Міністерство освіти і науки України
Національний технічний університет України
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики і обчислювальної техніки

Кафедра обчислювальної техніки

ПОЯСНЮВАЛЬНА ЗАПИСКА

з курсової роботи з навчальної дисципліни "Системне програмування"

Тема. Розробка компілятора з мови програмування C++ на мову асемблер x86 з підмножиною команд мови, що включає арифметичні дії, дужкові форми та оператори розгалуження

Захищено з оцінкою
Керівник роботи
_____ Павлов Валерій Георгійович
(підпис)
“___”_____ 20__ року

Допущено до захисту
Керівник роботи
_____ Павлов Валерій Георгійович
(підпис)
“___”_____ 20__ року

Роботу виконав
Студент групи ІО-52
_____ Березинець Артем Андрійович
(підпис студента)
”___” _____ 20__ р.

Київ 2017 р.

Національний технічний університет України
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»
Факультет інформатики і обчислювальної техніки

Кафедра обчислювальної техніки

ЗАВДАННЯ

на курсову роботу з навчальної дисципліни "Системне програмування"
Тема роботи: розробка компілятора з мови програмування C++ на мову асемблер x86 з підмножиною команд мови, що включає арифметичні дії, дужкові форми та оператори розгалуження.

Функціональні вимоги до програми: перевірити вхідний текст програми, написаний на мові C++. У разі коректності коду згенерувати вихідний код на мові асемблера x86, а інакше – вивести повідомлення про допущені помилки. Потрібно реалізувати підтримку типів Float, Int, Bool, а також вказівників на ці типи, арифметичні дії, оператори розгалуження, дужкові форми.

Вимоги до форматів вхідних і вихідних даних програми: вхідні дані подаються в текстовому форматі на мові C++. Вихідними даними є або повідомлення про допущені помилки у вхідному коді, або вихідний код на асемблері x86.

Вимоги до програмних та апаратних інструментів: код розробленої програми необхідно компілювати з використанням середовища програмування IntellijIDEA або Eclipse.

Рекомендована література: Ахо А.В., Лам М.С., Сети Р., Ульман Д.Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд.: Пер. с англ. – М.: ООО «И.Д. Вильямс», 2008. – 1184 с.: ил. – Парал. тит. англ.

Завдання на роботу видав	**Завдання на роботу прийняв**
Керівник роботи	**Студент групи ІО-52**
_____ Павлов Валерій Георгійович	_____ Березинець Артем Андрійович
(підпис)	(підпис студента)
“___”_____ 20__ р.	**”___” _____ 20__ р.**

Дата видачі завдання 01 листопада 2017 року.

КАЛЕНДАРНИЙ ПЛАН

№ п/п	Назва етапів виконання курсової роботи	Срок виконання етапів курсової роботи
1	Виконання розділу 1	10.11.17
2	Виконання розділу 2	20.11.17
3	Виконання розділу 3	30.11.17
4	Оформлення курсової роботи	8.12.17
5	Перевірка курсової роботи викладачем	11.12.17
6	Захист курсової роботи	18.12.17

Зміст

Вступ

Компілятор — програма, що переводить код, написаний на одній (зазвичай, високорівневій) мові програмування у іншу (зазвичай, низькорівневу) мову[1].

Потреба у компіляторі з’являється тоді, коли доцільно написати програму на одній мові, але безпосередньо виконати її можна лише на якійсь іншій мові. Наприклад, людині зручніше писати програму на високорівневій мові, адже це дозволяє абстрагуватися від деталей машинної реалізацій та сфокусуватися на головному — на вирішенні задачі. Це в свою чергу значно зменшує час, необхідний для написання коду, а також прибирає велику кількість помилок, що могли б виникнути інакше. Також зачасту дуже зменшується об’єм вихідного коду.

В якості вхідної мови компілятора був обраний C++, адже це одна з найстарших, проте й по сьогодні популярна мова програмування, що поєднує в собі позитивні стороні з обох світів: високорівневі концепції та структури разом з можливістю писати надшвидкий код, що майже однозначно переводиться у машинний. За свої роки існування C++ неодноразово покращувався та змінювався. Зараз він використовується скрізь, де важливим є час розв’язання задачі. Проте аби стати професіоналом у C++ потрібні роки практики.

У якості вихідної мови був обраний асемблер x86, тому що це найпопулярніший тип асемблера, який можна запустити майже на будь-якому сучасному комп’ютері. Через це програма на цьому асемблері є актуальною. Переведення вхідного тексту програми на низькорівневу мову дозволяє, за наявності гарної оптимізації в компіляторі, пришвидшити її виконання за рахунок використання особливостей архітектури цільової машини.

Вибір інструментів розв’язання задачі

Мовою реалізації компілятора було обрано Java, так як це відносно легка мова для освоєння, при цьому вона є добре читаємою, що дозволяє швидко зрозуміти, що відбувається.

Протягом попередніх років багато можна було почути, що швидкість виконання програми на Джаві залишає бажати кращого, що викликано виконанням програми у віртуальній машині. Проте останнім часом розробники вклали багато зусиль, аби прискорити цей процес, і тому в наш час цей аргумент не є актуальним.

Також в мову було додана значна кількість концепцій та бібліотек, що використовуються при написанні компілятора, наприклад `java.util.List` та `java.util.Map`, що дозволяють з легкістю зберігати списки (масиви) об’єктів будь-якого типу, або елементів у таблиці по ключах, і при цьому не обтяжувати себе думками про контролювання пам’яті. `java.util.Map` використовувався для створення таблиці, в якій кожному імені змінної відповідає тип даних, якому вона належить. Завдяки особливості бібліотеки, що не дозволяє зберігати по одному ключу декілька різних значень, при розробці компілятора з легкістю можна було перевіряти, що змінна не об’являється декілька разів.

У якості середи розробки була використана програма IntelliJ Idea компанії JetBrains. Вона зарекомендувала себе на ринку як найкраща для написання програм будь-якого рівня на мові Java.

Опис роботи програми

Створений компілятор складається з таких частин:

- Лексичний аналізатор

Лексичний аналізатор сканує вхідний текст, написаний на мові C++, а на виході видає набір лексем, допустимих у даній мові. Якщо ж створити лексему не вдається, то виводиться відповідне повідомлення про помилку. Даний етап не є обов'язковим у всіх компіляторах, проте дозволяє спростити подальші етапи, адже абстрагується від деяких особливостей вхідної мови;

- Синтаксичний аналізатор

Синтаксичний аналізатор на вхід отримує набір лексем, яку видає Лексичний аналізатор. Його завдання — продивитися набір цих лексем та знайти синтаксичні помилки. Якщо вони знайдені, то текст помилок виводиться користувачу, аби той мав змогу знайти місце помилки та виправити її. В даному компіляторі на цьому етапі також формуються проміжні структури (`Part`), що спрощують подальшу генерацію коду. Вони зберігають усю необхідну інформацію та будуть використовуватися на наступному етапі. Це робиться для пришвидшення компілятора загалом, а також для спрощення архітектури. Також на цьому етапі в процесі розбору виразів проводиться семантична перевірка, тобто робиться перевірка сумісності типів.

- Генератор коду

Генератор коду отримує на вхід список з `Part`, згенерований на попередньому етапі, а також таблицю використовуваних змінних. На основі цих змінних він може сформувати повноцінний вихідний код на мові асемблера.

Окремо прокоментуємо другий етап роботи компілятора, а саме алгоритм, за яким виконувався розбір вхідного коду.

Програма очікує, що в відповідних місцях з'являться ті чи інші лексеми. Якщо їх немає, то виводиться відповідне повідомлення про помилку. Далі

перевіряється коректність послідовності лексем. Якщо мова йде про арифметичний вираз, то тут шукалися наперед задані шаблони. Якщо якась частина виразу співпадає з шаблоном, то вона заміняється на щось інше, з урахуванням типу. Це відбувається доти, доки увесь вираз не буде замінений на одну змінну. Якщо ж на черговій ітерації не можна знайти відповідності шаблону, то це свідчить про те, що у вхідному виразі була допущена помилка. В цьому випадку виводиться повідомлення з вказанням місця помилки та її характеру.

Блок схема роботи компілятора наведена у додатку А.

Код програми компілятора наведений у додатку Б.

Інструкція для користувача

Для виконання програмного продукту необхідно використовувати відповідні класи та їх методи. Ось приклад функцій `main`,що виконує всі етапи роботи компілятора:

```
public static void main(String[] args) {
    final String input = "int b; bool c; if (3 > (4)) {} else c = true; b=5;";
    System.out.println("Input: " + input);

    LexicalAnalyzer lexical = new LexicalAnalyzer();
    final List<Token> tokens = lexical.analyze(input);
    if (tokens == null) {
        System.out.println("*** Lexical Analyzer Error ***: " + "Lexical analysis failed.");
        return;
    }

    SyntaxAnalyzer syntax = new SyntaxAnalyzer();
    final StringBuilder code = syntax.parse(tokens);
    if (code == null) {
        System.out.println("*** Syntax Analyzer Error ***: Syntax analysis failed.");
        return;
    }

    OutputGenerator outputGenerator = new OutputGenerator(code, syntax.getRegisteredIds());
    final String output = outputGenerator.generateCode();

    System.out.println("======================================");
    System.out.println(output);
}
```

Розглянемо детальніше цей код:

- Створення змінної типу `String`, що зберігає текст коду на мові C++, який потрібно зкомпілювати;
- Створюється змінна типу `LexicalAnalyzer`, а також список лексем, який ініціалізується результатом роботи методу `analyze` лексичного аналізатора;
- Перевірка рівності списку лексем `null` — це є сигналом, що знайдено помилку. У цьому випадку метод сам вже вивід необхідні повідомлення, нам залишилось тільки завершити роботу;

- Якщо помилок немає, то створення змінної типу `SyntaxAnalyzer`, для якої викликається метод `parse`, що бере на вхід список лексем, створений на попередньому етапі;
- Ініціалізація змінної типу `StringBuilder`, що зберігатиме частковий код, та ініціалізація її результатом роботи методу `parse`. Якщо було помилки, то метод вивів їх на екран, а змінна буде дорівнювати `null`. Нам залишається лише завершити роботу;
- Якщо помилок немає, то створюємо змінну типу `OutputGenerator`, що ініціалізується за допомогою створеного на попередньому етапі списку з часткових кодів, а також таблицею змінних з їх типами;
- Останній крок — вивід результату роботи на екран, тобто коду на мові асемблеру. Для цього викликаємо метод `generateCode`.

Тестування програми

- Перевірка лексем

```
Input: int b; float dd3; float 3dd;¬
*** Lexical Analyzer Error ***: Invalid Id: '3dd'. First char must be a letter.
*** Lexical Analyzer Error ***: Internal error at index 27.¬
*** Lexical Analyzer Error ***: Lexical analysis failed.¬
```

- Перевірка дужкових форм

```
Input: int b; b = ((4) + ((5-2))¬
*** Syntax Analyzer Error ***: Unmatched '(' parentheses at index 6.¬
*** Syntax Analyzer Error ***: Syntax analysis failed.¬
```

```
Input: int b; b = (4{)}¬
*** Syntax Analyzer Error ***: Unmatched ')' parentheses at index 8.¬
*** Syntax Analyzer Error ***: Syntax analysis failed.¬
```

- Перевірка типів

```
Input: int a; float b; a = 2 + b + 1.0;¬
*** Syntax Analyzer Error ***: The type on the right of the '=' does not match-
the type on the left side. At index 8.¬
*** Syntax Analyzer Error ***: Invalid expression at index 8.¬
*** Syntax Analyzer Error ***: Invalid assignment at index 6.¬
*** Syntax Analyzer Error ***: Syntax analysis failed.¬
```

```
Input: float a; a = true * 2;¬
*** Syntax Analyzer Error ***: Incompatible types with operator:¬
bool OperatorMultiply int.¬
*** Syntax Analyzer Error ***: Invalid expression at index 5.¬
*** Syntax Analyzer Error ***: Invalid assignment at index 3.¬
*** Syntax Analyzer Error ***: Syntax analysis failed.¬
```

- Перевірка вказівників

```
Input: float* a; a = a + 2; a = a * 2;¬
*** Syntax Analyzer Error ***: Incompatible types with operator:-
float* OperatorMultiply int.¬
*** Syntax Analyzer Error ***: Invalid expression at index 12.¬
*** Syntax Analyzer Error ***: Invalid assignment at index 10.¬
*** Syntax Analyzer Error ***: Syntax analysis failed.¬
```

```
Input: float* a; bool* b; b = a;¬
*** Syntax Analyzer Error ***: The type on the right of the '=' does not match-
the type on the left side. At 10.¬
*** Syntax Analyzer Error ***: Invalid expression at index 10.¬
*** Syntax Analyzer Error ***: Invalid assignment at index 8.¬
*** Syntax Analyzer Error ***: Syntax analysis failed.¬
```

- Перевірка операторів розгалудження

```
Input: int a; if (a > 2) {} else;¬
*** Syntax Analyzer Error ***: Unexpected Else block end at index 12.¬
*** Syntax Analyzer Error ***: Syntax analysis failed.¬
```

```
Input: int a; else a = 2;¬
*** Syntax Analyzer Error ***: Unexpected starting token at 3: KeywordElse.¬
*** Syntax Analyzer Error ***: Syntax analysis failed.¬
```

- Перевірка арифметичних та логічних операцій

```
Input: int b; bool c; c = 1 + (2 >= (2 / 4 * 2.0));¬
*** Syntax Analyzer Error ***: Incompatible types with operator:¬
int OperatorPlus bool.¬
*** Syntax Analyzer Error ***: Invalid expression at index 8.¬
*** Syntax Analyzer Error ***: Invalid assignment at index 6.¬
*** Syntax Analyzer Error ***: Syntax analysis failed.¬
```

- Загальна перевірка

```
Input:=int=b;=bool=c;=if=(3=>=(4))={}=else c = true; b=5;¬
=======================================¬
.586¬
¬
.data¬
▸   b DD ?¬
▸   c DB ?¬
¬
.code¬
main proc¬
▸   push EBP¬
▸   mov EBP, ESP¬
▸   mov EAX, 4¬
▸   push EAX¬
▸   mov EAX, 3¬
▸   pop EBX¬
▸   cmp EAX, EBX¬
▸   jg label0¬
▸   mov EAX, 0¬
▸   jmp label1¬
▸   label0:¬
▸   mov EAX, 1¬
▸   label1:¬
▸   cmp EAX, 1¬
▸   jne label2¬
▸   jmp label3¬
▸   label2:¬
▸   mov EAX, 1¬
▸   mov c, EAX¬
▸   label3:¬
▸   mov EAX, 5¬
▸   mov b, EAX¬
▸   mov ESP, EBP¬
▸   pop EBP¬
▸   ret¬
main endp¬
```

Висновки

Розроблений компілятор чітко і надійно працює на заданій множині вхідних операторів і команд. Завдяки правильній архітектурі вийшло зробити дуже гнучку систему.

Компілятор написаний на мові Java, а одже є крос-платформенним і може бути запущений майже на будь-якій сучасній системі, що є великим плюсом.

Завдяки `java.util.List` та `StringBuilder` було створено дуже красиву на елегантну систему, що надійно працює.

До недоліків варто підтримку лише окремої множини операторів вхідної мови, а також. Такой в подільшому можна було б оптимізувати вихідний асемблерний код.

Список використаної літератури

1. Ахо А.В., Лам М.С., Сети Р., Ульман Д.Д. Компиляторы: принципы, технологии и инструментарий, 2-е изд.: Пер. с англ. – М.: ООО «И.Д. Вильямс», 2008. – 1184 с.: ил. – Парал. тит. англ.
2. Свердлов С. Конструирование компиляторов. Учебное пособие// LAP Lambert Academic Publishing, 2015. ISBN 978-3-659-71665-2, 571 стр., илл.
3. Compiler [Електронний ресурс] – www.en.wikipedia.org/wiki/Compiler
4. Interface List<E> [Електронний ресурс] – ://docs.oracle.com/javase/8/docs/api/java/util/List.html
5. Interface Map<K,V> [Електронний ресурс] – https://docs.oracle.com/javase/8/docs/api/java/util/Map.html

Додаток А. Блок-схема алгоритму компілятору

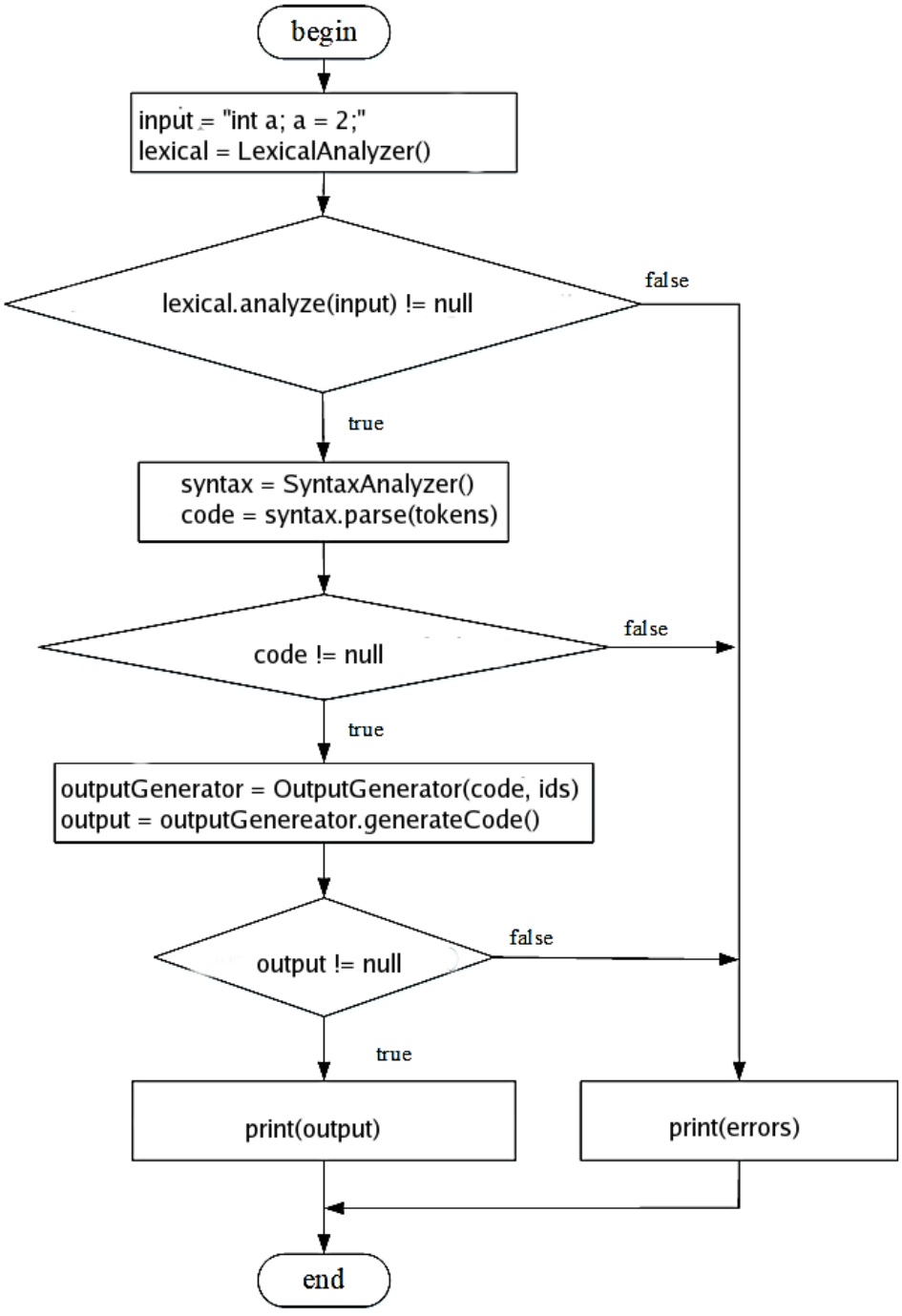

Рис. 1 – Блок-схема компілятору

Додаток Б. Вихідний код програмного продукту

LexicalAnalyzer.java

```
import java.util.ArrayList;
import java.util.List;

public class LexicalAnalyzer {

    public LexicalAnalyzer() {

    }

    public List<Token> analyze(String input) {
        List<Token> listOfTokens = new ArrayList<>();
        StringBuilder accumulator = new StringBuilder();
        boolean foundSeparator;

        for (int i = 0; i < input.length(); i++) {
            final char c = input.charAt(i);
            final boolean existsNextChar = (i + 1) < input.length();
            Token tokenToAdd = null;

            switch (c) {
                case '[':
                    tokenToAdd = new Token(TokenType.BracketsOpen, "[");
                    foundSeparator = true;
                    break;
                case ']':
                    tokenToAdd = new Token(TokenType.BracketsClose, "]");
                    foundSeparator = true;
                    break;
                case '(':
                    tokenToAdd = new Token(TokenType.ParenthesesOpen, "(");
                    foundSeparator = true;
                    break;
                case ')':
                    tokenToAdd = new Token(TokenType.ParenthesesClose, ")");
                    foundSeparator = true;
                    break;
                case '{':
                    tokenToAdd = new Token(TokenType.BracesOpen, "{");
                    foundSeparator = true;
                    break;
                case '}':
                    tokenToAdd = new Token(TokenType.BracesClose, "}");
                    foundSeparator = true;
                    break;

                case ';':
                    tokenToAdd = new Token(TokenType.Semicolon, ";");
                    foundSeparator = true;
```

```
                break;
            case ',':
                tokenToAdd = new Token(TokenType.Comma, ",");
                foundSeparator = true;
                break;

            case '+':
                tokenToAdd = new Token(TokenType.OperatorPlus, "+");
                foundSeparator = true;
                break;
            case '-':
                tokenToAdd = new Token(TokenType.OperatorMinus, "-");
                foundSeparator = true;
                break;
            case '*':
                tokenToAdd = new Token(TokenType.OperatorMultiply, "*");
                foundSeparator = true;
                break;
            case '/':
                tokenToAdd = new Token(TokenType.OperatorDivide, "/");
                foundSeparator = true;
                break;

            case '>':
                if (existsNextChar && (input.charAt(i + 1) == '=')) {
                    tokenToAdd = new Token(TokenType.OperatorGreaterOrEquals, ">=");
                    i++;
                } else {
                    tokenToAdd = new Token(TokenType.OperatorGreater, ">");
                }
                foundSeparator = true;
                break;
            case '<':
                if (existsNextChar && (input.charAt(i + 1) == '=')) {
                    tokenToAdd = new Token(TokenType.OperatorLessOrEquals, "<=");
                    i++;
                } else {
                    tokenToAdd = new Token(TokenType.OperatorLess, "<");
                }
                foundSeparator = true;
                break;

            case '!':
                if (existsNextChar && (input.charAt(i + 1) == '=')) {
                    tokenToAdd = new Token(TokenType.OperatorNotEquals, "!=");
                    i++;
                } else {
                    System.out.println("*** Lexical Analyzer Error ***: " + "Invalid token at " + i + ": " +
c + ".");
                    return null;
                }
                foundSeparator = true;
                break;
```

```
            case '=':
                if (existsNextChar && (input.charAt(i + 1) == '=')) {
                    tokenToAdd = new Token(TokenType.OperatorEquals, "==");
                    i++;
                } else {
                    tokenToAdd = new Token(TokenType.OperatorAssign, "=");
                }
                foundSeparator = true;
                break;

            case ' ':
                foundSeparator = true;
                break;

            default:
                if (!(isDigit(c) || isLetter(c) || c == '.')) {
                    System.out.println("*** Lexical Analyzer Error ***: " + "Invalid char at " + i + ": " + c
+ ".");
                    return null;
                }

                foundSeparator = false;
                accumulator.append(c);
        }

        if (foundSeparator || i == input.length() - 1) {
            if (accumulator.length() > 0) {
                if (isInt(accumulator)) {
                    listOfTokens.add(
                            new Token(TokenType.LiteralInt, accumulator.toString()));
                } else if (isFloat(accumulator)) {
                    listOfTokens.add(
                            new Token(TokenType.LiteralFloat, accumulator.toString()));
                } else if (isId(accumulator)) {
                    final String id = accumulator.toString();
                    if (id.equals("true") || id.equals("false")) {
                        listOfTokens.add(
                                new Token(TokenType.LiteralBool, id));
                    } else if (id.equals("if")) {
                        listOfTokens.add(
                                new Token(TokenType.KeywordIf, id));
                    } else if (id.equals("else")) {
                        listOfTokens.add(
                                new Token(TokenType.KeywordElse, id));
                    } else if (id.equals("int")) {
                        listOfTokens.add(
                                new Token(TokenType.KeywordInt, id));
                    } else if (id.equals("float")) {
                        listOfTokens.add(
                                new Token(TokenType.KeywordFloat, id));
                    } else if (id.equals("bool")) {
                        listOfTokens.add(
```

```
                        new Token(TokenType.KeywordBool, id));
                } else {
                    listOfTokens.add(
                        new Token(TokenType.Id, id));
                }
            } else {
                System.out.println("*** Lexical Analyzer Error ***: " + "Internal error at index " + i +
".");
                return null;
            }

            accumulator = new StringBuilder();
        }

        if (tokenToAdd != null) {
            listOfTokens.add(tokenToAdd);
            tokenToAdd = null;
        }
      }
    }

    return listOfTokens;
  }

  private boolean isId(StringBuilder token) {
    for (int i = 0; i < token.length(); i++) {
      final char c = token.charAt(i);
      if (!(isDigit(c) || isLetter(c))) {
        return false;
      }
    }

    if (!isLetter(token.charAt(0))) {
      System.out.println("*** Lexical Analyzer Error ***: " + "Invalid Id: '" + token + "'. First char
must be a letter.");
      return false;
    }

    return true;
  }

  private boolean isInt(StringBuilder token) {
    for (int i = 0; i < token.length(); i++) {
      if (!isDigit(token.charAt(i))) {
        return false;
      }
    }

    return true;
  }

  private boolean isFloat(StringBuilder token) {
    for (int i = 0; i < token.length(); i++) {
```

```
            final char c = token.charAt(i);
            if (!(isDigit(c) || c == '.')) {
                return false;
            }
        }

        return containsOnePeriod(token);
    }

    private boolean isDigit(char c) {
        return (c >= '0') && (c <= '9');
    }

    private boolean isLetter(char c) {
        return ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'));
    }

    private boolean isBoolLiteral(String input) {
        return (input.equals("true") || input.equals("false"));
    }

    private boolean containsOnePeriod(StringBuilder input) {
        boolean periodFound = false;
        for (int i = 0; i < input.length(); i++) {
            if (input.charAt(i) == '.') {
                if (periodFound) {
                    System.out.println("*** Lexical Analyzer Error ***: " + "Invalid usage of >= 2 '.'
symbols.");
                    return false;
                }
                periodFound = true;
            }
        }

        return periodFound;
    }
}
```

TokenType.java

```
public enum TokenType {
    Id,
    LiteralInt,
    LiteralFloat,
    LiteralBool,

    OperatorAssign,
    OperatorPlus,
    OperatorMinus,
    OperatorMultiply,
    OperatorDivide,
    OperatorEquals,
    OperatorNotEquals,
    OperatorLess,
```

```
    OperatorLessOrEquals,
    OperatorGreater,
    OperatorGreaterOrEquals,

    ParenthesesOpen,
    ParenthesesClose,
    BracketsOpen,
    BracketsClose,
    BracesOpen,
    BracesClose,

    Comma,
    Semicolon,

    KeywordInt,
    KeywordFloat,
    KeywordBool,
    KeywordIf,
    KeywordElse
}
```

Token.java

```
public class Token {

    public final TokenType type;
    public final String value;

    public Token(TokenType type, String value) {
        this.type = type;
        this.value = value;
    }

    @Override
    public String toString() {
        return "Token: " + value + " --- " + type.name();
    }
}
```

Type.java

```
public enum Type {
    Int, Float, Bool,
    PointerInt, PointerFloat, PointerBool;

    @Override
    public String toString() {
        switch (this) {
            case Int:
                return "int";
            case Float:
                return "float";
```

```
            case Bool:
                return "bool";
            case PointerInt:
                return "int*";
            case PointerFloat:
                return "float*";
            case PointerBool:
                return "bool*";
        }

        return "";
    }
}
```

SyntaxAnalyzer.java

```
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class SyntaxAnalyzer {

    private static int availableLabel = 0;
    private Map<String, Type> registeredIds = new HashMap<>();

    private StringBuilder code = new StringBuilder();

    public StringBuilder parse(List<Token> tokens) {
        if (!areParenthesesCorrect(tokens, 0, tokens.size())) {
            // Error message is printed by the function
            return null;
        }

        int index = 0;
        while (index < tokens.size()) {
            final Token currentToken = tokens.get(index);
            final TokenType currentTokenType = currentToken.type;
            switch (currentTokenType) {
                case KeywordInt:
                    {
                        final int declarationResultingIndex = processVariableDeclaration(tokens, index,
Type.Int, Type.PointerInt);
                        if (declarationResultingIndex == -1) {
                            System.out.println("*** Syntax Analyzer Error ***: " +
                                    "Invalid variable declaration at index " + index + ".");
                            return null;
                        }
                        index = declarationResultingIndex;
                        }
                    break;
                case KeywordFloat:
                    {
```

```
                final int declarationResultingIndex = processVariableDeclaration(tokens, index,
Type.Float, Type.PointerFloat);
                if (declarationResultingIndex == -1) {
                    System.out.println("*** Syntax Analyzer Error ***: " +
                            "Invalid variable declaration at index " + index + ".");
                    return null;
                }
                index = declarationResultingIndex;
            }
            break;
        case KeywordBool:
            {
                final int declarationResultingIndex = processVariableDeclaration(tokens, index,
Type.Bool, Type.PointerBool);
                if (declarationResultingIndex == -1) {
                    System.out.println("*** Syntax Analyzer Error ***: " +
                            "Invalid variable declaration at index " + index + ".");
                    return null;
                }
                index = declarationResultingIndex;
            }
            break;
        case Id:
            {
                final int assignmentResultingIndex = processAssignment(tokens, index);
                if (assignmentResultingIndex == -1) {
                    System.out.println("*** Syntax Analyzer Error ***: " +
                            "Invalid assignment at index " + index + ".");
                    return null;
                }
                index = assignmentResultingIndex;
            }
            break;
        case KeywordIf:
            {
                if (index + 5 >= tokens.size()) {
                    System.out.println("*** Syntax Analyzer Error ***: " +
                            "Unexpected input end. Too few tokens for an If block at index " + index + ".");
                    return null;
                }

                if (tokens.get(index + 1).type != TokenType.ParenthesesOpen) {
                    System.out.println("*** Syntax Analyzer Error ***: " +
                            "Invalid If condition: '(' expected at index " + index + ".");
                    return null;
                }
                index += 2;

                final int expressionResultingIndex = processExpression(tokens, index, Type.Bool);
                if (expressionResultingIndex == -1) {
                    System.out.println("*** Syntax Analyzer Error ***: " +
                            "Invalid If condition at index " + index + ".");
                    return null;
```

```
}
index = expressionResultingIndex;

if (tokens.get(index).type != TokenType.ParenthesesClose) {
   System.out.println("*** Syntax Analyzer Error ***: " +
         "Invalid If condition: ')' expected at index " + index + ".");
   return null;
}
index++;

// Adding the IF Code
code.append("\tcmp EAX, 1\n");
code.append("\tjne label" + availableLabel + "\n");

// The IF block
if (tokens.get(index).type == TokenType.BracesOpen) {
   index++; // skip {
   while (tokens.get(index).type != TokenType.BracesClose) {
      final int assignmentResultingIndex = processAssignment(tokens, index);
      if (assignmentResultingIndex == -1) {
         System.out.println("*** Syntax Analyzer Error ***: " +
               "Invalid assignment at index " + index + ".");
         return null;
      }
      index = assignmentResultingIndex;
   }

   // Found '}'
   index++; // skip it
} else {
   final int assignmentResultingIndex = processAssignment(tokens, index);
   if (assignmentResultingIndex == -1) {
      System.out.println("*** Syntax Analyzer Error ***: " +
            "Invalid assignment at index " + index + ".");
      return null;
   }
   index = assignmentResultingIndex;
}

if (index >= tokens.size()) {
   // The IF block was the last input
   break;
}

// End the True block
code.append("\tjmp label" + (availableLabel + 1) + "\n");

code.append("\tlabel" + availableLabel + ":\n");

// Now expecting either ELSE or something new
if (tokens.get(index).type == TokenType.KeywordElse) {
   index++;
   if (index + 1 >= tokens.size()) {
```

```
                    System.out.println("*** Syntax Analyzer Error ***: " +
                            "Unexpected Else block end at index " + index + ".");
                    return null;
                }

                // The ELSE block. Same block as before
                if (tokens.get(index).type == TokenType.BracesOpen) {
                    index++; // skip {
                    while (tokens.get(index).type != TokenType.BracesClose) {
                        final int assignmentResultingIndex = processAssignment(tokens, index);
                        if (assignmentResultingIndex == -1) {
                            System.out.println("*** Syntax Analyzer Error ***: " +
                                    "Invalid assignment at index " + index + ".");
                            return null;
                        }
                        index = assignmentResultingIndex;
                    }

                    // Found '}'
                    index++; // skip it
                } else {
                    final int assignmentResultingIndex = processAssignment(tokens, index);
                    if (assignmentResultingIndex == -1) {
                        System.out.println("*** Syntax Analyzer Error ***: " +
                                "Invalid assignment at index " + index + ".");
                        return null;
                    }
                    index = assignmentResultingIndex;
                }
            }

            // Finish the Code
            code.append("\tlabel" + (availableLabel + 1) + ":\n");

            availableLabel += 2;

            // Done with IF[-ELSE] statement
        }
        break;

    default:
        System.out.println("*** Syntax Analyzer Error ***: " +
                "Unexpected starting token at " + index + ": " + currentTokenType + ".");
        return null;
    }
  }

  return code;
}


// call me when you expect an assignment at this place and you are sure that
// the first token is an Id.
```

```
    // index === start index.
    // returns the new index(just after the assignment end), or -1 in case of error.
    private int processAssignment(List<Token> tokens, int index) {
        final String variableName = tokens.get(index).value;
        if (index + 3 >= tokens.size()) {
            System.out.println("*** Syntax Analyzer Error ***: " +
                    "Unexpected input end. Too few tokens for an assignment for Id " + variableName + " at
index " + index + ".");
            return -1;
        }

        final Token operatorToken = tokens.get(index + 1);
        if (operatorToken.type != TokenType.OperatorAssign) {
            System.out.println("*** Syntax Analyzer Error ***: " +
                    "Expected '=' after Id '" + variableName + "', but found " + operatorToken.type + " at
index " + index + " instead.");
            return -1;
        }
        index += 2;

        final Type variableType = getVariableType(variableName);
        if (variableType == null) {
            System.out.println("*** Syntax Analyzer Error ***: " +
                    "Assignment to an undeclared variable: " + variableName + ", at index " + index + ".");
            return -1;
        }

        final int expressionResultingIndex = processExpression(tokens, index, variableType);
        if (expressionResultingIndex == -1) {
            System.out.println("*** Syntax Analyzer Error ***: " +
                    "Invalid expression at index " + index + ".");
            return -1;
        }
        index = expressionResultingIndex;

        final Token lastToken = tokens.get(index);
        if (index >= tokens.size() || lastToken.type != TokenType.Semicolon) {
            System.out.println("*** Syntax Analyzer Error ***: " +
                    "Expected ';' at index " + index + ", but found " + lastToken.type + " instead.");
            return -1;
        }
        index++;

        // Add the appropriate Code
        code.append("\tmov " + variableName + ", EAX\n");

        return index;
    }


    // call me when you expect an expression at this place.
    // index === start index.
    // returns the new index(just after the expression end), or -1 in case of error.
```

```
// TODO: add code to Codes
private int processExpression(List<Token> tokens, int index, Type expectedType) {
    int parenthesesLeftToClose = 0;
    boolean endFound = false;
    int expressionEndIndex;
    for (expressionEndIndex = index; expressionEndIndex < tokens.size(); expressionEndIndex++) {
        final TokenType tokenType = tokens.get(expressionEndIndex).type;
        switch (tokenType) {
            case Id:
            case LiteralInt:
            case LiteralFloat:
            case LiteralBool:
            case OperatorPlus:
            case OperatorMinus:
            case OperatorMultiply:
            case OperatorDivide:
            case OperatorEquals:
            case OperatorNotEquals:
            case OperatorLess:
            case OperatorLessOrEquals:
            case OperatorGreater:
            case OperatorGreaterOrEquals:
                continue;
            case ParenthesesOpen:
                parenthesesLeftToClose++;
                continue;
            case ParenthesesClose:
                if (parenthesesLeftToClose == 0) {
                    endFound = true;
                } else {
                    parenthesesLeftToClose--;
                }
                break;

            default:
                endFound = true;
                break;
        }

        if (endFound) {
            break;
        }
    }

    if (!endFound) {
        System.out.println("*** Syntax Analyzer Error ***: " +
                "Could not find proper expression end.");
        return -1;
    }

    // convert
    final List<Part> parts = convertIntoParts(tokens, index, expressionEndIndex);
    if (parts == null) {
```

```
            // The error is already printed in the convertIntoParts function.
            return -1;
        }

        final PartType lastPartType = parts.get(parts.size() - 1).partType;
        if (lastPartType != PartType.Expression && lastPartType != PartType.ParenthesesClose) {
            System.out.println("*** Syntax Analyzer Error ***: " +
                    "Invalid expression: Invalid last token. Expected ')' or expression.");
            return -1;
        }

        if (!checkExpressionValidity(parts)) {
            // The error is already printed in the checkExpressionValidity function.
            return -1;
        }

        final Part part = reduceIntoPart(parts);
        if (part == null) {
            // The error is already printed in the checkExpressionValidity function.
            return -1;
        }

        if (part.type != expectedType && (!(part.type == Type.Int && expectedType == Type.Float))) {
            System.out.println("*** Syntax Analyzer Error ***: " +
                    "The type on the right of the '=' does not match the type on the left side. At " + index +
".");
            return -1;
        }

        // Add the appropriate Code
        code.append(part.code);

        return expressionEndIndex;
    }


    private Part reduceIntoPart(List<Part> parts) {
        boolean somethingChanged;
        do {
            // System.out.println("------Now:");
            // for (Part part : parts) {
            //     System.out.print(part.partType + " ");
            // }
            // System.out.println();
            somethingChanged = false;

            // ( EXPR )
            final int smallParenthesesPatternIndex = findSmallParenthesesPattern(parts);
            if (smallParenthesesPatternIndex != -1) {
                parts.remove(smallParenthesesPatternIndex + 2);
                parts.remove(smallParenthesesPatternIndex);

                somethingChanged = true;
```

```
            continue;
        }

        // EXPR OP EXPR
        final int operatorApplicationIndex = findOperatorApplication(parts);
        if (operatorApplicationIndex == -1) {
            continue;
        }

        final Part operator = parts.get(operatorApplicationIndex); // PartType.Operator..
        final Part left = parts.get(operatorApplicationIndex - 1); // PartType.Expression
        final Part right = parts.get(operatorApplicationIndex + 1); // PartType.Expression
        final Type leftType = left.type; // Int, PointerInt, ...
        final Type rightType = right.type; // Int, PointerInt, ...

        Part newPart = null;
        final String newCode =
            constructCode(left.code, operator.code, right.code);
        boolean typesWereViolated = false;
        switch (operator.partType) {
            case OperatorPlus:
            case OperatorMinus:
                if ((leftType == Type.Float || leftType == Type.Int) &&
                        (leftType == rightType)) {
                    newPart = new Part(PartType.Expression, leftType, newCode);
                } else if (leftType == Type.Float || rightType == Type.Float) {
                    newPart = new Part(PartType.Expression, Type.Float, newCode);
                } else if (isTypePointer(leftType) && rightType == Type.Int) {
                    newPart = new Part(PartType.Expression, leftType, newCode);
                } else if (isTypePointer(rightType) && leftType == Type.Int
                        && operator.partType == PartType.OperatorPlus) {
                    newPart = new Part(PartType.Expression, rightType, newCode);
                } else {
                    typesWereViolated = true;
                }
                break;
            case OperatorMultiply:
            case OperatorDivide:
                if ((leftType == Type.Float || leftType == Type.Int) &&
                        (leftType == rightType)) {
                    newPart = new Part(PartType.Expression, leftType, newCode);
                } else if ((leftType == Type.Float && rightType == Type.Int) || (rightType == Type.Float
&& leftType == Type.Int)) {
                    newPart = new Part(PartType.Expression, Type.Float, newCode);
                } else {
                    typesWereViolated = true;
                }
                break;
            case OperatorLess:
            case OperatorLessOrEquals:
            case OperatorGreater:
            case OperatorGreaterOrEquals:
                if ((leftType == Type.Float || leftType == Type.Int) &&
```

```
                (rightType == Type.Float || rightType == Type.Int)) {
              newPart = new Part(PartType.Expression, Type.Bool, newCode);
            } else {
              typesWereViolated = true;
            }
            break;
          case OperatorEquals:
          case OperatorNotEquals:
            if (leftType == rightType) {
              newPart = new Part(PartType.Expression, Type.Bool, newCode);
            } else {
              typesWereViolated = true;
            }
            break;
          default:
            System.out.println("*** Syntax Analyzer Error ***: " +
              "Internal unknown error.");
            return null;
        }

        if (typesWereViolated) {
          System.out.println("*** Syntax Analyzer Error ***: " +
            "Incompatible types with operator: " + left.type + " " +
            operator.partType + " " + right.type + ".");
          return null;
        }

        // Replace EXPR OP EXPR with NEW_EXPR
        parts.remove(operatorApplicationIndex + 1);
        parts.remove(operatorApplicationIndex + 0);
        parts.remove(operatorApplicationIndex - 1);
        parts.add(operatorApplicationIndex - 1, newPart);
        somethingChanged = true;

      } while (somethingChanged);

      if (parts.size() > 1) {
        System.out.println("^^^^^" + "more than 1");
        // Try to find the error

        // final int operatorApplicationIndex = findOperatorApplication(parts);
        // if (operatorApplicationIndex != -1) {
        //     final Part left = parts.get(operatorApplicationIndex - 1);
        //     final Part right = parts.get(operatorApplicationIndex + 1);
        //     // it did not get reduced => could not => incompatible types
        //     System.out.println("*** Syntax Analyzer Error ***: " +
        //             "Incompatible types: " + left.type + " and " + right.type + ".");
        // }

        // Hmmm... What other kind of error could it possibly be??
        // Let's hope none

        return null;
```

```
    }

    return parts.get(0);
}


private boolean isTypePointer(Type type) {
    return (type == Type.PointerInt || type == Type.PointerFloat || type == Type.PointerBool);
}


private String constructCode(String leftCode, String operatorCode, String rightCode) {
    return
        rightCode +
        "\tpush EAX\n" +
        leftCode +
        "\tpop EBX\n" +
        operatorCode;
}


// To be called right after the convertion into parts, so that
// all expressions are for sure Literals or Variables.
// Does not cover 100% of cases
private boolean checkExpressionValidity(List<Part> parts) {
    boolean lastWasExpression = false;
    boolean lastWasOperator = false;
    boolean lastWasParenthesesOpen = true;
    for (int i = 0; i < parts.size(); i++) {
        final PartType partType = parts.get(i).partType;
        if (lastWasExpression) {
            if (partType == PartType.Expression) {
                System.out.println("*** Syntax Analyzer Error ***: " +
                        "Invalid expression: Expression after Expression. Operator Expected.");
                return false;
            } else if (partType == PartType.ParenthesesOpen) {
                System.out.println("*** Syntax Analyzer Error ***: " +
                        "Invalid expression: '(' after an Expression.");
                return false;
            } else {
                lastWasExpression = false;
                lastWasOperator = isPartTypeOperator(partType);
                lastWasParenthesesOpen = false;
            }
        } else {
            if (lastWasOperator) {
                final boolean parenthesesOpen = partType == PartType.ParenthesesOpen;
                if (partType == PartType.Expression || parenthesesOpen) {
                    lastWasOperator = false;
                    lastWasExpression = !parenthesesOpen;
                    lastWasParenthesesOpen = parenthesesOpen;
                } else {
                    System.out.println("*** Syntax Analyzer Error ***: " +
```

```
                    "Invalid expression: Invalid token after operator.");
                return false;
            }
        } else { // parentheses
            if (lastWasParenthesesOpen) {
                if (partType == PartType.Expression) {
                    lastWasParenthesesOpen = false;
                    lastWasExpression = true;
                    lastWasOperator = false;
                } else if (partType == PartType.ParenthesesOpen) {
                    // nothing changes
                } else {
                    System.out.println("*** Syntax Analyzer Error ***: " +
                            "Invalid expression: Invalid token after '('. Expected '(' or expression.");
                    return false;
                }
            } else { // ')'
                if (partType == PartType.ParenthesesClose) {
                    // nothing changes
                } else if (isPartTypeOperator(partType)) {
                    lastWasOperator = true;
                    lastWasExpression = false;
                    lastWasParenthesesOpen = false;
                } else {
                    System.out.println("*** Syntax Analyzer Error ***: " +
                            "Invalid expression: Invalid token after ')'. Expected ')' or operator.");
                    return false;
                }
            }
        }
    }

    return true;
}

private boolean isPartTypeParentheses(PartType partType) {
    if (partType == PartType.ParenthesesOpen ||
        partType == PartType.ParenthesesClose) {
        return true;
    }

    return false;
}

private boolean isPartTypeOperator(PartType partType) {
    if (partType == PartType.ParenthesesOpen ||
        partType == PartType.ParenthesesClose ||
        partType == PartType.Expression) {
        return false;
    }

    return true;
```

```
}


private int findOperatorApplication(List<Part> parts) {
    for (int i = 1; i < parts.size() - 1; i++) {
        final Part operator = parts.get(i);
        final PartType operatorPartType = operator.partType;
        if (operatorPartType == PartType.ParenthesesOpen ||
            operatorPartType == PartType.ParenthesesClose ||
            operatorPartType == PartType.Expression) {
            continue;
        }

        // Definetely some kind of operator

        final Part left = parts.get(i - 1);
        final Part right = parts.get(i + 1);
        if (left.partType == PartType.Expression && right.partType == PartType.Expression) {
            return i;
        }
    }

    return -1;
}


// returns the position of the pattern, -1 otherwise
private int findSmallParenthesesPattern(List<Part> parts) {
    for (int i = 0; i < parts.size() - 2; i++) {
        final PartType part1 = parts.get(i).partType;
        final PartType part2 = parts.get(i + 1).partType;
        final PartType part3 = parts.get(i + 2).partType;

        if (part1 == PartType.ParenthesesOpen &&
            part2 == PartType.Expression &&
            part3 == PartType.ParenthesesClose) {
            return i;
        }
    }

    return -1;
}


// [start;end)
private List<Part> convertIntoParts(List<Token> tokens, int start, int end) {
    List<Part> parts = new ArrayList<>();
    for (int i = start; i < end; i++) {
        final TokenType tokenType = tokens.get(i).type;
        final String tokenValue = tokens.get(i).value;
        switch (tokenType) {
            case OperatorPlus:
                parts.add(new Part(PartType.OperatorPlus, null, "\tadd EAX, EBX\n"));
```

```
    break;
case OperatorMinus:
    parts.add(new Part(PartType.OperatorMinus, null, "\tsub EAX, EBX\n"));
    break;
case OperatorMultiply:
    parts.add(new Part(PartType.OperatorMultiply, null, "\tmul EBX\n"));
    break;
case OperatorDivide:
    parts.add(new Part(PartType.OperatorDivide, null, "\tdiv EBX\n"));
    break;

case OperatorLess:
    parts.add(new Part(PartType.OperatorLess, null,
            "\tcmp EAX, EBX\n" +
            "\tjl label" + availableLabel + "\n" +
            "\tmov EAX, 0\n" +
            "\tjmp label" + (availableLabel + 1) + "\n" +
            "\tlabel" + availableLabel++ + ":\n" +
            "\tmov EAX, 1\n" +
            "\tlabel" + availableLabel++ + ":\n" // +1'ed
    ));
    break;
case OperatorLessOrEquals:
    parts.add(new Part(PartType.OperatorLessOrEquals, null,
            "\tcmp EAX, EBX\n" +
            "\tjle label" + availableLabel + "\n" +
            "\tmov EAX, 0\n" +
            "\tjmp label" + (availableLabel + 1) + "\n" +
            "\tlabel" + availableLabel++ + ":\n" +
            "\tmov EAX, 1\n" +
            "\tlabel" + availableLabel++ + ":\n" // +1'ed
    ));
    break;
case OperatorGreater:
    parts.add(new Part(PartType.OperatorGreater, null,
            "\tcmp EAX, EBX\n" +
            "\tjg label" + availableLabel + "\n" +
            "\tmov EAX, 0\n" +
            "\tjmp label" + (availableLabel + 1) + "\n" +
            "\tlabel" + availableLabel++ + ":\n" +
            "\tmov EAX, 1\n" +
            "\tlabel" + availableLabel++ + ":\n" // +1'ed
    ));
    break;
case OperatorGreaterOrEquals:
    parts.add(new Part(PartType.OperatorGreaterOrEquals, null,
            "\tcmp EAX, EBX\n" +
            "\tjge label" + availableLabel + "\n" +
            "\tmov EAX, 0\n" +
            "\tjmp label" + (availableLabel + 1) + "\n" +
            "\tlabel" + availableLabel++ + ":\n" +
            "\tmov EAX, 1\n" +
            "\tlabel" + availableLabel++ + ":\n" // +1'ed
```

```
                ));
                break;
            case OperatorEquals:
                parts.add(new Part(PartType.OperatorEquals, null,
                        "\tcmp EAX, EBX\n" +
                        "\tje label" + availableLabel + "\n" +
                        "\tmov EAX, 0\n" +
                        "\tjmp label" + (availableLabel + 1) + "\n" +
                        "\tlabel" + availableLabel++ + ":\n" +
                        "\tmov EAX, 1\n" +
                        "\tlabel" + availableLabel++ + ":\n" // +1'ed
                ));
                break;
            case OperatorNotEquals:
                parts.add(new Part(PartType.OperatorNotEquals, null,
                        "\tcmp EAX, EBX\n" +
                        "\tjne label" + availableLabel + "\n" +
                        "\tmov EAX, 0\n" +
                        "\tjmp label" + (availableLabel + 1) + "\n" +
                        "\tlabel" + availableLabel++ + ":\n" +
                        "\tmov EAX, 1\n" +
                        "\tlabel" + availableLabel++ + ":\n" // +1'ed
                ));
                break;

            case ParenthesesOpen:
                parts.add(new Part(PartType.ParenthesesOpen, null, ""));
                break;
            case ParenthesesClose:
                parts.add(new Part(PartType.ParenthesesClose, null, ""));
                break;

            case Id:
                final Type variableType = getVariableType(tokenValue);
                if (variableType == null) {
                    System.out.println("*** Syntax Analyzer Error ***: " +
                            "Usage of undeclared variable at index " + i + ": " + tokenValue + ".");
                    return null;
                }
                parts.add(new Part(PartType.Expression, variableType, "\tmov EAX, " + tokenValue +
"\n"));
                break;
            case LiteralInt:
                parts.add(new Part(PartType.Expression, Type.Int, "\tmov EAX, " + tokenValue + "\n"));
                break;
            case LiteralFloat:
                parts.add(new Part(PartType.Expression, Type.Float, "\tmov EAX, " + tokenValue +
"\n"));
                break;
            case LiteralBool:
                parts.add(new Part(PartType.Expression, Type.Bool, "\tmov EAX, " +
(tokenValue.equals("true") ? 1 : 0) + "\n"));
                break;
```

```
            default:
                System.out.println("*** Syntax Analyzer Error ***: " +
                        "Invalid token in expression at index " + i + ": " + tokenType + ".");
                return null;
            }
        }

        return parts;
    }


    // call me when you expect a declaration at this place and you are sure that
    // the first token's type is a type keyword.
    // index === start index.
    // returns the new index(just after the ';'), or -1 in case of error.
    private int processVariableDeclaration(List<Token> tokens, int index, Type potentialRegularType,
Type potentialPointerType) {
        final Token currentToken = tokens.get(index);
        final int leftAmount = tokens.size() - index;
        final TokenType currentTokenType = currentToken.type;
        if (leftAmount < 3) {
            System.out.println("*** Syntax Analyzer Error ***: " +
                    "Unexpected input end. Too few tokens after " + currentToken + " at index " + index +
".");
            return -1;
        }

        final Token secondToken = tokens.get(index + 1);
        final Type type;
        final Token variable;
        final boolean declaringPointer = secondToken.type == TokenType.OperatorMultiply;
        if (declaringPointer) {
            type = potentialPointerType;
            variable = tokens.get(index + 2);
        } else {
            type = potentialRegularType;
            variable = secondToken;
        }

        if (variable.type != TokenType.Id) {
            System.out.println("*** Syntax Analyzer Error ***: " +
                    "Id expected, but " + variable.type + " found instead at index " + index + ".");
            return -1;
        }

        if (registeredIds.containsKey(variable.value)) {
            System.out.println("*** Syntax Analyzer Error ***: " +
                    "Redeclaration of variable " + variable.value + " at index " + index + ".");
            return -1;
        }

        registeredIds.put(variable.value, type);
```

```
        index += declaringPointer ? 3 : 2;

        while (tokens.get(index).type == TokenType.Comma) {
            if (index + 2 >= tokens.size()) {
                System.out.println("*** Syntax Analyzer Error ***: " +
                        "Unexpected input end. Too few tokens after ',' after index " + index + ".");
                return -1;
            }

            final Token nextVariable = tokens.get(index + 1);
            if (nextVariable.type != TokenType.Id) {
                System.out.println("*** Syntax Analyzer Error ***: " +
                        "Expected an Id, but found " + nextVariable.type + " at " + (index + 1) + " instead.");
                return -1;
            }

            registeredIds.put(nextVariable.value, type);
            index += 2;
        }

        final Token lastToken = tokens.get(index);
        if (lastToken.type != TokenType.Semicolon) {
            System.out.println("*** Syntax Analyzer Error ***: " +
                    "Expected ';' or ',', but found " + lastToken.type + " at " + index + " instead.");
            return -1;
        }
        index++;

        return index;
    }


    private boolean areParenthesesCorrect(List<Token> tokens, int start, int end) {
        final int parenthesesCode = 0; // ()
        final int bracesCode = 1; // {}
        final int bracketsCode = 2; // []

        int[] stack = new int[tokens.size()];
        int stackTop = -1;

        for (int i = start; i < end; i++) {
            final Token token = tokens.get(i);
            final TokenType tokenType = token.type;
            switch (tokenType) {
                case ParenthesesOpen:
                    stack[++stackTop] = parenthesesCode;
                    break;
                case ParenthesesClose:
                    if (stackTop < 0 || stack[stackTop] != parenthesesCode) {
                        System.out.println("*** Syntax Analyzer Error ***: " + "Unmatched ')' parentheses at
index " + i + ".");
                        return false;
                    }
```

```
                stackTop--;
                break;
            case BracketsOpen:
                stack[++stackTop] = bracketsCode;
                break;
            case BracketsClose:
                if (stackTop < 0 || stack[stackTop] != bracketsCode) {
                    System.out.println("*** Syntax Analyzer Error ***: " + "Unmatched ']' bracket.");
                    return false;
                }
                stackTop--;
                break;
            case BracesOpen:
                stack[++stackTop] = bracesCode;
                break;
            case BracesClose:
                if (stackTop < 0 || stack[stackTop] != bracesCode) {
                    System.out.println("*** Syntax Analyzer Error ***: " + "Unmatched '}' braces.");
                    return false;
                }
                stackTop--;
                break;
            default:
                break;
        }
    }

    final boolean correct = stackTop == -1;
    if (!correct) {
        switch (stack[0]) {
            case parenthesesCode:
                System.out.println("*** Syntax Analyzer Error ***: " + "Unmatched '(' parentheses.");
                break;
            case bracketsCode:
                System.out.println("*** Syntax Analyzer Error ***: " + "Unmatched '[' brackets.");
                break;
            case bracesCode:
                System.out.println("*** Syntax Analyzer Error ***: " + "Unmatched '{' braces.");
                break;
            default:
                break;
        }
    }

    return correct;
}


private Type getVariableType(String name) {
    return registeredIds.get(name);
}
```

```
    public Map<String, Type> getRegisteredIds() {
        return registeredIds;
    }
}
```

OutputGenerator.java

```
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class OutputGenerator {

    private final StringBuilder code;
    private final Map<String, Type> variables;

    public OutputGenerator(StringBuilder code, Map<String, Type> variables) {
        this.code = code;
        this.variables = variables;
    }

    private StringBuilder generateDeclarationCode(Map<String, Type> variables) {
        StringBuilder declaration = new StringBuilder();
        declaration.append(".data\n");
        for (Map.Entry<String, Type> var : variables.entrySet()) {
            declaration.append("\t" + var.getKey() + " ");
            switch (var.getValue()) {
                case Bool:
                    declaration.append("DB");
                    break;
                case Int:
                case PointerBool:
                case PointerInt:
                case PointerFloat:
                    declaration.append("DD");
                    break;
                case Float:
                    declaration.append("DQ");
                    break;
            }
            declaration.append(" ?\n");
        }

        return declaration;
    }

    public String generateCode() {
        StringBuilder result = new StringBuilder();

        result.append(".586\n");
        result.append("\n");
        result.append(generateDeclarationCode(variables));
```

```
        result.append("\n");
        result.append(".code\n");
        result.append("main proc\n");
        result.append("\tpush EBP\n");
        result.append("\tmov EBP, ESP\n");

        result.append(code);

        result.append("\tmov ESP, EBP\n");
        result.append("\tpop EBP\n");
        result.append("\tret\n");
        result.append("main endp\n");

        return result.toString();
    }
}
```