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 С на мову асемблеру x86 з підмножиною команд мови, що включає арифметичні та логічні дії, дужкові форми, оголошення використання функцій, 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
_____ Бояршин Ігор Іванович
(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 С на мову асемблер x86.

Функціональні вимоги до програми: компілятор має коректно розпізнавати та перевіряти на коректність вхідний текст, написаний на мові С, у разі некоректності виводити повідомлення про помилки. Компілятор має генерувати вихідний код на асемблері у разі коректності вхідного тексту. Потрібно реалізувати підтримку типів даних int, float та bool, оператори розгалуження, описання та використання функцій, перевірку коректності дужкових форм, арифметичні та логічні оператори.

Вимоги до форматів вхідних і вихідних даних програми: вхідні дані подаються в якості тексту, написаного на мові С. Вихідні дані — інформативні повідомлення про наявні помилки у вхідному коді та вихідний код на асемблері, якщо помилок немає.

Вимоги до програмних та апаратних інструментів: сирцевий код необхідно компілювати з використанням компілятора мови С++, що підтримує стандарт 2014 року.

Завдання на роботу видав
Керівник роботи

_____ Павлов Валерій Георгійович
(підпис)
“___”_____ 20__ р.

Завдання на роботу прийняв
Студент групи ІО-52

_____ Бояршин Ігор Іванович
(підпис студента)
”___” _____ 20__ р.

Зміст

Вступ 4

Вибір інструментів розв’язання задачі 5

Опис роботи програми 6

Інструкція для користувача 8

Тестування програми 10

Список використаної літератури 15

Додаток А. Блок-схема алгоритму компілятора 16

Додаток Б. Вихідний код програмного продукту(компілятора) 17

Вступ

Компілятор – програма, що перетворює вихідний код, написаний однією мовою, на семантично еквівалентний їй код, написаний іншою мовою[1].

У наш час цінність компілятора важко переоцінити, беручи до уваги кількість програмних продуктів, що випускаються кожного року та їх усе зростаючу комплексність. Раніше програмістам доводилося писати код напряму в машинних кодах, що призводило до непомірно громіздких лістингів, експоненціально зростаючій складності підтримки та знаходження помилки в коді зі збільшенням розміру програми, та й просто дуже великого часу на створення програм. Тому поступово перейшли до використання програм-компіляторів, що дозволили писати код на (як правило) більше високорівневій мові програмування, а на виході отримувати код на потрібній мові (наприклад, на низькорівневій). Це дозволило перемістити фокус з написання коду на безпосередньо створення програмного продукту.

У даній роботі був вибраний компілятор, що переводить вхідний код на мові програмування C у код, написанний на асемблері x86. Вибір такої пари мов можна пояснити наступним чином.

Мова програмування C є найбільш близькою до “заліза” серед мов високого рівня, що дозволяє створювати дуже швидкі та оптимізовані програми. В той же час вона вже абстрагується від низькорівневих операторів асемблера і ставить акцент на зручному для людини способу представлення коду. Тому мова C є найкращим компромісом між ефективним кодом та легкістю написання.

Мова асемблера є майже повним еквівалентом машинних кодів. Саме до неї зводиться створення програм. Вона була вибрана тому, що є найбільш розповсюдженою, її можна запустити на майже будь-якому сучасному комп’ютері під управлінням популярних операційних систем.

Вибір інструментів розв’язання задачі

Щодо мови написання самого компілятора, то тут вибір пав на мову C++. Це пояснюється, знову ж таки, тим, що ця мова дозволяє писати найбільш оптимізований та швидкий варіант програми. При написанні програмного продукту постійно виникає потреба перекомпілювати програму та подивитися на результати змін, тому дуже важливо, щоб цей процес віднімав якумога менше часу.

Мова C++ існує вже давно, і не дивлячись на це, актуальна і сьогодні. Її стандарт оновлюється кожні три роки (наприклад, нещодавно вийшов стандарт 17-го року). До мови додаються нові сучасні конструкції, що полегшують та пришвидшують процес створення програм, дозволяючи будувати як завгодно складну архітектуру проекту.

Серед інструментів, що були використані при створенні компілятора, необхідно особливо відмітити наступні компоненти стандартної бібліотеки:

- `std::vector` – тип даних, що дозволяє з легкістю зберігати та використовувати динамічні масиви. Він бере на себе слідкування за правильною роботою з пам’яттю, а також автоматично змінює свій внутрішній розмір, якщо необхідно зберігти більшу кількість елементів. Все це відбувається без втручання користувача, що робить процес надзвичайно легким та зрозумілим [3].
- `std::map` – тип даних, що ставить у відповідність елементу з однієї множини елемент у ішній множині. Тобто за ключем отримати значення. Він широко використовується, коли необхідно зберігти елемент не просто за індексом, а за більш складним ключем, як-от строка чи окрема структура даних [4].

Також у програмі широко використовується модифікатор `const` як для звичайних типів данних, так і для посилань та вказівників, а також для методів класів.

Опис роботи програми

Вся програма (компялітор) складається з наступних частин[5]:

1. *Лексичний аналізатор.* На вхід лексичного аналізатора надається вхідний текст програми, написаної на мові С. Задачею лексичного аналізатора є сканування цього коду та виявлення неналежності окремих його елементів до одного з наперед заданих типів лексем. У разі виявлення неналежності подальша коректна робота компілятора неможлива, тому на цьому компіляція зупиняється і користувачу виводиться вичерпна інформація щодо помилки, аби він мав можливість її виправити. Якщо ж вхідний текст був успішно розпізнаний, то на вихід лексичний аналізатор видає масив токенів, який буде використовуватися на наступних етапах роботи компілятора;
2. *Синтаксичний аналізатор*. На вхід синтаксичного аналізатора поступає масив токенів, що був згенерований на попередньому етапі за допомогою лексичного аналізатора. Задачею синтаксичного аналізатора є парсинг вхідного масиву токенів та створення абстрактного дерева розбору. На цьому етапі перевіряється відповідність вхідної програми наперед заданим правилам граматики вхідної мови. У разі невідповідності правилам граматики користувачу виводиться вичерпна інформація щодо типу помилки та ймовірного місця, де вона була допущена, і подальша робота компілятора переривається. Якщо ж помилок виявлено не було, то успішно будується дерево розбору, що несе в собі інформацію щодо підлеглості конструкцій та відповідає правилам граматики мови. Також синтаксичний аналізатор створює та зберігає таблицю використаних ідентифікаторів з вказанням їхнього типу. Це необхідно для перевірки відповідності типів. Ця таблиця може бути додатково передана на наступний етап роботи компялітора;
3. *Семантичний аналізатор.* До задач семантичного аналізатора входить перевірка коректності семантики вхідної програми. Зокрема, на цьому етапі відбувається приведення типів. Цей етап роботи компілятора був розподілений між синтаксичним аналізатором та генератором коду, і тому явно не присутній;

4. *Генератор коду.* Генератор коду є останньою ланкою роботи компілятора. На вхід йому надається абстрактне дерево розбору, створене на попередньому етапі, а також опціонально таблиця використаних ідентифікаторів з указанням їх типів. Задачею генератора коду є безпосереднє створення вихідного тексту програми на асемблері, що робиться на основі абстрактного дерева розбору. Цей етап у найпростішій своїй реалізації виконується майже лінійно, тобто неперервно зверху вниз, від першого до останнього елемента дерева розбору. Кожний блок коду генерується в залежності від типу вузла дерева та конкретній інформації, що там зберігається. У разі створення більш “розумного” компілятора генератор коду також намагається створити найбільш оптимізований під цільову платформу код, і тоді вже з’являється нелінійність проходу дерева і роботи алгоритму.

Окремо потрібно відмітити особливості роботи синтаксичного аналізатора, адже найчастіше саме він є найскладнішим етапом у роботі компілятора.

У даній роботі була використана модифікація алгоритму синтаксичного аналізатора, що має назву “shift-reduce”[1]. Її зміст наступний: аналізатор зчитує наступний токен з вхідного масиву. Далі він намагається “згорнути” якусь частину вже наявного набору елементів, починаючи з останніх. Якщо аналізатор знаходить відповідне правило з граматики, то відбувається “згортка”, в результаті якої остання частина поточного набору елементів замінюється її відповідністю з правила. Далі, в незалежності від того, чи відбулася згортка, продовжується зчитування наступного елемента з вхідного масива токенів.

З приводу генератора коду, то він “йде” по дереву розбору, що було отримано на вході, і в залежності від типу конструкції генерує наперед заданий шаблоном блок коду на асемблері, підставляючи необхідні дані, як-от назви змінних чи значення констант.

Наведемо опис алгоритму роботи компілятора на блок схемі, вона знаходиться в Додатку А.

Програмний код компілятора знаходиться в Додатку Б.

Інструкція для користувача

Використання програмного продукту зводиться до виклику відповідних складових компілятора та створення ланцюжка, в якому вихідні дані попереднього етапу подаються на вхід наступного:

```
const std::string input = "int a; int main(){ if (3 > 2) a = 2; }";
LexicalAnalyzer la;
SyntaxAnalyzer sa{la.analyze(input)};
const bool inputCorrect = sa.parse();
CodeGenerator cg(sa.getRootNode());
const auto code = cg.generate();
std::cout << code;
```

Розглянемо детальніше кожну строчку:

- Створюється змінна типу `std::string`, що зберігає вхідний код. Це робиться виключно для зручності, щоб потім передати це на вхід лексичного аналізатора. На цьому етапі можна легко організувати зчитування вхідного коду з файлу, головна ідея підходу в тому, що подальші кроки ніяк від цього не зміняться;
- Створюється змінна типу `LexicalAnalyzer`. Він буде виконувати створення масиву лексем;
- Створюється змінна типу `SyntaxAnalyzer`, в конструктор якій передається результат роботи методу `analyze()` лексичного аналізатора, якому на вхід дали вхідний код.
- Створюється змінна типу `bool`, що ініціалізується результатом роботи синтаксичного аналізатора. Цю змінну можна буде використовувати аби зрозуміти, чи успішно відбулася побудова абстрактного дерева розбору.
- Створюється змінна типу `CodeGenerator`, до конструктора якої передається кореневий елемент абстрактного дерева розбору (з якого можна дістатися до будь-якого елемента дерева).

- Створюється змінна, що ініціалізується вихідним кодом, сформованим в результаті роботи генератора коду. Тобто в цій змінній зберігається сам вихідний код.
- У консоль виводиться вихідний код, що є результатом роботи компілятора. Потрібно звернути увагу, що можна було б також легко реалізувати запис у файл вихідного коду, при цьому всі інші частини при цьому ніяк не зазнали б змін. Це свідчить про якість розробленої архітектури програми.

По ходу роботи окремих складових компілятора до консолі виводяться повідомлення про завершення та початок кожного з етапів компіляції, а також детальні повідомлення про помилки, які були виявлені у вхідному коді.

Тестування програми

1. Перевірка операторів

```
---------------- Program start ----------------¬
:> Input: ¬
int a; int main(){ a = 3 + (5 * 8); }¬
---------------- Lexical Analyzer ----------------¬
---------------- Syntax Analyzer ----------------¬
---------------- Code Generator ----------------¬
:> Output code: ¬
.386¬
.model flat¬
.stack 4096¬
¬
.data¬
    a dd 0¬
¬
.code¬
¬
main proc¬
    push ebp¬
    mov ebp, esp¬
¬
    mov eax, 8¬
    mov ebx, eax¬
    mov eax, 5¬
    mul ebx¬
    mov eax, edx¬
    mov ebx, eax¬
    mov eax, 3¬
    add eax, ebx¬
¬
    mov a, eax¬
¬
    mov esp, ebp¬
    pop ebp¬
    ret 0¬
main endp¬
```

```
---------------- Program start ----------------¬
:> Input: ¬
int a; int b; int main(){ a = 3 + * (5 * 8); }¬
---------------- Lexical Analyzer ----------------¬
---------------- Syntax Analyzer ----------------¬
:> [Parse Error]: After DECLARATION_LIST expected one of the following:['DECLARATION' ], but found
'TYPED_ID'¬
:> [Parse Error]: After ) expected one of the following:['BLOCK' 'STATEMENT' ], but found '{'¬
:> [Parse Error]: After { expected one of the following:['}' 'STATEMENT' 'STATEMENT_LIST' ], but fo
und 'ID_INT'¬
:> [Parse Error]: After + expected one of the following:['EXPRESSION_INT' 'EXPRESSION_FLOAT' ], but
 found '*'¬
:> [Parse Error]: Unexpected input end.¬
:> [Syntax Analyzer Error]: Invalid input. Terminating.¬
```

2. Перевірка дужкових форм

```
---------------- Program start ----------------¬
:> Input: ¬
int a; int main}({ a = 3 + (5 * 8); }¬
---------------- Lexical Analyzer ----------------¬
---------------- Syntax Analyzer ----------------¬
:> [Parse Error]: After ( expected one of the following:[')' 'TYPED_ID' 'TYPED_ID_LIST' 'EXPRESSION
_BOOL' 'EXPRESSION_INT' 'EXPRESSION_FLOAT' 'EXPRESSION_LIST' ], but found 'STATEMENT'¬
:> [Parse Error]: Unexpected input end.¬
:> [Syntax Analyzer Error]: Invalid input. Terminating.¬
```

```
---------------- Program start ----------------¬
:> Input: ¬
int a; int main(){ a = 3 + ((5 * 8); }¬
---------------- Lexical Analyzer ----------------¬
---------------- Syntax Analyzer ----------------¬
:> [Parse Error]: Unmatched token '('.¬
:> [Parse Error]: After + expected one of the following:['EXPRESSION_INT' 'EXPRESSION_FLOAT' ], but
 found '('¬
:> [Parse Error]: Unexpected input end.¬
:> [Syntax Analyzer Error]: Invalid input. Terminating.¬
```

3. Перевірка типів даних

```
---------------- Program start ----------------¬
:> Input: ¬
int a; float b; int main(){ a = 3 + (5 * 8); b = 1.0 / 2.5 * 3.3; }¬
---------------- Lexical Analyzer ----------------¬
---------------- Syntax Analyzer ----------------¬
---------------- Code Generator ----------------¬
:> Output code: ¬
.386¬
.model flat¬
.stack 4096¬
¬
.data¬
    a dd 0¬
    b dq 0.0¬
¬
.code¬
¬
main proc¬
    push ebp¬
    mov ebp, esp¬
¬
    mov eax, 8¬
    mov ebx, eax¬
    mov eax, 5¬
    mul ebx¬
    mov eax, edx¬
    mov ebx, eax¬
    mov eax, 3¬
    add eax, ebx¬
¬
    mov a, eax¬
¬
    mov esp, ebp¬
    pop ebp¬
    ret 0¬
main endp¬
```

4. Перевірка функцій

```
---------------- Program start ----------------¬
:> Input: ¬
int a; int f(float b){b = 5.5;} int main(){ a = f(3.3); }¬
---------------- Lexical Analyzer ----------------¬
---------------- Syntax Analyzer ----------------¬
---------------- Code Generator ----------------¬
:> Output code: ¬
.386¬
.model flat¬
.stack 4096¬
¬
.data¬
    a dd 0¬
    b dq 0.0¬
¬
.code¬
¬
f proc¬
    push ebp¬
    mov ebp, esp¬
¬
    mov eax, dword ptr[ebp + 8]¬
    mov b, eax¬
¬
    mov esp, ebp¬
    pop ebp¬
    ret 4¬
f endp¬
¬
main proc¬
    push ebp¬
    mov ebp, esp¬
¬
    mov eax, 3.3¬
    push eax¬
    call f¬
¬
    mov a, eax¬
¬
    mov esp, ebp¬
    pop ebp¬
    ret 0¬
main endp¬
```

```
---------------- Program start ----------------¬
:> Input: ¬
int a; int f(float b){b = 5.5;} int main(){ a = f(5.5 > 1.0); }¬
---------------- Lexical Analyzer ----------------¬
---------------- Syntax Analyzer ----------------¬
---------------- Code Generator ----------------¬
:> [Parse Error]: Specified function arguments do not match with those specified at function declar
ation(for function f).¬
:> [Parse Error]: Expected:Found: FLOAT:BOOL¬
:> [Code Generator Error]: Invalid input. Terminating.¬
```

5. Перевірка розгалуження

```
---------------- Program start ----------------¬
:> Input: ¬
int a; int main(){ if(4 < 5) {a = 5;} else a = 6;}¬
---------------- Lexical Analyzer ----------------¬
---------------- Syntax Analyzer ----------------¬
---------------- Code Generator ----------------¬
:> Output code: ¬
.386¬
.model flat¬
.stack 4096¬
¬
.data¬
    a dd 0¬
¬
.code¬
¬
main proc¬
    push ebp¬
    mov ebp, esp¬
    mov eax, 5¬
    mov ebx, eax¬
    mov eax, 4¬
    cmp eax, ebx¬
    jl loop0¬
    mov eax, 0¬
    jmp loop1¬
loop0:¬
    mov eax, 1¬
loop1:¬
    cmp eax, 1¬
    jne loop2¬
    mov eax, 5¬
    mov a, eax¬
    jmp loop3¬
loop2:¬
    mov eax, 6¬
    mov a, eax¬
loop3:¬
    mov esp, ebp¬
    pop ebp¬
    ret 0¬
main endp¬
```

```
---------------- Program start ----------------¬
:> Input: ¬
int a; int main(){ if(true) else a = 6;}¬
---------------- Lexical Analyzer ----------------¬
---------------- Syntax Analyzer ----------------¬
:> [Parse Error]: After DECLARATION expected one of the following:['DECLARATION' ], but found 'TYPE
D_ID'¬
:> [Parse Error]: After ) expected one of the following:['BLOCK' 'STATEMENT' ], but found '{'¬
:> [Parse Error]: After { expected one of the following:['}' 'STATEMENT' 'STATEMENT_LIST' ], but fo
und 'IF'¬
:> [Parse Error]: After ) expected one of the following:['BLOCK' 'STATEMENT' ], but found 'ELSE'¬
:> [Parse Error]: Unexpected input end.¬
:> [Syntax Analyzer Error]: Invalid input. Terminating.¬
```

Висновки

На даний момент компілятор відмінно виконує поставлене перед ним завдання в рамках тієї підмножини команд, для якої від був заданий. Була розроблена дуже гнучка та якісна архітектура, що дозволяє робити як завгодно комплексні структури з простих блоків, а також забезпечує стабільну надійність при рості програми.

Подальші поліпшення компілятора можуть виконатися стосовно розширення множини конструкцій, які підтримуються.

Також не останню роль у якості програмного продукту (компілятора) відіграє якість написання коду логіки, адже допущені помилки чи неточності призводять до небажаних результатів роботи. Підтримувати належний стан коду допіг модифікатор мови `const`, а також складові бібліотеки `std::vector` та `std::map`, що значно полегшили створення необхідних структур та взяли на себе шматок роботи, пов’язаний з прямою роботою з пам’яттю.

До недоліків поточного компілятора можна віднести те, що вихідний код на асемлері не є оптимізованим. Тобто він є семантично еквівалентним вхідному кодові на мові C, проте його швидкодія могла бути б кращою, якщо розробити етап генерації коду та додати логіку створення оптимізації.

Список використаної літератури

1. Morgan Kauffmann. Advanced Compiler Design and Implementation, 1st edition – Oxford; New York: Oxford University Press, 1997. – 495 p.
2. Compiler [Електронний ресурс] – https://en.wikipedia.org/wiki/Compiler
3. C++ Reference: vector [Електронний ресурс] – http://www.cplusplus.com/reference/vector/vector/
4. C++ Reference: map [Електронний ресурс] – http://www.cplusplus.com/reference/map/map/
5. Tutorials point: Compiler Design [Електронний ресурс] – https://www.tutorialspoint.com/compiler_design/
6. Конспект лекцій з дисципліни «Системне програмування – 2» за 2017н.р.

Додаток А. Блок-схема алгоритму компілятора

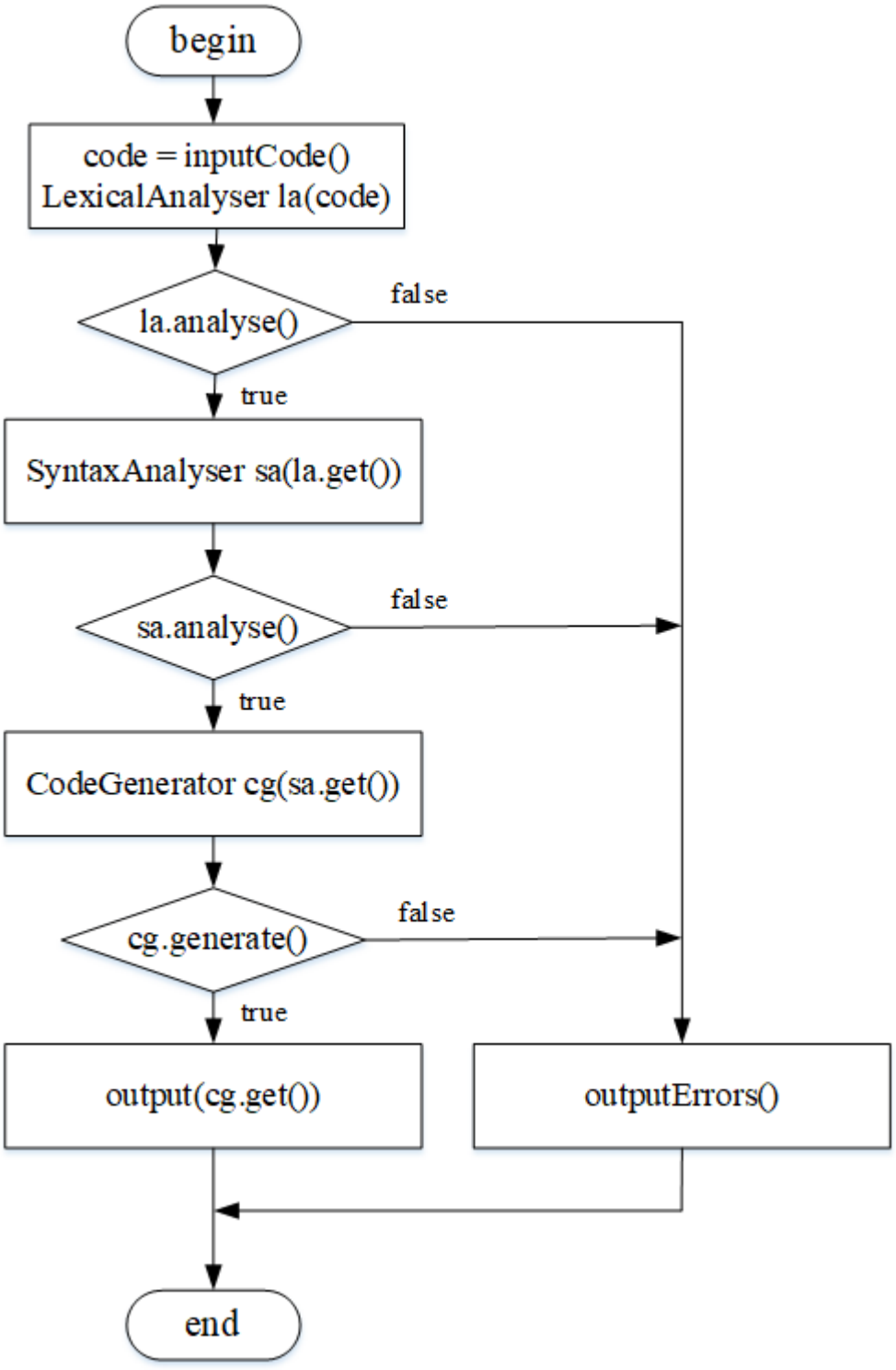

Рис. 1 – Блок-схема компілятору

Додаток Б. Вихідний код програмного продукту(компілятора)

TokenType.h

```
#ifndef TOKENTYPE_H
#define TOKENTYPE_H

#include <iostream>
#include <string>

enum TokenType {
    TokenType_Identifier,
    TokenType_LiteralBool,
    TokenType_LiteralInt,
    TokenType_LiteralFloat,
    TokenType_Keyword_If,
    TokenType_Keyword_Else,
    TokenType_Keyword_Int,
    TokenType_Keyword_Float,
    TokenType_Keyword_Bool,
    TokenType_Keyword_Void,
    TokenType_Operator,
    TokenType_Operator_Assign,
    TokenType_Brackets_Open, // [
    TokenType_Brackets_Close, // ]
    TokenType_Parentheses_Open, // (
    TokenType_Parentheses_Close, // )
    TokenType_Braces_Open, // {
    TokenType_Braces_Close, // }
    TokenType_Semicolon,
    TokenType_Comma,
    TokenType_Unknown
};


TokenType getTokenTypeFromString(const std::string& str);
std::string toString(TokenType tokenType);

std::ostream& operator<<(std::ostream& os, const TokenType& tokenType);
TokenType getIfKeyword(const std::string& lexeme);

#endif
```

Token.h

```
#ifndef TOKEN_H
#define TOKEN_H

#include <string>
#include "TokenType.h"

class Token {
    public:
        Token(const std::string& value);
        Token(const TokenType& type, const std::string& value);

        Token& operator=(const Token& token);

        friend std::ostream& operator<<(std::ostream& os, const Token& token);
```

```
        TokenType getType() const;
        std::string getValue() const;

    private:
        TokenType m_Type;
        std::string m_Value;
};


std::ostream& operator<<(std::ostream& os, const Token& token);

#endif
```

Token.cpp

```
#include "Token.h"


Token::Token(const std::string& value) :
    m_Type(TokenType_Unknown), m_Value(value) {}


Token::Token(const TokenType& type, const std::string& value) :
    m_Type(type), m_Value(value) {}


TokenType Token::getType() const {
    return m_Type;
}

std::string Token::getValue() const {
    return m_Value;
}

Token& Token::operator=(const Token& token) {
    this->m_Type = token.m_Type;
    this->m_Value = token.m_Value;

    return *this;
}


std::ostream& operator<<(std::ostream& os, const Token& token) {
    os << token.m_Value << " :: " << token.m_Type;
    return os;
}
```

NodeType.h

```
#ifndef NODETYPE_H
#define NODETYPE_H

#include <string>
#include <iostream>

enum NodeType {
    // Terminal symbols. Runtime-defined
    NodeType_Id,
    NodeType_IdUndefined,
    NodeType_IdVoid,
    NodeType_IdBool,
```

```
    NodeType_IdInt,
    NodeType_IdFloat,
    NodeType_LiteralBool,
    NodeType_LiteralInt,
    NodeType_LiteralFloat,


    // Terminal symbols. Compiletime-defined
    NodeType_KeywordIf,
    NodeType_KeywordElse,
    NodeType_KeywordVoid,
    NodeType_KeywordBool,
    NodeType_KeywordInt,
    NodeType_KeywordFloat,
    NodeType_BracketsOpen, // [
    NodeType_BracketsClose, // ]
    NodeType_ParenthesesOpen, // (
    NodeType_ParenthesesClose, // )
    NodeType_BracesOpen, // {
    NodeType_BracesClose, // }
    NodeType_OperatorPlus,
    NodeType_OperatorMinus,
    NodeType_OperatorMultiply,
    NodeType_OperatorDivide,
    NodeType_OperatorEquals,
    NodeType_OperatorNotEquals,
    NodeType_OperatorLess,
    NodeType_OperatorLessOrEquals,
    NodeType_OperatorGreater,
    NodeType_OperatorGreaterOrEquals,
    NodeType_OperatorAssign,
    NodeType_Comma,
    NodeType_Semicolon,


    // Non-terminal symbols. Compiletime-defined
    NodeType_Block,
    NodeType_TypedIdList,
    NodeType_TypedId,
    NodeType_DeclarationList,
    NodeType_Declaration,
    NodeType_StatementList,
    NodeType_Statement,
    NodeType_ExpressionList,
    NodeType_ExpressionBool,
    NodeType_ExpressionInt,
    NodeType_ExpressionFloat,
    NodeType_FunctionCallVoid,
    NodeType_FunctionCallBool,
    NodeType_FunctionCallInt,
    NodeType_FunctionCallFloat,
};


NodeType getNodeTypeFromString(const std::string& str);
std::string toString(NodeType nodeType);

std::ostream& operator<<(std::ostream& os, const NodeType& nodeType);

#endif
```

Node.h

```
#ifndef NODE_H
#define NODE_H

#include <vector>
#include <string>
#include <ostream>
#include <iostream>
#include "NodeType.h"


class Node {
    public:
        Node(
            const NodeType nodeType
        );

        Node(
            const NodeType nodeType,
            const std::vector<const Node*>& children
        );

        Node(
            const NodeType nodeType,
            const std::string& nodeValue
        );

    public:
        NodeType m_NodeType;
        std::string m_NodeValue;
        std::vector<const Node*> m_Children;

        friend std::ostream& operator<<(std::ostream& os, const Node& node);
};

std::ostream& operator<<(std::ostream& os, const Node& node);

#endif
```

LexicalAnalyzer.cpp

```
#include "LexicalAnalyzer.h"


LexicalAnalyzer::LexicalAnalyzer() : m_TokensTable(std::vector<Token>()) {}


std::vector<Token> LexicalAnalyzer::analyze(const std::string& input) {
    const unsigned int size = input.size();
    if (size == 0) {
        return m_TokensTable;
    }

    unsigned int currentIndex = 0;
    std::string accumulator("");
    bool accumulatingNumber;

    while (currentIndex < size) {
        const char c = input[currentIndex];
        std::string cc;
        TokenType tokenType;
```

```
bool separatorFound = false;

switch(c) {
    case '(':
        tokenType = TokenType_Parentheses_Open;
        separatorFound = true;
        break;
    case ')':
        tokenType = TokenType_Parentheses_Close;
        separatorFound = true;
        break;

    case '{':
        tokenType = TokenType_Braces_Open;
        separatorFound = true;
        break;
    case '}':
        tokenType = TokenType_Braces_Close;
        separatorFound = true;
        break;

    case '[':
        tokenType = TokenType_Brackets_Open;
        separatorFound = true;
        break;
    case ']':
        tokenType = TokenType_Brackets_Close;
        separatorFound = true;
        break;

    // TODO: implement unary !
    case '!':
        if (currentIndex + 1 < size - 1 && input[currentIndex + 1] == '=') {
            currentIndex++;
            tokenType = TokenType_Operator;
            cc = "!=";
        } else {
            tokenType = TokenType_Unknown;
        }
        separatorFound = true;
        break;

    case '=':
        if (currentIndex + 1 < size - 1 && input[currentIndex + 1] == '=') {
            currentIndex++;
            tokenType = TokenType_Operator;
            cc = "==";
        } else {
            tokenType = TokenType_Operator_Assign;
        }
        separatorFound = true;
        break;

    case '<':
        if (currentIndex + 1 < size - 1 && input[currentIndex + 1] == '=') {
            currentIndex++;
            cc = "<=";
        }
        tokenType = TokenType_Operator;
        separatorFound = true;
        break;
    case '>':
        if (currentIndex + 1 < size - 1 && input[currentIndex + 1] == '=') {
            currentIndex++;
            cc = ">=";
```

```
                }
                tokenType = TokenType_Operator;
                separatorFound = true;
                break;

            case '+':
            case '-':
            case '*':
            case '/':
                tokenType = TokenType_Operator;
                separatorFound = true;
                break;

            case ';':
                tokenType = TokenType_Semicolon;
                separatorFound = true;
                break;

            case ',':
                tokenType = TokenType_Comma;
                separatorFound = true;
                break;

            case ' ':
                separatorFound = true;
                break;

            default:
                separatorFound = false;
                if (isAlpha(c)) {
                    if (accumulator.size() == 0) {
                        accumulatingNumber = false;
                    } else if (accumulatingNumber) {
                        // something like "35var" => invalid
                        std::cout << ":> [Lexical Analyzer Error]: Bad format(at char " <<
currentIndex
                                  << "): " << accumulator << std::endl;
                        return m_TokensTable;
                    }

                    accumulator += c;
                } else if (isNumber(c) || c == '.') {
                    if (accumulator.size() == 0) {
                        accumulatingNumber = true;
                    }

                    accumulator += c;
                } else {
                    std::cout << ":> [Lexical Analyzer Error]: Bad format: Unknown char at "
                              << currentIndex << ": " << c << std::endl;
                    return m_TokensTable;
                }
        }

        if (separatorFound) {
            const bool accumulating = accumulator.size() > 0;
            if (accumulating) {
                // flush the accumulator
                if (accumulatingNumber) {
                    if (accumulator.find('.') != std::string::npos) {
                        // TODO: prone to be incorrect
                        m_TokensTable.emplace_back(TokenType_LiteralFloat, accumulator);
                    } else {
                        m_TokensTable.emplace_back(TokenType_LiteralInt, accumulator);
                    }
```

```
                } else {
                    if (accumulator.compare("true") == 0 ||
                            accumulator.compare("false") == 0) {
                        m_TokensTable.emplace_back(TokenType_LiteralBool, accumulator);
                    } else {
                        const TokenType keyword = getIfKeyword(accumulator);
                        m_TokensTable.emplace_back(
                                keyword != TokenType_Unknown ? keyword : TokenType_Identifier,
                                accumulator);
                    }
                }

                accumulator.clear();
            }

            // Regardless of whether we were accumulating or not,
            // It is time to insert the signle token
            // (unless it is a space, in this case just skip it)
            if (c != ' ') {
                if (cc.size() > 0) {
                    m_TokensTable.push_back(Token(tokenType, cc));
                } else {
                    m_TokensTable.push_back(Token(tokenType, std::string(1, c)));
                }
                // m_TokensTable.emplace_back(tokenType, "" + c);
            }

        }
        // If separatorFound == false => we either don't care or it's impossible

        currentIndex++;
    }

    return m_TokensTable;
}

std::ostream& operator<<(std::ostream& os, const LexicalAnalyzer& lexicalAnalyzer) {
    os << ":> Tokens table:" << std::endl;

    for (const Token& token : lexicalAnalyzer.m_TokensTable) {
        os << token << std::endl;
    }

    return os;
}


bool isAlpha(char c) {
    return ((c >= 'a' && c <= 'z') || (c >= 'A' && c <= 'Z'));
}

bool isNumber(char c) {
    return (c >= '0' && c <= '9');
}
```

SyntaxAnalyzer.cpp

```
#include "SyntaxAnalyzer.h"


SyntaxAnalyzer::SyntaxAnalyzer(
        const std::vector<Token> tokensTable) :
    m_ConvertedTokensTable(std::vector<Node>()),
    m_NextTokensTableIndex(0),
```

```
    m_IdsTable(std::map<std::string, std::vector<NodeType>>()),
    m_ReductionPatterns(readReductionPatterns("rules.txt")),
    m_AllowedSequences(readAllowedSequences("sequences.txt"))
{
    // Convert all tokens into their Node equivalents
    for (Token token : tokensTable) {
        m_ConvertedTokensTable.push_back(convertIntoNodeType(token));
        // std::cout << token << std::endl;
    }


    // const auto rp = readReductionPatterns("rules.txt");
    // for (const auto r : rp) {
    //     std::cout << std::get<1>(r) << std::endl;
    //     for (const auto t : std::get<0>(r)) {
    //         std::cout << " " << t;
    //     }
    //     std::cout << std::endl;
    // }


    // const auto rp = readAllowedSequences("sequences.txt");
    // for (const auto& x : rp) {
    //     std::cout << x.first << std::endl;
    //     for (const auto t : x.second) {
    //         std::cout << " " << t;
    //     }
    //     std::cout << std::endl;
    // }
}


const Node* SyntaxAnalyzer::getNext() {
    const Node* node = peekNext();
    if (node) {
        m_NextTokensTableIndex++;
    }

    return node;
}

const Node* SyntaxAnalyzer::peekNext() const {
    if (m_NextTokensTableIndex == m_ConvertedTokensTable.size()) {
        return nullptr;
    }

    return &m_ConvertedTokensTable[m_NextTokensTableIndex];
}


std::vector<ReductionPattern> SyntaxAnalyzer::getPotentialReductionPatterns() const {
    std::vector<ReductionPattern> potentialReductionPatterns;
    for (ReductionPattern reductionPattern : m_ReductionPatterns) {
        const std::vector<NodeType> sequence = std::get<0>(reductionPattern);
        const unsigned int sequenceLength = sequence.size();
        if (m_Nodes.size() < sequenceLength) { // potential pattern is too long
            continue;
        }

        bool mismatchFound = false;
        unsigned int reductionPatternIndex = 0;
        for (unsigned int i = m_Nodes.size() - sequenceLength;
                i < m_Nodes.size(); i++, reductionPatternIndex++) {
            const NodeType nodeTypeInReductionPattern =
                sequence.at(reductionPatternIndex);
```

```
            if (m_Nodes.at(i)->m_NodeType != nodeTypeInReductionPattern) {
                mismatchFound = true;
                break;
            }
        }

        if (!mismatchFound) {
            potentialReductionPatterns.push_back(reductionPattern);
        }
    }

    return potentialReductionPatterns;
}

bool SyntaxAnalyzer::reduceIfCan() {
    const std::vector<ReductionPattern> potentialReductionPatterns =
getPotentialReductionPatterns();
    if (potentialReductionPatterns.size() == 0) {
        // Can't reduce => nothing to do here
        return false;
    }

    const unsigned int currentNodeIndex = m_Nodes.size() - 1;
    const Node* const currentNode = m_Nodes[currentNodeIndex];
    const NodeType currentNodeType = currentNode->m_NodeType;
    const Node* const nextNode = peekNext();
    const ReductionPattern firstReductionPattern = potentialReductionPatterns.at(0);

    // std::cout << "Could reduce into: ";
    // for (ReductionPattern rp : potentialReductionPatterns) {
    //     std::cout << std::get<1>(rp) << " or ";
    // }
    // std::cout << std::endl;


    // Encountered an ID => put into the IDs table if this is a declaration,
    // otherwise convert into the appropriate type using the IDs table. If there
    // is no entry for this ID in the table => Error.
    if (currentNodeType == NodeType_Id) {
        // std::cout << "000000:" << currentNode->m_NodeValue << std::endl;
        const std::string& idValue = currentNode->m_NodeValue;
        const bool knownId =
            m_IdsTable.find(idValue) != m_IdsTable.end();
        const bool idDeclaration =
            std::get<1>(firstReductionPattern) == NodeType_TypedId;

        if (knownId && idDeclaration) {
            std::cerr << ":> [Parse Error]: Redeclaration of Id: '"
                      << idValue << "'"
                      << std::endl;
            return false;
        }

        if (!knownId && !idDeclaration) {
            std::cerr << ":> [Parse Error]: Usage of undeclared Id: '"
                      << idValue << "'"
                      << std::endl;
            return false;
        }

        // The index is valid, becase otherwise we'd have (!knownId && !idDeclaration)
        const NodeType previousNodeType = m_Nodes.at(currentNodeIndex - 1)->m_NodeType;
        if (idDeclaration) {
            m_IdsTable.insert(
                std::pair<std::string, std::vector<NodeType>>(
```

```
                idValue,
                {previousNodeType}
            )
        );

        // Now continue(will be reduced)
    } else { // knownId
        // So, this is a valid usage of an ID. We need to convert it into the
        // appropriate type(ID_*).

        // the 0th type is the return type
        const NodeType idType = m_IdsTable.at(idValue).at(0);
        switch (idType) {
            case NodeType_KeywordVoid:
                m_Nodes[currentNodeIndex] = new Node(
                    NodeType_IdVoid,
                    idValue
                );
                break;
            case NodeType_KeywordBool:
                m_Nodes[currentNodeIndex] = new Node(
                    NodeType_IdBool,
                    idValue
                );
                break;
            case NodeType_KeywordInt:
                m_Nodes[currentNodeIndex] = new Node(
                    NodeType_IdInt,
                    idValue
                );
                break;
            case NodeType_KeywordFloat:
                m_Nodes[currentNodeIndex] = new Node(
                    NodeType_IdFloat,
                    idValue
                );
                break;
            default:
                std::cerr << ":> [Parse Error]: reduceIfCan(): Invalid Type for ID: "
                          << idType
                          << std::endl;
                return false;
        }

        // Current reduction is done
        return true;
    }
}


// Don't allow an Id to become an expression if it is an assignment or a function
if ((currentNodeType == NodeType_IdBool ||
        currentNodeType == NodeType_IdInt ||
        currentNodeType == NodeType_IdFloat)
        && nextNode) {
    const NodeType nextNodeType = nextNode->m_NodeType;
    if (nextNodeType == NodeType_ParenthesesOpen
            || nextNodeType == NodeType_OperatorAssign) {
        return false;
    }
}


// If the Node before the beginning of the pattern is IF =>
// => don't allow (EXPR) to fold into an EXPR (because be need () for the if condition).
```

```
    if (std::get<1>(firstReductionPattern) == NodeType_ExpressionBool) {
        const int indexOfPotentialIf =
            currentNodeIndex - std::get<0>(firstReductionPattern).size();
        if (indexOfPotentialIf >= 0) {
            if (m_Nodes.at(indexOfPotentialIf)->m_NodeType == NodeType_KeywordIf) {
                return false;
            }
        }
    }


    // If it wants to reduce into a bool EXPR => don't allow if
    // there is an arithmetic operator coming up next.
    if (std::get<1>(firstReductionPattern) == NodeType_ExpressionBool) {
        const unsigned int indexOfPotentialOperator = m_NextTokensTableIndex + 0;
        if (indexOfPotentialOperator < m_ConvertedTokensTable.size()) {
            const NodeType operatorType =
                m_ConvertedTokensTable.at(indexOfPotentialOperator).m_NodeType;
            if (
                    operatorType == NodeType_OperatorPlus ||
                    operatorType == NodeType_OperatorMinus ||
                    operatorType == NodeType_OperatorMultiply ||
                    operatorType == NodeType_OperatorDivide ||
                    operatorType == NodeType_OperatorLess ||
                    operatorType == NodeType_OperatorLessOrEquals ||
                    operatorType == NodeType_OperatorGreater ||
                    operatorType == NodeType_OperatorGreaterOrEquals
                    ) {
                return false;
            }
        }
    }


    // TODO: implement
    // If it wants to reduce the (+) or (-) operator => don't allow if there
    // is an (*) or (/) operator coming up next
    // if (std::get<0>(firstReductionPattern).size() > 0) {
    //     if (std::get<0>(firstReductionPattern).at(1) == NodeType_OperatorPlus ||
    //             std::get<0>(firstReductionPattern).at(1) == NodeType_OperatorMinus) {
    //         const unsigned int indexOfPotentialMoreSignificantOperator =
m_NextTokensTableIndex;
    //         if (indexOfPotentialMoreSignificantOperator < m_ConvertedTokensTable.size()) {
    //             const NodeType operatorType =
    //
m_ConvertedTokensTable.at(indexOfPotentialMoreSignificantOperator).m_NodeType;
    //             if (
    //                     operatorType == NodeType_OperatorMultiply ||
    //                     operatorType == NodeType_OperatorDivide
    //                     ) {
    //                 return false;
    //             }
    //         }
    //     }
    // }

    // If it wants to reduce onto IF => allow only if there is no
    // following ELSE(otherwise force IF_ELSE). The size of the IF pattern is 5
    if (std::get<0>(firstReductionPattern).at(0) == NodeType_KeywordIf
            && std::get<0>(firstReductionPattern).size() == 5 && nextNode) {
        const NodeType nextNodeType = nextNode->m_NodeType;
        if (nextNodeType == NodeType_KeywordElse) {
            return false;
        }
    }
```

```
    // Alles gut => pick the best(first is best) pattern

    // std::cout << ":> Before reduction: ";
    // for (const Node* node : m_Nodes) {
    //     std::cout << node->m_NodeType << " ";
    // }
    // std::cout << std::endl;
    //

    const unsigned int patternSize = std::get<0>(firstReductionPattern).size();
    const unsigned int startIndex = m_Nodes.size() - patternSize;
    std::vector<const Node*> children;
    for (unsigned int i = startIndex; i < startIndex + patternSize; i++) {
        children.push_back(m_Nodes[i]);
    }

    // Remove the reduced part
    m_Nodes.erase(m_Nodes.begin() + startIndex, m_Nodes.end());

    // Insert a new Node instead of the reduced part
    m_Nodes.push_back(new Node(
        std::get<1>(firstReductionPattern),
        children
    ));

    // std::cout << ":> After reduction: ";
    // for (const Node* node : m_Nodes) {
    //     std::cout << node->m_NodeType << " ";
    // }
    // std::cout << std::endl;
    //

    return true;
}

bool SyntaxAnalyzer::parse() {
    while (peekNext()) {
        const Node* nodeCurrent = getNext();
        m_Nodes.push_back(nodeCurrent);
        bool keepReducing = true;
        while(keepReducing) {
            keepReducing = reduceIfCan();
        }
    }

    // std::cout << ":> Parsed tree:" << std::endl;
    // std::cout << *m_Nodes[0] << std::endl;

    if (m_Nodes.size() > 1) {
        // There is an error for sure, let's try to deduce where it is
        for (unsigned int i = 1; i < m_Nodes.size(); i++) {
            const NodeType previousNodeType = m_Nodes[i - 1]->m_NodeType;
            const NodeType currentNodeType = m_Nodes[i]->m_NodeType;

            if (!isAllowed(previousNodeType, currentNodeType)) {
                continue;
            }
        }

        std::cout << ":> [Parse Error]: Unexpected input end." << std::endl;
        return false;
    }

    const auto topLevelNodeType = m_Nodes[0]->m_NodeType;
```

```
    if (!(topLevelNodeType == NodeType_DeclarationList ||
            topLevelNodeType == NodeType_Declaration)) {
        std::cout << ":> [Parse Error]: Unexpected top-level entry." << std::endl;
        return false;
    }



    return true;
}


bool SyntaxAnalyzer::isAllowed(
        const NodeType nodeType1,
        const NodeType nodeType2) const {

    const auto entryForNodeType1 = m_AllowedSequences.find(nodeType1);
    if (entryForNodeType1 != m_AllowedSequences.end()) {
        const std::vector<NodeType> allowedSequences = m_AllowedSequences.at(nodeType1);
        for (NodeType nodeType : allowedSequences) {
            if (nodeType2 == nodeType) {
                return true;
            }
        }

        std::cout << ":> [Parse Error]: After " << nodeType1 << " expected one of the
following:[";
        for (NodeType nodeType : allowedSequences) {
            std::cout << "'" << nodeType << "' ";
        }
        std::cout << "], but found '" << nodeType2 << "'" << std::endl;
    }

    return true;
}


Node SyntaxAnalyzer::convertIntoNodeType(const Token token) const {
    const TokenType tokenType = token.getType();
    const std::string tokenValue = token.getValue();
    switch(tokenType) {
        case TokenType_Identifier:
            return Node(NodeType_Id, tokenValue);
        case TokenType_LiteralBool:
            return Node(NodeType_LiteralBool, tokenValue);
        case TokenType_LiteralInt:
            return Node(NodeType_LiteralInt, tokenValue);
        case TokenType_LiteralFloat:
            return Node(NodeType_LiteralFloat, tokenValue);
        case TokenType_Keyword_If:
            return Node(NodeType_KeywordIf);
        case TokenType_Keyword_Else:
            return Node(NodeType_KeywordElse);
        case TokenType_Keyword_Void:
            return Node(NodeType_KeywordVoid);
        case TokenType_Keyword_Bool:
            return Node(NodeType_KeywordBool);
        case TokenType_Keyword_Int:
            return Node(NodeType_KeywordInt);
        case TokenType_Keyword_Float:
            return Node(NodeType_KeywordFloat);
        case TokenType_Operator:
            if (tokenValue.compare("+") == 0) {
                return Node(NodeType_OperatorPlus);
            } else if (tokenValue.compare("-") == 0) {
```

```
                return Node(NodeType_OperatorMinus);
            } else if (tokenValue.compare("*") == 0) {
                return Node(NodeType_OperatorMultiply);
            } else if (tokenValue.compare("/") == 0) {
                return Node(NodeType_OperatorDivide);
            } else if (tokenValue.compare("==") == 0) {
                return Node(NodeType_OperatorEquals);
            } else if (tokenValue.compare("!=") == 0) {
                return Node(NodeType_OperatorNotEquals);
            } else if (tokenValue.compare("<") == 0) {
                return Node(NodeType_OperatorLess);
            } else if (tokenValue.compare("<=") == 0) {
                return Node(NodeType_OperatorLessOrEquals);
            } else if (tokenValue.compare(">") == 0) {
                return Node(NodeType_OperatorGreater);
            } else if (tokenValue.compare(">=") == 0) {
                return Node(NodeType_OperatorGreaterOrEquals);
            } else {
                std::cout << ":> Error: Unknown operator in convertIntoNodeType(): "
                    << tokenValue << std::endl;
            }
        case TokenType_Operator_Assign:
            return Node(NodeType_OperatorAssign);
        case TokenType_Brackets_Open:
            return Node(NodeType_BracketsOpen);
        case TokenType_Brackets_Close:
            return Node(NodeType_BracketsClose);
        case TokenType_Parentheses_Open:
            return Node(NodeType_ParenthesesOpen);
        case TokenType_Parentheses_Close:
            return Node(NodeType_ParenthesesClose);
        case TokenType_Braces_Open:
            return Node(NodeType_BracesOpen);
        case TokenType_Braces_Close:
            return Node(NodeType_BracesClose);
        case TokenType_Semicolon:
            return Node(NodeType_Semicolon);
        case TokenType_Comma:
            return Node(NodeType_Comma);
        case TokenType_Unknown:
        default:
            std::cout << ":> Error: Encountered Unknown Token in convertIntoNodeType():"
                << tokenValue;
    }

    return Node(NodeType_Semicolon);
}


template<typename Out>
void split(const std::string &s, char delim, Out result) {
    std::stringstream ss(s);
    std::string item;
    while (std::getline(ss, item, delim)) {
        *(result++) = item;
    }
}

std::vector<std::string> split(const std::string &s, char delim) {
    std::vector<std::string> elems;
    split(s, delim, std::back_inserter(elems));
    return elems;
}
```

```
std::vector<ReductionPattern> readReductionPatterns(const std::string& fileName) {
    std::ifstream in(fileName);
    std::vector<ReductionPattern> reductionPatterns;
    if (!in.is_open()) {
        std::cerr << ":> Unable to open: " << fileName << std::endl;
        return reductionPatterns;
    }

    std::string line;
    bool insideRule = false;
    std::vector<NodeType> from;
    NodeType to;
    while (std::getline(in, line)) {
        if (line.size() == 0) {
            insideRule = false;
            continue;
        }

        // Skip comments
        if (line[0] == '*') {
            continue;
        }

        const std::vector<std::string> words = split(line, ' ');
        if (insideRule) {
            for (const std::string& word : words) {
                if (word.size() == 0) {
                    continue;
                }

                // std::cout << "Read line " << lineCounter << std::endl;
                from.push_back(getNodeTypeFromString(word));
            }

            reductionPatterns.push_back(
                std::make_tuple<std::vector<NodeType>, NodeType>(
                    std::move(from),
                    std::move(to)
                )
            );

            from.clear();
        } else {
            // Prepare to read rules
            insideRule = true;
            to = getNodeTypeFromString(words[0]);
        }
    }

    in.close();

    return reductionPatterns;
}


std::map<NodeType, std::vector<NodeType>> readAllowedSequences(const std::string& fileName) {
    std::ifstream in(fileName);
    std::map<NodeType, std::vector<NodeType>> allowedSequences;
    if (!in.is_open()) {
        std::cerr << ":> Unable to open: " << fileName << std::endl;
        return allowedSequences;
    }

    std::string line;
    bool insideRule = false;
```

```
    std::vector<NodeType> to;
    NodeType from;
    while (std::getline(in, line)) {
        if (line.size() == 0) {
            insideRule = false;
            continue;
        }

        // Skip comments
        if (line[0] == '%') {
            continue;
        }

        const std::vector<std::string> words = split(line, ' ');
        if (insideRule) {
            for (const std::string& word : words) {
                if (word.size() == 0) {
                    continue;
                }

                to.push_back(getNodeTypeFromString(word));
            }

            allowedSequences.insert(
                std::pair<NodeType, std::vector<NodeType>>(
                    std::move(from),
                    std::move(to)
                )
            );

            to.clear();
        } else {
            // Prepare to read rules
            insideRule = true;
            from = getNodeTypeFromString(words[0]);
        }
    }

    in.close();

    return allowedSequences;
}


const Node* SyntaxAnalyzer::getRootNode() const {
    return m_Nodes[0];
}
```

CodeGenerator.cpp

```
#include "CodeGenerator.h"


CodeGenerator::CodeGenerator(const Node* rootNode)
    : m_RootNode(rootNode),
    m_Variables(getAllVariableDeclarations(rootNode)),
    m_Functions(getAllFunctionDeclarations(rootNode)), m_ErrorFound(false),
    m_NextAvailablyLoopIndex(0) {
        // for (const auto& pair : m_Variables) {
        //     std::cout << "Var: " << pair.second << " " << pair.first << std::endl;
        // }
        //

        // for (const auto& function : m_Functions) {
```

```
        //     std::cout << "Func: " << function.second[0].second << " "
        //               << function.second[0].first << ": ";
        //     for (unsigned int i = 1; i < function.second.size(); i++) {
        //         std::cout << " " << function.second[i].second << " "
        //                   << function.second[i].first << ", ";
        //     }
        //     std::cout << std::endl;
        // }
    }


std::string CodeGenerator::generate() {
    // We know that the top-level entry is either DECLARATION or DECLARATION_LIST.
    // Search among them for function declarations and implement their bodies.

    std::stringstream code;
    code << ".386" << std::endl;
    code << ".model flat" << std::endl;
    code << ".stack 4096" << std::endl;
    code << std::endl;

    code << ".data" << std::endl;
    generateVariableDeclarationsCode(code);
    code << std::endl;

    code << ".code" << std::endl;
    code << std::endl;

    // Will generate function declarations
    // (which is essentially all there is left to do)
    generateDeclarationCode(code, m_RootNode);

    return code.str();
}


void CodeGenerator::generateDeclarationCode(
        std::stringstream& code,
        const Node* node) const {

    if (
            node->m_NodeType != NodeType_Declaration &&
            node->m_NodeType != NodeType_DeclarationList) {
        return;
    }

    if (node->m_NodeType == NodeType_DeclarationList) {
        generateDeclarationCode(code, node->m_Children[0]);
        generateDeclarationCode(code, node->m_Children[1]);
        return;
    }

    // The type is Declaration for sure
    // The next function will return safely even if it is called upon not a function
declaration
    generateFunctionDeclarationCode(code, node);
}


void CodeGenerator::generateVariableDeclarationsCode(std::stringstream& code) const {
    for (const auto& var : m_Variables) {
        code << shift() << var.first << " " << getAssemblyTypeFor(var.second)
             << " " << getAssemblyZeroFor(var.second) << std::endl;
    }
}
```

```
void CodeGenerator::generateBlockCode(
    std::stringstream& code,
    const Node* blockNode) const {

    if (blockNode->m_NodeType != NodeType_Block) {
        return;
    }

    // A Node is either {} or {STMT} or {STMT_LIST}

    if (blockNode->m_Children.size() == 2) {
        // The {} case.
        // No statements => no code.
        // No money => no honey.
        return;
    }

    const Node* blockChild = blockNode->m_Children[1];
    if (blockChild->m_NodeType == NodeType_StatementList) {
        generateStatementListCode(code, blockChild);
    } else if (blockChild->m_NodeType == NodeType_Statement) {
        generateStatementCode(code, blockChild);
    }
}


void CodeGenerator::generateStatementListCode(
    std::stringstream& code,
    const Node* statementListNode) const {

    if (statementListNode->m_NodeType != NodeType_StatementList) {
        return;
    }

    const std::vector<const Node*> children =
        statementListNode->m_Children;

    if (children[0]->m_NodeType == NodeType_StatementList) {
        generateStatementListCode(code, children[0]);
    } else if (children[0]->m_NodeType == NodeType_Statement) {
        generateStatementCode(code, children[0]);
    }

    generateStatementCode(code, children[1]);
}


void CodeGenerator::generateStatementCode(
    std::stringstream& code,
    const Node* statementNode) const {

    if (statementNode->m_NodeType != NodeType_Statement) {
        return;
    }

    const std::vector<const Node*> children = statementNode->m_Children;
    const NodeType firstNodeType = children[0]->m_NodeType;
    switch (firstNodeType) {
        case NodeType_Block:
            generateBlockCode(code, children[0]);
            break;
        case NodeType_FunctionCallVoid:
            generateFunctionCallCode(code, children[0]);
```

```
            break;
        case NodeType_IdInt: { // assignment
            const Node* expressionRoot = children[2];
            generateExpressionEvaluation(code, expressionRoot);
            code << std::endl;

            // The previous function will have stored the result of the
            // expression in the EAX register
            code << shift() << "mov " << children[0]->m_NodeValue << ", eax" << std::endl
                 << std::endl;
            break;
        }
        case NodeType_KeywordIf:
            if (children.size() >= 5) { //if statement OR ifelse statement
                // Generate the first part
                generateExpressionEvaluation(code, children[2]);
                code << shift() << "cmp eax, 1" << std::endl
                     << shift() << "jne loop" << m_NextAvailablyLoopIndex << std::endl;
                generateStatementCode(code, children[4]);
                code << std::endl;
                m_NextAvailablyLoopIndex++;
                code << shift() << "jmp loop" << m_NextAvailablyLoopIndex << std::endl;
            }

            code << "loop" << (m_NextAvailablyLoopIndex - 1) << ":" << std::endl;
            m_NextAvailablyLoopIndex++;

            if (children.size() == 7) { // ifelse statement
                // Generate the 'else' part if it exists(if size == 7)
                generateStatementCode(code, children[6]);
                code << "loop" << (m_NextAvailablyLoopIndex - 1) << ":" << std::endl;
                code << std::endl;
                m_NextAvailablyLoopIndex++;
            }
            break;

        default:
            break;
    }
}


// The result of evaluation is always stored in the EAX register
void CodeGenerator::generateExpressionEvaluation(
    std::stringstream& code,
    const Node* expressionRoot) const {

    const NodeType expressionType = expressionRoot->m_NodeType;
    const std::vector<const Node*> children = expressionRoot->m_Children;
    const Node* firstNode = children[0];
    const NodeType firstNodeType = firstNode->m_NodeType;
    switch (expressionType) {
        case NodeType_ExpressionBool:
        case NodeType_ExpressionFloat:
        case NodeType_ExpressionInt: {
            switch (firstNodeType) {
                case NodeType_IdBool:
                case NodeType_IdFloat:
                case NodeType_IdInt: {
                    code << shift() << "mov eax, " << firstNode->m_NodeValue
                                    << "" << std::endl;
                    break;
                }
                case NodeType_LiteralBool:
                    code << shift() << "mov eax, ";
```

```
                    if (firstNode->m_NodeValue == "true") {
                       code << "1";
                    } else if (firstNode->m_NodeValue == "false"){
                       code << "0";
                    }
                    code << std::endl;
                    break;
                case NodeType_LiteralFloat:
                case NodeType_LiteralInt: {
                    code << shift() << "mov eax, " << firstNode->m_NodeValue << std::endl;
                    break;
                }
                case NodeType_FunctionCallBool:
                case NodeType_FunctionCallFloat:
                case NodeType_FunctionCallInt:
                    generateFunctionCallCode(code, firstNode);
                    break;
                case NodeType_ParenthesesOpen:
                    generateExpressionEvaluation(code, children[1]);
                    break;
                case NodeType_ExpressionBool:
                case NodeType_ExpressionFloat:
                case NodeType_ExpressionInt: { // arithmetic operations
                    const Node* operand1 = children[0];
                    const Node* operand2 = children[2];
                    const NodeType operation = children[1]->m_NodeType;

                    generateExpressionEvaluation(code, operand2);
                    code << shift() << "mov ebx, eax" << std::endl;
                    generateExpressionEvaluation(code, operand1);

                    // The arguments are now in EAX and EBX

                    std::string addedF{
                        operand1->m_NodeType == NodeType_ExpressionFloat ? "f" : ""
                    };
                    switch (operation) {
                        case NodeType_OperatorPlus:
                            code << shift() << addedF << "add eax, ebx" << std::endl;
                            break;
                        case NodeType_OperatorMinus:
                            code << shift() << addedF <<  "sub eax, ebx" << std::endl;
                            break;
                        case NodeType_OperatorMultiply:
                            code << shift() << addedF << "mul ebx" << std::endl
                                 << shift() << "mov eax, edx" << std::endl;
                            break;
                        case NodeType_OperatorDivide:
                            code << shift() << "xor edx, edx" << std::endl
                                 << shift() << addedF << "div ebx" << std::endl;
                            break;

                        case NodeType_OperatorEquals:
                            code << shift() << addedF << "cmp eax, ebx" << std::endl;
                            code << shift() << "je loop" << m_NextAvailablyLoopIndex <<
std::endl;
                            // Not equals
                            code << shift() << "mov eax, 0" << std::endl;
                            code << shift() << "jmp loop" << (m_NextAvailablyLoopIndex + 1) <<
std::endl;
                            // Equals
                            code << "loop" << m_NextAvailablyLoopIndex << ":" << std::endl;
                            code << shift() << "mov eax, 1" << std::endl;
                            code << "loop" << (m_NextAvailablyLoopIndex + 1) << ":" <<
std::endl;
```

```
                m_NextAvailablyLoopIndex += 2;
                break;
            case NodeType_OperatorNotEquals:
                code << shift() << addedF << "cmp eax, ebx" << std::endl;
                code << shift() << "je loop" << m_NextAvailablyLoopIndex <<
std::endl;
                // Not equals
                code << shift() << "mov eax, 1" << std::endl;
                code << shift() << "jmp loop" << (m_NextAvailablyLoopIndex + 1) <<
std::endl;
                // Equals
                code << "loop" << m_NextAvailablyLoopIndex << ":" << std::endl;
                code << shift() << "mov eax, 0" << std::endl;
                code << "loop" << (m_NextAvailablyLoopIndex + 1) << ":" <<
std::endl;
                m_NextAvailablyLoopIndex += 2;
                break;
            case NodeType_OperatorLess:
                code << shift() << addedF << "cmp eax, ebx" << std::endl;
                code << shift() << "jl loop" << m_NextAvailablyLoopIndex <<
std::endl;
                // False
                code << shift() << "mov eax, 0" << std::endl;
                code << shift() << "jmp loop" << (m_NextAvailablyLoopIndex + 1) <<
std::endl;
                // True
                code << "loop" << m_NextAvailablyLoopIndex << ":" << std::endl;
                code << shift() << "mov eax, 1" << std::endl;
                code << "loop" << (m_NextAvailablyLoopIndex + 1) << ":" <<
std::endl;
                m_NextAvailablyLoopIndex += 2;
                break;
            case NodeType_OperatorGreater:
                code << shift() << addedF << "cmp eax, ebx" << std::endl;
                code << shift() << "jg loop" << m_NextAvailablyLoopIndex <<
std::endl;
                // False
                code << shift() << "mov eax, 0" << std::endl;
                code << shift() << "jmp loop" << (m_NextAvailablyLoopIndex + 1) <<
std::endl;
                // True
                code << "loop" << m_NextAvailablyLoopIndex << ":" << std::endl;
                code << shift() << "mov eax, 1" << std::endl;
                code << "loop" << (m_NextAvailablyLoopIndex + 1) << ":" <<
std::endl;
                m_NextAvailablyLoopIndex += 2;
                break;
            case NodeType_OperatorLessOrEquals:
                code << shift() << addedF << "cmp eax, ebx" << std::endl;
                code << shift() << "jle loop" << m_NextAvailablyLoopIndex <<
std::endl;
                // False
                code << shift() << "mov eax, 0" << std::endl;
                code << shift() << "jmp loop" << (m_NextAvailablyLoopIndex + 1) <<
std::endl;
                // True
                code << "loop" << m_NextAvailablyLoopIndex << ":" << std::endl;
                code << shift() << "mov eax, 1" << std::endl;
                code << "loop" << (m_NextAvailablyLoopIndex + 1) << ":" <<
std::endl;
                m_NextAvailablyLoopIndex += 2;
                break;
            case NodeType_OperatorGreaterOrEquals:
                code << shift() << addedF << "cmp eax, ebx" << std::endl;
```

```
                            code << shift() << "jge loop" << m_NextAvailablyLoopIndex <<
std::endl;
                            // False
                            code << shift() << "mov eax, 0" << std::endl;
                            code << shift() << "jmp loop" << (m_NextAvailablyLoopIndex + 1) <<
std::endl;
                            // True
                            code << "loop" << m_NextAvailablyLoopIndex << ":" << std::endl;
                            code << shift() << "mov eax, 1" << std::endl;
                            code << "loop" << (m_NextAvailablyLoopIndex + 1) << ":" <<
std::endl;
                            m_NextAvailablyLoopIndex += 2;
                            break;
                        default:
                            break;
                    }

                    break;
                }
                default:
                    break;
            }
            break;
        }
        default:
            break;
    }
}


// In order to call a function we need to evaluate all arguments(expressions)
// and push them, then call the function.
void CodeGenerator::generateFunctionCallCode(
    std::stringstream& code,
    const Node* functionNode) const {

    const std::vector<const Node*> children = functionNode->m_Children;
    const NodeType functionType = functionNode->m_NodeType;
    const std::string functionName = children[0]->m_NodeValue;
    switch (functionType) {
        case NodeType_FunctionCallInt:
        case NodeType_FunctionCallFloat:
        case NodeType_FunctionCallBool:
        case NodeType_FunctionCallVoid: {
            const bool argumentsPresent = children.size() > 3;
            const auto functionExpectedTypes = m_Functions.at(functionName);
            if (argumentsPresent) {
                const auto types = generateExpressionListPush(code, children[2]);
                if (functionExpectedTypes.size() - 1 != types.size()) {
                    m_ErrorFound = true;
                    std::cout << ":> [Parse Error]: specified function arguments do not match
with those specified at function declaration(for function " << functionName << ")." <<
std::endl;
                    return;
                }

                for (unsigned int i = 0; i < types.size(); i++) {
                    if (functionExpectedTypes[i+1].second != types[i]) {
                        m_ErrorFound = true;
                        std::cout << ":> [Parse Error]: Specified function arguments do not
match with those specified at function declaration(for function " << functionName << ")." <<
std::endl;
                        std::cout << ":> [Parse Error]: Expected:Found: " <<
functionExpectedTypes[i+1].second << ":" << types[i] << std::endl;
                        return;
```

```
                    }
                }
            }
            code << shift() << "call " << functionName << std::endl;
            code << std::endl;
            break;
        }
        default:
            break;
    }
}


// Handles individual Expressions as well
// Returns the list of arguments' types
std::vector<NodeType> CodeGenerator::generateExpressionListPush(
    std::stringstream& code,
    const Node* expressionListRoot) const {

    std::vector<NodeType> types;
    const NodeType expressionListRootType = expressionListRoot->m_NodeType;
    if (
            expressionListRootType == NodeType_ExpressionBool ||
            expressionListRootType == NodeType_ExpressionInt ||
            expressionListRootType == NodeType_ExpressionFloat
        ) {
        generateExpressionEvaluation(code, expressionListRoot);
        code << shift() << "push eax" << std::endl;

        if (expressionListRootType == NodeType_ExpressionBool) {
            types.push_back(NodeType_KeywordBool);
        } else if (expressionListRootType == NodeType_ExpressionInt) {
            types.push_back(NodeType_KeywordInt);
        } else if (expressionListRootType == NodeType_ExpressionFloat) {
            types.push_back(NodeType_KeywordFloat);
        }
    } else if (expressionListRootType == NodeType_ExpressionList) {
        const Node* child1 = expressionListRoot->m_Children[0];
        const Node* child2 = expressionListRoot->m_Children[2];
        const auto res1 = generateExpressionListPush(code, child1);
        const auto res2 = generateExpressionListPush(code, child2);
        types.insert(
                types.end(),
                res1.begin(),
                res1.end()
        );
        types.insert(
                types.end(),
                res2.begin(),
                res2.end()
        );
    }

    return types;
}


// Will safely return if it is not a function declaration
void CodeGenerator::generateFunctionDeclarationCode(
    std::stringstream& code,
    const Node* functionDeclarationNode) const {

    if (functionDeclarationNode->m_NodeType != NodeType_Declaration) {
        return;
    }
```

```
    // The BLOCK is always the last child
    const unsigned int amountOfChildren = functionDeclarationNode->m_Children.size();
    const Node* lastChildNode = functionDeclarationNode->m_Children[amountOfChildren - 1];

    if (lastChildNode->m_NodeType != NodeType_Block) {
        // If it ain't a function declaration => no business here
        return;
    }

    // Generate function header

    const std::string functionName =
        functionDeclarationNode->m_Children[0]->m_Children[1]->m_NodeValue;
    const std::vector< std::pair<std::string, NodeType> > functionArguments =
        m_Functions.at(functionName);

    // Function title
    code << functionName << " proc" << std::endl;
    code << shift() << "push ebp" << std::endl
         << shift() << "mov ebp, esp" << std::endl;
    code << std::endl;

    // Retrieve arguments from the srack
    const unsigned int amountOfArguments = functionArguments.size() - 1; // 0th is the return
type
    const unsigned int firstArgShift = (amountOfArguments + 1) * 4;
    for (unsigned int i = 1; i < functionArguments.size(); i++) {
        code << shift() << "mov eax, dword ptr[ebp + " << (firstArgShift - (i - 1) * 4) << "]"
<< std::endl;
        code << shift() << "mov " << functionArguments[i].first << ", eax" << std::endl;
        code << std::endl;
    }

    // Generate function body
    generateBlockCode(code, lastChildNode);


    // Generate function end
    code << shift() << "mov esp, ebp" << std::endl
         << shift() << "pop ebp" << std::endl
         << shift() << "ret " << (amountOfArguments * 4) << std::endl;
    code << functionName << " endp" << std::endl;
    code << std::endl;
}



std::vector< std::pair<std::string, NodeType> >
    CodeGenerator::getAllVariableDeclarations(const Node* parentNode) const {

    std::vector< std::pair<std::string, NodeType> > declarations;

    const std::vector<const Node*> children = parentNode->m_Children;
    for (unsigned int index = 0; index < children.size(); index++) {
        const Node* child = children[index];

        if (
                child->m_NodeType == NodeType_Declaration ||
                child->m_NodeType == NodeType_DeclarationList ||
                child->m_NodeType == NodeType_TypedIdList
           ) {
            const std::vector< std::pair<std::string, NodeType> >
                childDeclarations = getAllVariableDeclarations(child);
            declarations.insert(
```

```
                    declarations.end(),
                    childDeclarations.begin(),
                    childDeclarations.end()
            );
        } else if (child->m_NodeType == NodeType_TypedId) {
            const bool functionId = (index + 1) < children.size() &&
                children[index + 1]->m_NodeType == NodeType_ParenthesesOpen;
            if (!functionId) {
                const std::vector<const Node*> typedIdChildren = child->m_Children;
                const NodeType type = typedIdChildren[0]->m_NodeType;
                const std::string name = typedIdChildren[1]->m_NodeValue;
                declarations.push_back(std::pair<std::string, NodeType>(name, type));
            }
        }
    }

    return declarations;
}

std::map< std::string, std::vector< std::pair<std::string, NodeType> > >
    CodeGenerator::getAllFunctionDeclarations(const Node* parentNode) const {

    std::map< std::string, std::vector< std::pair<std::string, NodeType> > > declarations;

    const std::vector<const Node*> children = parentNode->m_Children;
    for (unsigned int index = 0; index < children.size(); index++) {
        const Node* child = children[index];

        if (
                child->m_NodeType == NodeType_Declaration ||
                child->m_NodeType == NodeType_DeclarationList ||
                child->m_NodeType == NodeType_TypedIdList
           ) {
            std::map< std::string, std::vector< std::pair<std::string, NodeType> > >
                childDeclarations = getAllFunctionDeclarations(child);
            declarations.insert(
                    childDeclarations.begin(),
                    childDeclarations.end()
            );
        } else if (child->m_NodeType == NodeType_TypedId) {
            const bool functionId = (index + 1) < children.size() &&
                children[index + 1]->m_NodeType == NodeType_ParenthesesOpen;
            if (functionId) {
                const std::vector<const Node*> typedIdChildren = child->m_Children;
                const NodeType functionType = typedIdChildren[0]->m_NodeType;
                const std::string functionName = typedIdChildren[1]->m_NodeValue;
                const std::vector<std::pair<std::string, NodeType>> variableDeclarations =
                    getAllVariableDeclarations(parentNode);
                std::vector< std::pair<std::string, NodeType> > arguments;
                arguments.push_back(std::pair<std::string, NodeType>(functionName,
functionType)); // the 0th type is the function's return type
                for (const auto& argument : variableDeclarations) {
                    arguments.push_back(std::pair<std::string, NodeType>(
                        argument.first, argument.second
                    ));
                }
                declarations.insert(
                    std::pair<std::string, std::vector< std::pair<std::string, NodeType>> >(
                        functionName, arguments
                ));
            }
        }
    }

    return declarations;
```

```
}


std::string CodeGenerator::getAssemblyTypeFor(NodeType nodeType) const {
    switch(nodeType) {
        case NodeType_KeywordInt:
            return "dd";
        case NodeType_KeywordBool:
            return "db";
        case NodeType_KeywordFloat:
            return "dq";
        default:
            return "??";
    }
}

std::string CodeGenerator::getAssemblyZeroFor(NodeType nodeType) const {
    switch(nodeType) {
        case NodeType_KeywordInt:
            return "0";
        case NodeType_KeywordBool:
            return "0";
        case NodeType_KeywordFloat:
            return "0.0";
        default:
            return "??";
    }
}


std::string CodeGenerator::shift() const {
    return "    ";
}


bool CodeGenerator::errorWasFound() const {
    return m_ErrorFound;
}
```

main.cpp

```
/*
 * System Programming
 * "Compiler from C to x86 Assembly"
 *
 * Author: Igor Boyarshin, #5207, IO-52, FIOT
 * Date: 30.11.17
 */

#include <iostream>
#include <vector>
#include "LexicalAnalyzer.h"
#include "SyntaxAnalyzer.h"
#include "Token.h"
#include "CodeGenerator.h"


int main() {
    std::cout << "---------------- Program start ----------------"
              << std::endl;

    const std::string input =
        "int a; int main(){ if(true) else a = 6;}";
```

```
    std::cout << ":> Input: " << std::endl << input << std::endl;

    std::cout << "---------------- Lexical Analyzer ----------------"
              << std::endl;

    LexicalAnalyzer la;
    const auto tokens = la.analyze(input);
    if (tokens.size() == 0) {
        std::cout << ":> [Lexical Analyzer Error]: Invalid input. Terminating." << std::endl;
        return -1;
    }

    std::cout << "---------------- Syntax Analyzer ----------------"
              << std::endl;

    SyntaxAnalyzer sa{tokens};
    const bool inputCorrect = sa.parse();
    if (!inputCorrect) {
        std::cout << ":> [Syntax Analyzer Error]: Invalid input. Terminating." << std::endl;
        return -1;
    }

    std::cout << "---------------- Code Generator ----------------"
              << std::endl;

    CodeGenerator cg(sa.getRootNode());
    const auto code = cg.generate();
    if (cg.errorWasFound()) {
        std::cout << ":> [Code Generator Error]: Invalid input. Terminating." << std::endl;
        return -1;
    }
    std::cout << ":> Output code: " << std::endl
              << code
              << std::endl;


    return 0;
}
```