[TKOM]

Język Przetwarzania Obrazu

iml

Piotr Frątczak

(300207)

Kwiecień 2021

Streszczenie

Projekt polega na stworzeniu interpretera języka przetwarzenia obrazu wykonanym w języku Python.

1 Opis funkcjonalny

Język służy do wykonywania operacji na obrazach wyrażonych w postaci macierzy pikseli w formacie RGB. Język posiada wbudowany typ podstawowy - piksel, posiadający 3 opisujące go wartości: R, G, B (kanały czerwony, zielony, niebieski), które łatwo odczytać oraz zmodyfikować. Ponadto wbudowany jest typ macierzy, który może składać się z pojedyńczych pikseli lub liczb. Język umożliwia definiowanie własnych przekształceń oraz definiowanie własnych operatorów służących do manipulacji obrazami.

Język ułatwia przeprowadzanie przekształceń na obrazach, np. przeprowadzanie konwolucji, wygładzanie obrazka, znajdowanie krawędzi, używanie filtrów na obrazach.

1.1 Język umożliwia:

- wykonywanie podstawowych operacji arytmetycznych:
  - dodawanie,
  - odejmowanie,
  - mnożenie,
  - dzielenie,
  - dzielenie z resztą,

  z zachowaniem priorytetów operatorów,
- wykonywanie operacji matematycznych na macierzach:
  - mnożenie macierzy,
  - mnożenie macierzy przez liczbę,
  - dodawanie macierzy,
- wykonywanie operacji na wierszach macierzy,
- definiowanie własnych operacji na macierzach, w tym:
  - definiowanie własnych operatorów:

    operator jest definiowany dla konkretnego typu danych - ten sam operator może oznaczać dwie inne operacje dla dwóch różnych typów danych,
- definiowanie własnych funkcji,
- wczytywanie obrazów z pliku,

  przykład: `python3 imli.py img.jpg`;
- zapisywanie obrazów do pliku,
- dodawanie komentarzy liniowych.
- wykonywanie instrukcji warunkowych,
- wykonywanie instrukcji pętli,
- komentowanie kodu.

1.2 Założenia

- Wartości kanałów dla typu pixel są liczbami całkowitymi należącymi do zakresu `[0, 255]`. Próba przypisania wartości niecałkowitej kończy się przypisaniem zaokrąglenia tej liczby, jeżeli liczba nie należy do ustalonego przedziału, przypisana zostanie liczba minimalna z przedziału dla wartości mniejszych od `0` i maksymalna dla wartości większych od `255`.
- Nowo utowrzona macierz zawiera same zera, jeżeli zostały podane same wymiary.
- Białe znaki są ignorowane.
- Wyrażenia niekończące się klamrą `}` są zakończone średnikiem `;`.
- Nowo utworzone operatory mają najniższy priorytet.
- Definicja funkcji musi kończyć się słowem kluczowym `return`.

1.3 Uruchamianie

Aby uruchomić interpreter języka na podstawie kodu źródłowego należy wykonać:
`$ python imli.py <source_file>`

2 Typy danych

Język oferuje 3 wbudowane typy danych: liczbę, macierz oraz piksel.

2.1 Liczba

Liczba (`number`) jest typem podstawowym. Język obsługuje tylko liczby całkowite, dodatnie i ujemne.
Przykład: `2, 3.1, -5, -2.3`.

2.2 Piksel

Piksel (`pixel`) jest typem złożonym z 3 liczb całkowitych z zakresu `[0, 255]`. Trzy liczby odpowiadają trzem kanałom RGB, można łatwo odczytać i modyfikować wartości kolorów piksela.
Przykład: piksel o kolorze zielonym można wyrazić przez `pixel(0, 255, 0) lub (0, 255, 0)`.

2.3 Macierz

Macierz (`matrix`) jest typem złożonym z wektorów liczb lub pikseli. Można łatwo odczytać dany wiersz, kolumnę lub pojedyńczeą wartość macierzy lub je zmodyfikować.
Przykład: `[[1, 2], [3, 4]], [[pixel(0,0,5), pixel(1,1,1)]] == [[0,1], [0,1], [5,1]]`.

3 Wymagania

3.1 Wymagania funkcjonalne

- obrazy (macierze) można pobierać z pliku i zapisywać do pliku.
- interpreter działa na plikach tekstowych - odczytuje, parsuje i analizuje programy,
- interpreter analizuje poprawność kodu i zgłasza wykryte błędy.

3.2 Wymagania niefunkcjonalne

- komunikaty o błędach powinny być czytelne i powinny pomagać w odnalezieniu błędu, powinny wskazywać nr linii i kolumnę błędu oraz fragment błędnego kodu,
- język nie powinien być podatny na złośliwe ataki,
- język powinny mieć praktyczne zastosowanie w operowaniu macierzami,
- język i interpreter powinien dać się rozbudować w sposób prosty.

4 Elementy języka

4.1 Słowa klucze

Język zawiera poniżej wymienione słowa i znaki klucze oznaczające:

- operatory logiczne: `!`, `and`, `or`, `<`, `>`, `==`, `!=`, `<=`, `>=`;
- operatory arytmetyczne: `+`, `-`, `*`, `/`;
- dzielenie z resztą (modulo): `%`;
- specjalne mnożenie macierzy (element po elemencie): `@`;
- początek komentarza: `#`;
- konstrukcje pętlowe: `for`, `in`;
- konstrukcje warunkowe: `if`, `else`;
- konstrukcje funkcji: `return`;
- konstrukcja nowego operatora: `of`;
- przypisanie wartości: `=`;
- odwołanie do metody lub atrybutu: `.`;
- separator argumentów: `,`;
- koniec wyrażenia: `;`;
- początek i koniec wyrażenia lub bloku: `(`, `)`, `{`, `}`;
- odwołanie do elementu macierzy: `[`, `]`;
- typy danych: `pixel`, `matrix`, `number`.

4.2 Funkcje wbudowane

Funkcje należące do biblioteki standardowej:

- `print()`,
- `number()`,
- `pixel()`,
- `matrix()`,
- `rand_matrix()`,
- `rand_pixel()`,
- `rand_integer()`,
- `rand_float()`,
- `<matrix>.convolute()`,
- `<matrix>.svd()`,
- `<matrix>.det()`,

5 Przykładowy program

```
main() {
    # podstawowe operacje na macierzach
    px = pixel(0, 0, 255); # niebieski piksel
    mx = matrix(3, 3, px); # macierz 3x3 zlozona z niebieskich pikseli
    mx.dims; # 2
    mx.xdim; # 3
    mx.ydim; # 3

    m1 = matrix(2,4);
    m2 = matrix(4,3);
    m3 = m1 * m2;
    m3.xdim; # 2
    m3.ydim; # 3

    m4 = matrix(2,4);
    m5 = m1 @ m4;
    m5.xdim; # 2
    m5.ydim; # 4

    m6 = matrix([[1, 2],
                 [3, 4]]);
    m7 = 2 * m6 ; # [[2, 4],
                  #  [6, 8]]


    # definiowanie nowego operatora dla typu macierzy
    newop(avg, m1 of matrix, m2 of matrix){
        return (m1 + m2) / 2;
    }

    m8 = matrix([[1, 2],
                 [3, 4]]);
    m9 = matrix([[3, 2],
                [3, 2]]);
    m10 = m8 avg m9; # [[2, 2],
                     #  [3, 3]]
    m8[1, 0] # 3 #

    p1 = pixel(20, 90, 120);
    p2 = pixel(10, 10, 200);
    p3 = p1 + p2; # pixel(30, 100, 255)
    p4 = p1 - p2; # pixel(10, 80, 0)

    a = p1.r; # 20
    b = p1.g; # 90
    c = p1.b: # 120

    d = a + b + c; # 230

    # konstrukcja petli
    f = 0;
    for (i in 5){
    f = f + 1;
    }
    # po wykonaniu f wynosi 5
}
```

6 Gramatyka

```
program                    = {function_definition | comment};
comment                    = "#", { ? any character ? }, ? carriage return ?;

function_definition        = id, "(", argument_list, ")", block;
function_call              = id, "(", argument_list, ")";
argument_list              = [ expression, {",", expression} ];

if_statement               = "if", "(", condition, ")", block , {"else", block};
init_statement             = type, "(", argument_list, ")", ";";
return_statement           = "return", expression, ";";
block                      = statement | ( "{", {statement}, "}" );

assignment                 = (id_or_member | matrix_lookup), assignment_operator,
                             expression, ";";
id_or_member               = id, {member_operator, id},
                             [member_operator, matrix_lookup];
member_reference           = (id | function_call), {member_reference}, ";";
member                     = member_operator, (id | function_call);

operator_definition        = new_operator, "(", id, of_operator, type, ",",
                             id, of_operator, type, ")", block;
matrix_lookup              = id, "[", arithmetic_expression,
                             {",", arithmetic_expression} "]";

for_loop                   = "for", "(", expression, "in", expression, ")",
                             block;
while_loop                 = "while", "(", condition, ")", block;

statement                  = operator_definition | for_loop | while_loop |
                             if_statement | init_statement | return_statement |
                             assignment | (function_call, ";") | comment;

expression                 = multiplicative_expression,
                             {additive_operator, mulitiplicative_expression};
multiplicative_expression  = base_expression,
                             {mulitiplicative_operator, base_expression};
base_expression            = [subtraction_operator],
                             (expression_in_parenthesis | number |
                             matrix_lookup | function_call |
                             member_reference);
expression_in_parenthesis  = "(", expression, ")";

condition                  = and_condition,
                             {alternative_operator, and_condition};
and_condition              = equality_condition,
                             {conjunction_operator, equality_condition};
equality_condition         = relation_condition,
                             [equal_operator, relation_condition];
relation_condition         = base_condition,
                             [comparison_operator, base_condition];
base_condition             = [negation_operator],
                             (condition_in_parenthesis | expression);
condition_in_parenthesis   = "(", condition, ")";
```

Nie można wstawiać białych znaków między elementami poniższych produkcji

```
id                        = alpha , {word};
alpha                     = "a"..."z" | "A"..."Z";
non_zero                  = "1"..."9";
digit                     = "0" | non_zero;
number                    = "0" | ( non_zero, {digit} );
word                      = alpha | digit | "_";

negation_operator         = "!";
member_operator           = ".";
alternative_operator      = "or";
conjunction_operator      = "and";
equal_operator            = "==" | "!=";
matrix_mult_operator      = "@";
modulo_operator           = "%";
comparison_operator       = "<" | ">" | "<=" | ">=";
additive_operator         = "+" | subtraction_operator;
subtraction_operator      = "-";
multiplicative_operator = "*" | "/";
assignment_operator       = "=";
of_operator               = "of";
new_operator              = "newop";
type                      = "pixel" | "matrix" | "number";
```

7 Struktura projektu

Projekt będzie złożony z poniższych modułów:

- **Types** - zawiera definicje wszystkich typów danych (`matrix, pixel, number`).
- **Source** - odpowiada za wczytywanie kodu ze źródła znak po znaku i przekazywaniu tych znaków do Lexera.
- **Lexer** - odpowiada za analizę leksykalną - pobieranie znaków i budowanie tokenów, a następnie przekazanie je do Parsera.
- **Parser** - odpowiada za analizę składniową - sprawdza poprawność gramatyczną otrzymanych tokenów i tworzy elementy potrzebne do zbudowania drzewa składniowego i przekazuje do Analysera.
- **Analyser** - odpowiada za analizę semantyczną otrzymanych elementów, sprawdza poprawność programu zależnie od kontekstu i generuje drzewo instrukcji.
- **Interpreter** - odpowiada za wykonanie programu po zakończeniu działania poprzednich modułów, na podstawie gotowego drzewa instrukcji.
- **ErrorHandler** - odpowiada za obsługę błędów i generowanie odpowiednich komunikatów.

7.1 Testy

W celu sprawdzenia poprawności działania interpretera, zostanie stowrzony dodatkowy moduł - **Tests**, odpowiadający za testy dla trzech głównych modułów: Leksera, Parsera i Interpretera.