Interpreter prostego języka skryptowego

Dokumentacja końcowa

Paweł Kaczyński

# Opis projektu

Celem projektu była implementacja interpretera prostego języka skryptowego. Składnia języka umożliwia operacje na prostych typach wbudowanych, używanie pętli oraz instrukcji warunkowych, wykonywanie operacji arytmetycznych oraz definiowanie własnych funkcji i ich wywoływanie.

### Założenia

* Instrukcje kończą się średnikiem.
* Białe znaki pomiędzy instrukcjami są ignorowane.
* Dozwolone komentarze liniowe (zaczynające się od „//”)
* Funkcje definiowane są w głównym zasięgu programu. Każda z nich musi mieć unikalną nazwę i musi kończyć się instrukcją return (pustą lub nie). Kolejność definiowana nie ma znaczenia.
* Program musi zawierać funkcję „program NAZWA”, która jest punktem startowym programu.
* Wywołanie funkcji musi zawierać wszystkie argumenty i dokładnie tyle. Brak argumentów domyślnych.

# Tokeny i gramatyka

### Zbiór używanych tokenów

program, function, return, var, const, int, bool, string, float, true, false, if, while, else

( ) { } , ; . + - / \* %

### Produkcje gramatyki

--- Program ---

Program = "program" Name FuncArgs FuncBody { Function }

--- Functions -----------------------------------------------

Function = "function" Name FuncArgs FuncBody

FuncArgs = "(" ( [Type Name] | (Type Name { "," Type Name }) ) ")"

FuncBody = "{" { Statement } ReturnInstr ";" "}"

ReturnInstr = "return" [ RValue ]

--- Statements ------------------------------------------

Statement = Instruction | IfBlock | WhileStatement | LineComment

Instruction = (VarDeclaration | ConstDeclaration | Assignment | FuncCall | ReturnInstr) ";"

IfBlock = IfStatement { "else" IfStatement } [ "else" (Body | SingleInstr) ]

IfStatement = "if" "(" LogicalExpr ")" (Body | SingleInstr)

WhileStatement = "while" "(" LogicalExpr ")" Body

LineComment = "//" String "\n"

--- Instructions -----------------------------------------

VarDeclaration = "var" ( (Type Name) | TypeAssignment )

ConstDeclaration = "const" TypeAssignment

FuncCall = Name "(" ([ RValue ] | RValue { "," RValue }) ")"

Assignment = Name AssignmentOp [ "<" Type ">" ] RValueAssign

TypeAssignment = IntAssign | FloatAssign | StringAssignment | BoolAssign

IntAssign = IntType Name AssignmentOp [ "<" IntType | FloatType | BoolType ">" ] RValue

FloatAssign = FloatType Name AssignmentOp [ "<" IntType | FloatType ">" ] RValue

StringAssign = StringType Name AssignmentOp [ "<" IntType | FloatType | BoolType | StringAssign ">" ] RValue

BoolAssign = BoolType Name AssignmentOp [ "<" IntType | BoolType ">" ] RValue

AssignmentOp = "="

--- Expressions ------------------------------------------------

LogicalExpr = WeakLogicalExpr

WeakLogicalExpr = StrongLogicalExpr { WeakLogicalOp StrongLogicalExpr }

StrongLogicalExpr = RelationExpr { StrongLogicalOp RelationExpr }

RelationExpr = LogicalOperand [ RelationOp LogicalOperand ]

LogicalOperand = [ NegationOp ] RValue | "(" LogicalExpr ")"

StrongLogicalOp = "&&"

WeakLogicalOp = "||"

RelationOp = "==" | "!=" | "<" | ">" | "<=" | ">="

NegationOp = "!"

ArithmeticExpr = WeakArithmeticExpr

WeakArithmeticExpr = StrongArithmeticExpr { WeakArithmeticOp StrongArithmeticExpr }

StrongArithmeticExpr= ArithmeticOperand { StrongArithmeticOp ArithmeticOperand }

ArithmeticOperand = Number | Name | FuncCall | "(" ArithmeticExpr ")"

StrongArithmeticOp = "\*" | "/" | "%"

WeakArithmeticOp = "+" | "-"

--- Types ---

Type = IntType | FloatType | StringType | BoolType

IntType = "int"

FloatType = "float"

StringType = "string"

BoolType = "bool"

--- Other ---

Body = "{" { Statement } "}"

SingleInstr = {LineComment} Statement {LineComment}

RValue = Value | Name | ArithmeticExpr | LogicalExpr

RValueAssign = Value | Name | “(“ArithmeticExpr “)” | “(“LogicalExpr “)”

Name = (Letter | "\_") {Letter | Digit | "\_" }

Value = String | Number | BoolValue

String = """ { AsciiChar } """

Letter = "a".."z" | "A".."Z"

Number = [Sign] Digit { Digit } [ "." Digit { Digit } ]

Digit = "0".."9"

BoolValue = "true" | "false"

Sign = [ "-" ]

AsciiChar = <every ASCII character>

# 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 źródłowych.
* Poprawne wykonywanie wszystkich poprawnie zapisanych instrukcji w plikach źródłowych.
* Możliwość definiowania własnych funkcji oraz ich późniejszego wywoływania.
* Przestrzeganie logicznego porządku instrukcji sterujących.
* Przeprowadzanie operacji logicznych i arytmetycznych na typach wbudowanych.

### Niefunkcjonalne

* Uruchomienie aplikacji z niepoprawnymi parametrami powinno poinformować użytkownika o możliwych parametrach startowych.
* Komunikaty o błędach analizy plików powinny być proste i w sposób jednoznaczny wskazywać popełnione błędy.

# Szczegóły implementacyjne

### Kompilacja

Program można kompilować zarówno na platformie Windows jak i Linux. Dla Windowsa jest przygotowany projekt Visual Studio 2013. Dla Linuxa, kompilacja odbywa się za pomocą skryptu ‘install.sh’ który wykorzystuje narzędzie CMake. Kompilator musi obsługiwać standard c++11.

Program zawiera wersje zwykłą oraz debug dla obu platform. Wersja debug wyświetla informację o żródłach, tokenach i innych dodatkowych informacjach.

### Uruchamianie

Program jest aplikacją konsolową. Po nazwie podawana jest ścieżka do pliku, a potem opcjonalnie parametry wejściowe. Np.  
paweł~$ program /home/pawel/source.txt 1 „test”

Program ma przebieg 3 etapowy. Domyślna kompilacja zakłada przerwanie toku analizy i wykonania w przypadku napotkania pierwszego błędu, tzn. jeżeli analiza składniowa zgłosiła błąd, analiza semantyczna nie będzie wykonywana. Po każ

### Budowa programu

Program złożony jest z głównego modułu interpretera (Interpreter.h) sterującego przepływem analizy pliku źródłowego, trzech modułów analizy, modułu wykonania oraz modułów pomocniczych. Dodatkowo, moduły są wspierane przez grupę klas drzewa składniowego oraz grupę klas reprezentacji pośredniej. Wszystkie moduły korzystają z funkcji obsługi wiadomości – MessageHandler.h.

##### Analiza leksykalna (Lexer.h)

Moduł lexera rozbija plik źródłowy na tokeny. Wczytywanie następuje znak po znaku aż do napotkania pierwszego pasującego tokenu. Korzysta z modułu wczytywania (InputManager.h) oraz tablicy tokenów (TokenType.h).

##### Analiza składniowa (Parser.h).

Moduł parser, przy pomocy lexera, ładuje kolejne tokeny. Następnie sprawdza, czy tokeny są zgodne z przyjętą gramatyką języka. Następne tokeny decydują o przebiegu ścieżki analizy (wywołując odpowiednie metody). Zatwierdzone struktury gramatyki będą tworzyć drzewo składniowe (struktury drzewa poniżej).

##### Analiza semantyczna (SemanticChecker.h)

Do zadań analizy semantycznej należy:

* Poprawność identyfikatorów
* Poprawność konwersji
* Sprawdzenie nadpisywania zmiennych/stałych oraz funkcji.
* Zgodność operacji arytmetycznych i logicznych.
* Zgodność argumentów wywołania funkcji itd.

W trakcie analizy generowane są struktury reprezentacji pośredniej, przeznaczonej dla modułu wykonania. Moduł współpracuje także z biblioteką funkcji wbudowanych – Library.h

##### Wykonanie (Executor.h)

Wykonuje sekwencyjnie instrukcje, zaczynając od punktu startowego, którym jest funkcja główna „program” i wypisuje wyniki na ekran (jeśli jakieś są).

Poniżej klasy przedstawione z uwzględnieniem dziedziczenia.

### Klasy drzewa składniowego

* Node
  + RValue
    - ArithmeticExpression
    - LogicalExpression
    - Literal
      * Bool
      * String
      * Number
    - Call
  + ConstDeclaration
  + VarDeclaration
  + FunctionDefinition
  + IfStatement
  + WhileStatement
  + Program
  + ReturnStatement
  + StatetementBlock
  + Variable
  + Assignment

### Klasy reprezentacji pośredniej

* Executable
  + Assignable
    - ArithmeticOperand
      * ArithmeticExpression
      * CallInstr
      * Literal
      * Variable
    - LogicalOperand
      * Logical Expression
      * *ArithmeticExpression*
      * *CallInstr*
      * *Literal*
      * *Variable*
  + Instruction
    - AssignmentInstr
    - Block
      * Function
    - *CallInstr*
    - IfInstr
    - ReturnInstr
    - WhileInstr
* ScopePrototype
* ScopeInstance

### Biblioteka funkcji standardowych

Interpreter posiada kilka funkcji wbudowanych, a dodawanie kolejnych jest bardzo łatwe.

* print (x) – wyświetla x na ekrean z uwzględnieniem typu.
* println (x) – to samo co print, ale kończy znakiem nowej linii.
* nl() – przechodzi do nowej linii.

# Możliwości języka

### Opis funkcji – przykłady

##### Komentarze

// Komentarze można wstawiać…

program PROG () {

// .. w dowolnym ..

return 0; // Return 0, because that’s why.

// .. z tych miejsc...

}

// Tu też.

##### Zmienne i stałe.

// Zmienne i stałe muszą mieć określony typ.

var int i1; // OK.

const string S1; // OK.

var v1; // ERROR.

int i2; // ERROR.

// Mogą też być inicjalizowane.

// Zakomentowane inicjalizacje są nieprawidłowe, ponieważ nie wszystkie konwersje

// implicite są dozwolone.

### var bool b1 = false;

### //var bool b2 = 2;

### //var bool b3 = 3.1415;

### //var bool b4 = "bool";

### var int i1 = true;

### const int i2 = 0000;

### //const int i3 = 2.5;

### //const int i4 = "int";

### var float f1 = true;

### var float f2 = 1;

### var float f3 = 2.5;

### //var float f4 = "float";

### var string s1 = false;

const string s2 = 4;

const string s3 = 2.73;

const string s4 = "string";

// Do zmiennych można przypisać wartość, zmienną, wyrażenie bądź wywołanie funkcji.

var float f;

const int i; // WARNING.

f = 2.456; // OK.

i = 3; // ERROR.

f = 2 + 2 \* 2;

f = i; // Implicit conversion.

f = myFunction(i);

##### Wyrażenia logiczne

// W warunku logicznym można używać wartości true oraz false, zmiennych logicznych i wyrażeń relacyjnych.

const bool logical1 = false;

const int i1 = 4;

if (true && (false || logical1))

println("ERROR");

else

println("OK");

while (logical1 || i1 == 3)

println("ERROR");

println("OK");

// Można definiować wiele zagnieżdżonych wyrażeń warunkowych oraz wiele else-if’ów.

if (logical1)

{

if (i1 == 1)

print (“i1 = 1”);

else if (i1 == 2)

print (“i1 = 2”);

else

print (“i1 > 2”);

}

else

{

if (!logical1)

if (!logical1)

if (!logical1)

if (!logical1)

println(“logical1 = false”);

}

##### Wyrażenia arytmetyczne

// Wyrażenia matematyczne uwzględniają prriorytety operatorów.

var float arithm = (2 + 2 \* 2 / 1 - 3 / 3);

print(arithm); // Wyświetli 5, a nie 5/3.

##### Funkcje

// Deklaracje funkcji i wywoływanie

function Nazwa () { return; }

function Nazwa2 (int argument, string argument 2) { return argument };

const float returnValue = Nazwa2(123, “test”);

### Przykładowy skrypt

program SampleProgram (int arg)

{

var bool b;

var int i;

var float f;

var string s;

var bool b1 = false;

//var bool b2 = 2;

//var bool b3 = 3.1415;

//var bool b4 = "bool";

var int i1 = true;

const int i2 = 0000;

//const int i3 = 2.5;

//const int i4 = "int";

var float f1 = true;

var float f2 = 1;

var float f3 = 2.5;

//var float f4 = "float";

var string s1 = false;

const string s2 = 4;

const string s3 = 2.73;

const string s4 = "string";

b1 = true;

//b1 = 2;

//b1 = 2.5;

//b1 = "bool";

i1 = false;

i1 = 123456;

//i1 = 3.5;

//i1 = "int";

f1 = true;

f1 = 4;

f1 = 4.5678;

//f1 = "float";

s1 = false;

s1 = 10;

s1 = 456.345;

s1 = "string";

print(b1);

print(" ");

print(i2);

print(" ");

print(f1);

print(" ");

print(s4);

print(" ");

println(true);

println(false);

println(1);

println(2.345);

println("TEST");

i1 = 3;

if (i1 < 3)

print("NOT OK");

else

print("OK");

i1 = 100;

nl();

while (i1 > 3)

{

var int local = 1;

print(i1);

i1 = MySubtraction(i1, 5);

print(" ");

}

nl();

const bool logical1 = false;

if (true && (false || logical1))

println("ERROR");

else

println("OK");

if (logical1 || i1 == 3)

println("ERROR");

else

println("OK 2");

if(!logical1)

{

if(!logical1)

if(!logical1)

if(!logical1)

if(!logical1)

println("logical1 == false");

}

var float arithm = (2 + 2 \* 2 / 1 - 3 / 3);

var float arithm2 = (123 / 37);

println(arithm);

println(arithm2);

return;

}

// KOMENTARZ POZA FUNKCJA

function MySubtraction (float x, int y)

{

const int result = x - y;

return result;

}