

TKOM Projekt Final

Anton Masiukevich

Maj 2022

1 Opis

Implementacja interpretera języka programowania umożliwiającego tworzenie obiektów o zmiennej strukturze oraz operowanie na tych obiektach.

2 Funkcjonalność

- Obsługa definiowania funkcji i klas
- Obsługa zmiennych lokalnych i globalnych
- Obsługa tworzenia obiektu danej klasy
- Obsługa dodawania oraz modyfikacji wartości atrybutów obiektu bez sztywnej definicji w postaci pól klasy
- Obsługa deklaracji zmiennych
- Obsługa zaawansowanych wyrażeń arytmetycznych i logicznych z implementacją operacji porównania, mnożenia, dzielenia, modulo, dodawania, odejmowania, negacji, alternatywy, koniunkcji
- Obsługa priorytetów operatorów
- Obsługa instrukcji warunkowych
- Obsługa pętli **while** oraz **foreach**
- Obsługa manipulacji na atrybutach klasy (dodawanie, wypisanie, modyfikacja)
- Obsługa rzutowania typów
- Obsługa komentarzy
- Obsługa zapytania obiektu o istnienie atrybutu (metoda **has_attr()**)
- Możliwość wypisania wartości wyrażenia na konsolę

- Możliwość wypisania atrybutów klasy (pierwszego poziomu i rekursywnego)
- Możliwość rozpoczęcia wykonania programu od podanej funkcji

3 Założenia

- Kod programu powinien się znajdować wyłącznie w jednym pliku
- Plik powinien zawierać co najmniej jedną funkcję nie przyjmującą argumentów, od której się zacznie wykonywanie programu
- Język nie obsługuje pracy na wskaźnikach, operacji na fragmentach pamięcią
- Interpreter języka ignoruje znaki białe
- Język nie obsługuje wszystkich kombinacji rzutowania typów

4 Gramatyka

```

program ::= declaration, {declaration};

# In the IDLE mode there will be a function
declaration ::= function_definition | class_definition | statement;

function_definition ::= identifier, "(", parameters, ")", block;
class_definition ::= "class", identifier, class_block;

parameters ::= [ { this_kw | identifier }, { ",", identifier } ] ;
block ::= "{", {statement}, "}" ;

class_block ::= "{", {function_definition}, "}" ;

statement ::=      conditional |
                  loop |
                  return |
                  assign |
                  function_call |
                  comment ;

conditional ::= "if", or_expression, block,
               { "else if", or_expression, block },
               ["else", block] ;

loop ::= foreach_loop | while_loop ;
return ::= "return", {or_expression}, ";" ;
comment ::= "#", comment_body;

assign ::= complex_getter, "=", or_expression;
function_call ::= complex_getter, ";" ;

```

```

foreach_loop ::= "foreach", identifier, "in", or_expression, block;
while_loop  ::= "while", or_expression, block;

# Expressions
or_expression ::= and_expression, { or_oper, and_expression } ;
and_expression ::= equality_expression, { and_oper,
    equality_expression } ;
equality_expression ::= relation_expression, [ eq_oper,
    relation_expression ] ;
relation_expression ::= add_expression, [ rel_oper, add_expression
    ] ;
add_expression ::= mult_expression, { add_oper, mult_expression } ;
mult_expression ::= unary_expression, { gen_mult_oper,
    unary_expression } ;
unary_expression ::= [unary_oper], generalized_value;
generalized_value ::= "(", or_expression, ")" | value ;

# Value getting stuff
complex_getter ::= basic_getter, {access_oper, iterative_getter} ;
basic_getter  ::= (this_kw | iterative_getter) ;
iterative_getter ::= identifier, [rest_of_funcnt_call], ["[",
    add_expression, "]" ] ;
rest_of_funcnt_call ::= "(", arguments, ")" ;
arguments ::= { or_expression } ;

this_kw ::= "this"
comment_body ::= {special_char | string | number | other_char},
    newline;

identifier ::= (underscore | dollar_sign | letter), {(letter |
    digit | underscore)} ;

# Operators
access_oper ::= "." ;
or_oper ::= "||" ;
and_oper ::= "&&" ;
eq_oper ::= "==" |
    "!=" ;
rel_oper ::= ">" |
    "<" |
    ">=" |
    "<=" ;

add_oper ::= "+" |
    "-" ;

gen_mult_oper ::= mult_oper |
    "%" ;

mult_oper ::= "*" |
    "/" ;

unary_oper ::= neg_oper |
    not_oper ;
neg_oper ::= "-" ;

```

```

not_oper ::= "!" ;

value ::= literal | complex_getter ;

# Literals

literal ::= string |
           number |
           bool |
           "null";

string = "\"", {character}, "\"" ;

character = ( letter | digit | special_char ) ;
number = ( integer_part | float_number ) ;
float_number = integer_part, ".", fractional_part ;

integer_part ::= non_zero_number |
                 "0";
fractional_part ::= digit, {digit} ;

bool ::= "true" | "false" ;

non_zero_number ::= non_zero_digit, {digit} ;
digit ::= non_zero_digit |
          "0" ;
non_zero_digit ::= '1' - '9' ;

letter ::= 'A' - 'Z' |
          'a' - 'z' ;

special_char ::= ":" |
                 ";" |
                 "\" |
                 "'" |
                 "-" |
                 "." |
                 "," |
                 "/" |
                 "\\" |
                 "#" |
                 dollar_sign |
                 underscore ;

other_char ::= " " |
              "\t" ;

dollar_sign ::= "$" ;
newline ::= "\n" ;
underscore ::= "_" ;

```

5 Moduły

5.1 Moduł źródła

Abstrakcja źródła danych. Przekazuje do analizatora leksykalnego kolejne znaki oraz sygnalizuje koniec odczytu. W przypadku błędu zwróci pozycję zaistniałego błędu.

5.2 Moduł analizatora leksykalnego (Scanner)

Moduł analizatora leksykalnego jest odpowiedzialny za pobieranie ze źródła kolejnych symboli i tworzenia z nich tokenów języka. Przekazuje utworzone tokeny do analizatora składniowego.

5.3 Moduł analizatora składniowego (Parser)

Moduł ten pobiera od modułu analizatora leksykalnego kolejne tokeny i na ich podstawie buduje drzewo programu zgodnie z gramatyką języka. Drzewo zostaje przekazane do modułu interpretera. Typ analizatora - to analizator rekursywny zstępujący.

5.4 Moduł interpretera

Niniejszy moduł jest odpowiedzialny za wykonanie programu. Pobiera drzewo programu od modułu analizatora składniowego. Rozpoczyna wykonanie od funkcji podanej przez użytkownika w argumentach wiersza polecenia (domyślnie: *main*), wykonując instrukcje zapisane w kodzie. Jest również odpowiedzialny za analizę semantyczną. Do implementacji został wykorzystany wzorzec *Od-wiedzający* (Visitor).

5.5 Moduł wyjątków

Moduł zawierający implementację wyjątków wykorzystywanych przez poszczególne moduły (na różnych etapach interpretacji).

5.6 Moduł dodatków

Moduł zawiera implementację wszelkich niezbędnych komponentów, na których operują moduły programu: struktury danych oraz słowniki. M.in. abstrakcyjne drzewo rozbioru programu, struktury danych niezbędne do funkcjonowania źródła, interpretera.

6 Struktury Danych

6.1 Token

Token jest strukturą danych utworzoną przez analizator leksykalny na podstawie kolejnych symboli za źródła. Typy tokenów są zdefiniowane na podstawie gramatyki. Token jako struktura przechowuje swój typ, pozycję wystąpienia w kodzie oraz wartość (w przypadku literalów boolowskich oraz numerycznych - odpowiedniego typu, w pozostałych przypadkach - wartości typu string), istnieje specjalny typ atrybutu "null".

6.2 Pozycja

Jest to struktura przechowująca pozycję w kodzie. Użytkowana w obsłudze wyjątków. Zawiera informację w postaci numeru linii i kolumny w kodzie bieżącego obsługiwanego symbola.

6.3 Drzewo programu

Drzewo programu jest strukturą danych utworzoną przez analizator składniowy na podstawie tokenów wejściowych (otrzymanych od modułu analizatora leksykalnego). Reprezentuje ona pełne drzewo rozbioru interpretowanego programu, w którym węzłami są poszczególne instrukcje / wartości / zmienne, krawędzie zaś są reprezentowane przez relację zawierania. Na jej podstawie są wykonywane instrukcje przez interpreter.

6.4 Środowisko (Environment)

Środowisko jest "pamięcią dynamiczną" interpretera. Przechowuje stos kontekstów wywołań, realizuje metody dodawania nowych kontekstów, dostępu do zmiennych, przechowuje fakt zwracania i zwracane wartości po powrotach z wywołań funkcji/metod.

6.5 Kontekst wywołania (Call Context)

Jest strukturą danych przechowującą stos zakresów widoczności poszczególnych zmiennych w ramach wywołania funkcji/metody. Realizuje metody dodawania nowych zakresów, dostępu do zmiennych w ramach kontekstu wywołania.

6.6 Zakres widoczności (Scope)

Zakres przechowuje mapę zmiennych zdefiniowanych w ramach siebie. Realizuje metody dostępu do nich.

6.7 Instancja

Instancja jest abstrakcją obiektu języka programowania. Zawiera informacje o tym, jakiej klasy jest to instancja, zakres, w którym są widoczne pola i metody danego obiektu.

7 Obsługa błędów

Na poszczególnych poziomach program różnie reaguje na zaistniałą sytuację wyjątkową. Komunikat o błędzie zawiera dostosowaną do okoliczności wyjątku wiadomość bazową, wyświetlona zostanie również pozycja błędu z kawałkiem tekstu w programie, obok którego błąd zaistniał.

8 Testowanie

Do testowania modułów źródła, analizatora leksykalnego, analizatora składniowego zostały napisane testy jednostkowe.

9 Sposób realizacji

Program został napisany w zupełności w języku programowania **Python**, z wykorzystaniem modułów biblioteki standardowej tego języka:

- **io** - do modułu źródła
- **sys** oraz **argparse** - do obsługi zmiennych z wiersza polecenia
- **unittest** - do testów jednostkowych

10 Sposób uruchomienia

Program jest uruchamiany w konsoli poleceniem:

```
#!/bin/bash  
python main.py plik_z_kodem_źródłowym [-sf main]
```

Opcjonalny argument *-sf* (start function) oznacza nazwę funkcji, od której powinno się zacząć wykonanie programu (domyślnie *main*). Program może wyświetlać ewentualne komunikaty o błędach.

11 Przykładowe programy

11.1 Przykład 1

- Definicja funkcji
- Definicja zmiennej lokalnej
- Wywołanie funkcji
- Wypisanie wyniku

```
calc_price_total(item_price, tax) {  
    return item_price * (1 + tax / 100);  
}  
  
main() {  
    total_price = calc_price_total(1000, 10);  
    print(total_price);  
}
```

11.2 Przykład 2

- Definicja klas
- Stworzenie obiektów
- Dynamiczne dodanie atrybutu do obiektu
- Wywołanie metody obiektu

```
class Person {  
  
    Person(this, name, age) {  
        this.name = name;  
        this.age = age;  
    }  
  
    get_name(this) {  
        return this.name;  
    }  
}  
  
class Pet {  
  
    Pet(this, name) {  
        this.name = name;  
    }  
  
    get_name(this) {  
        return this.name;  
    }  
}
```



```

main() {

    david = Person("David", 45);
    hannah = Person("Hannah", 40);

    sarah = Person("Sarah", 15);
    andy = Person("Andy", 21);
    gosha = Pet("Gosha");

    david.child1 = sarah;
    david.child2 = andy;

    david.child1 = andy;

    hannah.child = sarah;
    sarah.pet = gosha;

    print(hannah.child.pet.get_name());

}

```

11.3 Przykład 3

- Użycie wywołania funkcji jako argumentu innego wywołania
- Użycie pętli while

```

objective(x, A, B, C) {
    return A * x * x + B * x + C;
}

gradient(x, A, B, C) {
    return 2 * A * x + B;
}

step_gradient(x, grad, rate) {
    return x - rate * grad;
}

main() {

    i = 0;
    point = -10000;
    A = 1;
    B = 2;
    C = -3;
    rate = 0.01;

    # Gradient descend
    while i < 1000 {
        loss = objective(point, A, B, C);
        point = step_gradient(point, gradient(point, A, B, C), rate
    );
}

```

```

        if i % 100 == 0 {
            print("Iteration " + i);
            print("Current point: " + point);
            print("Current loss: " + loss);
            print();
        }

        i = i + 1;
    }
}

```

11.4 Przykład 4

- Wypisanie atrybutów pierwszego poziomu
- Wypisanie atrybutów na wszystkich poziomach
- Użycie pętli foreach

```

# Killer comment

class Person {

    Person(this, name, age) {
        this.name = name;
        this.age = age;
    }
}

main() {

    david = Person("David", 85);
    andy = Person("Andy", 45);
    mike = Person("Mike", 15);
    emily = Person("Emily", 18);

    david.son = andy;
    andy.son = mike;
    andy.daughter = emily;

    foreach attr in david.attributes() {
        print(attr);
    }

    print();

    foreach attr in david.rec_attributes() {
        print(attr);
    }
}

```

11.5 Przykład 5

- Start wykonania programu z innej funkcji niż main

```
main(a, b) {  
    return a * b;  
}  
  
perform_calc() {  
    a = 15;  
    calculation = main(a, 15);  
    print(calculation);  
}
```

11.6 Przykład 6

- Użycie rekurencji

```
factorial (n) {  
    if n == 0 || n == 1 {  
        return 1;  
    } else {  
        return n * factorial(n - 1);  
    }  
}  
  
fibonacci(n) {  
    if n == 0 {  
        return 0;  
    } else if n == 1 {  
        return 1;  
    } else {  
        return fibonacci(n - 1) + fibonacci(n - 2);  
    }  
}  
  
main() {  
    print(fibonacci(10));  
    print(factorial(5));  
}
```