Matrixlang - Język z macierzami

Projekt wstępny

Piotr Satała

Spis treści

Opis zakładanej funkcjonalności

Głównym założeniem języka Matrixlang jest umożliwienie programiście korzystania z wbudowanego typu macierzy (Matrix). Pomocniczy typ wektor (Vector) będzie zrealizowany jako macierz n x 1, lecz nie będzie to widoczne dla programisty. Pozostałe typy to: int, float oraz string. 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).
Matrixlang jest językiem interpretowanym, interpreter napisany zostanie w języku C++. Zawiera on jednak 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);
}
```

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;
}
```

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;

}
```

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;
}
```

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;
}
```

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.

```
int main() {
      Vector<int>[2] vector;
      print((string)vector[0]);  #prints 0
      print((string)vector[1]);  #prints 0
      vector[0] = 3;
      vector[1] = 5;

      Vector<int>[2] vectorCopy = vector;
      print((string)vectorCopy[0]);   #prints 3
      print((string)vectorCopy[1]);   #prints 5
}
```

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

```
int main() {
      {
            int a = 10;
      }
      print((string)a);     #error - no such variable "a"
}
```

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(a);  #error - value to print must be of type string
}
```

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    = “a” | “b” | “c” | “d” | “e” | “f” | “g” | “h” | “i”
| “j” | “k” | “l” | “m” | “n” | “o” | “p” | “q” | “r” | “s” | “t” |
“u” | “v” | “w” | “x” | “y” | “z”;
upper letter    = “A” | “B” | “C” | “D” | “E” | “F” | “G” | “H” | “I”
| “J” | “K” | “L” | “M” | “N” | “O” | “P” | “Q” | “R” | “S” | “T” |
“U” | “V” | “W” | “X” | “Y” | “Z”;
letter          = lower letter | upper letter;
integer constant     = (non zero digit, {digit}) | “0”;
float constant  = integer constant, “.”, {digit};
string constant = “\””, {(letter | digit | “_”)}, “\””;
variable        = letter, [{letter | digit | “_”}];
```

Poziom składni

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

constant   = integer constant
           | float constant
           | string constant

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   = {mult exp, (“*” | “/” | “%”)}, factor;
```

```
add exp    = {add exp, ("+" | "-")}, mult exp;
rel exp    = {rel exp, ("<" | "<=" | ">" | ">=")}, add exp;
eq exp     = {eq exp, ("==" | "!=")}, rel exp;
and exp    = {and exp, "&&"}, eq exp;
or exp     = {or exp, "||"}, and exp;
assign exp = {assign exp, ("=" | "+=" | "-=" | "*=" | "/=" | "%=")},
             or exp;
expression = assign exp;
exp list   = {expression list, ","}, 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. Kopiowanie 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. Zaimplementowanie operatorów dla typów prostych (operator konkatenacji dla typu string i operatory operacji liczbowych dla typów int i float).
6. Zaimplementowanie 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. Wprowadzenie lokalności zmiennych (wewnątrz danego bloku kodu).
11. Obsługa tworzenia funkcji, które będą przyjmowały argumenty przez wartość. Funkcje te będą mogły być wywoływane rekurencyjnie.
12. Napisanie funkcji systemowych, takich jak print() czy input().
13. Obsługa wielu plików źródłowych przez operacje include.

Lista wymagań niefunkcjonalnych

1. Interpreter powinien być możliwy do uruchomienia na systemie operacyjnym Linux z wersją jądra 5.4 lub nowszą.

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 nie będzie kończył pracy interpretera, a spowoduje jedynie wyświetlenie ostrzeżenia, sam token natomiast zostanie zignorowany. Lekser będzie zmuszony przerwać pracę jeśli zacznie rozpoznawać jakiś token, a potem okaże się, że jest on nieprawidłowo zapisany (np. ciąg znaków “`zmienna^`” zacznie być rozpoznawany jako literał, 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.

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.
Głównym obiektem będzie `Interpreter` - parametry konstruktora to strumień wejścia, strumień wyjścia oraz strumień błędów.
Kolejnym obiektem jest `Lexer` - parametry jego konstruktora będą identyczne jak parametry dla obiektu `Interpreter` (nie będzie jedynie strumienia wyjścia). `Lexer` musi budować tokeny, stąd też w programie musi znaleźć się obiekt `Token`, po którym dziedziczyć będą bardziej sprecyzowane tokeny, m. in. `TokenLiteral` (po nim `TokenKeyword` i `TokenVariable`), `TokenConstant` oraz `TokenOther` (do oznaczania m. in. znaku "`[`"). Token będzie miał swój typ, który będzie dokładniej określał rodzaj tokena (np. `TokenKeyword` może mieć typ `FOR`, który będzie oznaczał token rozpoczynający pętle for). `Lexer` będzie posiadał metodę `getToken()`, która będzie zwracać jeden obiekt typu `Token`.
Metoda ta będzie wywoływana wyłącznie przez obiekt `Parser`, który będzie zawarty w obiekcie `Interpreter`. Obiekt ten będzie potrzebował strumienia błędów (który będzie taki sam jak strumień błędów `Interpreter`'a), a także obiektu `Lexer`'a jako argumentów konstruktora. `Parser` będzie posiadał metodę `constructDocumentTree()`, która korzystając z metody `getToken() Lexer`'a będzie pobierała kolejne tokeny i tworzyła drzewo całego programu (dokumentu). Powstanie obiekt `DocumentTreeNode`, który będzie reprezentował części tego drzewa. Metoda `constructDocumentTree()` zwróci obiekt typu `DocumentTreeNode`, który będzie korzeniem całego dokumentu.
Obiekt `Executor` będzie odpowiedzialny za wykonanie kodu programu w języku Matrixlang. Konstruktor tego obiektu potrzebować będzie strumienia wyjścia i strumienia błędów, takich samych jak obiekt-rodzic `Interpreter`. `Executor` będzie posiadał metodę `execute(DocumentTreeNode* root)`, której parametrem będzie korzeń drzewa dokumentu. Metoda ta będzie wykonywała kod programu pobrany wcześniej przez `Lexer` ze strumienia wejścia, a następnie przetworzony przez `Parser`. Konieczne będzie utworzenie pomocniczych struktur dla `Executor`'a, m. in. tabeli symboli, która zostanie zapewne zrealizowana jako wektor obiektów typu `Symbol`.
Wszystkie te obiekty będą mogły korzystać ze strumienia błędów, jeśli otrzymany kod źródłowy nie jest poprawny.

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).