Interpreter prostego języka

- Dokumentacja

Autor: Aleksander Zamojski

# Opis projektu

Projekt ma na celu wykonanie interpretatora prostego języka. Język ma być wyposażony w zmienne z zasięgiem, dwie podstawowe konstrukcje sterujące (pętla oraz instrukcja warunkowa), możliwość definiowania funkcji oraz wbudowany typ wektorowy (2-, 3-wymiarowy). Dodatkowo język powinien obsługiwać wyrażenia matematyczne uwzględniając priorytet operatorów.

# Funkcjonalność

* Odczytanie, parsowanie i analiza skryptów zapisanych w plikach tekstowych
* Kontrola poprawności wprowadzonych danych oraz zgłaszanie błędów wykrytych podczas kolejnych etapów analizy plików
* Wykonywanie poprawnie zapisanych instrukcji, nie produkujących błędów, z plikach wejściowych
* Możliwość definiowania typów:
  + number (liczba całkowita)
  + vec, (2,3-wymiarowe wektory)
  + string (typ znakowy istniejący tylko w funkcji print)
* Wykonanie wyrażeń matematycznych uwzględniając priorytet operatorów ((),\*,/,+,-)
* Wykonanie wyrażeń logicznych uwzględniając priorytet operatorów ((),==,||,&&)
* Wykonywanie operacji na wektorach (dodawanie, odejmowanie, iloczyn skalarny, iloczyn wektorowy)
* Możliwość używania instrukcji warunkowych oraz pętli
* Funkcja print, wypisująca informacje podane przez użytkownika
* Możliwość definiowania własnych funkcji oraz ich późniejszego wywoływania w skryptach
* Użycie typizacji silnej i dynamicznej

# Biblioteka standardowa

Wstępnie przewidywana biblioteka funkcji wbudowanych:

* Podstawowe
  + print (…)  
    Wypisuje zawartość na standardowe wyjście.
* Operacje na wektorach
  + convertFrom2dTo3d(vec, vec)  
    Zmienia wektor dwuwymiarowy na trójwymiarowy.
  + convertFrom3dTo32(vec, vec)  
    Zmienia wektor trójwymiarowy na dwuwymiarowy.
  + crossProduct2(vec, vec)  
    Wykonuje iloczyn wektorowy na dwóch dwuwymiarowych wektorach.
  + crossProduct3(vec, vec)  
    Wykonuje iloczyn wektorowy na dwóch trójwymiarowych wektorach.
  + scalarProduct2(vec, vec)  
    Wykonuje iloczyn skalarny na dwóch dwuwymiarowych wektorach.
  + scalarProduct3(vec, vec)  
    Wykonuje iloczyn skalarny na dwóch trójwymiarowych wektorach

# Gramatyka

**program** = { functionDef } ;

**functionDef** = "function" identifier parameters statementBlock ;

**parameters** = "(" [ identifier { "," identifier } ] ")" ;

**statementBlock** = "{" { initStatement | assignStatement | returnStatement |

ifStatement | whileStatement | functionCall ";" | "continue" ";" |

"break" ";" | printStatement | statementBlock } "}" ;

**returnStatement** = "return" logicExpr ";" ;

**initStatement** = "var" identifier [ "=" logicExpr ] ";" ;

**assignStatement**​ = variable "=" logicExpr ";" ;

**ifStatement** = "if" "(" logicExpr ")" statementBlock [ "else" statementBlock ] ;

**whileStatement** = "while" "(" logicExpr ")" statementBlock ;

**functionCall** = identifier arguments ;

**arguments**​ = "(" [ logicExpr { "," logicExpr } ] ")" ;

**printStatement** = "print" "(" (stringLiteral | logicExpr) {"," (stringLiteral | logicExpr)} ")" ";"

**logicExpr**​ = andExpr { orOp andExpr } ;

**andExpr**​ = relationalExpr { andOp relationalExpr } ;

**relationalExpr** ​= baseLogicExpr [ relationOp baseLogicExpr ] ;

**baseLogicExpr**​ = [ unaryLogicOp ] mathExpr ;

**mathExpr** = multiplicativeExpr { additiveOp multiplicativeExpr } ;

**multiplicativeExpr** ​= baseMathExpr { multiplicativeOp baseMathExpr } ;

**baseMathExpr** ​= [unaryMathOp ] (value | parentLogicExpr) ;

**parentLogicExpr**​ = "(" logicExpr ")" ;

**value** = numberLiteral |vectorLiteral | variable | functionCall ;

**unaryMathOp**​ = "-" ;

**unaryLogicOp**​ = "!" ;

**additiveOp** ​= "+" | "­" ;

**multiplicativeOp** ​= "\*" | "/" | "%" ;

**orOp** ​= "||" ;

**andOp**​ = "&&" ;

**relationOp** ​= "==" | "!=" | "<" | ">" | "<=" | ">=" ;

**variable**​ = identifier [ index ] ;

**index** = "[" numberLiteral "]" ;

**stringLiteral** = '"' { allCharacters – '"' } '"' ;

**vectorLiteral** = "vec" "(" numberLiteral "," numberLiteral ["," numberLiteral] ")" ;

**numberLiteral** = digit { digit } ;

**identifier** = letter { letter | digit | specialElement } ;

**specialElement** = "\_" | “@" ;

**letter** = "a".."z" | "A".."Z" ;

**digit** = "0".."9";

**allCharacters** = ? all visible characters ? ;

# Informacje techniczne

## Środowisko

Projekt zaimplementowany w języku C++, wykorzystuje bibliotekę do testów jednostkowych: "Catch". Całość jest budowana za pomocą "CMake".

## Obsługa programu

Program jest prostą aplikacja konsolową, uruchamianą wraz z parametrem reprezentującym ścieżkę do pliku ze skryptem do interpretacji.  
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 leksykalny, 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).

# Krótki kurs języka

Podstawowym obiektem języka są funkcje, to oznacza, że zawsze trzeba zacząć pisanie od definicji funkcji:

function id (variable, …) {…}

gdzie „function” to słowo klucz oznaczające rozpoczęcie definiowania funkcji. „id” to unikalna nazwa, w nawiasach wpisujemy nazwy parametrów wywołania funkcji, jeśli chcemy, aby funkcja je miała, a ich ilość jest nie ograniczona. Następnie musimy otworzyć blok funkcji, w którym jesteśmy w stanie wpisać resztę komend. Ich ilość jest nie ograniczona. Na końcu trzeba zamknąć blok.

Typy danych

* 1 - liczba całkowita
* vec(1,1) - wektor dwuwymiarowy
* vec(1,1,1) - wektor trójwymiarowy

Komendy:

Definicja zmiennej „a”:

* var a; - pusta definicja zmiennej.
* var a = 1; - definicja wraz z inicjalizacją.

Zmiana wartości zmiennej „a”:

* a = 1;

Blok warunkowy „if”:

* if( 1 ) { … }
* if( 1 ) { … } else { … }

Blok warunkowy „while”:

* while( 1 ) { … }

Wywołanie funkcji „fun” z jednym argumentem.

* fun( 1 ); - liczba argumentów w nawiasach musi się zgadzać z liczbą parametrów wywoływanej funkcji,   
   a każdy kolejny parametr musi być oddzielony od poprzedniego przecinkiem: 1,1,1,…

Wyrażenia matematycze:

* W każdym z miejsc w którym jest wpisana 1, można wpisać wyrażenie arytmetyczne, typ danych, wywołanie funkcji lub ich kombinacje:
  + 1 + 2 - dodanie do siebie dwóch liczb
  + 1 + a - dodanie 1 i zmiennej „a”
  + fun(a) - wywołanie funkcji “fun” podając w argumencie zmienną “a”

# Przykłady

|  |  |  |
| --- | --- | --- |
| function fun(a) {  print(a);  return 3; }  function main() {  fun(1);  fun(fun(2));  print(vec(1,2));  print(vec(1,2,3));   var a = 0;  if(a == 0) {  print("a= ", a);  }   var b = a;  while(b < 10) {  b = b + 1;  }  print("b= ", b); }  Wynik:  (1)  (2)  (3)  (1,2)  (1,2,3)  a= (0)  b= (10) | function main() {  print(vec(1,2),  " to ",  convertFrom2dTo3d(vec(1,2)) );  print(vec(1,2,3),  " to ",  convertFrom3dTo2d(vec(1,2,3)) );   print(vec(1,2)," X ",vec(1,2),  " = ",  crossProduct2(vec(1,2),vec(1,2)) );  print(vec(1,2,3)," x ",vec(1,2,3),  " = ",  crossProduct3(vec(1,2,3),vec(1,2,3)) );   print(vec(1,2)," o ",vec(1,2),  " = ",  scalarProduct2(vec(1,2),vec(1,2)) );  print(vec(1,2,3)," o ",vec(1,2,3),  " = ",  scalarProduct3(vec(1,2,3),vec(1,2,3))); }  Wynik:  (1,2) to (1,2,0)  (1,2,3) to (1,2)  (1,2) X (1,2) = (0,0)  (1,2,3) x (1,2,3) = (0,0,0)  (1,2) o (1,2) = (5)  (1,2,3) o (1,2,3) = (14) | function fun(a) {  {  a = a \* 2;  if (a < 8) {  fun(a);  }  print("a ", a);  return a;  } }  function main() {  print(fun(2)); }  Wynik:  a (8)  a (4)  (4) |