Dokumentacja Interpretera Języka z Definiowaniem Obiektów i Type Matchingiem

Mikołaj Taudul

March 22, 2024

1 Wstęp

Język programowania skoncentrowany na definiowaniu i manipulacji obiektami oraz zastosowaniu mechanizmu type matching. Jego główną cechą jest umożliwienie użytkownikom tworzenia własnych typów obiektów, wraz z ich atrybutami i metodami. Kluczową funkcjonalnością jest mechanizm type matching, który pozwala na wykonanie różnych operacji w zależności od typu przekazanego obiektu.
Interpreter dla tego języka jest tworzony w Pythonie.

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 **dynamiczne i silne typowanie**. Zmienne są **mutowalne**.

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, obiekt).

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 Funkcjonalności dodatkowe

Definiowanie obiektów Język umożliwia użytkownikom definiowanie własnych obiektów. Definicja obiektu może zawierać konstruktory, atrybuty i metody.

Przykład:

```
class Student {
    var name;
    fun greet() {print("Hello, " + name)};
}
```

Type Matching Mechanizm type matching pozwala na dynamiczne wykonanie różnych operacji w zależności od typu obiektu. Jest to przydatne w obsłudze struktur danych i scenariuszy, gdzie zachowanie programu powinno się różnić w zależności od konkretnego typu przekazanego obiektu.

Przykład:

```
fun process_object(obj) {
    obj match {
        Student => {print("Hello student ", obj.name)}
        Teacher => {print("Good morning sir")}
        _ => {print("Unknown object type")}
    }
}
```

2.3 Wymagania Niefunkcjonalne

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

3 Składnia języka

3.1 Definiowanie zmiennych

Jak definiować zmienne. Na przykład:

```
var name = "Ala";
```

3.2 Instrukcja warunkowa

Przedstawienie składni instrukcji warunkowych. Na przykład:

```
if (warunek){
    instrukcja;
} else {
    instrukcja;
};
```

3.3 Pętle

Opis składni pętli while w języku:

```
while (warunek) {
    instrukcje;
}
```

3.4 Definiowanie funkcji

Jak definiować funkcje:

```
fun nazwaFunkcji(parametry) {
    instrukcje;
}
```

3.5 Definiowanie obiektów

Opis składni dotyczącej definiowania obiektów:

```
class nazwa_obiektu {
    var nazwa_atrybutu;
    fun nazwa_metody();
}
```

3.6 Type matching

Opis składni dotyczącej type matchingu:

```
fun process_object(obj) {
    obj match {
        typ => {instrukcja;}
        typ => {instrukcja;}
    }
}
```

Przykład z wykorzystaniem **is**:

```
var ten = 10;
print(ten is int); #true
```

4 Gramatyka

```
program ::= { global_statement };

global_statement ::= declaration
                   | assignment
                   | if_statement
                   | while_loop
                   | function_def
                   | type_match
                   | expression_statement
                   | object_def ;

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

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

declaration ::= "var", identifier, [ "=", expression ], ";" ;
assignment ::= identifier, "=", expression, ";" ;
```

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

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

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


type_match ::= identifier, "match", "{", { match_case }, "}" ;
match_case ::= type, "=>", block
             | "_", "=>", block ;

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

expression_statement ::= expression, ";" ;

expression ::= logical_expression ;
logical_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 ;


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 ? ;
```

5 Obsługa Błędów

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

5.1 Klasyfikacja Błędów

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

5.2 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`.

5.3 Przykłady Błędów

Kod:

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

Komunikat błędu:

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

Kod:

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

Komunikat błędu:

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

6 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.

6.1 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.

6.2 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.

7 Sposób Testowania

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

7.1 Instalacja `pytest`

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

```
pip install pytest
```

7.2 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.