

JP - Język Programowania

- Radosław Kostrzewski, 310757
- Gitlab: @rkosrze

Wymagania

- Język obiektowy
- Możliwość rozszerzania języka o nowe typy
- Wszystkie zmienne są nullovalne
- Typowanie dynamiczne i silne
- Obsługa wyjątków
 - Możliwość rzucenia wyjątku
 - Możliwość wyłapania wyjątku
 - * Przechwycenie wyjątku bez dostępu do szczegółowych informacji
 - * Przechwycenie wyjątku po jednym lub kilku typach z dostępem do szczegółowych informacji
- Każde wyrażenie musi zakończyć się średnikiem
- Rzutowanie na inne typy odbywa się poprzez wywołanie konstruktora typu
- Gdy zmienna zostanie zadeklarowana ponownie w tym samym scope zostanie ona zastąpiona nową wartością. Wyjątkiem są zmienne tworzone przy łapaniu wyjątku oraz przy tworzeniu pętli for, wtedy zostanie zgłoszony błąd.
- Gdy funkcja zostanie zadeklarowana ponownie w tym samym scope zostanie zgłoszony błąd
- Po wyjściu ze scopeu identyfikator zostaje usunięty

Wbudowane typy

- Podstawowe typy
 - int
 - float
 - string
 - bool
 - null
- Typy złożone
 - Array
 - * Metody
 - add(element)
 - remove(element)
 - removeAt(index)
 - clear()
 - get(index)
 - set(index, element)
 - size()
 - contains(element)

· indexOf(element)

Wbudowane funkcje

- print(args); - wypisuje na ekranie wartości przekazane jako argumenty
- read(); - odczytuje wartość z konsoli i zwraca ją jako string
- exit(); - zamyka program

Właściwości zmiennych

- Wszystkie zmienne są nullowalne
- Zmienne domyślnie są typu null i są mutowalne
- Zmienne mogą być typu const, wtedy ich wartość nie może zostać zmieniona. Gdy zmienna jest const należy ją zainicjalizować przy jej tworzeniu.

Operatory i ich priorytety

Priorytety operatorów są od najwyższego do najniższego. Priorytety operatorów są uzależnione od zakresu w jakim się znajdują.

1. Operator referencji (@)
2. Operator dostępu do elementu obiektu (., ?.)
3. Operator negacji (!)
4. Operatory mnożeniowe (*, /, %)
5. Operatory adytywne (+, -)
6. Operatory porównania (==, !=, <, >, <=, >=)
7. Logiczny AND (&)
8. Logiczny OR (|)
9. Operator przypisania (=, +=, -=, *=, /=, %=)

Instrukcje warunkowe

- if (warunek) { instrukcje }
- if (warunek) { instrukcje } else { instrukcje }
- if (warunek) { instrukcje } else if (warunek) { instrukcje } else { instrukcje }

warunek może być wyrażeniem logicznym lub wyrażeniem zwracającym wartość typu bool

Pętle

- while (warunek) { instrukcje }
- for (zmienna: tablica) { instrukcje }

warunek może być wyrażeniem logicznym lub wyrażeniem zwracającym wartość typu bool

Rzucanie wyjątków

Aby rzucić wyjątek należy użyć słowa kluczowego `throw` a następnie wywołać konstruktor klasy błędu. Konstruktor klasy błędu będzie zawierał dwa opcjonalne pola, pierwszym będzie komunikat który ma zostać wyświetlony po rzuceniu wyjątku, a drugim wartość zmiennej której dotyczy błąd.

```
throw ArgumentError("Argument x cannot be negative. Value of x  
is: ", x);
```

Error message będzie wyglądał mniej więcej tak:

```
[ARGUMENT ERROR] Argument x cannot be negative. Value of x is: -1 in x:y ->  
throw ArgumentError("x cannot be negative", x);
```

Wyłapywanie wyjątków

Aby wyłapać wyjątek musimy umieścić metodę która rzuca wyjątek w nawiasach klamrowych po słowie kluczowym `try`. Funkcję która ma się wywołać po rzuceniu wyjątku umieszczamy w nawiasach klamrowych po słowie kluczowym `catch`.

Słowo `catch` bez żadnych argumentów będzie przechwytywało każdy wyjątek bez dostępu do jego szczegółowych informacji.

Słowo `catch` z argumentem typu `Error` będzie przechwytywało każdy wyjątek i zapisze jego szczegółowe informacje w tym argumentcie.

Słowo `catch` z argumentem o szczegółowym typie błędu będzie przechwytywało wyjątek tego samego typu co argument i zapisze jego szczegółowe informacje w tym argumentcie.

Możemy przechwycić wiele wyjątków w jednym bloku `catch`.

Może być wiele słów kluczowych `catch` w jednym bloku `try`.

Wyjątek rzucony w scopie występującym po słowie kluczowym `catch` musi zostać obsługiwany oddzielnie.

Komentarze

Komentarz jednoliniowy

Analiza leksykalna

Analizator leksykalny (Lexer) jest częścią kompilatora odpowiedzialną za podział kodu źródłowego na tokeny. Lexer będzie działał leniwie, czyli będzie odczytywał kod znak po znaku i tworzył tokeny dopiero gdy będzie miał wystarczającą ilość znaków do stworzenia tokena.

Tokeny

```
class TokenType(Enum):
    # --- Literals ---
    T_INT_LITERAL = 256
    T_FLOAT_LITERAL = 257
    T_STRING_LITERAL = 258
    T_BOOL_LITERAL = 259
    T_NULL_LITERAL = 260

    # --- Functions ---
    T_RETURN = 261
    T_BREAK = 262
    T_CONTINUE = 263

    # --- Statements ---
    T_IF = 264
    T_ELSE = 265
    T_WHILE = 266
    T_FOR = 267

    # --- Operators ---

    # ----- Arithmetic -----
    T_PLUS = 268 # +
    T_MINUS = 269 # -
    T_MULTIPLY = 270 # *
    T_DIVIDE = 271 # /
    T_MODULO = 272 # %

    # ----- Assignment -----
    T_ASSIGN = 273 # =
    T_ASSIGN_PLUS = 274 # +=
    T_ASSIGN_MINUS = 275 # -=
    T_ASSIGN_MULTIPLY = 276 # *=
    T_ASSIGN_DIVIDE = 277 # /=
    T_ASSIGN_MODULO = 278 # %=

    # ----- Comparison -----
    T_GREATER = 279 # >
    T_LESS = 280 # <
    T_GREATER_EQUAL = 281 # >=
    T_LESS_EQUAL = 282 # <=
    T_EQUAL = 283 # ==
    T_NOT_EQUAL = 284 # !=
```

```

# ----- Logic -----
T_AND = 285 # &
T_OR = 286 # |

# ----- Access -----
T_ACCESS = 287 # .
T_NULLABLE_ACCESS = 288 # ?.

# ----- Other -----
T_NOT = 289 # !
T_REF = 290 # @

# --- Brackets ---
T_LEFT_BRACKET = 291 # (
T_RIGHT_BRACKET = 292 # )
T_LEFT_CURLY_BRACKET = 293 # {
T_RIGHT_CURLY_BRACKET = 294 # }

# --- Other ---
T_IDENTIFIER = 295
T_SEMICOLON = 296 # ;
T_COMMA = 297 # ,
T_COLON = 298 # :
T_COMMENT = 299 # #
T_OPTIONAL = 300 # ?
T_EOF = 301

```

Analiza składniowa

Analizator składniowy (Parser) jest częścią kompilatora odpowiedzialną za sprawdzenie poprawności składni kodu źródłowego. Parser będzie odczytywał tokeny jedno po drugim i tworzył drzewo składniowe.

Interpreter

Interpreter jest częścią kompilatora odpowiedzialną za wykonanie kodu źródłowego. Interpreter będzie wykonywał kod źródłowy w oparciu o drzewo składniowe.

Gramatyka

Dokument zawierający gramatykę języka znajduje się w folderze **grammar**. Znajduje się tam zarówno gramatyka w formacie EBNF jak i graficzna reprezentacja.

Testowanie

Testy będą wykonywane za pomocą biblioteki `pytest`. Testy będą umieszczone w folderze `tests` w pliku o nazwie odpowiadającej nazwie pliku z kodem źródłowym. Testy będą wykonywane za pomocą komendy:

```
pytest
```

Uruchamianie

Aby uruchomić interpreter należy uruchomić skrypt pythonowy interpretera. Interpreter będzie uruchamiany za pomocą komendy:

```
./interpreter.py [plik]
```

Przykładowe programy

Przykładowe programy będą umieszczone w folderze `code_examples`.

Przykładowe wbudowane błędy

[VALUE ERROR] Value of constant variable cannot be changed in x:y -> **x = 2;**

Dla kodu:

```
x const = 1;  
x = 2;
```

[ARGUMENT ERROR] Mandatory argument of function **addOne(x)** wasn't provided in x:y -> **addOne();**

Dla kodu:

```
addOne(x) {  
    return x + 1;  
}
```

```
addOne();
```

[ERROR] Missing semicolon in x:y -> **x = 2**

[ERROR] Const variable wasn't initialized in x:y -> **x const;**

[ERROR] Use of unknown variable in x:y -> **x = y + 2;**

Dla kodu:

```
x = 2;  
x = y + 2;
```