Interpreter języka skryptowego z wbudowanym typem macierzy - Dokumentacja końcowa

Autor: Michał Dziekoński

Opis projektu

Ideą projektu jest stworzenie interpretera prostego języka skryptowego, wyposażonego w podstawowe instrukcje sterujące (instrukcja warunkowa i pętla), możliwość definiowania własnych funkcji oraz wbudowany typ macierzowy.

Dla uproszczenia języka, przyjęte zostały następujące założenia:

- Jedynym typem wbudowanym jest typ macierzowy, który wewnątrz siebie przechowuje tylko i wyłącznie liczby. Obsługiwane są macierze jedno i dwu-wymiarowe.
- Wszystkie „wolne” liczby są automatycznie konwertowane do macierzy rozmiaru 1x1
- W wyrażeniach warunkowych (dla instrukcji sterujących), wszelkie „wolne” (niezwiązane z żadną operacją porównania) zmienne lub bezpośrednie wypisania macierzy są rzutowane na wynik operacji logicznej - „puste” macierze (zawierające same zera) są zamieniane na „fałsz”, każda inna macierz na „prawdę”
- W wyrażeniach warunkowych można używać tylko i wyłącznie operacji logicznych, operacji porównania oraz zmiennych i literałów. Jeśli wymagane są operacje arytmetyczne (np. dodanie dwóch macierzy i sprawdzenie, czy wynik jest “zerowy”), należy wykonać je przed wyrażeniem warunkowym, przypisując wynik do zmiennej.
- Wszystkie zmienne i literały przekazywane są do funkcji jako referencje
- Każda zmienna ma stały rozmiar, aby zmienić jej rozmiar należy nadpisać jej zawartość.
- Funkcje mogą być definiowane tylko i wyłącznie w głównym zasięgu programu i nie mogą być nadpisywane.
- Część operacji (jak utworzenie macierzy o zadanym rozmiarze czy wypisanie macierzy na standardowe wyjście, funkcje zwracające rozmiar macierzy, itd.) będzie zdefiniowana jako „biblioteka standardowa” języka.
- Każdy program musi zawierać funkcję „main”, będącą punktem wejściowym dla jednostki wykonującej program.

Lista zdefiniowanych tokenów

“function”, “(“, “)”, “,”, “{“, “}”, “;”, “if”, “while”, “else”, “return”, “var”, “!”, “=”, “or”, “||”, “and”, “&&”, “==”, “!=”, “<”, “>”, “<=”, “>=”, “+”, “-”, “*”, “/”, “%”, “[“, “]”, “Infinity”, “.”

Gramatyka

*(W stosunku do dokumentacji wstępnej poprawiony został wpis **`equalityCond i relationalCond`** – gdzie należało wymusić "binarność" operacji)*

```
program = { functionDef } .
functionDef = "function" id parameters statementBlock .
parameters = "(" [ id { "," id } ] ")" .
arguments = "(" [ assignable { "," assignable } ] ")" .
statementBlock = "{" { ifStatement | whileStatement | returnStatement ";" |
       initStatement ";" | assignStatement ";" | funCall ";" | "continue" ";" |
       "break" ";" | statementBlock } "}" .
ifStatement = "if" "(" condition ")" statementBlock [ "else" statementBlock ] .
whileStatement = "while" "(" condition ")" statementBlock .
returnStatement = "return" assignable .
initStatement = "var" id [ assignmentOp assignable ] .
assignStatement = variable assignmentOp assignable .

assignable = funCall | expression .
expression = multiplicativeExpr { additiveOp multiplicativeExpr } .
multiplicativeExpr = primaryExpr { multiplicativeOp primaryExpr } .
primaryExpr = ( literal | variable | parenthExpr )
parenthExpr = "(" expression ")" .

condition = andCond { orOp andCond } .
andCond = equalityCond { andOp equalityCond } .
equalityCond = relationalCond [ equalOp relationalCond ] .
relationalCond = primaryCond [ relationOp primaryCond ] .
primaryCond = [ unaryOp ] ( parenthCond | variable | literal ) .
parenthCond = "(" condition ")" .

unaryOp = "!" .
assignmentOp = "=" .
orOp = "or" | "||" .
andOp = "and" | "&&" .
equalOp = "==" | "!=" .
relationOp = "<" | ">" | "<=" | ">=" .
additiveOp = "+" | "-" .
multiplicativeOp = "*" | "/" | "%" .
indexOp = "[" assignable "]" [ "[" assignable "]" ] .

literal = numberLiteral | matrixLiteral .
variable = id [ indexOp ] .
funCall = id arguments .

numberLiteral = [ "-" ] "Infinity" | finiteNumber .
finiteNumber = digit { digit } [ "." { digit } ] .
matrixLiteral = "[" numberLiteral { "," numberLiteral } { ";" numberLiteral { ","
numberLiteral } } "]" .

id = letter { digit | letter }
letter = "a".."z" | "A".."Z" | "_" | "$" .
digit = "0".."9" .
```

Wymagania funkcjonalne

- Odczytywanie, parsowanie i analiza skryptów zapisanych w plikach tekstowych
- Kontrola poprawności wprowadzonych danych oraz poprawne zgłaszanie błędów wykrytych podczas kolejnych etapów analizy plików
- Poprawne wykonywanie wszystkich poprawnie zapisanych instrukcji w plikach skryptowych
- Możliwość definiowania własnych funkcji oraz ich późniejszego wywoływania w skryptach
- Przestrzeganie logicznego porządku instrukcji sterujących
- Przeprowadzanie operacji arytmetycznych na macierzach
- Wyświetlanie wyników operacji macierzowych na standardowym wyjściu terminala

Wymagania niefunkcjonalne

- Uruchomienie aplikacji z niepoprawnymi parametrami powinno w przejrzysty sposób poinformować użytkownika o możliwych parametrach startowych
- Komunikaty o błędach analizy plików powinny być proste i przejrzyste dla użytkownika, i w sposób jednoznaczny wskazywać popełnione błędy
- Wyniki uruchomienia skryptów powinny być prezentowane w formacie identycznym lub zbliżonym do formatu wejściowego, by ułatwić ponowne wykorzystanie danych wyjściowych w następnych uruchomieniach skryptów

Idea budowy i działania projektu

Środowisko uruchomieniowe i wybrane technologie

Projekt będzie pisany z myślą o systemach z rodziny **Linux**, jako aplikacja czysto konsolowa. Projekt będzie pisany w języku **C++** (standard C++11), zaś do jego kompilacji zostanie wykorzystany kompilator **GCC** lub **Clang** oraz program **Scons** do konfiguracji procesu budowania.

Obsługa programu

Program będzie prostą aplikacja konsolową, uruchamianą poprzez wywołanie wraz z parametrem uruchomieniowym reprezentującym ścieżkę do pliku ze skryptem do interpretacji, oraz ewentualnymi flagami (np. uruchomienie dokładniejszego trybu zgłaszania błędów, tryb debugowania - dokładne wypisywanie działań poszczególnych modułów).
Wynik poszczególnych etapów analizy pliku oraz samego wyniku interpretacji końcowej i wykonania będzie wyświetlany na standardowym wyjściu. W zależności od ogólnego wyniku analizy, na standardowe wyjście mogą być zgłaszane: błędy analizy pliku, błędy składniowe, błędy semantyczne lub wynik wykonania skryptu (wraz z możliwymi błędami czasu wykonania). Jako że jest to aplikacja konsolowa, nie przewiduję zapisywania wyników do pliku (można to zrobić przekierowując wyjście bezpośrednio do pliku).

Budowa programu

Program będzie złożony z modułów odpowiedzialnych za kolejne etapy analizy plików wejściowych, oraz z dodatkowych modułów pomocniczych, wspomagających cały proces.

Cały proces analizy i wykonywania skryptów będzie odbywał się w następujących etapach:

1. Analiza leksykalna (moduł lexera) + Analiza składniowa (moduł parsera)
2. Analiza semantyczna (moduł analizatora semantycznego)
3. Wykonanie zbioru instrukcji (moduł wykonawczy)

Analiza leksykalna

Moduł lexera będzie odpowiedzialny za rozbicie otwartego pliku wejściowego na tokeny, które są potrzebne dla analizatora składniowego. Lexer będzie odczytywał znak po znaku, do czasu stwierdzenia odczytania całości sekwencji odpowiadającej jednemu z rozpoznawanych i akceptowanych tokenów języka. Tokeny te będą zwracane do parsera, jeden po drugim, każdy dla pojedynczego wywołania metody lexera “odczytaj następny token” przez parser.
Moduł ten będzie wspierany przez moduł obsługi pliku wejściowego oraz tablicę akceptowalnych tokenów, będzie miał również połączenie z modułem obsługi błędów.

Analiza składniowa

Moduł parsera będzie pracował przy współpracy analizatora leksykalnego, ładując za jego pomocą kolejne tokeny. Jego zadaniem jest sprawdzenie, czy wszystkie rozpoznane tokeny są ułożone zgodnie ze zdefiniowaną gramatyką języka. Po pobraniu tokena, analizator decyduje którą ścieżką ma teraz podążać, stopniowo schodząc w głąb analizowanego skryptu. Wymagane będą do tego odpowiednie procedury rozbioru. Zatwierdzone struktury gramatyczne będą tworzyć drzewo składniowe, korzystając z niżej wymienionych klas do utworzenia odpowiednich instancji struktur języka.
Moduł ten będzie miał połączenie z modułem obsługi błędów.

Analiza semantyczna

Moduł analizatora semantycznego ma za zadanie sprawdzić poprawność ”znaczenia” utworzonego przez analizator składniowy drzewa. Analizator zstępuje w głąb drzewa, analizując zgodność skryptu, tj. poprawność używanych identyfikatorów, brak nadpisywania identyfikatorów funkcji (zarówno tych ze “standardowej biblioteki” jak i zdefiniowanych przez użytkownika), zgodność kontekstów operacji arytmetycznych czy logicznych, zgodność ilości parametrów wywołań funkcyjnych itd.
Podczas sprawdzania, generowane jest drzewo reprezentacji instrukcji do wykonania.
Moduł będzie miał połączenie z modułem obsługi błędów, będzie również konieczna współpraca z modułem zdefiniowanych nazw (wraz z tablicą biblioteki standardowej).

Wykonanie zbioru instrukcji

Moduł wykonawczy ma za zadanie sekwencyjne wykonanie instrukcji zawartych w drzewie wytworzonym na poprzednich etapach. Na tym poziomie jedynymi błędami jakie mogą się

pojawić, są błędy związane z zawartościami zmiennych - np. nieodpowiednie argumenty do operacji dodawania dwóch macierzy (niezgodność rozmiarów macierzy).
Po wykonaniu wszystkich instrukcji program kończy swoje działanie.
Moduł będzie miał połączenie z modułem obsługi błędów, z modułem zdefiniowanych identyfikatorów oraz modułem obsługi wyjścia programu.

Dodatkowe moduły

- **Moduł obsługi plików** - organizuje operacje odczytywania zawartości z pliku
- **Tablica akceptowalnych tokenów** - predefiniowana tablica zawierająca wykaz wszystkich symboli, jakie powinien rozpoznawać lexer
- **Moduł obsługi błędów** - organizuje prezentację błędów w różnych etapach analizy plików, w sposób czytelny dla użytkownika programu
- **Moduł zdefiniowanych nazw** - organizuje strukturę wszystkich zdefiniowanych przez użytkownika identyfikatorów. Moduł będzie współpracował z tablicą biblioteki standardowej.
- **Tablica biblioteki standardowej** - predefiniowana tablica wbudowanych funkcji, dostępnych do użycia przez użytkownika w skrypcie.
- **Moduł obsługi wyjścia** - organizuje prezentację wyników wykonania skryptów, w sposób czytelny do użytkownika programu

Klasy drzewa składniowego (AST)

- **Node -** główna klasa dla obiektów w drzewie
- **Assignable** - dodatkowa klasa pośrednia reprezentująca wartość “do przypisania”
- **FunDefinition** - definicja funkcji
- **StatementBlock -** przechowuje informacje na temat instrukcji zawartych wewnątrz tego bloku
- **IfStatement -** przechowuje warunek, blok do wykonania w przypadku “prawdy” oraz ewentualny blok do wykonania w przypadku “fałszu”
- **WhileStatement -** przechowuje warunek kontynuacji oraz blok do wykonywania
- **Condition -** warunek logiczny wejścia w instrukcję warunkową
- **ReturnStatement** - operacja zwrócenia wartości z funkcji
- **VarDeclaration** - deklaracja zmiennej w danym bloku
- **Assignment -** przypisanie wartości do zmiennej
- **Expression** - wyrażenie przypisywane do zmiennej
- **Call -** wywołanie zdefiniowanej funkcji lub funkcji ze standardowej biblioteki
- **Variable** - deskryptor zmiennej, przechowuje jej wartość i ewentualne dane statystyczne
- **Matrix** - reprezentacja macierzy
- **LoopJump** - operacje przerywania/kontynuacji w pętli
- **Program** - wektor zdefiniowanych funkcji

Klasy drzewa wykonywania (IR)

- **Executable** - klasa bazowa dla wszystkich elementów, które mogą zostać wykonane
- **Assignable** - klasa pośrednicząca reprezentująca wartość “do przypisania”

- **Block** - klasa reprezentująca zestaw instrukcji z określonym kontekstem zmiennych
- **Condition** - klasa obliczająca wartość wyrażeń logicznych dla instrukcji sterujących
- **ConditionOperand** - klasa pośrednicząca, stanowiąca element po jednej ze stron operatora logicznego
- **Expression** - klasa obliczająca wartość wyrazenia arytmetycznego dla instrukcji przypisania
- **ExpressionOperand** - klasa pośrednicząca, stanowiąca element po jednej ze stron operatora arytmetycznego
- **Function** - klasa reprezentująca właściwe wywołanie funkcji, wraz z generacją nowego kontekstu
- **Instruction** - klasa bazowa dla instrukcji
- **Literal** - klasa reprezentująca dane (wartość zmiennych)
- **Variable** - klasa reprezentująca identyfikator dostępowy do zmiennej w kontekście wykonania
- **Assignment** - instrukcja przypisania wartości do zmiennej
- **Call** - instrukcja wywołania funkcji, służy do “ukonkretnienia” zmiennych i znalezienia odpowiedniego dowiązania do funkcji
- **IfStatement** - instrukcja warunkowa
- **LoopJump** - instrukcja wyskoku lub kontynuacji pętli
- **Return** - instrukcja służy do ukonkretnienia wartości zwracanej przez funkcję
- **WhileStatement** - instrukcja pętli
- **ScopeProto** - prototyp dla kontekstu, służy za fabrykę dla późniejszych instancji przy wykonaniu programu
- **ScopeInst** - instancja kontekstu, używana przy wykonywaniu programu

Biblioteka standardowa

Wstępnie przewidywana biblioteka funkcji wbudowanych:

- **Generalne funkcje macierzowe**
    - **generateMatrix**(width, height, fill = 0)
      Generuje macierz o zadanym rozmiarze i wypełnieniu (domyślnie, każde pole macierzy ma wartość **0**)
    - **resizeMatrix**(matrix, newWidth, newHeight, fill = 0)
      Zmienia rozmiar zadanej macierzy, wypełniając ewentualne nowe pola zadaną liczbą (domyślnie, wartość **0**)
    - **size**(matrix)
      Zwraca rozmiar zadanej macierzy, jako wektor o dwóch wartościach:
      [szerokość, wysokość]
    - **print**(matrix)
      Wypisuje zawartość macierzy na standardowe wyjście
    - **printEmpty**()
      Wyświetla pustą linię
- **Operacje na macierzach**

- **transpose**(matrix)
  Wykonuje transpozycję na zadanej macierzy
- **operacje arytmetyczne** (dodawanie, odejmowanie, mnożenie)
  Zostaną zintegrowane z operatorami arytmetycznymi

Przykład poprawnego skryptu obsługiwanego przez interpreter

```
function isDiag(matrix)
{
	if (!(matrix))
	{
		return 0;
	}
	var mSize = size(matrix);
	print(mSize);
	if (mSize[0][0] != mSize[0][1])
	{
		return 0;
	}

	var i = 0;
	while (i < mSize[0][0])
	{
		var j = 0;
		while (j < mSize[0][1])
		{
			if (i != j && matrix[i][j] != 0)
			{
				return 0;
			}

			j = j + 1;
		}

		i = i + 1;
	}

	return 1;
}



function main()
{
	var gen = generate(3, 3, 0);

	gen[1] = [ 0, 2, 0];
	print(gen);
	print(isDiag(gen));

	printEmpty();

	gen = resize(gen, 4, 4, 1);
	print(gen);
	print(isDiag(resize(gen, 4, 4, 1)));

	return 0;
}
```