Projekt TKOM – Dokumentacja wstępna

Bartłomiej Niewiarowski, semestr 2024L

Koncepcja

Temat:

Celem projektu jest wykonanie interpretera własnego języka programowania ogólnego przeznaczenia.

Projekt będzie realizowany z użyciem języka Python z którego to będą czerpane poszczególne mechanizmy działania tworzonego języka.

Założenia:

- Typowanie dynamiczne i słabe
- Zmienne mutowalne
- Argumenty przekazywane do funkcji przez wartość
- Kolekcja w postaci listy jako typ wbudowany, lista będzie mogła zawierać elementy różnych typów
- Na kolekcji będzie można wykonywać operacje podobne do mechanizmu LINQ
- Możliwe będzie importowanie zewnętrznych bibliotek

Wbudowane typy języka:

- bool – typ logiczny
- int – liczba całkowita, o zakresie wartości: +-(2^63-1)
- float – liczba zmienno przecinkowa, z dokładnością do 15 cyfr dziesiętnych
- string - ciąg znaków
- array – lista elementów

Założenia funkcjonalne i niefunkcjonalne:

- Każdy program napisany w języku będzie musiał zawierać funkcję main() która to będzie zawierać główne ciało programu.
- Funkcja main nie musi być definiowana w konkretnym miejscu w kodzie
- Każda linia kodu musi się kończyć ; a w przypadku funkcji, jej ciało będzie zawierać się w { }
- Konstrukcja pętli:

```
While ( *warunek kończący* )
{
    *instrukcja*
}
```

- Konstrukcja break umożliwiająca przerwanie pętli.

- Konstrukcja instrukcji warunkowych:

  if (*warunek*) {

  instrukcja

  }

  else {

  instrukcja

  }

- Wypisywanie tekstu na ekranie:

  print(*tekst*);

- Wczytywanie wejścia użytkownika:

  input();

  funkcja input czeka na wpisanie przez użytkownika wejścia, wejście wczytywane jest jako tekst, wciśnięcie enter będzie oznaczać koniec wprowadzania wejścia, funkcja będzie mogła przyjmować ciąg tekstowy jako monit ktory powinna wyswietlic

- Kolejność wykonywania operacji:

  1. * , /

  2. + , -

  3. ==, !=, <, >, <=, >=

  4. and

  5. or

  Operacje zawarte w nawiasach będą wykonywane w pierwszej kolejności

- Język umożliwia pisanie funkcji z wykorzystaniem rekurencji, z ograniczeniem rekursji do 1000 wywołań. // mozliwosc konfiguracji max ilosci wywolan, kwestia przypisania
- Argumenty funkcji przekazywane są przez wartość dla typów prostych: bool, int, string, float oraz przez referencję dla typu złożonego tablicy
- Funkcje nie mogą być przeciążane
- Definicja funkcji:

  def nazwa_funkcji(*parametry*)

  {

  instrukcje

  return; / return expression;

  }

- Wywołanie funkcji:

  nazwa_funkcji(argumenty);

- Komentarze jednolinijkowe, zaczynające się od #

- Definiowanie zmiennej:

  nazwa_zmiennej = wartość;

- Definiowanie kolekcji:

  nazwa_zmiennej = [wartość_a, wartość_b, ...];

  kolekcja iterowana od 0

- Wykonywanie operacji na kolekcji:
  1. append - dodanie elementu na końcu kolekcji
  2. remove(n) - usunięcie n-tego elementu z kolekcji
  3. foreach
  4. where
  5. sort - dla listy zawierającej typy liczbowe, int lub float.
  6. get - odczytanie elementu znjadującego się pod danym indeksem.
- Kontakencja na stringu:

  imie = ‘Bartek’, nazwisko = ‘Niewiarowski’

  imie + nazwisko = BartekNiewiarowski

- Długość stringa będzie ograniczona do miliarda znaków, rozwiązanie to umożliwia obsługę znacznej ilości danych tekstowych, a jednocześnie zapewnia bezpieczne ograniczenie, które nie powinno prowadzić do problemów z wydajnością ani przepełnieniem pamięci.
- Długość identyfikatora będzie ograniczona do 40 znaków, zapewni to czytelność i zrozumienie w nazwach funkcji, a zarazem uchroni przed zbyt długimi identyfikatorami
- Brak zmiennych globalnych
- Zmienne deklarowane tylko i wyłącznie w ciele funkcji, { }
- Możliwe będzie powoływanie obiektów innych typów, typy te są definiowane w innym języku, nasz interpreter importuje dane typy z zewnętrznej biblioteki. Możliwe będzie powołanie nowego obiektu oraz wykonanie na nim podstawowych operacji:

  Załóżmy, że importujemy typ Student:

  from School import Student;

  nowy_uczen = Student();

Gramatyka:

- Gramatyka ze względu na czytelność opisana w innym pliku

Przykładowy kod:

```
from school import Student;

varable = 10; # blad, zmienna zadeklarowana w niedozwolonym miejscu

def exampleFunction () {
    variable_a = 4;
    variable_b = 5;
    return(variable_a + variable_b);
}

def fibonacci (x) {
    #przyklad funckji napisanej z wykorzystaniem rekurencji
    if (x <= 1) {
        return x;
    }
    else {
        return( fibonacci(x-1) + fibonacci(x-2));
    }
}

def listOperations() {
    # przykłady operacji na liscie
    numbers = [1, 2, 3, 4, 5];
    numbers.append(6);

    numbers.foreach(num =>
    {
       if(num != 3) {
           print(num);
       }
    });
    numbers.sort();

    numbers.append("7");
    number.sort() #podczas wykonania '7' konwertuje sie na liczbe

    numbers.append([1, 2, 3]);
    numbers.sort() #blad niezgodnosc typow
}

```

```
41 def operacjeNaZmiennych() {
42     A = 5;
43     B = 1.55;
44     C = A + B; #C = 6.55
45 
46     D = "7";
47     E = A + D; # konwersja D na liczbe calkowita
48 
49     F = "abc";
50     G = A + F; #blad, niezgodnosc typow
51 
52     H = [1, 2, 3];
53     I = A + H; #blad, niezgodnosc typow
54 }
55 
56 def operacjeLogiczne() {
57     if(5 >= 3) {
58         print("Tak");
59     }
60     else {
61         print("Nie");
62     }
63     A = 5;
64     zmienna_logiczna = A && True;
65     A = 5.55;
66     zmienna_logiczna = A && True; # blad, typ float niedozwolony
67 }
```

```
69 def main() {
70     A = 5;
71     B = 10;
72 
73     print("Podaj imie:");
74     imie = input();
75     print("Podaj nazwisko:");
76     nazwisko = input();
77 
78     dane = imie + " " + nazwisko;
79     print(dane);
80 
81     C = A + B;
82     print(fibonacci(C));
83 
84     print(D); #blad, nieznana zmienna
85 
86     C = C/0; #blad, dzielenie przez 0
87 
88     uczen = Student();
89 
90     jakasFunkcja(); #blad, wywolanie nieistniejacej funkcji
91 
92     result1 = fibonacci(2, 3, 4); # blad, niepoprawna ilsoc argumentow
93 
94     result2 = fibonacci("ABCDE") # blad, niepoprawne argument wywoalnia funkcji
95 }
```

Rodzaje komunikatów o błędach:

- Błędy składniowe
    - Rodzaj błędu: Nieprawidłowa składnia w kodzie.
    - Obsługa: Wyświetlenie komunikatu o błędzie z informacją o nieprawidłowej składni w kodzie wraz z ewentualnym wskazaniem na miejsce w kodzie, gdzie błąd napotkano.
- Błędy semantyczne
    - Rodzaj błędu: Nieprawidłowe użycie konstrukcji języka, które są poprawne składniowo, ale niepoprawne semantycznie.
    - Obsługa: Wyświetlanie komunikatu o błędzie z opisem problemu oraz, jeśli to możliwe, wskazaniem na miejsce w kodzie, gdzie wystąpił błąd semantyczny.
- Błędy wewnętrzne:
    - Rodzaj błędu: Błąd, który wystąpił wewnętrznie w kompilatorze lub interpreterze.
    - Obsługa: Wyświetlanie komunikatu o błędzie z informacją o przyczynie

Przykłady komunikatów:

- Błędy składniowe:
    - Nieprawidłowa instrukcja. Oczekiwano zakończenia linii lub średnika.
    - Nieprawidłowe użycie operatora w linii: x.
- Błędy semantyczne:

  Błędy związane ze zmiennymi:

    - Zmienna x zadeklarowana, jednak nigdy nie została użyta.
    - Użyta, ale nie zadeklarowana
    - Deklaracja zmiennej w niedozwolonym miejscu

    Błędy związane z operacjami arytmetycznymi i logicznymi:

    - Dzielenie przez 0
    - Niezgodność typów

  Błędy związane z funkcjami:

    - Wywołanie niezdefiniowanej funkcji
    - Redefinicja funkcji
    - Błędne argumenty wywołania funkcji.

    Błędy związane z wbudowanymi ograniczeniami:

    - Zbyt duża wartość zmiennej liczbowej
    - Zbyt duża długość zmiennej tekstowej
    - Zbyt duża długość identyfikatora
    - Zbyt duża ilość wywołań rekurencyjnych

Struktura projektu:

- Analizator leksykalny:

Analizator leksykalny będzie przyjmował dwa rodzaje źródeł danych: plik oraz ciąg znaków. Głównym zadaniem lexera będzie przekształcenie kodu źródłowego na strumień tokenów w których można wyróżnić: identyfikatory, słowa kluczowe, operatory, liczby, znaki specjalne. Lexer będzie również odpowiedzialny za identyfikację błędów leksykalnych.

- Analizator składniowy:

Analizator składniowy będzie przekształcał strumień tokenów wyprodukowanych przez analizator składniowy, w drzewo składniowe.

Konwersja typów:

Konwersja będzie realizowana w momencie próby wykonania na danej zmiennej działania wraz z innym typem zmiennej. W niektórych przypadkach będzie to niemożliwe, w tej sytuacji zostanie zwrócony określony błąd.

typ A	**typ B**	**operacje**	**wykonywana konwersja**
int	float	+, -, *, /, >, <. >=, <=	konwersja int na float
string	int	+	int zamieniony na string i kontakencja dwóch stringów
string	int	*	powielenie string tyle razy ile wynosi wartość int, jeśli jest ona >=0
int	string	-, /, >, <. >=, <=	**operacje niedozwolone, zostanie zwrócony błąd**
float	string	+	float zamieniony na string i kontakencja dwóch stringów
float	string	-, *, /, >, <. >=, <=	**operacje niedozwolone, zostanie zwrócony błąd**
int	bool	and, or	konwersja int na zmienna binarną, 0 -> false, inna wartość -> true
int	bool	+, -, *, /, >, <. >=, <=	**operacje niedozwolone, zostanie zwrócony błąd**
float	bool	and, or	**operacje niedozwolone, zostanie zwrócony błąd**
float	bool	+, -, *, /, >, <. >=, <=	**operacje niedozwolone, zostanie zwrócony błąd**
array	string, int, float, bool	+, -, *, /, >, <, >=, <=, and, or	**operacje niedozwolone, zostanie zwrócony błąd**
bool	string	+, -, *, /, >, <, >=, <=, and, or	**operacje niedozwolone, zostanie zwrócony błąd**

Dodatkowo w języku przewidziana jest funkcja do konwersji string na int lub float, za mocą dwóch funkcji int(argument typu string), float(argument typu string)

jeśli obiekty są tego samego typu wyrażenia zachowują się tak jak w python.

Testowanie:

Testy rozwiązania będą stworzone z wykorzystaniem:

- Testów jednostkowych
- Przykłady testowe dla lexera
- Testy integracyjne z lexerem dla parsera.

Sposób uruchamiania:

python3 interpreter.py *ścieżka do pliku z kodem źródłowym*

Interpreter na wejście otrzyma kod źródłowy w postaci pliku bądź ciągu znaków, natomiast na wyjściu otrzymamy wykonanie kodu bądź komunikaty ewentualnych błędów.