Techniki Kompilacji - Projekt

Interpreter prostego języka z typem wektorowym

Stawczyk Przemysław 293153

1 Opis Projektu

Projekt zakłada wykonanie interpretera prostego języka z obsługą wbudowanego typu *wektora* o *dim = {2, 3}*. Język ten ma obsługiwać : zmienne z zasięgiem, instrukcje wykonywane na żądanie *funkcje*, instrukcje wykonywane nie zawsze *warunkowe*, wyrażenia matematyczne oraz operatory wraz z priorytetami.

1.1 Ogólne założenia

- Wartości liczbowe są reprezentowane przez liczby całkowite.
- Obsługiwane są typy :
  - Typ liczbowy *skalar*.
  - Typ wektorowy *vec* o *dim = {2,3}*
    np. *vec(1,2), vec(4,5,6)*.
- Na typach liczbowych można definiować wyrażenia arytmetyczne za pomocą operatorów: *+ - * / ( )* z uwzględnieniem ich priorytetów.
- Wykonywanie operacji na wektorach : iloczynu skalarnego oraz wektorowego. *z użyciem wbudowanych funkcji*
- Na typach *vec* i liczbowych można użyć operatorów przyrównania *== !=* oraz łączyć w wyrażenia za pomocą *|| && ( )*.
- Możliwy jest dostęp indeksowy do zawartości zmiennej np *zmienna[1]* daje dostęp do wartości 2 wymiaru wektora.
- Można definiować instrukcje warunkowe za pomocą konstrukcji *if()* oraz *else*.
- Można tworzyć pętle korzystając z konstrukcji *while()*.
- Można definiować funkcje z użycie słowa kluczowego *fun*.
- Zmienne są przekazywane do funkcji przez referencje.
- Program zaczyna wykonanie od bezparametrycznej funkcji *main()*
- Wartości logiczne sa reprezentowane przez liczby gdzie 0 = fałsz, !0 = prawda. Każdy wektor niezerowy jest ewaluowany do prawdy.
- Język wspierać ma operację *print(...)* przyjmująca oprócz typów numerycznych stałe tekstowe *[w cudzysłowach]* istniejące wyłącznie na potrzeby *print*.

1.2 Funkcje Biblioteczne

Funkcje biblioteczne miały być ładowane przed właściwym programem i zaimplementowane w docelowym języku.

- fun **product3**(vec1, vec2) - *iloczyn wektorowy wektorów wymiaru 3*
- fun **product2**(vec1, vec2) - *iloczyn wektorowy wektorów wymiaru 2*
- fun **scalar3**(vec1, vec2) - *iloczyn skalarny wektorów wymiaru 3*
- fun **scalar2**(vec1, vec2) - *iloczyn skalarny wektorów wymiaru 2*

2 Przykłady

Przykład 1 - funkcja z rekurencją

```
fun licz(a) {
    a = a * 2;
    if (a < 10) {
        fun(a);
    }
    return a;
}

fun main() {
    if(11){
        print(fun(2));
    }
}
```

Przykład 2 - funkcje biblioteczne

```
fun main() {
    var a = vec(1,2);
    var b = vec(3,4);
    print("a = ", a);
    print("b = ", b);
    print("a + b = ", a+b);
    print("a * b = ", scalar2(a, b));
    print("a o b = ", product2(a, b));
    print("2 * a = ", 2*a);
    print("a * 2 = ", a*2);
}
```

Przykład 3 - zmienne

```
fun printAndRet(a) {
    print(a);
    return a+1;
}
```

```
fun main() {
    var a = 0;
    if(a == 0) {
        print("a");
        print(a);
    }
    print(vec(1,2));
    print(vec(1,2,3));

    var b = a;
    while(b < 10) {
        b = b + 1;
    }
    print("b = ", b);

    fun(1);
    fun(fun(2));
}
```

Przykład 4 - product2

```
fun product2(vec1, vec2) {
    var res=vec(1,2);"
    res[0] = vec1[1] - vec2[1];
    res[1] = vec2[0] - vec1[0];
    return res;
}
```

Przykład 5 - product3

```
fun product3(vec1, vec2) {
    var res = vec(0,0,0);
    res[0] = vec1[1] * vec2[2] - vec1[2] * vec2[1];
    res[1] = vec1[2] * vec2[0] - vec1[0] * vec2[2];
    res[2] = vec1[0] * vec2[1] - vec1[1] * vec2[0];
    return res;
}
```

Przykład 6 - scalar2

```
fun scalar2(vec1, vec2) {
    var res = vec1[0] * vec2[0] + vec1[1] * vec2[1];
    return res;
}
```

Przykład 7 - scalar3

```
fun scalar3(vec1, vec2) {
    var res = vec1[0] * vec2[0] + vec1[1] * vec2[1] + vec1[2] * vec2[2];
    return res;
}
```

3 Opis Struktury Języka

program = { *functionDef* } ;
functionDef = "fun" *identifier parameters statementBlock* ;
parameters = "(" [*identifier* { "," *identifier* }] ")" ;

statementBlock = "{" { *initStatement* | *assignStatement* | *returnStatement* | *ifStatement* | *whileStatement* | *functionCallStatement* | *printStatement* | *statementBlock* } "}" ;
returnStatement = "return" *orExpr* ";" ;

initStatement = "var" *identifier* ["=" *orExpr*] ";" ;
assignStatement = *variable* "=" *orExpr* ";" ;
ifStatement = "if" "(" *orExpr* ")" *statementBlock* ["else" *statementBlock*] ;
whileStatement = "while" "(" *orExpr* ")" *statementBlock* ;
functionCallStatement = *functionCall* ";"
printStatement = "print" "(" (*charString* | *orExpr*) {"," (*charString* | *orExpr*)} ")" ";";

functionCall = *identifier arguments* ;
arguments = "(" [*orExpr* { "," *orExpr* }] ")" ;

parentExpr = "(" *orExpr* ")" ;
orExpr = *andExpr* { *orOp andExpr* } ;
andExpr = *relationalExpr* { *andOp relationalExpr* } ;
relationalExpr = *baseLogicExpr* [*relationOp baseLogicExpr*] ;
baseLogicExpr = [*unaryNegation*] *additiveExpr* ;

additiveExpr = *multiplyExpr* { *additiveOp multiplyExpr* } ;
multiplyExpr = *baseMathExpr* { *multiplyOp baseMathExpr* } ;
baseMathExpr = [*unaryMinus*] (*value* | *parentExpr*) ;

value = *numberString* | *vectorValue* | *variable* | *functionCall* ;

additiveOp = "+" | "-" ;
multiplyOp = "*" | "/" | "%" ;
orOp = "or" ;
andOp = "and" ;
unaryMinus = "-" ;
unaryNegation = "!" ;
relationOp = "==" | "!=" | "<" | ">" | "<=" | ">=" ;

identifier = *letter* { *letter* | *digit* | *underscore* } ;
variable = *identifier* [*index*] ;
index = "[" *numberString* "]" ;

vectorValue = "vec" "(" *numberString* "," *numberString* ["," *numberString*] ")" ;
numberString = *digit* { *digit* } ;
charString = "" { *allCharacters* − "" } "" ;

digit = "0".."9";
underscore = "_";
letter = "a".."z" | "A".."Z" ;

allCharacters = *? all visible characters ? ;*

4 Założenia Implementacyjne

4.1 Produkt Końcowy

Finalny program ma być konsolowa aplikacja uruchamianą wraz z parametrem reprezentującym ścieżkę do pliku do interpretacji. Wynik działania skryptu będzie wypisywane na standardowe wyjście *stdout*. W przypadku błędów kompilacji lub wykonania będą one wypisywane na standardowe wyjście błędów *stderr*. Umożliwi to łatwe przekierowanie poszczególnych wyjść do pliku/innej konsoli etc.

4.2 Struktura i Narzędzia

Projekt miałby zostać zaimplementowany w języku *C++* z użyciem biblioteki *boost* do testów jednostkowych *boost::unit_test* oraz parsowania argumentów *boost::program_options*. Całość korzystać ma z narzędzia *CMake* do zarządzania procesem budowania.

Program miałby składać się z następujących modułów :

- analizator leksykalny
- analizator składniowy
- interpreter

5 Instrukcja Kompilacji i Użytkowania

5.1 Instrukcja Kompilacji

Projekt kompilowany jest z użyciem narzędzia *CMake*.

5.2 Instrukcja Użycia