Projekt TKOM semestr 18L – dokumentacja końcowa

Temat**: Interpreter językowy z dynamiczną biblioteką typów i jednostek SI**
Autor**: Michał Smoła**

Opis projektu

Ideą projektu jest stworzenie prostego interpretera językowego z możliwością importowania typów i jednostek SI przy pomocy zdefiniowanych bibliotek. Może mieć zastosowanie w zakresie przeliczania różnych jednostek tego samego typu.

Przyjęto następujące założenia:

- wykonywanie operacji na liczbach zmiennoprzecinkowych typu *double*;
- dynamiczne typowanie ;
- w przypadku operacji arytmetycznych dodawania i odejmowania, jeśli nie zostanie podana docelowa jednostka wynik zostanie zwrócony w jednostce podstawowej dla typu;
- definicje funkcji tylko i wyłącznie w głównym zasięgu programu bez możliwości nadpisywania, ale z możliwością rekurencyjnego wywołania;
- definicja funkcja *main* na początku każdego programu, będącą punktem wejściowym dla jednostki wykonującej;
- definicja każdego typu w osobnym pliku bibliotecznym lub poprzez odpowiedni zapis u góry programu;
- przykładowy plik biblioteczny:
  length | m : 1 | km : 1000 | mile : 1609.344 | dm : 0.1 | cm : 0.01 | mm : 0.001
  w przypadku braku operacji mnożenia i dzielenia typów, albo
  length : area | m : 1 | km : 1000 | mile : 1609.344 | dm : 0.1 | cm : 0.01 | mm : 0.001
  w przypadku możliwych operacji mnożenia i dzielenia typów
- przeparsowanie całości programu, zapamiętanie w kolekcji nazw typów, które zostały zdefiniowane w plikach, następnie przeczytanie wszystkich plików (biblioteki dynamicznej) i zapamiętanie ich zawartości w tekstowym typie danych, a w dalszej kolejności przeprasowanie samego stringa i połączenie z całością programu;
- konwersja jednostek poprzez przelicznie jednostki źródłowej do jednostki głównej SI w obrębie typu, a następnie z jednostki SI do jednostki docelowej;
- sprawdzanie poprawności po wykonaniu parsowania całego pliku programu tj. na etapie analizy semantycznej oraz samego wykonywania;

Lista zdefiniowanych tokenów

“import”, “function”, “(”, “)”, “{”, “}”, “if”, “else”, “loop”, “return”, “.”, “+”, “-”, “*”, “/”, “%”, “=”, “!”, “||”, “&&”, “==”, “!=”, “<”, “>”, “<=”, “>=”, “;”, “|”, “:”

Gramatyka

```
program = { [ importStatement ] funDefinition }

importDirective = “import” { id | “[” iDefinition “]” } “;”
importDefinition = id { “:” id } iContent { iContent }
importContent = “|” id “:” number

funDefinition = “function” id “(” [ parameters ] “)” body
parameters = id { “,” id }
body = “{” [ statement ] “}”
statement = { ifStatement | loopStatement | returnStatement “;”
      | assignStatement “;” | funCall “;”| body }
ifStatement = “if” “(” condition “)” body [ “else” body ]
loopStatement = “loop” “(” condition “)” body
returnStatement = “return” expression
assignStatement = id assignmentOp expression

expression = multiExp { addOp multiExp }
multiExp = [ unaryOp ] simpleExp { multiOp simpleExp }
simpleExp =  primaryExp [ convertOp ]
primaryExp = “(” expression “)” | id | literal | funCall
funCall = id “(” [ arguments ] “)”
arguments = expression { “,” expression }

condition = andCond { orOp andCond }
andCond = comparisonCond { andOp comparisonCond }
comparisonCond = primaryCond [ comparisonOp primaryCond ]
primaryCond = [ notOp ] ( parentCond | id | literal | funCall )
parentCond = “(” condition “)”

addOp = “+” | unaryOp
unaryOp = “-”
multiOp = “*” | “/” | “%”
assignmentOp = “=”
notOp = “!”
orOp = “||”
andOp = “&&”
comparisonOp = “==” | “!=” | “<” | “>” | “<=” | “>=”
convertOp = “.” id

funCall = id “(” [ parameters ] “)”
literal = number [ id ]
number = digit { digit } [ “.” { digit } ]
id = letter { digit | letter }
letter = “a”..“z” | “A”..“Z”
digit = “0”..”9”
```

Przykładowe konstrukcje językowe

1)
```
import length;

function example(size) {
      counter = 0;
      value = size;

      if (!size) {
            return 1mile;
      }
      else {
            jump = 10mm.cm;
            loop(counter < 10) {
                  value = (value + jump).m;
                  counter = counter + 1;
            }
      }

      return (value + 1).mile;
}

function main() {
      print(example(2.5m));
      return 0;
}
```

2)
```
function main() {
      value = 10miles;
      flag;
      flag = 1;

      if (value.km > 15km && flag != 0) {
            return 0;
      }
      else {
            return 1;
      }
}
```

Wymagania funkcjonalne

- odczytywanie, parsowanie i analiza skryptów zapisanych w plikach tekstowych
- sprawdzanie poprawności wprowadzonych danych oraz poprawne zgłaszanie błędów wykrytych podczas kolejnych etapów analizy pliku
- poprawne wykonywanie wszystkich poprawnie zapisanych instrukcji w plikach tekstowych
- poprawne parsowanie wszystkich poprawnie zapisanych informacji w dołączanych plikach bibliotecznych
- przestrzeganie logicznego porządku instrukcji sterujących
- przeprowadzanie operacji arytmetycznych na jednostkach
- wyświetlanie otrzymanych wyników operacji na standardowym wyjściu terminala

Wymagania niefunkcjonalne

- informowanie użytkownika o możliwych parametrach startowych w przypadku uruchomienia aplikacji z niepoprawnymi parametrami
- wyświetlanie prostych i przejrzystych dla użytkownika komunikatów o błędach analizy plików i wskazywanie popełnionych błędów w sposób jednoznaczny

Biblioteka standardowa

Zakres funkcjonalności zawarty w projekcie:

- operacje arytmetyczne na jednostkach (dodawanie, odejmowanie, mnożenie, dzielenie), przy założeniu, że mnożenie i dzielenie jednostek w obrębie typu jest jasno określone w pliku bibliotecznym, np. length : area | m : 1 itd… oznacza, że area jest oczekiwanym typem dla mnożenia dwóch wartości typu length
- konwersja jednostek przy pomocy wyrażenia *.dstUnit*
- wypisywanie wyniku obliczeń na ekran

Idea budowy i działania projektu

Projekt został napisany jako aplikacja konsolowa z przeznaczenie dla środowiska Linux, w języku Java (standard Java 8) z wykorzystaniem biblioteki ANTLR jako biblioteka parsująca oraz narzędzia Maven do automatyzacji budowania aplikacji.

Wynik poszczególnych etapów analizy pliku oraz samego wyniku interpretacji końcowej i wykonania jest wyświetlany na standardowym wyjściu. W zależności od ogólnego wyniku analizy, na standardowe wyjście są zgłaszane: błędy analizy pliku, błędy składniowe, błędy semantyczne lub wynik wykonania skryptu. Nie przewiduje się zapisywania wyników wykonania do pliku.

Program składa się z modułów odpowiedzialnych za kolejne etapy analizy plików wejściowych, oraz z dodatkowych modułów pomocniczych. Cały proces analizy i wykonywania skryptów odbywa się w następujących etapach:

1. Analiza leksykalna (moduł lexera) oraz analiza składniowa (moduł parsera)
2. Analiza semantyczna (moduł analizatora semantycznego)
3. Wykonanie zbioru instrukcji (moduł wykonawczy)

Moduły pomocnicze dotyczą takich kwestii jak obsługa plików, prezentacja wyników na standardowym wyjściu czy obsługa błędów.

Etap pierwszy został zaimplementowany w osobnym pakiecie *analyzer*, który składa się z następujących klas:

- **UnitLanguageLexer** – wygenerowany lexer;
- **UnitLanguageBaseVisitor** i **UnitLanguageParser** - wygenerowany parser;
- **UnitLanguageVisitor** – wygenerowana klasa pomocnicza używana podczas budowania drzewa rozbioru;
- **BuildingParser** – klasa odpowiadająca za budowanie drzewa rozbioru i umieszczania obiektów w odpowiednich strukturach;

Kolejne etapy zdefiniowana w następujących klasach

- **SemChecker** – klasa odpowiadająca za ogólną analizę semantyczną programu, tj. wykrycie nieprawidłowości na poziomie nazw funkcji czy plików bibliotecznych;
- **Helper** – klasa pomocnicza implementująca metody odpowiedzialne za obsługę biblioteki dynamicznej;
- **Interpreter** – klasa odpowiadająca za punkt wejściowy do programu, poprzez podanie argumentu wskazanie lokalizacji pliku z programem;
- **Executor** – klasa kontrolująca wykonywanie wszystkich faz programu;

Dodatkowo zdefiniowana moduł obsługi błędów jako pakiet klas wyjątków mogących pojawić się na etapie analizy semantycznej oraz wykonywania programu. W komunikacie o błędzie na początku linijki podawana jest informacja w formacie [*a*:*b*], gdzie *a* to numer linii, natomiast *b* to numer kolumny, w której wystąpiła niezgodność.

Klasy odpowiedzialne za drzewo rozbioru zaimplementowane zgodnie z wyżej wymienioną gramatyką i wyglądają one następująco:

- **condition** – pakiet klas obliczających wartość wyrażeń logicznych dla instrukcji sterujących
- **element** – pakiet klas odpowiadających za obiekty terminalne w programie, zawierający także klasę **Element**, pochodną dla wszystkich klas drzewa rozbioru
- **expression** – pakiet klas obliczających wartość wyrażenia arytmetycznego
- **importSt** – pakiet klas odpowiedzialnych za przechowywanie informacji o typach i przelicznikach jednostek
- **statement** – pakiet klas instrukcji możliwych do wykorzystania w programie
- **Body** – klasa przechowująca zestaw instrukcji z określonym kontekstem zmiennych
- **Common** – klasa pomocnicza implementująca statyczne metody pomocnicze wykorzystywane podczas wykonywania programu
- **Executable** – interfejs implementowany przez klasy bezpośrednio wykorzystywane podczas realizacji modułu wykonywania i składający się z metod
    - *execute* – odpowiedzialna za realizację wykonywania operacji
    - *canReturn* – informująca czy dana wykonywana operacja może zwracać wartość
- **FunDefinition** – klasa zwierająca definicję funkcji
- **NodeType** – wyliczeniowy typ danych określający typy obiektów dla drzewa rozbioru
- **OperatorType** – wyliczeniowy typ danych wiążący informację o typie operatora wraz z odpowiadającym mu kodem znakowym
- **PrintFunction** – klasa realizująca metodę biblioteki standardowej *print*, odpowiadającą za wyświetlanie wyniku wykonania na konsoli
- **Program** – główna klasa przechowująca całe drzewo rozbioru
- **Scope** – klasa kontekstu zmiennych

Testowanie

Program składa się także z testów, sprawdzających poprawność działania zaimplementowanych funkcjonalności. Należą do nich testy analizatora składniowego oraz testy aplikacyjne, sprawdzające ostateczny wynik wykonania. Część z nich przestawiono na ilustracji poniżej.

```
@Before
public void setUp() {
    System.setOut(new PrintStream(outContent));
    parser = new BuildingParser();
    importSource =
            "import [length : area | m : 1 | km : 1000 | mile : 1609.344 " +
                    "| cm : 0.01 | mm : 0.001];" +
            "import [area | m2:1 | a:100 | ha:10000 | km2:1000000];";
}

@Test
public void print_shouldShowCorrectValueByReturnStatement() {
    String source = importSource +
            "function foo(a) {" +
                "return ((a*a / 100cm ) * 10).km + 1km;" +
            "}" +
            "function main() {" +
            "    print(foo(foo(1m)));" +
            "}";
    Program program = parser.parse(source);
    SemChecker s = new SemChecker(program);
    s.check();
    program.execute( scope: null,  f: null,  i: null);
    assertEquals( expected: "10202km\r\n", outContent.toString());
}

@Test
public void print_shouldShowCorrectValueByLoopStatement() {
    String source = importSource +
            "function foo(a) {" +
                "i = 0;" +
                "loop(i < 10) {" +
                    "a = a + 1cm;" +
                    "i = i + 1;" +
                "}" +
                "return ((a / 10).cm).mm;" +
            "}" +
            "function main() {" +
                "    print(foo(1mm));" +
            "}";
    Program program = parser.parse(source);
    SemChecker s = new SemChecker(program);
    s.check();
    program.execute( scope: null,  f: null,  i: null);
    assertEquals( expected: "10,1mm\r\n", outContent.toString());
}
```

Kompilacja i uruchamianie

Do projektu został dodany plik README, w którym znajdują się niezbędne informacje do tego, jak należy skompilować program i następnie go uruchomić.

Propozycje nowych funkcjonalności

- dodanie interfejsu graficznego umożliwiającego łatwiejszą obsługę i użytkowanie programu;
- umożliwienie definiowanie zależności pomiędzy jednostkami jako wyrażeń arytmetycznych;
- zaimplementowanie funkcjonalności dodawania bibliotek z innych ścieżek, niż ścieżka programu ;