Matrixlang - Język z macierzami

Projekt wstępny

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 (ta druga jest zaimplementowana przy użyciu pierwszej).
5. Matrixlang jest językiem interpretowanym, interpreter napisany zostanie w języku C++.
6. Zawiera on typowanie silne oraz statyczne. Wszystkie zmienne są mutowalne, ale są one 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(string(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(string(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(string(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 / float(n);
}

int main() {
      Vector<float>[2] vector;
      vector[0] = 2;
      vector[1] = 3;
      print(string(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 = int(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 ” + string(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(string(a)); #prints 3

	a = addOne(a);
	print(string(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(string(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;
      print(string(a)); #ok
      print("$a$"); #ok
      print(a);  #error - value to print must be of type string
      return 0;
}
```

13. Bardziej złożony przykład wykorzystania języka - algorytm MergeSort. Kod napisany na podstawie implementacji w języku C.

```
# Merges two subarrays of arr[].
# First subarray is arr[l..m]
# Second subarray is arr[m+1..r]
void merge(Vector<int>[size] arr, int l, int m, int r)
{
      int i, j, k;
      int n1 = m - l + 1;
      int n2 = r - m;

      # create temp arrays
      Vector<int>[n1] L;
      Vector<int>[n2] R;

      # Copy data to temp arrays L[] and R[]
      for (i = 0; i < n1; i++)
            L[i] = arr[l + i];
      for (j = 0; j < n2; j++)
            R[j] = arr[m + 1 + j];

      # Merge the temp arrays back into arr[l..r]
      i = 0; # Initial index of first subarray
      j = 0; # Initial index of second subarray
      k = l; # Initial index of merged subarray
      for ( ; i < n1 && j < n2 ; ) {
            if (L[i] <= R[j]) {
                  arr[k] = L[i];
                  i++;
            }
            else {
                  arr[k] = R[j];
                  j++;
            }
            k++;
      }

      ``` Copy the remaining elements of L[], if there
      are any ```
      for ( ; i < n1 ; ) {
            arr[k] = L[i];
            i++;
            k++;
```
```

```
      }

      ``` Copy the remaining elements of R[], if there
      are any ```
      for ( ; j < n2 ; ) {
            arr[k] = R[j];
            j++;
            k++;
      }
}

``` l is for left index and r is right index of the
sub-array of arr to be sorted ```
void mergeSort(Vector<int>[size] arr, int l, int r)
{
      if (l < r) {
            // Same as (l+r)/2, but avoids overflow for
            // large l and h
            int m = l + (r - l) / 2;

            // Sort first and second halves
            mergeSort(arr, l, m);
            mergeSort(arr, m + 1, r);

            merge(arr, l, m, r);
      }
}

# UTILITY FUNCTIONS
# Function to print an array
void printArray(Vector<int>[size] A)
{
      int i;
      for (i = 0; i < size; i++)
            print("$A[i]$ ");
      print("\n");
}

int main()
{
      int arr_size = 6;
      Vector<int>[arr_size] arr;
      arr[0] = 12;
      arr[1] = 11;
      arr[2] = 13;
```

```
    arr[3] = 5;
    arr[4] = 6;
    arr[5] = 7;

    print("Given array is \n");
    printArray(arr); #prints “12 11 13 5 6 7 “

    mergeSort(arr, 0, arr_size - 1);

    print("\nSorted array is \n");
    printArray(arr); #prints “5 6 7 11 12 13 “
    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};
variable        = letter, [{letter | digit | “_”}];
```

Poziom składni

```
/* for print function */
string exp = string literal, {“$”, expression, “$”, string literal};

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

literal    = integer literal
           | float literal
           | “\””, string exp, “\””

prim exp   = “(“, expression, “)”
           | variable
           | constant
post exp   = (post exp, “(“, [exp list], “)”
           | post exp, “[“, expression, “]”
           | post exp, (“++” | “--”)
           | prim exp);
unary exp  = ((“++” | “--” | “+” | “-” | “!”), unary exp
           | post exp);
cast exp   = (unary exp
           | (“(“, type, “)”, cast exp);
mult exp   = cast exp, {(“*” | “/” | “%”), cast 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;
lval exp   = variable, {"[", expression, "]"};
assign exp = {lval exp, ("=" | "+=" | "-=" | "*=" | "/=" | "%=")},
             rval exp;
expression = assign exp;
exp list   = expression, {",", expression};

declaration= type, variable, ["=", expression], ";";

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

instruction= ([expression], ";")
           | if
           | switch
           | for
           | return;
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 constant, ":", instr list;
default    = "default", ":", instr list;
switch go  = "switch", "{", {case go}, [default], "}";
switch c   = "switch", "{", {case c}, [default], "}";
switch     = switch go | switch c;

for        = "for", "(", declaration, ";", expression, ";",
expression, ")", statement;

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

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. string().
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 zostanie przekazany 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 i miejscu wystąpienia błędu zostaną przekazane do strumienia błędów, będącego atrybutem egzekutora.

Sposób uruchomienia

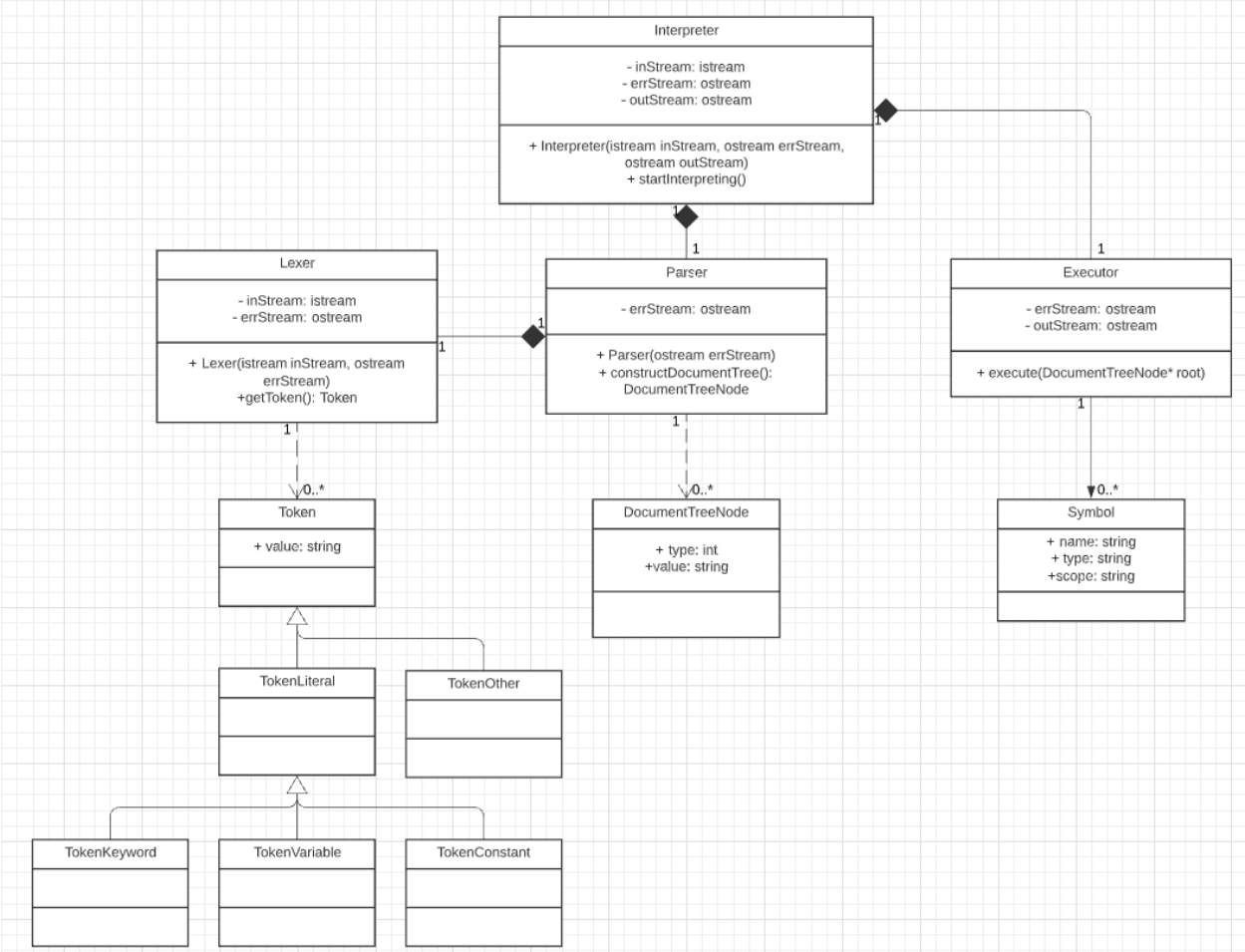
Aby uruchomić program napisany w Matrixlang'u, należy podać ścieżkę do skompilowanego interpretera języka Matrixlang, oraz ścieżkę do pliku wejściowego (z funkcją main). Ilustruje to poniższy przykład:

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

Strumieniem wyjściowym dla tak uruchomionego programu będzie domyślny strumień wyjścia, a strumieniem błędów domyślny strumień błędów. 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`".
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 domyślny strumień wejścia). Wyjście przekazywane jest na drugi strumień, również podany jako argument do obiektu leksera. Strumień błędów będzie trzecim strumieniem podanym jako argument obiektu leksera.

Sposób realizacji

Program interpretera zostanie napisany w języku C++, stąd też jego struktura będzie oparta na obiektach. Ponadto przydadzą się oferowane przez ten język strumienie.
Poniżej znajduje się wstępny diagram klas interpretera:



Sposób testowania

- Podstawą testowania będą przykłady umieszczone w sekcji “Przykłady wykorzystania języka”. Będą one testowane z punktu widzenia leksera, parsera, a także egzekutora.
    - Testowanie leksera będzie polegać na sprawdzaniu czy utworzone zostały tokeny prawidłowych typów i z prawidłowymi atrybutami.
    - Testowanie parsera to z kolei sprawdzanie czy zwrócone drzewo wykonania ma oczekiwaną strukturę.
    - Testowanie egzekutora będzie odbywać się przez sprawdzenie poprawności danych przekazywanych do strumienia wyjścia.
- Ponadto dla każdego z etapów sprawdzany będzie też strumień błędów (niektóre testy są “negatywne” - pokazują nieprawidłowe użycie języka).
- Poza testami umieszczonymi w tym dokumencie, powstaną prostsze przypadki testowe, których celem będzie sprawdzenie prostych struktur językowych (np. test sprawdzający czy instrukcja warunkowa została przetworzona poprawnie).