Matrixlang - Język z macierzami

Dokumentacja

Piotr Satała

Spis treści

Opis zakładanej funkcjonalności

1. Głównym założeniem języka Matrixlang jest umożliwienie programiście korzystania z wbudowanego typu macierzy (Matrix).
2. Dostępny będzie też typ wektor (Vector). Typ macierzy będzie zrealizowany jako wektor wektorów, lecz nie będzie to widoczne dla programisty.
3. Pozostałe typy to: int, float oraz string.
4. Język umożliwia tworzenie funkcji (w tym rekurencyjnych), obsługuje prostą pętle (for), a także posiada 2 instrukcje warunkowe - switch oraz if.
5. Matrixlang jest językiem interpretowanym, interpreter napisany został w języku C++.
6. Zawiera on typowanie silne oraz statyczne. Wszystkie zmienne są mutowalne.
7. Zmienne są lokalne - można na nich operować tylko wewnątrz danego bloku kodu.

Następna sekcja prezentuję zakładaną funkcjonalność bardziej szczegółowo przy użyciu przykładów.

Przykłady wykorzystania języka

1. Funkcja wyznaczająca wartość n-tego elementu ciągu Fibonacciego (rekurencyjnie).

```
int fibonacciRec(int n) {
    if(n == 0) { return 0; }
    if(n == 1) { return 1; }
    return fibonacciRec(n - 1) + fibonacciRec(n - 2);
}

int main() {
    print(intToString(fibonacciRec(10))); #prints 55
    return 0;
}
```

2. Funkcja wyznaczająca wartość n-tego elementu ciągu Fibonacciego (iteracyjnie).

```
int fibonacciIter(int n) {
     #special cases
     if(n == 0) { return 0; }
     if(n == 1) { return 1; }

     #declare variables
     int prevprev = 0;
     int prev = 1;
     int current;

     #main loop
     for(int i = 2; i < n; ++i) {
          current = prev + prevprev;
          prevprev = prev;
          prev = current;
     }

     return current;
}

int main() {
     print(intToString(fibonacciIter(10))); #prints 55
     return 0;
}
```

3. Funkcja znajdująca największą wartość w macierzy liczb całkowitych.

```
int findMax(Matrix<int>[n_cols, n_rows] matrix) {
      #n_cols and n_rows always an int

      int max = INT_MIN; #environmental constant
      for(int i = 0; i < n_cols; ++i) {
            for(int j = 0; j < n_rows; ++j) {
                  if(matrix[i, j] > max) { max = matrix[i, j]; }
            }
      }
      return max;

}

int main() {
      Matrix<int>[2, 2] matrix;
      matrix[0, 0] = -3;
      matrix[0, 1] = 3;
      matrix[1, 0] = 14;
      matrix[1, 1] = -14;
      print(intToString(findMax(matrix))); #prints 14
      return 0;
}
```

4. Funkcja konkatenująca wszystkie łańcuchy znaków w macierzy w jeden łańcuch znaków.

```
string concatStringMatrix(Matrix<string>[n_cols, n_rows] matrix) {
      string concatString = “”;
      for(int i = 0; i < n_cols; ++i) {
            for(int j = 0; j < n_rows; ++j) {
                  concatString += matrix[i, j];
            }
      }
      return concatString;
}

int main() {
      Matrix<string>[2, 2] matrix;
      matrix[0, 0] = “ala ”;
      matrix[0, 1] = “ma “;
      matrix[1, 0] = “kota ”;
      matrix[1, 1] = “i psa”;
      print(concatStringMatrix(matrix)); #prints “ala ma kota i psa”
      return 0;
}
```

5. Funkcja wyznaczająca średnią wartość elementów w wektorze.

```
float average(Vector<float>[n] vector) {
      float sum = 0.0;
      for(int i = 0; i < n; ++i) {
            sum += vector[i];
      }
      return sum / intToFloat(n);
}

int main() {
      Vector<float>[2] vector;
      vector[0] = 2;
      vector[1] = 3;
      print(floatToString(average(vector))); #prints 2.5
      return 0;
}
```

6. Program wyświetlający na konsoli informację tekstową na podstawie podanego przez użytkownika wieku.

```
void printAgeDescription(int age) {
      ```
      In this function a switch statement is used to print
information about your age.
      ```
      switch {
      case age < 0:
            print(“Your age is incorrect”);
      case age < 18:
            print(“You are a kid.”);
      case age < 60:
            print(“You are an adult.”);
      default:
            print(“You are a pensioner.”);
      }
}

int main() {
      int age = stringToInt(input());
      printAgeDescription(age);
      return 0;
}
```

7. Program umieszczony w różnych plikach wyświetlający pozdrowienie.

Zawartość pliku “yearInfo.ml”:

```
void printYearInfo(int currentYear) {
      print(“Hello from year ” currentYear “”);
}
```

Zawartość pliku “main.ml”:

```
@include “yearInfo.ml”

int main() {
      printYearInfo(2021);
      return 0;
}
```

8. Program pokazujący, że zmienne przekazywane są przez wartość.

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

int main() {
      int a = 3;

      addOne(a);
      print(intToString(a)); #prints 3

      a = addOne(a);
      print(intToString(a)); #prints 4

      return 0;
}
```

9. Program pokazujący działanie kopiowania na przykładzie Vectora. Pokazana jest też alternatywna forma użycia funkcji print().

```
int main() {
      Vector<int>[2] vector;
      print("0: "vector[0]", 1: "vector[1]"\n"); #prints "0: 0, 1: 0"
      vector[0] = 3;
      vector[1] = 5;

      Vector<int>[2] vectorCopy = vector;
      print("0: "vectorCopy[0]", 1: "vectorCopy[1]"\n"); #prints "0:
3, 1: 5"
      return 0;
}
```

10. Program pokazujący lokalność zmiennych (widoczne tylko wewnątrz danego bloku kodu).

```
int main() {
      {
            int a = 10;
      }
      print(intToString(a));     #error - no such variable “a”
      return 0;
}
```

11. Program pokazujący błąd przy próbie dzielenia przez zero.

```
int main() {
      int a = 1 / 0;  #error - cannot divide by 0
      return 0;
}
```

12. Program pokazujący silne typowanie.

```
int main() {
    int a = 0;
    float b = floatToInt(a); #ok
    float c = a; #error
    return 0;
}
```

Formalna specyfikacja i składnia - gramatyka w postaci EBNF

Poziom leksyki

```
non zero digit  = “1” | “2” | “3” | “4” | “5” | “6” | “7” | “8” |
“9”;
digit           = “0” | non zero digit;
lower letter    = /* all lowercase letters */
upper letter    = /* all uppercase letters */
character       = /* all UTF-8 encoded characters */
letter          = lower letter | upper letter;
integer literal = (non zero digit, {digit}) | “0”;
float literal   = integer literal, “.”, {digit};
string literal  = “\””, {character}, “\””;
identifier      = letter, [{letter | digit | “_”}];
```

Poziom składni

```
simple type= “int” | “float” | “string”;
vector type= “Vector”, “<”, type, “>”, “[“, expression, “]”;
matrix type= “Matrix”, “<”, type, “>”, “[“, expression, “,”,
expression, “]”;
type       = simple type | vector type | matrix type;

string exp = string literal, {[expression], string literal};
literal    = integer literal
           | float literal
           | string exp
funcall exp= identifier, “(“, exp list, “)”;
prim exp   = “(“, expression, “)”
           | literal
           | funcall exp

var exp    = identifier
lval exp   = var exp, {“[“, expression, [“,”, expression] “]”};
post exp   = lval exp, { “++” | “--” };

unary inc exp   = {(“++” | “--”)}, post exp;
unary exp  = {(“+” | “-” | “!”)}, unary inc exp | prim exp;

mult exp   = unary exp, {(“*” | “/” | “%”), unary exp};
add exp    = mult exp, {(“+” | “-”), mult exp};
```

```
rel exp    = add exp, {("<" | "<=" | ">" | ">=" | "==" | "!="), add
exp};
and exp    = rel exp, {"&&", rel exp};
or exp     = and exp, {"||", and exp};
rval exp   = or exp;
assign exp = {lval exp, ("=" | "+=" | "-=" | "*=" | "/=" | "%=")},
             rval exp;
expression = assign exp;
exp list   = [expression, {",", expression}];

declaration= type, identifier, ["=", expression];
declaration instruction    = declaration, ";";

arg list   = type, identifier, {",", type, identifier};
function   = type, identifier, "(", [arg list], ")", statement;
return     = "return", [expression], ";";

instruction= ([expression], ";")
           | if
           | switch
           | for
           | return
           | declaration instruction
           | block;
instr list = {instruction};
block      = "{", instr list, "}";
statement  = instruction | block;


if         = "if", "(", expression, ")", statement, ["else",
statement];

case go    = "case", expression, ":", instr list;
case c     = "case", integer literal | float literal | string
literal, ":", instr list;
default    = "default", ":", instr list;

//switch bez żadnego case rozpoznawany jako switch c
switch go end   = "{", {case go}, [default],  "}";
switch c end    = "(", lval exp, ")", "{", {case c}, [default],  "}";
switch          = "switch", switch c end | switch go end;

for        = "for", "(", [declaration], ";", [expression], ";",
[expression], ")", statement;
```

```
program    = {declaration instruction | function};
```

Wymagania funkcjonalne i niefunkcjonalne

Lista wymagań funkcjonalnych

1. Możliwość tworzenia zmiennych typów prostych (int, float, string).
2. Możliwość tworzenia zmiennych typów złożonych (Matrix, Vector).
3. Obsługa kopiowania zmiennych tych samych typów przy użyciu operatora “=”.
4. Obsługa rzutowania z jednego typu na drugi - nie wszystkie kombinacje będą dostępne (np. rzutowanie z Matrix na float), inne mogą generować błędy (np. string na int), inne powinny powieść się za każdym razem (np. int na string).
5. Obsługa operatorów dla typów prostych (operator konkatenacji dla typu string i operatory operacji liczbowych dla typów int i float).
6. Obsługa operatorów dla typów złożonych (np. operator “[ ]” dla typu Vector).
7. Możliwość tworzenia wyrażeń, za pomocą zmiennych oraz operatorów (powinny one zwracać wartość logiczną, która będzie konieczna do realizacji instrukcji warunkowej i pętli).
8. Możliwość tworzenia instrukcji warunkowych (if/else oraz switch - przy czym switch może nie mieć po sobie podanej nazwy zmiennej - wtedy po każdym słowie kluczowym case powinno nastąpić wyrażenie, którego wartość logiczna ma zostać sprawdzona).
9. Możliwość tworzenia klasycznej pętli for.
10. Obsługa zakresu widoczności zmiennych (wewnątrz danego bloku kodu).
11. Obsługa tworzenia funkcji, które będą przyjmowały argumenty przez wartość. Funkcje te będą mogły być wywoływane rekurencyjnie.
12. Możliwość wywoływania funkcji systemowych, takich jak print() czy input(), a także funkcji rzutowania, np. intToString().
13. Obsługa wielu plików źródłowych przez operacje @include.
14. Interpreter działa w trybie wsadowym.
15. Interpreter zapewnia obsługę błędów - na poziomie leksera, parsera oraz egzekutora. Przerywa on swoje działanie w przypadku gdy natrafi na błąd, a także przekazuje informacje diagnostyczne, takie jak miejsce wystąpienia błędu i jego przyczyna.

Lista wymagań niefunkcjonalnych

1. Interpreter ma być możliwy do uruchomienia w systemie operacyjnym Linux.
2. Interpreter nie może kończyć działania wskutek błędu w kodzie interpretera.
3. Czas odpowiedzi interpretera (rozpoczęcie interpretacji kodu w języku Matrixlang) nie przekracza jednej sekundy dla strumieni wejścia zawierających nie więcej niż 1 KB danych.

Obsługa błędów

Błędy mogą być generowane przez lekser, parser bądź też egzekutor. Komunikat o każdym błędzie jest przekazywany do strumienia błędów, a będzie w nim zawarta informacja o rodzaju błędu, a także o miejscu jego wystąpienia (numer linii i wiersza).

- Lekser będzie generował błąd, jeśli dany ciąg znaków nie będzie można przypisać żadnemu rodzajowi tokenu (np. samodzielny znak “^”) - taki błąd przerwie pracę interpretera, a sam token zostanie zapisany jako token nieprawidłowy. Lekser będzie zmuszony przerwać pracę również wtedy, gdy zacznie rozpoznawać jakiś token, a potem okaże się, że jest on nieprawidłowo zapisany (np. ciąg znaków “``a” zacznie być rozpoznawany jako początek wieloliniowego komentarza, lecz pojawienie się tam nielegalnego znaku wygeneruje błąd).

- Parser będzie generował błąd, jeśli otrzymane od leksera tokeny nie zgadzają się z gramatyką, np. pojawienie się tokena “{“ otwierającego blok kodu, a nie pojawienie się nigdy tokena “}”, który by ten blok kodu zamknął - taki błąd przerwie działanie interpretera.

- Egzekutor będzie generował błąd, jeśli napotkany symbol nie będzie się znajdował w tablicy symboli - błąd ten przerwie działanie interpretera. Próba dzielenia przez 0 bądź inne podobne niedozwolone operacje również będą skutkować błędem przerywającym pracę interpretera, a bardziej szczegółowe informacje o typie błędu zostaną przekazane do strumienia błędów, będącego atrybutem egzekutora.

Sposób uruchomienia

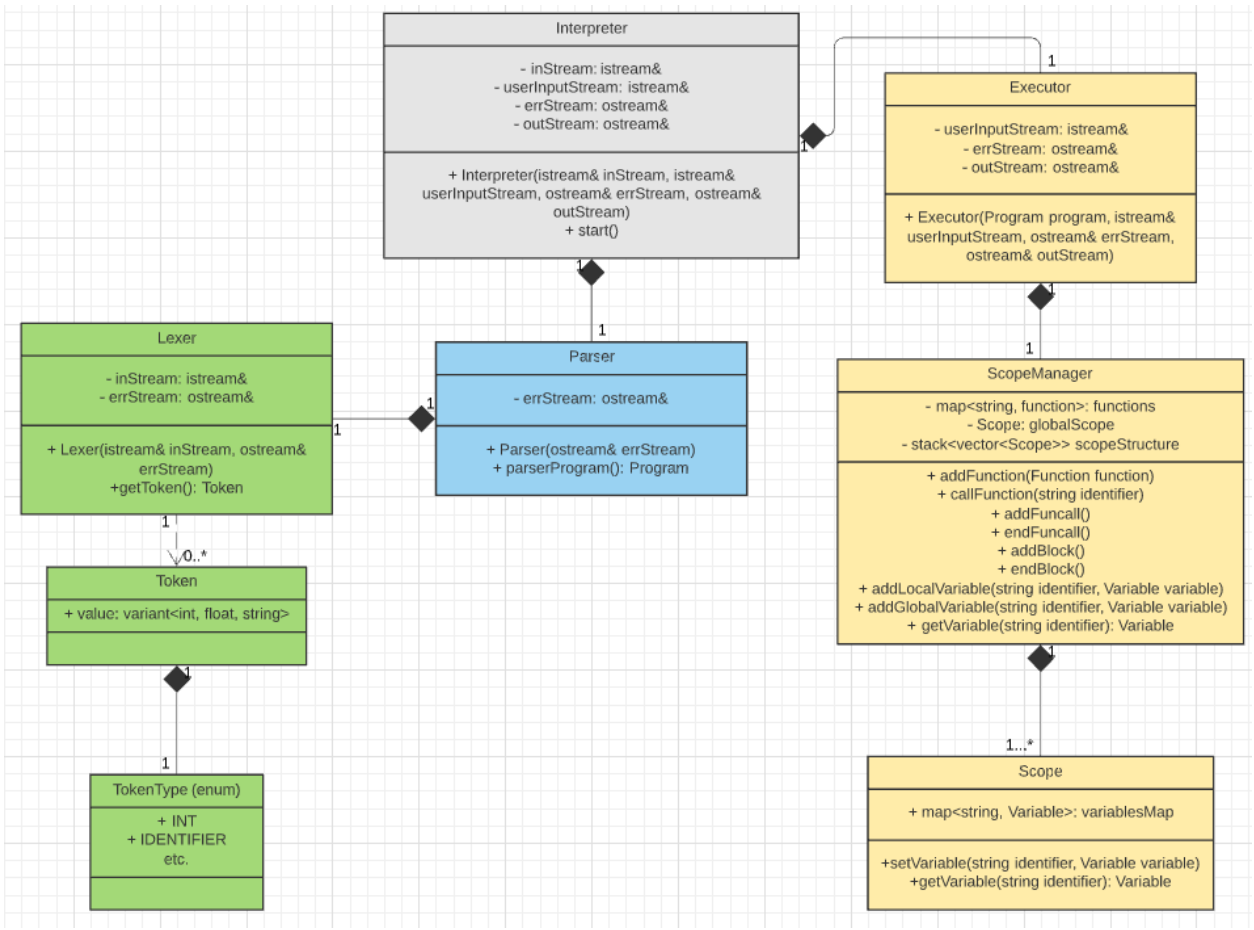
Aby uruchomić program napisany w Matrixlang'u, należy podać ścieżkę do skompilowanego interpretera języka Matrixlang, oraz ścieżkę do pliku wejściowego (z funkcją main). Ilustruje to poniższy przykład:

```
./matrixlang main.ml
```

Strumieniem wyjściowym dla tak uruchomionego programu będzie domyślny strumień wyjścia, a strumieniem błędów domyślny strumień błędów. Strumieniem wejścia użytkownika będzie domyślny strumień wejścia. Wszystkie pliki źródłowe języka Matrixlang powinny mieć rozszerzenie `.ml`. Jeśli kod znajduje się w kilku różnych plikach, należy wtedy użyć instrukcji leksera "`@include file.ml`", która zmieni strumień wejściowy na "`file.ml`".
Każdy program napisany w języku Matrixlang powinien zawierać funkcję `main`, która nie przyjmuje żadnych parametrów i zwraca wartość typu `int`. Jeśli zwrócona wartość będzie inna od 0, to interpreter poinformuje o tym użytkownika.
Aby ułatwić proces testowania, Matrixlang umożliwia także inny sposób uruchomienia, lecz jest on dostępny tylko wewnątrz interpretera - należy przekazać testowany kod do strumienia, a następnie przekazać owy strumień jako argument do obiektu leksera (pierwszy sposób będzie zrealizowany bardzo podobnie, z tym że owym strumieniem będzie strumień wejścia z pliku).

Sposób realizacji

Program interpretera został napisany w języku C++, stąd też jego struktura jest oparta na obiektach. Ponadto przydały się oferowane przez ten język strumienie.
Poniżej znajduje się uproszczony diagram klas interpretera:



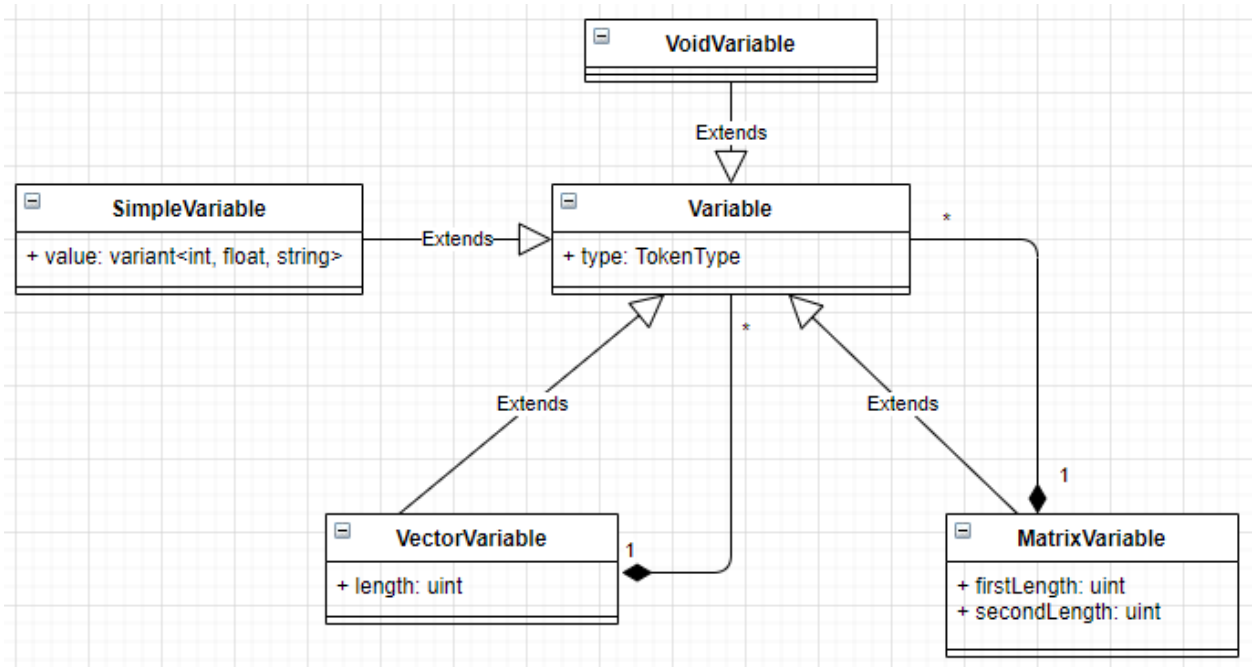
Warto zwrócić uwagę na obiekt ScopeManager. Jest on odpowiedzialny za realizację lokalności zmiennych. Przechowuje on też wszystkie zadeklarowane funkcje oraz zmienne globalne. Obiekt ten zawiera zmienną scopeStructure, na którą składa się:

- Stos wywołań funkcji
- Wektor bloków wewnątrz danego wywołania funkcji
- Mapa o kluczach typu `string` i wartościach typu `Variable` przechowująca zmienne (obiekt `Scope`), która jest elementem powyższego wektora

Przy takiej strukturze dodanie zmiennej lokalnej spowoduje dodanie jej do mapy będącej ostatnim elementem wektora bloków, który z kolei znajduje się na szczycie stosu wywołań funkcji. Dzięki temu, jeśli w programie mamy kilka zmiennych o tych samych identyfikatorach, zwrócona zostanie ta najbardziej lokalna (znajdująca się na końcu wektora). Przy poszukiwaniu zmiennej pod uwagę brany jest tylko wektor znajdujący się na szczycie stosu oraz mapa globalna.

Struktura zmiennych

Poniższy diagram przedstawia rekursywną strukturę zmiennych. Zmienne typu Vector oraz Matrix mogą zawierać wiele zmiennych dowolnego (tego samego) typu (a zatem Vector może zawierać w sobie Vector). Zmienna typu VoidVariable używana jest tylko do określenia funkcji, która nic nie zwraca - nie można utworzyć zmiennej tego typu.



Sposób testowania

- Testy leksera
    1. Sprawdzenie czy poprawnie tworzone są podstawowe tokeny.
    2. Sprawdzenie czy lekser reaguje odpowiednio na niepoprawne wejście.
    3. Sprawdzenie przypadków testowych z sekcji “Przykłady wykorzystania języka”.
- Testy parsera
    1. Sprawdzenie poprawności drzewa parsowania.
    2. Sprawdzenie czy parser reaguje na błędy składniowe.
    3. Sprawdzenie czy przypadki testowe z sekcji “Przykłady wykorzystania języka” generują błędy składniowe.
- Testy egzekutora
    1. Sprawdzenie wewnętrznych struktur egzekutora.
    2. Sprawdzenie czy egzekutor reaguje na błędy semantyczne.
    3. Sprawdzenie przypadków testowych z sekcji “Przykłady wykorzystania języka”.