Семантичен анализ

1. Въведение

Семантичният анализатор използва синтактичното дърво, за да провери дали входната програма е семантично коректна според дефиницията на езика.

Семантичният анализатор изпълнява две основни задачи:

- Проверка за област на видимост (scope) на променливи

Областта на видимост (scope) се определя от блоковата структура на езика. Блок в програмата представлява подпрограма функция или цялата програма. Правилото за scope на променлива е, че дадена променлива е видима в текущия блок, в който е декларирана и във вложените (т.е. вътрешните) блокове, но не е видима във външните блокове. Ако има декларация на еднакъв идентификатор на променлива във вътрешен и външен блок, вътрешната декларация припокрива външната.

- Проверка за съвместимост на типове

Семантичният анализатор проверява дали всеки оператор има операнди от коректен тип. Например входният език на учебния компилатор изисква за индекс на масив да се използва целочислена стойност. Ако семантичният анализатор открие, че за индекс е използвана реална числена стойност, той трябва да докладва семантична грешка.

2. Символи

Символ, съдържащ се във входната програма се описва от следния клас:

```
public abstract class Symbol {
   private String name;
   private Type type;

   public Symbol(String name, Type type) {
      this.name = name;
      this.type = type;
   }

   public String getName() {
      return name;
   }
```

```
    public Type getType() {
        return type;
    }
}
```

- name – името на символа (get).
- type – тип на символа (get).

2.1. Символ за функция

Символ за функция се описва със следния клас:

```
public class FunctionSymbol extends Symbol {
    private List<Type> params;

    public FunctionSymbol(String name, Type type, List<Type> params) {
        super(name, type);
        this.params = params;
        if (this.params == null) this.params = new ArrayList<>();
    }

    public FunctionSymbol(String name, Type type) {
        this(name, type, null);
    }

    public List<Type> getParams() {
        return params;
    }
}
```

- params – Списък от типовете на параметрите за функцията (get).

2.2. Символ за променлива

Символ за променлива се описва от следния клас:

```
public class VariableSymbol extends Symbol {
   public VariableSymbol(String name, Type type) {
      super(name, type);
   }
}
```

3. Типове

Всеки символ има тип, който се описва със следния интерфейс:

```
public interface Type {
   public String getName();
}
```

- getName() - име на типа.

3.1. Символ за примитивен тип

Символ за примитивен тип наследява абстрактния клас Symbol и имплементира интерфейса Type, защото е както символ така и тип:

```
public class PrimitiveTypeSymbol extends Symbol implements Type {
   public PrimitiveTypeSymbol(String name) {
      super(name, null);
   }
}
```

3.2. Тип масив

Тип масив имплементира интерфейса Type, съдържа в себе си референция към типа на елементите му и се описва от следния клас:

```
public class ArrayType implements Type {
    private Type type;

    public ArrayType(Type type) {
        this.type = type;
    }

    public Type getType() {
        return type;
    }

    @Override
    public String getName() {
        return type.getName();
    }
}
```

- type – тип на елементите на масива.

4. Област на видимост (scope)

Област на видимост се описва от следния интерфейс:

```
public interface Scope {
    public Scope getEnclosingScope();
    public void addSymbol(Symbol symbol);
    public Symbol findSymbol(String symbolName);
}
```

- getEnclosingScope() - обхващащ scope.
- addSymbol(Symbol symbol) – добавяне на символ в символната таблица на текущия scope.

- findSymbol(String symbolName) – търсене на символ в символната таблица. Ако символът не е намерен в текущия scope, търсенето продължава в обхващащите го scope-ове.

Абстрактният клас AbstractScope предлага следната базова имплементация на интерфейса Scope:

```
public abstract class AbstractScope implements Scope {
    protected Map<String, Symbol> symbolTable;
    protected Scope enclosingScope;

    public AbstractScope() {
        this(null);
    }

    public AbstractScope(Scope enclosingScope) {
        this.enclosingScope = enclosingScope;
        this.symbolTable = new HashMap<>();
    }

    @Override
    public Scope getEnclosingScope() {
        return enclosingScope;
    }

    @Override
    public void addSymbol(Symbol symbol) {
        if (symbolTable.get(symbol.getName()) != null) {
            throw new RuntimeException("Symbol " + symbol.getName() + " is already defined!");
        }
```

```
        symbolTable.put(symbol.getName(), symbol);
    }

    @Override
    public Symbol findSymbol(String symbolName) {
        Symbol symbol = symbolTable.get(symbolName);
        if (symbol != null) {
            return symbol;
        }
        if (enclosingScope != null) {
            return enclosingScope.findSymbol(symbolName);
        }
        return null;
    }
}
```

- symbolTable – символна таблица.
- enclosingScope – обхващащият scope.

4.1. Глобален scope

Глобалният scope се представя от следния клас:

```
public class GlobalScope extends AbstractScope {
    public GlobalScope() {
        super();
        addPrimitiveTypes();
    }

    private void addPrimitiveTypes() {
        addSymbol(new PrimitiveTypeSymbol("int"));
```

```
        addSymbol(new PrimitiveTypeSymbol("char"));
        addSymbol(new PrimitiveTypeSymbol("void"));
        addSymbol(new PrimitiveTypeSymbol("boolean"));
    }
}
```

- addPrimitiveTypes() - добавяне на примитивните типове в символната таблица на глобалния scope.

4.2. Scope на функция

Scope на функция се описва от следния клас:

```
public class FunctionScope extends AbstractScope {
    private FunctionSymbol symbol;

    public FunctionScope(Scope enclosingScope, FunctionSymbol symbol){
        super(enclosingScope);
        this.symbol = symbol;
    }

    public FunctionSymbol getSymbol() {
        return symbol;
    }
}
```

- symbol – Символът на функцията.

4.3. Scope на съставен оператор

Scope на съставен оператор се описва от следния клас:

```
public class StatementScope extends AbstractScope {
    public StatementScope(Scope enclosingScope) {
        super(enclosingScope);
        if (enclosingScope instanceof FunctionScope) {
            this.symbolTable = ((FunctionScope) enclosingScope).symbolTable;
        }
    }
}
```

Ако обхващащият scope е от тип FunctionScope, тогава се ползва неговата символна таблица.

5. Съвместимост на типове

Пример за проверка на типовете при присвояване на стойност на променлива:

```
public void visit(AssignmentNode node) {
    node.getChildNodes().forEach(n -> n.accept(this));
    VariableNode variable = node.getVariable();
    AssignableNode assignable = node.getAssignable();
    if (!variable.equals(assignable)) {
        throw new SemanticException("Type mismatch!", assignable.getToken());
    }
}
```

След обхождане на наследниците на върха са вече известни типовете на променливата и на стойността за присвояване. Извършва се проверка за еквивалентност. При несъвпадение се генерира изключение.

Пример за оператор while:

```
public void visit(WhileStatementNode node) {
   ExpressionNode expression = node.getExpression();
   expression.accept(this);
   if (expression.getType() != ExpressionType.BOOLEAN.ordinal()) {
      throw new SemanticException("Expression must be of boolean type!", expression.getToken());
   }
   node.getBlock().accept(this);
}
```

Проверява се дали изразът в началото на оператора е от булев тип.

Пример за унарен оператор:

```
public void visit(MinusNode node) {
   node.getOperand().accept(this);
   SemanticUtils.handleUnaryOperators(node, ExpressionType.INT.ordinal());
   node.setType(ExpressionType.INT.ordinal());
}
```

Методът visit(MinusNode node) използва следния помощен метод за изпълнение на проверката за съвместимост на типове:

```
public static void handleUnaryOperators(UnaryOperator node, int validOperandType) {
      ExpressionNode operand = node.getOperand();
      if (operand.getType() != validOperandType) {
         throw new SemanticException("Operand of operator '" + node.getToken().getText() +
'" must be of type" + ExpressionType.getName(validOperandType) + "!", node.getToken());
      }
```

```
}
```

След посещаване на наследника на върха за оператора се проверява дали типът на операнда е валиден.

Задачи:

1. Във файла SemanticAnalyzer.java да се довършат телата на семантичните функции, маркирани с коментар /* ToDo */.

За стартиране на семантичния анализатор да се добави следния метод main:

```
public static void main(String[] args) throws IOException {
   Lexer<TokenType> lexer = new LexerImpl(new SourceImpl("resources/Fib.txt"));
   Parser<TokenType, AST> parser = new ParserImpl(lexer);
   ProgramBodyNode root = (ProgramBodyNode) parser.entryRule();
   SemanticVisitor semanticVisitor = new SemanticAnalyzer();
   semanticVisitor.visit(root);
   Map<VariableNode, Symbol> map = semanticVisitor.getVarSymbolMap();
}
```

2. Във входната програма на семантичния анализатор да се направят грешки, свързани с неправилна област на видимост (scope) на променливи:

- двукратно деклариране на променлива;

- използване на променлива извън нейния scope.

Да се провери дали семантичният анализатор извежда подходящи съобщения за семантични грешки.

3. Във входната програма на семантичния анализатор да се направят грешки, свързани с несъвместимост на типове:

- несъвместими типове на операндите на различни операции;

- индекс на масив различен от целочислен тип данни.

Да се провери дали семантичният анализатор извежда подходящи съобщения за семантични грешки.