Projekt TKOM – Dokumentacja Końcowa

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 typów prostych przekazywane do funkcji przez wartość, kolekcje przez referencje
- Kolekcja w postaci listy jako typ wbudowany, lista może zawierać elementy różnych typów
- Na kolekcji 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:

  scan();

  funkcja scan 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 100 wywołań. Jednak ograniczenie może być określone również przez użytkownika.
- 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 jest 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 jest 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 jest 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, jedynym zastrzeżeniem jest fakt, że plik, z którego importujemy powinien znajdować się w tym samym katalogu co plik main służący do interpretacji plików wejściowych.

  Załóżmy, że importujemy typ Student:

  from School import Student;

  nowy_uczen = Student();

Gramatyka:

- Gramatyka ze względu na czytelność opisana w innym pliku

Błędy i ich obsługa:

W przypadku napotkania błędu jest on rzucany a użytkownik jest informowany o tym fakcie.

- Błędy składniowe:

    - pozostałe tokeny po zakończeniu parsowania programu
    - invalid variable assignment, nieprawidłowa próba przypisania wartości
    - invalid statement - nieprawidłowy warunek pętli while
    - invalid factor in negation - nieprawidłowe wyrażenie po znaku negacji
    - invalid array definition - nieprawidłowa definicja tablicy
    - invalid logic expression - niepoprawna operacja logiczna
    - invalid arth expression - niepoprawne wyrażenie arytmetyczne
- Błędy semantyczne:
    - przekroczona maksymalna długość identyfikatora
    - nieprawidłowa sekwencja escape
    - znak nowej linii w stringu
    - znak EOF w stringu
    - przekroczona maksymalna długość string
    - liczba float z zbyt dużą dokładnością
    - zbyt duża wartość liczby całkowitej
    - redefinition function error - redeginicja funkcji
    - empty block of statements - pusty blok instrukcji
    - empty if condition - pusty warunek instrukcji warunkowej
    - two parameters with the same name
    - invalid parameters definition
    - invalid or expression
    - invalid and expression
    - importError - obiekt nie może zostać zaimportowany
    - mainFunctionRequired - brak zdefiniowanej funkcji main
    - orOperationErrot i andOperationError - operatory and lub or napotkały nikompatybline typy danych.
    - zero division error - dzielenie przez zero
    - type error - niekompatybline typy danych z wykonywanymi operacjami
    - attributeError - rzucany gdy nie można ustawić atrybutu na obiekcie
    - FunctionDoesNotExist - wywoływana funkcja nie istnieje
    - RecursionLimitExceeded - zbyt duża ilość wywołań rekursywnych
- Błędy kontrolne:
    - ExpectedExpressionError - oczekiwano wyrażenia ale niemożna go znaleźć
    - Parsing Error - ogólna kategoria błędów związanych z parsowaniem kodu

Struktura projektu:

- Analizator leksykalny:

  Lexer analizuje strumień wejściowy znak po znaku w celu utworzenia tokenów, które reprezentują elementy składniowe języka, takie jak operatory, identyfikatory, liczby, i stringi. Wykorzystuje zdefiniowane zasady (mapowania słów kluczowych, operatorów, znaków specjalnych i sekwencji escape) do rozróżnienia różnych typów tokenów i obsługuje błędy poprzez rzucanie wyjątków LexerError w przypadku wykrycia nieprawidłowości, takich jak przekroczenie maksymalnej długości identyfikatora lub stringa. Po przetworzeniu całego strumienia wejściowego, lexer generuje tokeny aż do osiągnięcia końca pliku (EOF), służąc jako podstawa dla dalszej analizy syntaktycznej w parserze.

- Analizator składniowy:
  Parser, analizuje strumień tokenów generowanych przez Lexer, aby zbudować drzewo składni (syntax tree) odpowiadające strukturze źródłowego kodu programu. Wykorzystuje metody takie jak consume_token do przesuwania się przez strumień tokenów, try_consume i must_be do sprawdzania i wymuszania obecności oczekiwanych tokenów, oraz serię metod parse_... do rozpoznawania i przetwarzania konkretnych konstrukcji językowych, takich jak definicje funkcji, wyrażenia arytmetyczne i logiczne, instrukcje sterujące oraz wywołania funkcji.

  Parser jest zaprojektowany, by konstruować złożone struktury takie jak programy, funkcje, wyrażenia, a także instrukcje sterujące, zwracając odpowiednie obiekty drzewa składni dla każdego rodzaju konstrukcji. Błędy składniowe są obsługiwane poprzez rzucanie wyjątków zdefiniowanych w module syntax_error, które mogą wskazywać na różne problemy, takie jak niespójności w składni, brak oczekiwanych tokenów, czy nieprawidłowe definiowanie zmiennych.

  Cały proces analizy składniowej polega na iteracyjnym przetwarzaniu tokenów i konstrukcji składniowych, korzystając z odpowiednio zagnieżdżonych wywołań funkcji parsujących, które współpracują, aby zbudować kompleksową reprezentację programu, gotową do dalszej analizy semantycznej lub bezpośredniej interpretacji.

- Interpreter:

  Interpreter zaimplementowany w klasie ExecuteVisitor służy do wykonania drzewa składniowego utworzonego przez parser. Wzorzec projektowy "wizytator" użyty w tej klasie umożliwia obsługę różnych typów węzłów składniowych poprzez dedykowane metody visit_.... Stan wykonania programu, w tym zmienne, funkcje, i bieżące wyniki, jest zarządzany przez instancję klasy Context.

  Podczas wykonywania, interpreter przetwarza węzły takie jak definicje funkcji, instrukcje warunkowe, pętle oraz wyrażenia, wykorzystując kontekst do przechowywania i modyfikacji zmiennych oraz do kontroli przepływu programu. Obsługa błędów takich jak błędne typy danych czy przekroczenie limitu rekursji jest integralną częścią mechanizmu wykonawczego, co zapewnia stabilność i przejrzystość procesu wykonania. Importowanie modułów i zarządzanie zależnościami również jest wspierane, co pozwala na rozszerzanie funkcjonalności programu poprzez zewnętrzne biblioteki.

  Każda metoda visit_... odpowiada za odpowiednie przetworzenie węzła i aktualizację kontekstu wykonania, co umożliwia dynamiczne zarządzanie stanem aplikacji w trakcie jej działania. To sprawia, że interpreter jest nie tylko efektywnym narzędziem do wykonywania skomplikowanego kodu, ale także elastycznym rozwiązaniem adaptującym się do zmieniających się warunków wykonania kodu.

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)

Testowanie:

Testy rozwiązania będą stworzone z wykorzystaniem:

- Testów jednostkowych dla źródła
- Testów jednostkowych dla lexera
- Testy integracyjne z lexerem dla parsera.
- Testy integracyjne dla lexera, parsera i interpretera
- Przykłady testowe kodu

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.