Dokumentacja Interpretera Języka ze Statycznym Typowaniem, Typem Słownikowym Niehomogenicznym, Zmiennymi Nullable i Type Matchingiem

Mikołaj Taudul

November 5, 2024

1 Wstęp

Język programowania, napisany w Pythonie, który wyróżnia się silnym typowaniem, oraz typem słownikowym niehomogenicznym, pozwalającym na przechowywanie danych o różnych typach w jednej strukturze. Obsługuje również zmienne nullable, co ułatwia pracę z opcjonalnymi wartościami, oraz type matching, umożliwiający dynamiczne dopasowanie działań do rzeczywistego typu wartości.

2 Opis Zakładanej Funkcjonalności

2.1 Funkcjonalności standardowe

Podstawowe typy danych i operacje Język obsługuje typy liczbowe **int**/**float** z operacjami matematycznymi (np. `+`, `-`, `*`, `/`) z zachowaniem priorytetu i obsługą nawiasów, typ **bool**, a także typy znakowe **string** z możliwością konkatenacji.

Zmienne Możliwe jest tworzenie zmiennych, przypisywanie do nich wartości oraz odczytywanie tych wartości. Język oferuje **statyczne i silne typowanie**. Zmienne są **mutowalne** i mogą być **nullable**, co umożliwia przypisanie im wartości `null` w przypadku braku danych lub gdy wartość jest opcjonalna.

Komentarze Wsparcie dla komentarzy w jednej linii, rozpoczynających się od znaku `#`.

Instrukcje warunkowe i pętle Język wspiera konstrukcje warunkowe i pętle.

Funkcje Możliwe jest definiowanie własnych funkcji. Argumenty funkcji przekazywane są przez wartość dla typów prostych (int, float, bool) i przez referencję dla typów złożonych (string, słownik).

Rekursja Język wspiera rekursywne wywołania funkcji. Limit głębokości rekursji 1000.

Obsługa błędów Zaimplementowano mechanizmy obsługi błędów, polegający na wyświetlaniu odpowiednich komunikatów przy niepoprawnym kodzie.

2.2 Przekazywanie zmiennych

W języku istnieją dwa główne sposoby przekazywania zmiennych do funkcji: przez wartość oraz przez referencję. W zależności od typu zmiennej, język automatycznie określa sposób przekazania.

Przekazywanie przez wartość Zmienne prostych typów, takich jak `int`, `float`, `bool`, oraz `string`, są przekazywane przez wartość. Oznacza to, że przekazywana jest kopia zmiennej, a zmiany wprowadzone w funkcji nie wpływają na oryginalną zmienną.

```
fun update_text(string text) {
    text = "new text";
}

string message = "original text";
update_text(message);
print(message);  # original text
```

Przekazywanie przez referencję Typy złożone, takie jak słowniki, są przekazywane przez referencję. Oznacza to, że zmiany wprowadzone na słowniku wewnątrz funkcji wpływają na oryginalną zmienną, co pozwala na wydajne operacje na dużych strukturach danych.

```
fun update_value(dict data, string key, int new_value) {
    data.update(key, new_value);
}

dict myDict = {"age": 25};
update_value(MyDict, "age", 30);
print(myDict.get(age));  # 30
```

Decyzja zależna od typu danych W języku zmienne są przekazywane przez wartość dla prostych typów danych, takich jak liczby, wartości logiczne oraz łańcuchy tekstowe, natomiast przez referencję dla typów złożonych, takich jak słowniki. Dzięki temu zarządzanie pamięcią jest bardziej efektywne, a jednocześnie minimalizowane jest ryzyko nieoczekiwanych modyfikacji prostych danych.

3 Funkcjonalności dodatkowe

3.1 Typ słownikowy

W języku zaimplementowano typ danych pozwalający na przechowywanie niehomogenicznych słowników, czyli struktur danych, w których zarówno klucze, jak i wartości mogą być różnych typów. Użytkownik może przypisać dowolną kombinację typów do par klucz-wartość, co umożliwia elastyczne przechowywanie różnorodnych danych. Obsługa słownika jest realizowana przez dedykowane funkcje do dodawania, usuwania i modyfikowania elementów. Słownik tworzony jest poprzez formułę `dict`.

```
dict myDict = { "name" : "Alice", "age" : 30, "is_student" :
    true, 101 : "student ID" };
print(myDict.get("name"));          # Alice
print(myDict.get("age"));           # 30
print(myDict.get("is_student"));    # true
print(myDict.get(101));             # student ID
```

Dodawanie nowych elementów: Nowe elementy można dodawać za pomocą funkcji `add`, która przyjmuje klucz i wartość:

```
myDict.add("graduation_year", 2025);
print(myDict.get("graduation_year"));    # 2025
```

Modyfikowanie wartości: Wartości przypisane do kluczy można modyfikować za pomocą funkcji `update`, która przyjmuje istniejący klucz i nową wartość:

```
myDict.update("age", 31);
print(myDict.get("age"));   # 31
```

Usuwanie elementów: Elementy ze słownika można usuwać za pomocą funkcji `remove`, która przyjmuje klucz elementu do usunięcia:

```
myDict.remove("is_student");
print(myDict.get("is_student"));   # null
```

3.2 Type Matching

Mechanizm type matching pozwala na dynamiczne wykonanie różnych operacji w zależności od typu przekazanej wartości. Jest to przydatne w sytuacjach, gdzie zachowanie programu powinno różnić się w zależności od typu danych.

Przykład zastosowania type matchingu:

```
fun process_value(object value) {
    value match {
        int => { print("This is an integer with value " + value)
    ;}
        float => { print("This is a float with value " + value)
    ;}
        string => { print("This is a string with value: " +
    value);}
        null => { print("This is a nullable");}
        _ => { print("Everything different"); }
    }
}

process_value(10);        # This is an integer with value 10
process_value(3.14);      # This is a floating-point number with
     value 3.14
process_value("Hello");   # This is a string with value: Hello
process_value(true);      # This is a boolean: true
process_value(null);      # Unknown type.
```

3.3 Rzutowanie danych

Język obsługuje rzutowanie typów, co pozwala na zmianę typu zmiennych w czasie wykonywania programu. Użytkownik może jawnie rzutować typy

danych, np. z `float` na `int` lub z `string` na `int`, jeśli takie rzutowanie jest logicznie możliwe.

Przykład rzutowania:

```
float number = 42.5;
int integer_number = int(number);  # Rzutowanie float na int
print(integer_number);  # 42
```

Jeśli rzutowanie nie jest możliwe (np. z `string` na `int` dla niepoprawnego ciągu znaków), interpreter wygeneruje błąd typu `TypeError`.

3.4 Wymagania Niefunkcjonalne

- Język programowania: Python
- Podział na moduły: lexer, parser, interpreter
- Proste uruchomienie z linii poleceń

4 Składnia języka

4.1 Definiowanie zmiennych

Zmienne w języku definiuje się poprzez podanie typu, nazwy zmiennej oraz przypisanie wartości. Przykład:

```
int age = 25;
string name = "Alice";
float temperature = 36.6;
```

W powyższym przykładzie, zmienne są przypisane do różnych typów: `int`, `string`, i `float`. Każda zmienna musi mieć jasno określony typ.

4.2 Instrukcja warunkowa

Instrukcje warunkowe pozwalają na wykonanie kodu w zależności od spełnienia określonego warunku. Przykład:

```
int age = 18;

if (age >= 18) {
    print("You are an adult.");
} else {
    print("You are a minor.");
}
```

W powyższym przykładzie, warunek sprawdza, czy zmienna `age` jest większa lub równa 18. W zależności od spełnienia tego warunku, wyświetlana jest odpowiednia wiadomość.

4.3 Pętle

Pętla `while` wykonuje kod tak długo, jak spełniony jest określony warunek. Przykład:

```
int count = 0;

while (count < 5) {
    print("Count is " + count);
    count = count + 1;
}
```

W tym przykładzie pętla `while` wykonuje się pięciokrotnie, zwiększając wartość zmiennej `count` za każdym razem i wyświetlając jej aktualną wartość.

4.4 Definiowanie funkcji

Funkcje definiuje się, podając nazwę funkcji, listę parametrów oraz blok kodu. Przykład:

```
fun add(int a, int b) {
    return a + b;
}

int result = add(5, 3);
print(result);  # 8
```

W powyższym przykładzie, funkcja `add` przyjmuje dwa argumenty typu `int` i zwraca ich sumę.

4.5 Type matching

Type matching pozwala na wykonanie różnych operacji w zależności od typu przekazywanej zmiennej. Przykład:

```
fun identify_object(object obj) {
    obj match {
        int => { print("This is an integer."); }
        string => { print("This is a string."); }
        null => {print("This is nullable."); }
        _ => { print("Unknown type."); }
    }
```

```
}

identify_object(10);  # This is an integer.
identify_object("Hello");  # This is a string.
identify_object(null);  # This is nullable.
identify_object(true);  # Unkonown type.
```

5 Gramatyka

```
program ::= { global_statement };

global_statement ::= function_def
                   | object_def ;

block_statement ::= declaration
                  | assignment
                  | if_statement
                  | while_loop
                  | type_match
                  | expression
                  | return_statement ;

object_def ::= "class", identifier, "{", { class_member }, "}" ;
class_member ::= declaration | function_def;

declaration ::= type, identifier, [ "=", expression ], ";" ;
assignment ::= obj_access, "=", expression, ";" ;

if_statement ::= "if", "(", expression, ")", block, [ "else", block ] ;
while_loop ::= "while", "(", expression, ")", block ;

return_statement ::= "return", [ expression ], ";" ;

function_def ::= func_type, identifier, "(", [ parameters ], ")", block ;
parameters ::= type, identifier, { ",", type, identifier } ;
identifier_or_function_call ::= identifier, [ "(", [ arguments ], ")" ];
arguments ::= expression, { ",", expression } ;

type_match ::= expression, "as", identifier, "match", "{", { match_case }, "}" ;
match_case ::= type, "=>", block
```

```
              | "_", "=>", block ;

block ::= "{", { block_statement }, "}" ;

expression ::= or_expression ;
or_expression ::= and_expression, { "||", and_expression } ;
and_expression ::= equality_expression, { "&&", equality_expression } ;
equality_expression ::= relational_expression, [ "==" | "!=", relational_expression ]
relational_expression ::= add_expression, [ "<" | ">" | "<=" | ">=", add_expression ]
add_expression ::= mul_expression, { "+", | "-", mul_expression } ;
mul_expression ::= unary_expression, { "*", | "/", unary_expression } ;
unary_expression ::= [ "-", | "not", | "!" ], type_expression ;
type_expression ::= factor, [ "is", type ] ;

factor ::= literal | "(", expression, ")", | obj_access ;

obj_access ::= item, { ".", item } ;
item ::= identifier_or_function_call ;

func_type ::= "void" | type ;

type ::= "int" | "float" | "bool" | "string" | custom_type ;
custom_type ::= identifier;

literal ::= integer | float | bool | string ;
float ::= integer, ".", digit, { digit } ;
integer  ::= zero | (non_zero_digit, {digit}) ;
bool ::= "true" | "false" ;
string ::= '"', { any_character - '"' }, '"' ;

identifier ::= letter, { letter | digit | "_" } ;

digit ::= non_zero | zero ;
non_zero ::= "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9" ;
zero ::= "0" ;
letter ::= "A" | "B" | "C" | ... | "Z" | "a" | "b" | "c" | ... | "z" ;
any_character ::= ? all visible characters ? ;
```

6 Obsługa Błędów

Interpreter wykorzystuje `ErrorManager` do klasyfikacji błędów ze względu na krytyczność, umożliwiając adekwatną reakcję na napotkane problemy.

6.1 Klasyfikacja Błędów

- **Krytyczne:** Zatrzymują wykonanie programu, np. `DivisionByZero`.
- **Niekrytyczne:** Informują o problemie, ale nie przerywają działania, np. `MissingSemicolon`.

6.2 Mechanizm Raportowania

`ErrorManager` dostarcza szczegółowe komunikaty o błędach, wskazując lokalizację oraz opis problemu. Każdy komunikat o błędzie zawiera:

- **Typ błędu** - kategoria błędu.
- **Opis** - szczegółowy opis błędu.
- **Lokalizacja** - wskazuje lokalizację błędu w kodzie, uwzględniając numer linii.

Poniżej przedstawiono przykład komunikatu o błędzie, który może zostać wygenerowany przez interpreter:

```
TypeError at line 15: Unsupported operand type(s) for '+': 'int' and 'str'
```

Ten komunikat informuje użytkownika o błędzie typu (`TypeError`), który wystąpił w linii 15. Błąd dotyczy próby użycia operatora dodawania (+) na dwóch operandach niekompatybilnych typów: `int` i `str`.

6.3 Przykłady Błędów

Kod:

```
1 fun main(){
2     x = 123$123;
3 }
```

Komunikat błędu:

```
SyntaxError at line 2: Unexpected character: '$'.
```

Kod:

```
fun main() {
    x 0;
}
```

Komunikat błędu:

```
SyntaxError at line 2: Missing '=' in variable initialization.
```

7 Sposób Uruchomienia

Aby korzystać z interpretera, upewnij się, że na Twoim systemie zainstalowana jest odpowiednia wersja Pythona. Interpreter został przetestowany i jest zgodny z Python 3.12 i nowszymi wersjami.

7.1 Uruchamianie interaktywnego interpretera

Aby uruchomić interaktywny tryb interpretera, otwórz terminal lub wiersz poleceń i wpisz następujące polecenie:

```
python my_interpreter.py
```

Po wpisaniu tego polecenia i naciśnięciu Enter, powinieneś zobaczyć powitanie oraz monit, w którym możesz zacząć wpisywać komendy.

7.2 Wykonanie skryptu

Aby uruchomić skrypt, musisz podać ścieżkę do pliku jako argument przy uruchamianiu interpretera. Przykładowe polecenie wygląda następująco:

```
python my_interpreter.py sciezka/do/pliku
```

Zamień `sciezka/do/pliku` na własną ścieżkę.

8 Sposób Testowania

Testowanie języka poprzez testy jednostkowe i integracyjne przy pomocy biblioteki `pytest`.

8.1 Instalacja `pytest`

Przed przystąpieniem do uruchamiania testów, należy upewnić się, że `pytest` jest zainstalowany. Można to zrobić za pomocą poniższego polecenia:

```
pip install pytest
```

8.2 Uruchamianie testów

Aby uruchomić testy za pomocą `pytest`, należy otworzyć terminal w katalogu głównym projektu i wykonać polecenie:

```
pytest
```

`pytest` automatycznie znajdzie i uruchomi wszystkie testy zdefiniowane w plikach, których nazwy zaczynają się od `test_` lub kończą na `_test`, w katalogu bieżącym i jego podkatalogach.