Paweł Świątkowski

**TKOM – dokumentacja końcowa**

Temat: Język konwersji rzędów wielkości (przedrostków)

1. **Opis funkcjonalny**

* Celem projektu jest stworzenie interpretera prostego języka wspomagającego obliczenia rzędów wielkości (przedrostków jednostek).
* Zaimplementowane moduły:
  + Moduł odczytywanie pliku źródłowego (FileReader)
  + Analizator leksykalny (Lexer)
  + Analizator składniowy (Parser)
  + Analizator semantyczny (SemanticAnalyzer)
  + Moduł obsługi błędów (ErrorHandler)
  + Moduł wykonywania kodu (Executor)
* Język ma wbudowanych kilka typów przedrostków jednostek SI tj. MILI, CENTY, KILO, MEGA, GIGA
* Język zawiera takie konstrukcje jak: instrukcje warunkowe, pętle, funkcje.
* Stworzony interpreter jest aplikacją konsolową.
* Podczas analizy leksykalnej, składniowej bądź semantycznej pierwszy napotkany błąd będzie zgłaszany użytkownikowi

1. **Założenia**

* wartości logiczne reprezentowane są przez typ liczbowy, wartość równa 0 równoważna jest FALSE, każda inna wartość równoważna jest TRUE (w założeniu jest to wartość 1)
* dla programisty dostępna jest funkcja print(TYPE) umożliwiająca wyświetlenie wartości zmiennej przekazanej jako argument, może przyjmować również stringa miedzy cudzysłowami
* plik źródłowy musi mieć zdefiniowaną dokładnie jedną funkcję INTEGER main() zwracającą wartość liczbową typu INTEGER oraz nie przyjmującą żadnych argumentów. To od tej funkcji zacznie być wykonywany kod programu.
* separatorem dziesiętnym liczby jest „.” (kropka)
* jako argument wyrażeń warunkowych mogą występować jedynie operacje logiczne
* funkcji nie da się przeciążać
* cały program podany na wejście interpretera musi znajdować się w jednym pliku
* jeżeli przy deklaracji zmiennej nie przypisano jej żadnej wartości, domyślnie będzie ona przyjmowała wartość 0

1. **Tokeny:**

"function" "return" "if" "else" "while"

"{" "}" "(" ")"

"+" "-" "\*" "/"

"=" "!"

"&&" "||" "==" "!=" "<" "<=" ">" ">="

"," ";"

"MILI" "CENTY" "KILO" "MEGA" "GIGA" "DOUBLE"

1. **Formalny opis gramatyki**

Uwaga: w celu zwiększenia czytelności gramatyki, podczas jej definiowania zostały zastosowane pewne odstępstwa od notacji EBNF:

- nie jest stosowany operator konkatenacji ‘,’ (przecinek)

- zostały pominięte symbole białych znaków

- notację należy rozumieć w taki sposób, że spacja w notacji oznacza dowolną ilość białych znaków oraz konkatenację tokenów występujących po obu jej stronach

**program** = { functionDef };

**functionDef** = "function" signature parameters block;

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

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

**block**  = "{" { statement } "}"

**statement** = block | ifStatement | whileStatement | returnStatement |

initStatement | assignStatement | functionCall ";"

**ifStatement**​ = "if" "(" condition ")" statement [ "else" statement ];

**whileStatement**​ = "while" "(" condition ")" statement;

**returnStatement**​ = "return" expression ";";

**initStatement**​ = signature [ assignmentOp expression] ";";

**assignStatement**​ = id assignmentOp expression ";";

**expression**​ = multiplicativeExpr { additiveOp multiplicativeExpr };

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

**primaryExpr**​ = ( literal | id | parenthExpr | functionCall);

**parenthExpr**​ = "(" expression ")";

**condition** = andCond { orOp andCond };

**andCond**​ = equalityCond { andOp equalityCond };

**equalityCond**​ = relationalCond [ equalOp relationalCond ];

**relationalCond**​ = primaryCond [ relationOp primaryCond ];

**primaryCond**​ = [ unaryOp ] ( parenthCond | expression );

**parenthCond**​ = "(" condition ")";

**unaryOp**​ = "!";

**assignmentOp**​ = "=";

**orOp**​ = "||";

**andOp**​ = "&&";

**equalOp**​ = "==" | "!=";

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

**additiveOp**​ = "+" | "­";

**multiplicativeOp**​ = "\*" | "/";

**literal** = [ "­" ] | number;

**id** = letter { digit | letter }

**functionCall**​ = id arguments;

**number** = ( digit [ "." digit {digit} ] ) |

( naturalDigit {digit} [ "." digit {digit} ] );

**signature** = type id;

**type** = "MILI" | "CENTY" | "KILO" | "MEGA" | "GIGA" | "DOUBLE";

​ = letter { digit | letter };

**letter**​ = "a".."z" | "A".."Z" | "\_";

**digit**​ = "0".."9";

**naturalDigit** = "1".."9";

1. **Przykłady**

1) Prosta funkcja, instrukcja warunkowa, rzutowanie typów (automatyczna konwersja)

function KILO max(KILO unit1, KILO unit2) {

if(unit1 > unit2) {

return unit1;

} else {

return unit2;

}

}

function DOUBLE main() {

KILO smallUnit = 10;

MEGA bigUnit = 1;

KILO result = max(smallUnit, bigUnit);

print(“Evaluating result:”);

print(result);

return 0;

}

2) Condition, Loop, Conversion

function DOUBLE main() {

DOUBLE i = 0;

MILI unit = 10;

while(i<10) {

i = i+1;

unit = 100\*unit;

}

KILO biggerUnit = unit;

print(biggerUnit);

return 0;

}

1. **Opis techniczny realizacji**

* Projekt zostanie zaimplementowany w języku Java w wersji 1.8
* Do napisania testów jednostkowych posłuży biblioteka JUnit. Testy będą miały na celu np. weryfikację czy analizator leksykalny poprawnie rozpoznaje tokeny, czy parser buduje poprawne drzewo rozkładu gramatycznego czy executor poprawnie wykonuje polecenia
* Projekt będzie budowany przy wykorzystaniu narzędzia MAVEN.
* Program będzie prostą aplikacją konsolową. Przy uruchamianiu aplikacji będzie podawało się jako argumenty ścieżki plików wejściowych:
  + plik źródłowy napisanego programu w stworzonym języku (argument obligatoryjny)
  + plik konfiguracyjny (argument opcjonalny)

1. **Instrukcja obsługi**

* Do budowania projektu wykorzystywane jest narzędzie Maven.

Należy w konsoli wpisać komendę:

mvn clean install

* Do uruchomienia konieczne jest JRE 8.

Program uruchamia się poleceniem:

java -jar TKOM-UNIT-INTERPRETER.jar <path to file>

Konieczne jest podanie dokładnie jednego parametru, którym jest ścieżka do pliku wraz z rozszerzeniem w którym znajduje się kod, który ma zostać wykonany przez interpreter.