TKOM – sprawozdanie końcowe

Kalkulator "z jednostkami"

Michał Berliński – 290432

Oświadczam, że niniejsza praca stanowiąca podstawę do uznania osiągnięcia efektów uczenia się z przedmiotu Techniki kompilacji została wykonana przeze mnie samodzielnie.

Opis funkcjonalny:

Język ma umożliwiać proste przetwarzanie zmiennych zawierających wartości liczbowe razem z przypisanymi do nich jednostkami. Będzie umożliwiać użytkownikowi definiowanie własnych jednostek z określonych wymiarów (waga, odległość itp.). 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 (niektóre operacje będą zastrzeżone tylko dla konkretnych typów danych, nie można np. mnożyć typu napisowego)
- 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 przez użytkownika 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’, ‘(‘, CompareExp, ‘) ‘, ‘{‘, {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', '(', CompareExp, ')', '{', Statement, {Statement},
'}' ;

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

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

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

NegativeExp = (‘0’, ‘minusOp’, ‘number’) | (‘Unit’, ‘minusOp’,
‘Unit’)

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

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

Unit = ‘[‘, number, name, ‘]’;
```

Szczegóły implementacyjne

Moduł leksera :

Obiekt klasy lekser posiada 2 tryby działania – odczytywanie danych z pliku tekstowego lub ze stringa. Ich wybór jest dokonywany automatycznie na podstawie danych przekazanych do konstruktora klasy Skaner. Obiekt tej klasy jest wykorzystywany w funkcji *nextToken()* gdzie wczytywany jest pojedynczy znak. Na jego podstawie rozpoznawany jest rodzaj pobieranego tokenu. Jeśli informacja zawarta w pojedynczym znaku nie wystarcza do rozpoznania jaki token napotkano, lekser odkłada ten token na listę pomocniczą (*characters*), a następnie wczytuje następny znak. Jeśli token dalej nie będzie mógł być rozpoznany, operacja zostanie powtórzona. Jeśli wszystkie znaki zapisane w liście pomocniczej utworzą token zostanie ona wyczyszczona przez końcem działania funkcji wczytującej. W przeciwnym wypadku następnie jej wywołania będą zdejmować elementy z listy dopóki będzie ona niepusta.

W przypadku wczytania sekwencji znaków niebędącej żadnym z rozpoznawanych i akceptowanych tokenów, zostanie utworzony token typu *UNKNOWN* sygnalizujący błąd leksykalny (zostanie później obsłużony w parserze).

W skład modułu leksera wchodzą następujące pliki:

- klasa *Skaner* - wczytująca pojedynczy znak, zwracająca go oraz sprawdzająca fakt wystąpienia końca pliku
- klasa *Token* - reprezentująca obiekty będące tokenami. Pojedynczy Token jest tworzony na podstawie współrzędnych x i y ostatniego znaku tokenu, jego typu oraz wartości mogącej być *Stringiem, Integerem* lub *Doublem*. W zależności od jej rodzaju wywoływany jest odpowiedni konstruktor.
- Klasa *TokenPrefix* - posiadająca metody pozwalające rozpoznać jakiego typu jest aktualnie wczytany znak

- Klasa *TokenType* - będąca zmienną typu *enum* przechowująca wszystkie typy tokenów rozpoznawane przez program
- Klasa *Lekser* - wykorzystuje metody oraz obiekty wcześniej wspomnianych klas i zarządza (w funkcji *nextToken()*) procesem tworzenia nowego tokenu. Wartością zwracaną przez funkcję *nextToken()* jest nowy Obiekt klasy Token posiadający określone współrzędne, wartość oraz typ.

Działanie modułu leksykalnego jest sprawdzane przez klasę *LexerTest* znajdującą się w module testowym. Klasa ta przeprowadza testy jednostkowe z wykorzystaniem biblioteki JUnit.

Moduł parsera:

Moduł parsera otrzymuje od analizatora leksykalnego kolejne tokeny. Jego zadaniem jest sprawdzenie, czy wszystkie rozpoznane tokeny są ułożone zgodnie ze zdefiniowaną gramatyką języka. Parser tworzy drzewo składniowe, przyjmując kolejne tokeny od modułu leksykalnego i decydując w jakim miejscu drzewa powinny się one znaleźć. Podczas analizy składniowej przeprowadzana jest detekcja błędów. Jeśli przyjęty jest token typu *UNKNOWN* lub też sekwencja przyjętych tokenów nie jest dopuszczona przez składnię języka parser wyrzuca wyjątek klasy *ParserException* informując użytkownika o rodzaju oraz miejscu wystąpienia błędu.

W skład modułu parsera wchodzą następujące pliki:

- Interfejs *AST* będący podstawą wszystkich obiektów tworzących drzewo składniowe, wymusza implementację funkcji *accept* (wykorzystywanej w module interpretującym) w każdej z klas wykorzystujących go
- Klasa *ASTnode,* w której znajdują się zagnieżdżone klasy implementujące interfejs *AST* i reprezentujące wszystkie typy obiektów mogące znaleźć się w drzewie składniowym. Są to odpowiednio klasy:
    - *Program* – przechowuje mapę funkcji znajdujących się w programie
    - *FunctionDef* – reprezentuje deklarację funkcji, przechowuje swoją nazwę (będącą tokenem) oraz obiekty typu *AST* reprezentujące listę parametrów funkcji oraz jej ciało
    - *ParamList* – lista parametrów funkcji, przechowuje listę obiektów *AST,* z których każdy jest parametrem
    - *FunctionBody* – ciało funkcji, przechowuje swoją listę wyrażeń (obiekty typu *AST)*
    - *Variable* – reprezentuje zmienną, posiada nazwę (token) oraz wartość (*AST)*
    - *FunctionCall* – wywołanie funkcji, przechowuje nazwę funkcji (token) oraz listę parametrów (*AST*) z jakimi ją wywołano
    - *Assignment* – przypisanie, przechowuje zmienną (*AST*), do której dokonano przypisanie oraz przypisane wyrażenie (*AST*)
    - *VarDeclaration* – deklaracja zmiennej, posiada zmienną (*AST*), którą zadeklarowano oraz ewentualną przypisaną wartość (*AST*)
    - *IfStatement* – instrukcja warunkowa if, posiada warunek i bloki instrukcji w przypadku prawdy lub fałszu (wszystkie typu *AST*)
    - *WhileStatement* – instrukcja pętli, przechowuje warunek oraz blok instrukcji (oba AST)
    - *ReturnStatement* – przechowuje zwracaną wartość (*AST*)
    - *PrintCall* – przechowuje wypisywaną wartość
    - *BinOperator* – reprezentuje wyrażenie matematyczne dwuargumentowe (*AST*) oraz znak tej operacji (token)

    - *BinLogicOperator* - reprezentuje wyrażenie logiczne dwuargumentowe (*AST*) oraz znak tej operacji (token)
    - *UnOperator* – reprezentuje wyrażenie matematyczne jednoargumentowe, przechowuje argument (AST) i znak (token)
    - *IntNum* – liczba całkowita
    - *DoubleNum* – liczba zmiennoprzecinkowa
    - *StringVar* – zmienna typu napisowego
    - *BaseUnit* – jednostka typu podstawowego, posiada nazwę i kategorię
    - *Unit* – zwykła jednostka, posiada nazwę, krotność oraz nazwę jednostki bazowej swojej kategorii
    - *UnitResult* – jednostka ogółem, wykorzystywana jako zmienna posiada nazwę, krotność względem j. podstawowej, nazwę j. podstawowej swojej kategorii oraz wartość liczbową
- Klasa *Parser* – odpowiada za tworzenie drzewa składniowego z obiektów klas z klasy *ASTnode*
- Klasa *ParserException* – obsługuje błędy zgłaszane przez parser

Działanie modułu parsującego jest sprawdzane przez klasę *ParserTest* znajdującą się w module testowym. Klasa ta przeprowadza testy jednostkowe z wykorzystaniem biblioteki JUnit.

Moduł interpretujący:

Moduł analizatora semantycznego sprawdza poprawność drzewa składniowego utworzonego przez analizator składniowy. Analizator ocenia poprawność używanych identyfikatorów oraz przypadki ich nadpisania, poprawność użycia operatorów matematycznych, deklarowania funkcji, zmiennych i jednostek, zasięgi zadeklarowanych zmiennych oraz jednostek. W przypadku wystąpienia błędu semantycznego wywoływany jest wyjątek klasy InterpreterException informujący użytkownika o typie oraz miejscu wystąpienia błędu.

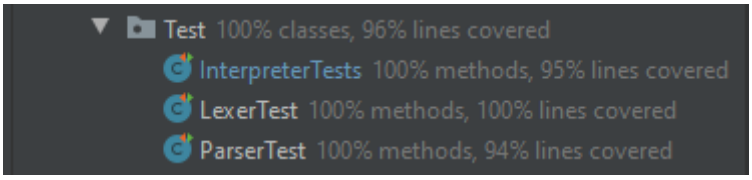
W skład modułu interpretera wchodzą następujące pliki:

- Klasa *Environment* – przechowuje i zarządza elementami, które wystąpiły w aktualnym wywołaniu programu. Są to stos kontekstów wywołań, mapa definicji funkcji występujących w programie, zmienna reprezentująca ostatnią natrafioną zmienną, globalne listy jednostek podstawowych i zwykłych, listy parametrów z definicji i wywołań funkcji
- Klasa *CallContext* – reprezentuje pojedynczy blok programu (ograniczony nawiasami klamrowymi). Przechowuje listę map zmiennych lokalnych, które wystąpiły w nim oraz kontekstach w nim występujących.
- Klasa *Interpreter* – odpowiada za konstrukcję modelu wizytora, przegląda drzewo składniowe interpretując na bieżąco jego konstrukcję oraz kontekst w jakim znalazły się znajdujące się w nim obiekty
- Klasa *InterpreterException* - obsługuje błędy zgłaszane przez interpreter

Działanie modułu interpretującego jest sprawdzane przez klasę *InterpreterTests* znajdującą się w module testowym. Klasa ta przeprowadza testy jednostkowe z wykorzystaniem biblioteki JUnit.

Moduł testujący:

W module znajdują się 3 klasy odpowiadające za przeprowadzenie testów jednostkowych Leksera, Parsera oraz Interpretera z wykorzystaniem biblioteki JUnit. Pokrycie testami wygląda następująco:



Uruchamianie:

Przed uruchomieniem programu należy pobrać ze strony Oracle'a i zainstalować najnowszą wersję Java SE Development Kit. Następnie korzystając np. z darmowego *Eclipse IDE for Java,* można sklonować projekt z repozytorium i uruchomić jako nowy projekt. W pliku main należy zmodyfikować ścieżkę do pliku z kodem.