_value() instoolstr
        + get_value_protected() instoolstr
    }
    class CREATEFRAME {
        + execute()
    }
    class PUSHFRAME {
        + execute()
    }

    Interpreter --> Logger : prints info about
    Interpreter --> SimpleType : operand has type
    Interpreter --> Instruction : operand has type
    Interpreter --> XML : is created by
    Interpreter --> Frames : creates
    Interpreter --> FlowControl : keeps state
    Interpreter --> XML : sends data filename
    Interpreter --> Instruction : enacts
    Instruction --> SimpleType : operand has type
    Instruction --> XML : is created by
    Instruction --> CREATEFRAME : 
    Instruction --> PUSHFRAME : 
    Instruction ..> PUSHFRAME : 
    
```

The diagram illustrates the architecture of the Interpreter project. The **Interpreter** class is the central component, managing the execution flow, state (stack, frames, logger), and interacting with various data structures and control elements. It delegates tasks like operand checking to **Instruction**, state management to **XML**, and frame management to **Frames**. The **Logger** class handles verbose output. **SimpleType**, **Instruction**, **XML**, **Frames**, and **FlowControl** are core data and control objects. **Const**, **Label**, **Type**, and **Var** represent different types of constants and variables. **CREATEFRAME** and **PUSHFRAME** are specialized execution methods for frame management.

Instance třídy **Interpret** řídí celý běh programu. Nejdůležitější metodou je `start()`, která vytvoří instanci třídy **XML**. Ta se postará o načtení, kontrolu a převedení veškerých částí kódů na předchystané objekty a **Interpretu** vrátí list po sobě jdoucích instancí instrukcí. V metodě `start()` je následně zavolána metoda `__set_labels()`, která před prováděním načte a zaznamená pozice pro instrukce **LABEL** a postará se přitom o jejich nahrazení za instrukce **NOP**. Následně již začne probíhat vykonávání kódu, které je ukončeno, načte-li se **None** z listu instrukcí, který slouží k zaznamenání konce.

### **Test.php**

Po ošetření argumentů volá skript externí příkaz `find`, který řeší problém hledání (v daném adresáři nebo rekurzivně) jednotlivých testů. `Find` vyhledává soubory s příponou `.src` a při následném zpracování zjišťuje a případně doplňuje chybějící soubory.

Hlavní část skriptu sestavuje soubor dat pro každý test, které následně odesílá funkci `addTest(data)`, která se stará o začlenění dat do vznikajícího html výstupu. Po skončení se vypíše html výstup, jehož součástí je tabulka s přehledem o tom, jak fungoval `parse.php` a `interpret.php` a zda se lišil výstup interpretu od referenčního, pokud oba skripty končili kódem 0.

Porovnání je opět prováděno externím programem `diff` a v případě, že je jeho výstup delší než tři řádky, v buňce se automaticky doplní javascriptový odkaz na zobrazení/skrytí celého obsahu.

### **Obecně**

Pro každý skript platí, že u něj lze aktivovat debugovací režim, kdy na `stderr` vypisuje poměrně podrobně jeho aktuální činnost a lze pak jednoduše zjistit, v jaké části se v průběhu vykonávání vyskytl problém. Zapíná se pomocí připsání dlouhého argumentu `--verbose` u všech skriptů, krátké verze jsou různé v závislosti na tom, zda se `-v` nevyužívá již k něčemu jinému dle zadání.