Paweł Świątkowski

**TKOM – dokumentacja wstępna**

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. MIKRO, MILI, CENTY, KILO, MEGA, GIGA. Istnieje jednak również możliwość deklarowania własnych typów zmiennych poprzez definicję ich w pliku konfiguracyjnym.
* 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 INTEGER, wartość równa 0 równoważna jest FALSE, każda inna wartość równoważna jest TRUE
* dla programisty dostępna jest funkcja print(TYPE) umożliwiająca wyświetlenie wartości zmiennej przekazanej jako argument
* 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"

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

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

"=" "!"

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

"," ";"

"MIKRO" "MILI" "CENTY" "KILO" "MEGA" "GIGA"

"INTEGER" "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" typedId parameters functionBody;

**functionBody** = "{" { statementBlock } "}"

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

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

**statementBlock**​ = ( "{" { singleStatement } "}" ) | singleStatement;

**singleStatement** = ifStatement | whileStatement | returnStatement |

initStatement | assignStatement | functionCall ";"

**ifStatement**​ = "if" "(" condition ")" statementBlock

[ "else" statementBlock ];

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

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

**initStatement**​ = typedId [ 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;

**functionCall**​ = id arguments;

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

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

**typedId** = type id;

**type** = "MIKRO" | "MILI" | "CENTY" | "KILO" | "MEGA" | "GIGA"|

"INTEGER" | "DOUBLE";

**id**​ = 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 INTEGER main() {

KILO smallUnit = 10;

MEGA bigUnit = 1;

KILO result = max(smallUnit, bigUnit);

print(result);

return 0;

}

2) Condition, Loop, Conversion

function INTEGER main() {

INTEGER i = 0;

MILI unit = 10;

while(i<10) {

i = i+1;

unit = 100\*unit;

}

KILO biggerUnit = unit;

print(biggerUnit);

return 0;

}

3) Przykładowy plik konfiguracyjny

DUUUUUZA\_JEDNOSTKA = 100000 MEGA

MAAAAALA\_JEDNOSTKA = 0.0001 MILI

1. **Opis techniczny realizacji**

* Projekt zostanie zaimplementowany w języku Java.
* Do napisania testów jednostkowych posłuży biblioteka JUnit. Testy będą miały na celu np. weryfikację czy analizator leksykalny poprawnie rozpoznaje tokeny
* 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)