TKOM - projekt wstępny

Kalkulator “z jednostkami”

Michał Berliński - 290432

Opis funkcjonalny:

Język ma umożliwiać proste przetwarzanie zmiennych zawierających wartości liczbowe razem z przypisanymi do nich jednostkami. Będzie zawierał pewną ilość definicji jednostek z systemów metrycznego i imperialnego wraz z przelicznikami, ale będzie w nim również możliwe definiowanie własnych jednostek. Główną cechą kalkulatora wyróżniającą go spośród innych będzie możliwość operowania na zmiennych o różnych jednostkach z tej samej dziedziny (np. dodawanie do siebie metrów i jardów). Każda dziedzina będzie posiadać swoją jednostkę podstawową, natomiast pozostałe należące do niej jednostki będą wyrażane jako krotność podstawowej. Krotność jednostki podstawowej będzie domyślnie równa 1.

Założenia:

- Język zostanie zaimplementowany w języku Java
- Każdy program musi posiadać funkcję main będącą punktem wejściowym dla wykonywania programu
- Typ danych opisujący jednostki
- Dynamiczny typ danych obsługujący liczby całkowite, rzeczywiste, napisy oraz operacje na nich
- W przypadku dodawania zmiennej typu liczbowego do zmiennej typu napisowego wynik będzie typem napisowym. Pozostałe operacje na różnych typach będą powodowały błąd
- Operatory matematyczne o różnym priorytecie wykonywania oraz nawiasy
- Język pozwala na tworzenie zmiennych oraz obsługuje zakres widoczności
- Język pozwala na tworzenie oraz wywoływanie funkcji
- Instrukcja warunkowa if-else
- Instrukcja pętli while

Tokeny:

„def”, „var”, „return”, „if”, „else”, „return”, „while”, „function”, „[”, „]”, „(”, „)”, „{”, „}”, „,”, „.”, „;”, „:”, „+”, „-”, „=”, „*”, „/”, „’”, „==”, „&&”, „|”, „>”, „<”, „!=”, „<=”, „>=”, „//”, „print”

Przykłady użycia:

1. Deklaracja zmiennej:

   def m: [LENGTH];
   def yd: 0.91 [m];

2. Przykład kodu wykorzystującego pętle while do obliczenia ile jardów liczy mila:

   def mile: 1609 [m];
   def yd: 0,91 [m];

```
var i = 0;

var x = [1 mile];
var y = [1 yd] ;

while(x>0)
{
        x = x – y;
        i = i + 1;
};
print(i);
```

3. Przykład funkcji dodającej do siebie 2 wartości wyrażone w różnych jednostkach:

```
var dist = [1200 m] ;
var distBis = [2 mile] ;

function dodaj (dist, distBis)
{
        var result = dist + distBis;
        return result;
}
```

Wynik zostanie zwrócony w jednostce podstawowej.

4. Funkcja przeliczająca wartości z metrów na mile angielskie

```
var dist = [5700 m];
function convert_m_to_mile (dist)
{
        var result = dist/1609;
        result = result mile ;
        return result;
}
```

5. Liczenie silni

```
var result = 1;
var x = 2;

while(x<15)
{
        result = result*x ;
        x=x+1 ;
}
```

6. Przykład funkcjonowania zakresów zmiennych.

```
var x=0;

dodaj(x, var a=5)       //funkcja do dodawania
{
        var x=3;
        return(x+a);    //zostanie zwrócona wartość 8
}
```

Wymagania funkcjonalne:

a) Odczytywanie, parsowanie i analiza kodu źródłowego zapisanego w pliku tekstowym
b) Kontrola poprawności wprowadzonych danych oraz umiejętne wykrywanie oraz zaznaczanie błędów
c) Wykonywanie operacji na zdefiniowanych wcześniej jednostkach
d) Kontrola poprawności definiowania jednostek

Wymagania niefunkcjonalne:

a) W przypadku uruchomienia programu z niepoprawnymi argumentami użytkownik powinien zostać poinformowany o możliwych parametrach startowych
b) Komunikaty o błędach powinny informować o typie i miejscu wystąpienia błędu

Sposób testowania:

Testy jednostkowe z wykorzystaniem biblioteki JUnit.

Analiza leksykalna:

Moduł leksera będzie przetwarzał otwarty plik wejściowy na tokeny, które będą potem badane w analizatorze składniowym. Lekser będzie odczytywał dane z pliku znak po znaku, do czasu stwierdzenia odczytania całości sekwencji odpowiadającej jednemu z rozpoznawanych i akceptowanych tokenów języka. Kolejne tokeny będą pojedynczo przekazywane do parsera.

Analiza składniowa:

Moduł parsera będzie otrzymywał od analizatora leksykalnego kolejne tokeny. Jego zadaniem jest sprawdzenie, czy wszystkie rozpoznane tokeny są ułożone zgodnie ze zdefiniowaną gramatyką języka. Podczas analizy składniowej będzie przeprowadzana detekcja błędów. W przypadku napotkania błędu będzie podawany jego typ i oraz miejsce wystąpienia. Następnie będzie tworzone drzewo składniowe, które zostanie przekazane do analizatora semantycznego.

Analiza semantyczna:

Moduł analizatora semantycznego będzie sprawdzał poprawność drzewa składniowego utworzonego przez analizator składniowy. Analizator będzie między innymi sprawdzał poprawność używanych identyfikatorów oraz przypadki ich nadpisania, poprawność użycia operatorów matematycznych czy deklarowania funkcji, zmiennych i jednostek.

Gramatyka:

```
assignmentOp = "="

orOp = "|"

andOp =  "&"

equalOp = "==" | "!="

relationOp = "<" | ">" | "<=" | ">="
```

additiveOp = "+"

minusOp = "-"

multiplicativeOp = "*"

divideOp = "/"

comment = "//"

digit = ‘0’ | non_zero_digit

non_zero_digit = ‘1’ | ‘2’ | … | ‘9’

smallletter = ‘a’ | … | ‘z’

bigletter = ‘A’ | … | ‘Z’

symbol = ‘”’ | ‘(‘ | ‘ ‘|…

letter = smallletter | bigletter

string = ‘”’, {letter | digit | symbol}, ‘”’

name = letter, {letter | digit | ‘_’}

BIGname = {bigletter} ;

number = ({digit}, ‘.’, {digit}) | (non_zero_digit, {digit})

Składnia:

Program = FunctionDef ;

FunctionDef = 'function', name, ParamList, FunctionBlock ;

ParamList = '(', Name, {'(', Name}, ')'

FunctionBlock = '{', {Statement}, ReturnStatement, '}'

Statement = (IfStatement | WhileStatement|Assignment | FunctionCall | PrintCall | ReturnStatement | VarDeclaration | UnitDeclaration), ';' ;

IfStatement = IfInstr, [ElseInstr] ;

WhileStatement = ‘while’, ‘(‘, [expression], ‘) ‘, ‘{‘, {Statement}, ‘}’

Assignment = name, ‘=’, AddExp ;

FunctionCall = name, ‘(‘, {BasicExp}, ‘)’ ;

PrintCall = ‘print’, ‘(‘, string | AddExp, ‘)’ ;

ReturnStatement = ‘return’, AddExp ;

VarDeclaration = ‘var’, name, [‘=’, AddExp] ;

UnitDeclaration = MainUnitDecl | OtherUnitDecl ;

MainUnitDecl = ‘def’, name, ‘ :’, ‘[‘,BIGname, ‘]’ ;

OtherUnitDecl = ‘def’, ‘name’, ‘ :’, Unit ;

IfInstr = 'if', '(', Expression, ')', '{', Statement, {Statement}, '}' ;

ElseInstr = 'else', '{', Statement, {Statement}, '}' ;

CompareExp = AddExp, (‘==’ | ‘ !=’ | ‘<=’ | ‘>=’ | ‘’), AddExp, {(‘==’ | ‘ !=’ | ‘<=’ | ‘>=’ | ‘’), CompareExp};

NegativeExp = ‘-‘, number ;

BasicExp = string | name | number | FunctionCall | Unit ;

AddExp = MultiExp, {(‘+’|’-’) MultiExp};

MultiExp = BasicExp, {(‘*’|’/’) BasicExp};

Unit = ‘[‘, number, name, ‘]’;