Karolina Deszcz

nr indeksu: 277284

Interpreter prostego języka – dokumentacja końcowa

1. Opis projektu

W ramach projektu został zaimplementowany interpreter własnego, prostego języka. Instrukcje zapisane w tym języku są wczytywane z pliku wejściowego. Lekser przekształca wejściowy ciąg znaków w sekwencję atomów leksykalnych – tokenów. Parser prosi lekser o dostarczenie kolejnych tokenów, a następnie przeprowadza analizę składniową grupując je w struktury składniowe, sprawdzając przy tym, czy tworzą poprawne konstrukcje, zgodne z gramatyką języka. Po utworzeniu drzewa składniowego przeprowadzana jest analiza semantyczna, podczas, której wykonywane są zapisane w kodzie instrukcje i sprawdzana jest ich poprawność semantyczna.

Interpreter został zaimplementowany w języku java jako aplikacja konsolowa, która jest wywoływana wraz z argumentem określającym ścieżkę do pliku z programem, który będzie interpretowany. Wynik programu oraz ewentualne błędy są wyświetlane na standardowym wyjściu.

Do projektu zostały dołączone testy jednostkowe, które sprawdzają poprawność zaimplementowanego projektu.

2. Opis języka – wymagania funkcjonalne

Każdy program musi zawierać zdefiniowaną funkcję main, która może być dowolnego typu (int, bool, string, Rectangle). Nie ma wymogu by funkcja main była zdefiniowana na początku programu.

Język umożliwia następujące operacje:

- Tworzenie zmiennych, operacje na zmiennych
    - Dostępne typy zmiennych: int, string, bool, Rectangle
- Wykonywanie instrukcji w pętli:

  while (warunek) { blok instrukcji }
- Wykonywanie instrukcji warunkowych:

  if (warunek) then {blok instrukcji}
  if (warunek) then {blok instrukcji} else {blok instrukcji}

- Wyświetlanie tekstu na ekranie

  print (tekst)

- Wykonywanie operacji przy zachowaniu priorytetu operatorów

Grupa I (najwyższy priorytet)	*, /
Grupa II (średni priorytet)	+ , -
Grupa III (najniższy priorytet)	&&, ||

- Wykonywanie operacji matematycznych na liczbach (dodawanie, odejmowanie, mnożenie, dzielenie)

- Wykonywanie operacji logicznych (negacja, suma i iloczyn logiczny)

- Definiowanie funkcji, język będzie umożliwiał tworzenie funkcji rekurencyjnych, parametry będą przekazywane do funkcji przez wartość.

```
typ  nazwaFunkcji (lista parametrów)  {
        blok instrukcji
        return wynik
}
```

- Wywoływanie funkcji

  nazwaFunkcji (lista argumentów)

- Dodawanie komentarzy

  # komentarz, tekst w tej linii nie będzie dalej przetwarzany

3. Specyfikacja własnego typu danych – Rectangle

pola składowe:

- x (współrzędna x w układzie współrzędnych)
- y (współrzędna y w układzie współrzędnych)
- width (szerokość prostokątna)
- length (długość prostokąta)
- area (pole prostokąta)
- perimeter (obwód prostokąta)

Pole (area) i obwód (perimeter) są obliczane na podstawie wartości width , length (ich wartość jest uaktualniana gdy zmieni się wartość któregoś z tych pól).

Relacja równości : prostokąty są równe gdy

```
(r1.width == r2.width && r1.length == r2.length)
||  (r1.width == r2.length && r1.length == r2.width)
```

Relacja mniejszości : prostokąt r1 jest mniejszy od prostokąta r2 gdy

```
r1. area <  r2.area
```

Suma prostokątów:

Prostokąt, który jest sumą dwóch innych prostokątów, to najmniejszy prostokąt, który zawiera w sobie oba prostokąty

Mnożenie prostokątów przez liczbę:

```
r2 = a * r1   <=>  r2.width = a * r1.width  &&  r2.length = a  * r1.length
```

Dzielenie prostokątów przez liczbę:

```
r2 = r1 / a  <=>  r2.width = r1.width / a  &&  r2.length = r1.length  / a
```

Mnożenie prostokątów (część wspólna):

```
r3 = r2 * r1  <=>  r3.width = MIN(r1.width, r2.width) &&
                   r3.length = MIN(r1.length, r2.length)
```

4. Opis języka – gramatyka

```
//Lexical grammar

digit  =  "0" .. "9";
nonZeroDigit =  "1" .. "9";
number = ["-"], nonZeroDigit, { digit } | ”0”;

letter = "A” .. "Z" |  "a" .. "z" ;
ident = letter, { letter | digit } [”.”ident ];
whiteSpace = ” ”;
txt = ” ” ”,   { letter | digit | whiteSpace},   ” ” ”;

// language syntax

program =  functionDefinition, { functionDefinition };
functionDefinition  = type, ident, "(", parameterList, ")", blockStmnt;
parameterList = [type, ident]  {”,” , type, ident };

blockStmnt = "{", {stmnt} ,"}";
stmnt = varDeclaration | assignStmnt | printStmnt | functionCallStmnt | ifStmnt |
          whileStmnt | blockStmnt | functionDefinitionStmnt | returnStmnt

type = "int" | "string" | "bool" | "Rectangle";
varDeclaration = type, ident, [”=” expr] ";"

assignStmnt = ident, "=", expr | functionCallStmnt | txt, ”;”
```

```
expr = AddExpr | OrCondition;
AddExpr = MulExpr, [ addOp, MulExpr ];
MulExpr = simpleExpr, [ mulOp, simpleExpr];
simpleExpr = [ ”-” ], ( number | ident | bracketExpr | functionCallStmnt);
bracketExpr = ”(”, AddExpr, ”)”;

OrCondition = AndCondition, [ "||", AndCondition];
AndCondition = Condition, [ "&&", Condition];
Condition = [ "!" ], ( "(", OrCondition, ")" | simpleExpr);
simpleCondition = simpleExpr, [ relOp, simpleExpr] ;
returnStmnt = "return", expr ,”;”
functionCallStmnt = ident, "(", argList, ")", ”;”;
argList = [ident]  {”,”, ident};

printStmnt = "print", "(", expr | (” ” ”, txt, ” ” ”), ")", ”;” ;

ifStmnt = "if", "(", OrCondition, ")", "then", blockStmnt [ "else", blockStmnt ];
whileStmnt = "while", "(", OrCondition, ")", blockStmnt;
blockStmnt = "{", {stmnt} ,"}";

addOp = "+" | "-" ;
mulOp = "*" | "/";
relOp = "==" | "!=" | "<" | ">" | "<=" | ">=";
logOp = "&&" | "||";

commentMark = "#";
```

5. Przykładowe skrypty

program wykorzystujący instrukcję warunkową:

```
bool main(){
	int a = 5;
	int b = 4;
	if (a < b) then {
		return true;
	}
	else {
		return false;
	}
}
```

program wykonujący instrukcje w pętli:

```
int main(){
	int a = 1;
	int b = 4;
	int result = 0;
	while (a < b){
		result = result * (a+b);
	}
	return result;
}
```

program wywołujący funkcję:

```
int add(int a, int b){
	return a+b;
}

int main(){
	int x = 5;
	int y  = -4;
	return add(x,y);
}
```

program wykorzystujący tryb Rectangle:

```
int main(){
	Rectangle r1(1,1,4,5);
	Rectangle r2(2,2,6,5);
	Rectangle r3();  # creates empty rectangle
	Rectangle r4();
	r3 = r1 + r2;
	r4 = r1 * r2;
	if (r3 > r4) then {
		return r3.area;
	}
	else {
		return r4.area;
	}
}
```