Projekt wstępny z przedmiotu TKOM

Radosław Radziukiewicz

Wstęp

Celem projektu jest stworzenie prostego ale użytecznego języka programowania ogólnego poziomu. Poza standardowymi typami danych (patrz sekcja standardowe typy danych) język będzie udzielał wsparcia dla specjalnego rodzaju danych: typu macierzowego. Typ macierzowy będzie udostępniał specjalne, dedykowane operacje: mnożenie przez liczbę, mnożenie macierzowe, dodawania liczby, dodawanie macierzowe oraz indeksowanie.

Paradygmaty języka i typowanie

Tworzony język będzie wspierał paradygmat imperatywny proceduralny. Przewidywanym rodzajem typowania będzie typowanie dynamiczne. Zmienne będą mutowalne i ich wartość oraz typ będzie zależał od chwili wykonania programu. Zakres widoczności zmiennych będzie statyczny (leksykalny). Oznacza to, że zmienna obecna w danym bloku będzie miała dostęp do wszystkich zmiennych zadeklarowanych w blokach otaczających blok danej zmiennej. W język nie będzie istniał mechanizm przesłaniania.

Standardowe typy danych

Typ liczbowy: W języku będzie istniał typ liczbowy. Typ ten obsługiwać będzie zarówno liczby całkowite jak i zmiennoprzecinkowe. Typ liczbowy będzie wspierał operacje:

- dodawania (+)
- odejmowania (-)
- mnożenia (*)
- dzielenia (/)
- porównania (<, >, <=, >=, ==, !=)

W języku nie będzie obecna notacja naukowa.

Typ napisowy: Język będzie udzielał wsparcia dla typu napisowego. Wartości typu napisowego będą zawarte pomiędzy znakami ‘ ‘, a znakiem ”ucieczki” będzie znak /. Typ napisowy będzie dostępny jedynie w formie stałych literałów, które będą wykorzystywane we wbudowanej funkcji print. Typ napisowy nie będzie posiadał słów kluczowych ani nie będzie wspierał żadnych dodatkowych operacji.

Typ logiczny: W języku nie będzie istniał typ logiczny. Ewaluacja wartości logicznej będzie odbywać się na zasadzie przyrównania wartości liczbowych do zera, macierzy do macierzy z samymi zerami oraz ewaluacji operacji porównywania liczb. Dostępne operacje logiczne to:

- negacja (!)
- alternatywa (or)
- koniunkcja (and)

Typ macierzowy

Typ macierzowy będzie specjalnym typem dostępnym w tworzonym języku. Będzie on wspierał operacje mnożenia (przez macierz i przez liczbę), dodawania i odejmowania (macierzy i liczby) oraz

indeksowania. Błędy związane z niepoprawną operację dodawania i mnożenia macierzowego będą wykrywane w fazie egzekucji skryptu. Macierze będą mutowalne. Indeksowanie odbywa się od liczby 0.

Instrukcje warunkowe, pętle oraz funkcje

Język będzie wspierał instrukcję warunkową if …. else …., pętlę until oraz definiowanie własnych funkcji. Semantyka funkcji until będzie identyczna z semantyką funkcji while znaną np. z języka c++ tzn. until (wyrażenia bool wynosi true) {wykonuj zadane instrukcje}.

Przekazywanie wartości

Wartości będą przekazywane do funkcji na 2 sposoby. Od tego, jaki sposób zostanie wykorzystany, zależy typ argumentu funkcji. Typ liczbowy oraz napisowy będzie przekazywany poprzez wartość. Umożliwia to zatem na przekazywanie stałych literałów do funkcji. Typ macierzowy będzie przekazywany poprzez referencję. Oznacza to, że żadna zmienna macierzowa nie może być przekazana do funkcji w postaci zadeklarowanego literału (błąd semantyczny).

Operator przypisania

W przeciwieństwie do przekazywania wartości do funkcji, operator przypisania zawsze będzie wykonywał głęboką kopię obiektu z prawej strony wyrażenia, i zapisywał ją pod zmienną obecną z lewej strony wyrażenia. Przykładowo, operacja *a = b*, wykona głęboką kopię obiektu *b* (może to być macierz lub liczba) i zapisze ją jako wartość obiektu *a*. Poprzednia wartość obiektu *a* zostanie bezpowrotnie stracona.

Komentarze

Język wspiera komentarze. Komentarz zaczyna się od znaku # a kończy wraz z wystąpieniem znaku końca linii.

Biblioteka standardowa

Język udostępniać będzie małą, przydatną bibliotekę standardową. Do biblioteki tej będą należały funkcja print, cin, transpozycja macierzy, konstruktor wypełniający macierzy, konstruktor identyczności macierzy, funkcja size oraz reshape. Parametry do funkcji print przekazywane będą w postaci listy argumentów o dowolnej długości, rozdzielonej znakiem przecinka. Funkcja cin będzie przyjmowała argumenty w taki sam sposób, jednak argumenty te będą musiały być identyfikatorami zmiennych. Funkcja cin będzie umożliwiała odczyt liczb oraz macierzy, a print wydruk liczby, macierzy i napisu.

Przykład: cin(a, b) # Odczyt dwóch danych i zapisanie ich do zmiennych a oraz b

Gramatyka

Poniżej zaprezentowana jest gramatyka opisanego języka:

```
program              = { functionDefinition }
functionDefinition   = identifier ”(” parameters ”)” statementBlock
parameters           = [ identifier { ”,” identifier } ]
statementBlock       = ”{” {
```

```
                                ifStatement |
                                untilStatement |
                                assignOrFuncCall |
                                statementBlock
                            } "}"

ifStatement     = "if" "(" condition ")" statementBlock
                [ "else" (ifStatement | statementBlock ) ]
untilStatement  = "until" "(" condition ")" statementBlock
assignOrFuncCall = identifier ( "(" arguments ")" |
                                [ indexOperator ] "=" expression )

arguments       = ( [ argument { "," argument } ] )
argument        = expression | string

expression      = mulExpression { ( "+" | "-" ) mulExpression }
mulExpression   = atomicExpression {( "*" | "/" ) atomicExpression }
atomicExpression= ( identOrFuncCall | literal | parenthExpression |
                    string )
identOrFuncCall = identifier [ "(" arguments ")" ]
parenthExpression = "(" condition ")"

condition       = andCondition { "or" andCondition }
andCondition    = relCondition { "and" ( relCondition |
                                         parenthCondition }
relCondition    = [ "!" ] ( atomicCondition [ ( "<" | "<=" | ">" |
                    ">=" | "==" | "!=" ) atomicCondition ] )
atomicCondition = expression

literal    = matrixLiteral | numberLiteral

indexOperator = "[" (":" | expression )(":" | expression ) "]"
```

```
matrixLiteral = ”[” expression { (”,” | ”;”) expression } ”]”
numberLiteral = ( 0|[1-9][0-9]*)(.[0-9]*)?
identifier    = [a-zA-Z][a-zA-Z0-9_-]*
string        = ” [znaki alfa-numeryczne]* ”
```

Przykładowy kod

Przykład 1

Przykład różnych deklaracji macierzy.

```
main() {
        Matrix1 = [ 1, 2, 3;
                    1, 2, 3;
                    1, 2, 3; ]
        Martrix2 = ident(3) # Stworzenie macierzy identycznościowej o wymiarach 3 x 3.
        resultAdd =Matrix1 + Matrix2
        Rot90Deg= [ 0, -1;
                     1,  0; ]
        myVector = [ 0; 2]
        rotatedVector = Rot90Deg * myVector # Wynikiem będzie wektor [ -2; 0 ]
}
```

Przykład 2

Przykład indeksowania macierzy.

```
main() {
        Matrix1 = [ 1, 2, 3;
                    4, 5, 6;
                    7, 8, 9; ]
        Num1 = Matrix1[0, 0]        # Wynik to 1
        Row1 = Matrix1[0 , :]       # Wynik to 1, 2, 3
        Col1 = Matrix1[:, 0]        # Wynik to 1, 4, 7
}
```

Przykład 3

Przykład działania pętli i instrukcji warunkowych.

```
main() {
        Matrix = [ 1, -3;
                   -9, 4; ]
        i = 0  j = 0
        matrixSize = size(Matrix)
        until (i  < matrixSize[0,0] and j < matrixSize[0,1]) {
                if (!Matrix[i, j]) {
                        if (j + 1 < matrixSize[0,1]) {
                                j = j + 1
                        } else {
                                I = i + 1
                        }
                } else if (Matrix[i, j] < 0 ) {
                        Matrix[i, j] = Matrix[i, j] + 1
                } else {
                        Matrix[i, j] = Matrix[i, j] – 1
                }
        }
}
```

Przykład 4

Przykład definiowania własnych funkcji i jej wykorzystania.

```
rot90Deg(matrix) {
        rotMatrix = [ 0, -1;
                      1, 0; ]
        return matrix * rotMatrix
}
main() {
        myVec = [ 2; 0; ]
        myVec = rot90Deg(myVec)          # Wynik to wektor 0; 2
        myVec = myVec * 0
        myVec = rot90Deg(myVec)          # Wynikiem będzie wektor zerowy.
```

```
}
```

Przykład 5

Przykład działania rekurencji.

```
factorial(number) {
        If (number == 1) {
                return number
        }
        return number * factorial(number-1)
}
main() {
        factorialOf5 = factorial(5)
        print(factorialOf5)             # Wynikiem będzie 120
}
```

Przykład 6

Przykład działania przekazywania zmiennych do funkcji.

```
changeMyMatrix(matrix) {
        matrix[0, 0] = -1000            # Macierz przekazywana przez referencję.
}
trytoChangeMyNumber(number) {
        number = -1000                  # Liczba przekazywana przez wartość.
}
main() {
        matrix = ident(2)               # Macierz identycznościowa o rozmiarze 2 x 2.
        number = 5
        changeMyMatrix(matrix)
        tryToChangeMyNumber(number)
        print(matrix, number)           # Wynik to macierz [-1000, 0; 0, 1] oraz numer 5.
}
```

Moduły

Sekcja ta ulegnie dużej modyfikacji w trakcie wykonania projektu. Nie powinna ona zatem być uważana z ostateczną lub „mocno wiążącą”.

Program obsługujący stworzony język powinien radzić sobie z obsługą błędów. Wykonanie tego zadania niesie za sobą potrzebę monitorowania stanu wejście do programu, aby móc zwrócić cenną informację o zaistniałym błędzie. W tym celu w module odpowiedzialnym za odczyt danych należy wyodrębnić klasę odpowiedzialną za zliczanie aktualnych wierszy i kolumn. Moduł obsługujący wejście będzie udostępniał możliwość wykonywania operacji wyj/wej na plikach, oraz na specjalnych interfejsach użytych do przeprowadzenia testów programu.

Analizator leksykalny podczas odczytu wejścia powinien usuwać redundantne białe znaki oraz komentarze. Rezultatem działania leksera będzie ciąg tokenów, wykorzystywany przez parser do stworzenia drzewa programu.

Klasa reprezentująca token będzie obiektem składającym się z trzech pól: typu, opcjonalnej wartości i wystąpieniu w źródle (numer wiersza i linii). Wartość tokenu będzie już przekonwertowaną liczbą lub napisem. Warto zaznaczyć tutaj, iż macierz będzie wytworem na poziomie składniowym, a nie leksykalnym. Nie będzie zatem istniał token reprezentujący macierz.

Analizator składniowy (parser) powinien być w stanie stwierdzić poprawność składniową stworzonego drzewa. W tym celu będzie on korzystał z tablic dostępnych symboli oraz drzewa wyprowadzenia. Ponadto należy zatroszczyć się o moduł zawierający definicje wszystkich napotkanych identyfikatorów (napotkanych oznacza dostępnych wdanym momencie działania programu).

Ostatnim modułem zawartym w programie będzie analizator semantyczny (interpreter). W trakcie jego działania będą wykrywane błędy związane z niepoprawnym wykorzystaniem zmiennych lub dostępnych operacji arytmetycznych.

Testowanie

Podstawą testowania poprawności programu będą testy jednostkowe oraz integracyjne . W przypadku ich realizacji zamierzam stosować odpowiednie wzorce projektowe zapewniające odpowiednie badanie stworzonych struktur danych i ich weryfikację (np. wzorzec odwiedzający).

Analizator leksykalny będzie testowany za pomocą dwóch rodzajów testów. Pierwsze z nich, czyli testy jednostkowe, będą sprawdzały poprawne rozpoznanie i ewaluację atomowych tworów języka. Testy te będą również sprawdzały, czy analizator zachowuje się w sposób poprawny w przypadku wykrycia błędu. Drugim rodzajem testu będą testy integracyjne, sprawdzające, czy analizator składniowy popranie wykrywa, oraz raportuje błędy z ciągu tokenów (przykładowo 4 – 5 obiektów).

Analizator składniowy będzie poddany, podobnie jak analizator leksykalny, testom jednostkowym oraz integracyjnym. Przy testowaniu tego modułu bardzo przydatny może okazać się wspomniany wzorzec odwiedzającego, który będzie weryfikował poprawność stworzonego drzewa wyprowadzenia. Testy integracyjne będą sprawdzały poprawne działania analizatora składniowego przy połączeniu z analizatorem leksykalnym.

Analizator semantyczny będzie podlegał jeszcze dodatkowo testom akceptacyjnym (analizator leksykalny + składniowy + semantyczny). Weryfikowane będzie, czy wynik działania całego programu jest zgodny z oczekiwanym lub czy wykonanie programu zakończyło się niepowodzeniem (również w sposób oczekiwany).