Dokumentacja wstępna TKOM – „Język do operacji na listach”

Ewelina Chmielewska
283714

1. Treść zadania

„Język do operacji na listach”
Język zorientowany wokół struktur listowych i operacjach na nich, takich jak filtrowanie, złączenia, dostęp do podlist etc.

2. Cel projektu

Celem projektu jest utworzenie języka umożliwiającego operowanie na listach, który swoją składnią będzie zbliżony do języka C++. Język ten powinien umożliwiać przede wszystkim filtrowanie zawartości list, złączenia ich oraz dostęp do poszczególnych ich elementów. Efektem prac ma być utworzenie programu, który otrzymawszy plik z poprawnie zapisanym ciągiem poleceń w tworzonym języku, właściwie zinterpretuje i wykona zadane komendy. W przypadku wystąpienia błędu leksykalnego lub składniowego program powinien wyświetlać komunikat o błędzie, wskazujący pozycję, na której błąd wystąpił.

3. Opis działania projektu

Po uruchomieniu aplikacji, z zadanego pliku odczytywany jest kod programu, który ma zostać wykonany. Źródło (Source) pobiera z pliku po jednym znaku, który następnie przekazuje Lekserowi (Lexer). Za sprawą leksera, kolejne znaki budowane są w słowo, mające znaczenie w projektowanym języku. Słowem tym może być między innymi liczba całkowita, wartość boolowska, identyfikator, znak specjalny lub słowo kluczowe. Kolejne słowa przekazywane są do parsera, który na ich podstawie tworzy bardziej złożone struktury, reprezentowane przez obiekty różnych typów – Number, Identifier, Function, itd. Ostatecznym efektem działania parsera jest utworzenie listy obiektów typu Function, stanowiących zapis programu w sposób zrozumiały dla interpretera. Lista ta przekazana zostaje Interpreterowi (Interpreter). Następnie spośród listy funkcji, interpreter wyszukuje funkcję o identyfikatorze main i wywołuje na niej metodę accept(). Metoda ta, odwołując się do odpowiadającej jej funkcji określonej w Wizytorze (Visitior), wykonuje ciało funkcji, odwołując się do metod accept() kolejnych obiektów. Rezultatem działania całego programu jest zwrócenie wartości określonej jako wartość zwracana w funkcji main (return znajdujący się w funkcji main) lub nie zwrócenie nic. Jeśli w kodzie, który został wykonany, znajduje się poprawne wywołanie procedury print(), to na konsoli wyświetlona zostanie wartość zadana jej jako argument.

4. Opis funkcjonalności:

- **Operacje specyficzne dla języka:**
  - Tworzenie i modyfikacja list
    - łączenie list
    - dodawanie elementów na początek i koniec listy
    - usuwanie elementów z listy
    - jednoczesna modyfikacja wszystkich elementów listy
  - Przeglądanie list
    - wyświetlanie zawartości list
  - Wyszukiwanie elementów listy
    - zwracanie elementów listy spełniających określone warunki - filtrowanie
    - zwracanie elementów listy o zadanym indeksie
- **Operacje standardowe:**
  - Operacje dodawania, odejmowania, mnożenia, dzielenia na liczbach całkowitych
  - Operacje dodawania i mnożenia na wartościach boolowskich
  - Definicje funkcji
  - Wywołania funkcji wewnątrz definicji ciała funkcji
  - Deklaracja i inicjalizacja zmiennych lokalnych w funkcji
  - Tworzenie skomplikowanych wyrażeń zawierających:
    - więcej niż dwa składniki
    - zmienne lokalne
    - operacje na listach
    - wywołania funkcji
  - Wyświetlanie:
    - zawartości zmiennych
    - pojedynczych wartości: liczb całkowitych, wartości boolowskich
    - wyników wyrażeń
    - rezultatów wywołania funkcji

5. Wymagania niefunkcjonalne:

- **Wymagana techniczne:**
  - Język programowania: Python
  - Uruchomienie aplikacji: z terminala
  - Aplikacja konsolowa
- **Pozostałe wymagania:**
  - Silne typowanie
  - Kolejność wykonywania operacji na liście: od lewej do prawej
  - Wyrażenia z priorytetami operatorów

1	=
2	+, -
3	*, /

6. Typy i operacje

W projekcie będą dostępne 3 główne typy danych:

- **list** – struktura listowa, umożliwiająca tworzenie zbioru elementów dowolnego typu. Każdy z nich może posiadać typ niezależny od pozostałych, dopuszczalne jest również utworzenie listy pustej. Na obiekcie listy dokonać można różnych operacji, których sposób wywołania jest następujący:

  lista.operacja(opcje)

  lista – identyfikator listy (nazwa wcześniej zdefiniowanej listy) lub lista w postaci standardowej (np. [1, 2, 3, 4])
  operacja – jedna z możliwych do wykonania operacji, których listę zamieszczono niżej
  opcje – opcje zależne od wykonywanej operacji. Konieczne do zawarcia opcje zamieszczono niżej.

  Operacje możliwe do wykonania na obiekcie typu list to:

  a. **filter(*warunki*)** – zwraca nową listę, składającą się z elementów listy źródłowej, które spełniły zadane *„warunki”*. Postać *„warunków”* jest następująca:

  $$\boldsymbol{warunek1}\ \&\ \boldsymbol{warunek2}\ \&\ \boldsymbol{warunek3}\ \&\ \ldots\ \&\ \boldsymbol{warunekN}$$

  Pojedynczy warunek jest postaci:

  $$\boldsymbol{x\ operacja\_porównania\ wyrażenie}$$

  x – symbol pomocniczy, mający reprezentować pojedynczy, dowolny element listy. Używanie go nie jest obowiązkowe, jednak stanowi dobrą praktykę
  operacja_porównania – jeden z symboli odpowiadających za porównanie:
  - \> „większy”
  - < „mniejszy”
  - >= „większy lub równy”
  - <= „mniejszy lub równy”
  - == „równy”
  - != „różny”

  wyrażenie – wyrażenie

  b. **each(*operator, wyrażenie*)** – wykonuje operację określoną przez operator oraz wartość wyrażenia na wszystkich elementach listy.

  operator – operator określający operację, jaka zostanie wykonana na elementach listy. Możliwe operatory:
  - \+ „dodawanie wartości wyrażenia do elementu listy”
  - – „odejmowanie wartości wyrażenia od elementu listy”
  - * „mnożenie wartości wyrażenia przez element listy”
  - / „dzielenie elementu listy przez wartość wyrażenia”

  wyrażenie – wyrażenie

  c. **get(*indeks*)** – zwraca element listy o zadanym *indeksie*
  d. **length()** – zwraca długość listy źródłowej
  e. **delete(*indeks*)** – usuwa element listy o zadanym *indeksie*

- **number** – typ danych dotyczący liczb całkowitych. Analogiczny do typu int w języku C++.

- **bool** – logiczny typ danych, składający się z dwóch elementów: prawda, fałsz.
  Operacje możliwe do wykonania na elementach typu bool:
  - \+ „suma logiczna”
  - * „iloczyn logiczny”

7. Instalacja i uruchomienie

- **Instalacja**
  1. **Przejście do katalogu TKOM**
  2. **Ustawienie lokalnej wersji pythona na 3.7.2**
     Przykład:
     - Instalacja pyenv ($ curl https://pyenv.run | bash)
     - Instalacja pythona
       - przejście do shella
         $ sh
       - instalacja pythona
         $ pyenv install 3.7.2
       - ustawienie lokalnej wersji pythona na 3.7.2
         $ pyenv local 3.7.2
  3. **Odczytanie ścieżki do pythona 3.7.2**
     $ which python
  4. **Skopiowanie wyniku komendy „which python"**
  5. **Instalacja środowiska w katalogu**
     $ virtualenv –python=sciezka_do_pythona_3.7.2 nazwa_folderu_srodowiska
     np. $ virtualenv –python=/home/ewelina/.pyenv/shims/python env
  6. **Uruchomienie środowiska**
     $ source nazwa_folderu_srodowiska/bin/activate
     np. $ source env/bin/activate
  7. **Instalacja najnowszej wersji pytest**
     $ pip install pytest
  8. **Ponowne uruchomienie środowiska**
     $ deactivate
     $ source nazwa_folderu_srodowiska/bin/activate
     np. $ deactivate
     $ source env/bin/activate
  9. **Test**
     $ pytest
     Jeśli wyświetlono komunikat o przejściu testów, to wszystko zostało zainstalowane poprawnie
- **Uruchamianie**
  $ python main.py <nazwa_pliku_z_kodem_do_interpretacji

8. Testy

Przed uruchomieniem testów należy przejść do katalogu TKOM.

- Uruchamianie testów automatycznych
  1. Uruchomienie środowiska
     $ source nazwa_folderu_srodowiska/bin/activate
  2. Uruchomianie testów
     $ pytest
- Uruchamianie testów manualnych
  1. Uruchomienie środowiska
     $ source nazwa_folderu_srodowiska/bin/activate
  2. Uruchomienie testu manualnego
     $ python main.py plik_z_testami
     np. $ python main.py test_file.txt

9. Przykłady w języku

TODO

10. Gramatyka

PROGRAM = [FUNCTION]*;

FUNCTION = function IDENTIFIER ‘(‘ [ARGUMENTS]‘) ’{’ FUNCTION_BODY ’}’;

IDENTIFIER = LETTER*[LETTER_OR_DIGIT]*

LETTER = UPPER|DOWN;

LETTER_OR_DIGIT = LETTER |DIGIT;

DOWN = ‘a’| ‘b’| ‘c’| ‘d’| ‘e’| ‘f’| ‘g’| ‘h’| ‘i’| ‘j’| ‘k’| ‘l’| ‘m’| ‘n’| ‘o’| ‘p’| ‘r’| ‘s’| ‘t’| ‘u’| ‘w’| ‘x’| ‘y’| ‘z’;

UPPER = ‘A’|’B’|’C’|’D’|’E’|’F’|’G’|’H’|’I’|’J’|’K’|’L’|’M’|’N’|’O’|’P’|’R’|’S’|’T’|’U’|’W’|’X’|’Y’|’Z;

DIGIT = ‘0’ | NON_ZERO_DIGIT;

NON_ZERO_DIGIT = ‘1’ | ‘2’ | ‘3’| ‘4’| ‘5’| ‘6’| ‘7’| ‘8’| ‘9’;

ARGUMENTS = TYPE IDENTIFIER [‘, ’ TYPE IDENTIFIER]*;

TYPE = ‘list’ | ‘number’ |‘bool’;

NUMBER = [-] NON_ZERO_NUMBER | 0;

NON_ZERO_NUMBER = NON_ZERO_DIGIT [DIGIT]*;

FUNCTION_BODY = "{" [FUNCTION_BODY_CONTENT] [FUNCTION_BODY_RETURN] "}";

FUNCTION_BODY_CONTENT = LINE [LINE]*;

FUNCTION_BODY_RETURN = "return" EXPRESSION);

BOOL = "true" | "false";

LIST = "[" [LIST_ELEMENTS] "]";

LIST_ELEMENTS = EXPRESSION [", " EXPRESSION]*;

LINE = (print(EXPRESSION)| DECLARATION | EXPRESSION) ’; ’;

EXPRESSION = (IDENTIFIER ’ = ’ EXPRESSION | MULTIPLICATION [ADR_OPERATOR MULTIPLICATION]*);

ADD_OPERATOR = ’ + ’|’ − ’;

MULTIPLICATION = FACTOR [MULTIPLICATION_OPERATOR FACTOR]* ;

MULTIPLICATION_OPERATOR = ’ ∗ ’ | ’/’;

FACTOR = ’(’ EXPRESSION ’)’ | COMPONENT;

COMPONENT = [’−’] (BOOL |NUMBER | COMPONENT_WITH_IDENTIFIER| COMPONENT_WITH_LIST);

COMPONENT_WITH_IDENTIFIER = IDENTIFIER [LIST_OPERATION | FUNCTION_CALL];

COMPONENT_WITH_LIST = LIST [LIST_OPEARTION];

LIST_OPEARTION = ’. ’ (FILTER|EACH|GET|LENGTH|DELETE);

DECLARATION = TYPE IDENTIFIER [" = " EXPRESSION];

LIST_OPERATION = "."(FILTER | EACH | GET | LENGTH | DELETE);

FILTER = "filter" "(" CONDITIONS ")";

CONDITIONS = SINGLE_CONDITION ["&"SINGLE_CONDITION]*

SINGLE_CONDITION = 'x' CMP_OPERATOR EXPRESSION);

CMP_OPERATOR = ' < ' | ' > ' | ' ≤ ' | ' ≥ ' | ' == ' | '! = ';

FUNCTION_CALL = "(" ELEMENTS ")";

ELEMENTS = EXPRESSION [',' EXPRESSION]*;

EACH = each((ADD_OPERATOR | MULTIPLICATION_OPERATOR) ', ' EXPRESSION ")";

OPERATION = " + " | " − " | "/" | " ∗ ";

GET = "get" "(" EXPRESSION ")";

LENGTH = "length" "(" ")";

DELETE = "delete" "(" EXPRESSION ")";