Implementační dokumentace k 1 úloze do IPP 2023/2024

Jméno a příjmení: Kininbayev Timur

Login: xkinin00

Introduction

This document outlines the design and implementation of a Python script dedicated to processing IPPcode24 source code, verifying its lexical and syntactic correctness, and converting it into a structured XML representation.

Design Philosophy

The script's design adheres to principles of modularity, readability, and robust error handling. Each component of the script - *argument parsing, input reading*, *lexical analysis*, *syntactic analysis*, and *XML generation* - is encapsulated in dedicated functions.

Solution Process

The script begins with parsing command-line arguments, specifically looking for a `--help` option to display usage information. It then reads the IPPcode24 source code, ensuring the presence of a correct header. Following this, it performs lexical analysis to tokenize the input and syntactic analysis to validate the structure of each instruction according to predefined rules. Finally, it generates an XML representation of the program, adhering to the specified format. Special attention is given to error handling, with detailed error messages guiding the correction of syntactic and lexical mistakes.

Internal Representation

Internally, the script utilizes dictionaries, tuples, and lists to manage the program's structure and its elements efficiently. Instructions and their expected operands are mapped in a dictionary (instruction_rules), facilitating quick validation checks during the syntactic analysis phase.

- Lexical Analysis:
    - Lexical analysis is implemented using regular expressions for tokenization. Tokens extracted from the source code are stored as tuples containing their value, type for compression, and actual type for containing all the information needed in syntactic analysis and XML generation
    - Example of type diversion:
        - In this case, the token will contain type `"CONSTANT"` and it will be compared with actual rules in syntactic analysis. Actual type is needed for proper XML generation.

```
elif re.match(r'^int@[-+]?\d+$', token):
    token_type = "CONSTANT"
    token_actual_type = "int"
```

- Syntactic Analysis:
  - Syntactic analysis verifies the structure of each instruction against predefined rules, ensuring each opcode is followed by the correct number and type of operands. This phase utilizes the `instruction_rules` dictionary, which maps each opcode to its expected operand types.
  - The analysis iterates over tokens identified during lexical analysis, checking each instruction's opcode and operands. A key aspect of this phase is the introduction of the `"SYMB"` type, which encompasses various operand types (variables, constants) not explicitly categorized during lexical analysis.
  - For each instruction, the syntactic analysis:
    - Confirms the first token as a valid opcode.
    - Matches the number of operands to the expected count for that opcode.
    - Validates each operand's type against the expected types, incorporating special handling for the "SYMB" type:

```
for idx, (operand_type, expected_type) in
enumerate(zip(operands, expected_operands)):

       if expected_type == "SYMB" and operand_type[1] in
["VARIABLE", "CONSTANT", "string", "bool", "int", "nil"]:

           continue  # argument type is correct
```

- XML generation:
  - The XML generation process converts validated IPPcode24 instructions into an XML document using the ElementTree API. This structured representation begins with a `<program>` root element, indicating the source language as IPPcode24.
  - For each instruction, an `<instruction>` element is created, featuring `order` and `opcode` attributes to reflect its sequence and type. Operand elements (`<arg1>`, `<arg2>`, etc.) are nested within their respective instruction elements, each marked with a `type` attribute (e.g., "var", "int", "bool", "string") and populated with the operand's value, adhering to XML standards.