|  |  |
| --- | --- |
| https://studfiles.net/html/2706/219/html_4NnFGVyFmL.LWVf/img-KuTuVC.png | МИНОБРНАУКИ РОССИИ  федеральное государственное бюджетное образовательное учреждение  высшего образования  **«Балтийский государственный технический университет «ВОЕНМЕХ» им. Д.Ф. Устинова»**  **(БГТУ «ВОЕНМЕХ» им. Д.Ф. Устинова»)** |
| БГТУ.СМК-Ф-4.2-К5-01 |

|  |  |  |  |
| --- | --- | --- | --- |
| Факультет | И |  | Информационные и управляющие системы |
|  | шифр |  | наименование |
| Кафедра | И5 |  | Информационные системы и программная инженерия |
|  | шифр |  | наименование |
| Дисциплина | Системное ПО | | |

КУРСОВАЯ РАБОТА

на тему

|  |
| --- |
| Создание компилятора |
| для подмножества языка TypeScript на ASM i386 |
|  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Выполнил студент группы | | | | | | И-582 |
| Махнев П.С. | | | | | | |
| Фамилия И.О. | | | | | | |
|  | | | | | | |
| **РУКОВОДИТЕЛЬ** | | | | | | |
|  | | | | | | |
| Иванов К.С. | | | |  |  | |
| Фамилия И.О. | | | |  | Подпись | |
|  | | | | | | |
| Оценка | | |  | | |  |
| « |  | » |  | | | 2020 г. |

Санкт-Петербург  
2020

**Балтийский государственный технический университет «ВОЕНМЕХ» им. Д.Ф. Устинова»  
(БГТУ «ВОЕНМЕХ» им. Д.Ф. Устинова»)**

|  |  |  |  |
| --- | --- | --- | --- |
| Факультет | И |  | Информационные и управляющие системы |
|  | шифр |  | наименование |
| Кафедра | И5 |  | Информационные системы и программная инженерия |
|  | шифр |  | наименование |
| Дисциплина | Системное ПО | | |

**ЗАДАНИЕ**

**НА КУРСОВУЮ РАБОТУ**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Студенту | | Махневу Петру Сергеевичу | | | | | | |
|  | | (фамилия, имя, отчество) | | | | | | |
| Научный руководитель | | |  | | | | | |
|  | | | (ученая степень, звание, Ф.И.О) | | | | | |
| Тема: | Создание компилятора для подмножества языка TypeScript на | | | | | | | |
| ASM i386 | | | | | | | | |
| Основный вопросы, подлежащие разработке: | | | |  | | | | |
| 1. Лексический анализ, алфавит, символы | | | | | | | | |
| 1. Синтаксический анализ, грамматика, дерево разбора | | | | | | | | |
| 1. Семантический анализ, проверка типов | | | | | | | | |
| 1. Генерация кода, создание программ на языке ассемблера | | | | | | | | |
| Задание выдал: Иванов Константин Сергеевич | | | | |  |  |  |  | |
|  | | | | |  | дата |  | подпись | |
| Задание принял: Иванов Константин Сергеевич | | | | |  |  |  |  | |
|  | | | | |  | дата |  | подпись | |

Санкт-Петербург  
2020

**Содержание**

[1 Введение 4](#_Toc39183115)

[2 Лексический анализ 6](#_Toc39183116)

[2.1 Описание 6](#_Toc39183117)

[2.2 Входной язык 6](#_Toc39183118)

[2.3 Выходной язык 7](#_Toc39183119)

[2.4 Реализация 8](#_Toc39183120)

[2.5 Результат работы 11](#_Toc39183121)

[3 Синтаксический анализ 12](#_Toc39183122)

[3.1 Описание 12](#_Toc39183123)

[3.1.1 Грамматика 12](#_Toc39183124)

[3.2 Реализация 19](#_Toc39183125)

[3.3 Результат работы 21](#_Toc39183126)

[4 Семантический анализ 22](#_Toc39183127)

[4.1 Описание 22](#_Toc39183128)

[4.2 Реализация 22](#_Toc39183129)

[4.3 Результат работы 24](#_Toc39183130)

[5 Генерация кода 25](#_Toc39183131)

[5.1 Описание 25](#_Toc39183132)

[5.2 Реализация 26](#_Toc39183133)

[5.3 Результат работы 29](#_Toc39183134)

[6 Заключение 30](#_Toc39183135)

[7 Тестовый пример 31](#_Toc39183136)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 37](#_Toc39183137)

[ПРИЛОЖЕНИЕ 38](#_Toc39183138)

# Введение

Целью данной курсовой работы стала разработка компилятора для языка высокого уровня TypeScript на язык ассемблера (MASM).

Компилятор — это специальная программа, которая переводит текст программы, написанный на языке программирования, в набор машинных кодов или на язык ассемблера.

Компиляция — сборка программы, включающая трансляцию всех модулей программы, написанных на одном или нескольких исходных языках программирования высокого уровня и/или языке ассемблера, в эквивалентные программные модули на низкоуровневом языке, близком машинному коду, иногда на язык ассемблера, или непосредственно на машинном языке или ином двоичнокодовом низкоуровневом командном языке и последующую сборку исполняемой машинной программы.

В случае данной курсовой работы, компилятор будет производить компиляцию программы из одного файла на языке TypeScript в один выходной файл на языке ассемблера.

Компиляцию можно разбить на несколько основных этапов:

1. Лексический анализ («токенизация», от англ. tokenizing) — процесс аналитического разбора входной последовательности символов на распознанные группы — лексемы, с целью получения на выходе идентифицированных последовательностей, называемых «токенами» (подобно группировке букв в словах).
2. Синтаксический анализ (или разбор, парсинг, англ. parsing) в лингвистике и информатике — процесс сопоставления линейной последовательности лексем (слов, токенов) естественного или формального языка с его формальной грамматикой. Результатом обычно является дерево разбора (синтаксическое дерево).
3. Семантический анализ — это этап компиляции, предназначенный для различных проверок синтаксической корректности входного кода, а также выделения составляющих, таких как переменные или функции.
4. Генерация кода — это автоматическое создание программного кода специальным приложением, при котором по заданным условиям полностью или частично формируется исходный код программы. Такое специальное приложение называется генератором кода.

Большинство компиляторов переводит программу с некоторого высокоуровневого языка программирования в машинный код, который может быть непосредственно выполнен физическим процессором, однако в данной работе компиляция будет производится на язык ассемблера. С языка ассемблера в машинный код будет переводить отдельный, уже готовый, компилятор, который входит в состав пакета MASM32.

Macro Assembler (MASM) — ассемблер для процессоров семейства x86. Первоначально был произведён компанией Microsoft для написания программ в операционной системе MS-DOS и был в течение некоторого времени самым популярным ассемблером, доступным для неё.

Компилятор, в данной работе, должен поддерживать следующие возможности языка:

1. Математические выражения
2. Логические выражения
3. Оператор(ы) цикла
4. Оператор(ы) ветвления
5. Функции и/или процедуры с корректной реализацией механизмов вызова, передачи и возврата параметров.
6. Функции ввода вывода или возможность использовать стандартные библиотеки.

# Лексический анализ

## Описание

Лексический анализатор (ЛА) — это первый этап процесса компиляции, на котором символы, составляющие исходную программу, группируются в отдельные минимальные единицы текста, несущие смысловую нагрузку — лексемы.

Задача лексического анализа - выделить лексемы и преобразовать их к виду, удобному для последующей обработки. ЛА использует контекстный анализ и таблицу состояний [1].

## Входной язык

Входным языком является подмножество языка TypeScript. В данной реализации обязательна строгая статическая типизация. Доступны следующие типы:

1. number — целое число;
2. boolean — логическая переменная;
3. string — только константные строки;
4. number[] — массив целых чисел;
5. boolean[] — массив логических переменных;

Алфавитязыка, следующий:

1. Строчные и прописные буквы латинского алфавита;
2. Цифры от 0 до 9;
3. Символ \_ (подчеркивания);
4. Набор специальных символов

* ’ " {} , | [ ] + - % /  ; ‘ : ? < > = ! & # ~ ^ . \*

1. Прочие символы.

Типы слов:

1. Ключевые слова.
2. Идентификаторы (имена переменных).
3. Символы (знаки) операций.
4. Разделители.
5. Литералы (константы) — фиксированные значения, которые программа не может изменять. Способ представления литерала зависит от его типа (например ‘a’ – символьная константа).

Из слов строятся выражения (предложения), команды.

В TypeScript признаком конца команды является символ точка с запятой.

## Выходной язык

В качестве выходного языка взят язык assembler, а точнее его реализация masm.

Алфавитязыка, следующий:

1. Латинские символы от А до Z, знаки @, #, \_, которые используются наравне с буквами. Строчные или прописные символы программой-ассемблером не различаются. Записи вида: mov eax, 9 и MOV EAX, 9 воспринимаются одинаково.
2. Цифры от 0 до 9
3. Набор специальных символов:

* + - \* / ( ) , . = : &
* Программа представляет собой набор предложений (операторов). Предложение состоит из нескольких полей. Поля записываются в определенном порядке с соблюдением принятых правил. Предложение имеет следующие поля:

[название] код операции [операнды] [комментарий]

Каждый оператор записывается с новой строки. Поля отделяются друг от друга одним или несколькими пробелами. Операнды (если их несколько) отделяются друг от друга запятой.

Названиепредложения — это имя, которое начинается с буквы, содержит не более 8 символов (букв или цифр).

Кодоперации — это мнемокод (условное обозначение) машинной команды, или имя команды транслятору, или имя макрокоманды.

Операнды —это объекты, используемые при выполнении данной операции.

Комментарий— это пояснение к исходной программе. В ассемблере комментарий ставится после точки с запятой:

Например: mov ah,9 ; это комментарий.

## Реализация

Основным классом, реализующим лексический анализ, является класс Lexer. Класс имеет следующие поля:

1. string m\_code — код из файла, который разбирается;
2. vector<Token\*> m\_tokens — массив токенов, на которые разбивается исходный код программы;
3. LexerState m\_state — текущее состояние лексического анализатора;
4. size\_t m\_current\_token\_index — индекс текущего токена. Данный индекс описывает токен, который сейчас обрабатывается в синтаксическом анализаторе.

Класс имеет основной метод void parse(), реализующий непосредственно лексический анализ.

Также класс имеет набор вспомогательных методов:

1. void open(const string& file\_path) — метод реализует открытие файла и считывания из него исходного кода;
2. void skip\_excess\_symbols(int& index, size\_t& current\_line, size\_t& current\_pos) — метод реализует пропуск неважных символов, таких как пробелы и переносы строк;
3. static bool is\_split\_symbol(const char& symbol) — метод проверяет, является ли текущий символ разделительным;
4. static bool is\_token\_symbol(const char& symbol) — метод проверяет, является ли текущий символ, таким символом, который должен быть представлен, как отдельный токен;
5. static bool next\_symbol\_is\_part\_of\_token(const char& token, const char& symbol) — метод проверяет, является ли следующий символ с текущим составным токеном (например, символ ! и символ =, это составной токен);
6. static bool is\_correct\_identifier(const string& token) — метод проверяет, является ли переданное значение корректным идентификатором;
7. void print\_tokens() — метод распечатывающий все токены, которые были выделены на стадии лексического анализа.

А также имеет набор методов для синтаксического анализа:

1. bool next\_token() — метод сдвигает индекс текущего токена на 1;
2. Token& current\_token() — метод возвращает текущий токен;
3. TokenType current\_token\_type() — метод возвращает тип текущего токена.

Работа лексического анализатора построена на однократном проходе строки с исходным кодом. Во время прохода происходит “накопление” текущего значения токена во временной переменной, когда анализатор встречает разделительный символ, он создает токен на базе той временной переменной, добавляет его в массив токенов и очищает временную переменную, для дальнейшего “накопления” нового токена.

Для того, чтобы корректно обрабатывать строковые константы, введены состояния. Анализатор имеет 2 состояния:

1. DEFAULT — описывает стандартное состояние;
2. IN\_STRING— описывает состояние, когда анализатор встретил строковую константу.

В зависимости от состояния, анализатор по-разному разбирает строку на токены. Так, встречая разделитель в строковой константе, анализатор не считает, что токен закончен и не добавляет его в массив токенов.

Для описания токена используется класс Token. Класс имеет два основных поля:

1. string m\_lexeme — значение токена;
2. TokenType m\_type — его тип.

Тип токена определяется на стадии создания объекта класса. Для этого используется метод static TokenType what\_type\_of\_lexeme(const string& lexeme), который принимает значение токена, а возвращает его тип. В конструкторе, это выглядит следующий образом:

Token(const std::string& lexeme, size\_t line = 0, size\_t pos = 0)  
{  
 this->m\_lexeme = lexeme;  
 this->m\_type = what\_type\_of\_lexeme(lexeme);  
  
 this->m\_line = line;  
 this->m\_pos = pos;  
}

Также класс содержит два вспомогательных поля:

1. size\_t m\_line — линия, где находится токен;
2. size\_t m\_pos — позиция в линии, где находится токен.

Метод, реализующий определение типа токена по лексеме, использует набор сравнений, в каждом из которых, значение токена сравнивается с предопределенным значение и в случае равенства, возвращается предопределенный тип. Ниже приведен пример такого рода проверок:

if (lexeme == "let")  
 return TokenType::LET;  
if (lexeme == "const")  
 return TokenType::CONST;

Тип токена описывается с помощью перечисления TokenType. Данное перечисление содержит типы всех допустимых токенов (59 штук).

Класс реализует методы для получения всех вышеобозначенных полей класса, а также набор вспомогательных, для синтаксического анализатора.

## Результат работы

Результатом работы лексического анализатора является набор токенов.

# Синтаксический анализ

## Описание

Задача синтаксического анализатора — провести разбор текста программы, сопоставив его с эталоном, данным в описании языках [1]. Структура конструкций синтаксического анализатора более сложна, чем структура идентификаторов и чисел. Поэтому для описания синтаксиса языка нужны более мощные грамматики нежели регулярные. Для этих целей были использованы контекстно-свободные грамматики.

### Грамматика

Входной язык имеет следующую грамматику. Грамматика описана в форме Бэкуса-Наура.

primary\_expression  
 : IDENTIFIER  
 | CONSTANT  
 | "true"  
 | "false"  
 | STRING\_LITERAL  
 | '(' expression ')'  
 ;  
  
type\_specifier  
 : "number"  
 | "boolean"  
 | "number[]"  
 | "boolean[]"  
 | "void"  
 ;  
  
type\_qualifier  
 : "let"  
 | "const"  
 ;  
  
direct\_declarator  
 : IDENTIFIER ':' type\_specifier  
 ;  
   
direct\_declarator\_list  
 : direct\_declarator  
 | direct\_declarator\_list ',' direct\_declarator  
  
declaration\_specifiers  
 : type\_qualifier direct\_declarator ';'  
 | type\_qualifier direct\_declarator '=' expression  
 ;  
  
postfix\_expression  
 : primary\_expression  
 | postfix\_expression '[' expression ']'  
 | postfix\_expression '(' ')'  
 | postfix\_expression '(' argument\_expression\_list ')'  
 | postfix\_expression "++"  
 | postfix\_expression "--"  
 ;  
  
argument\_expression\_list  
 : assignment\_expression  
 | argument\_expression\_list ',' assignment\_expression  
 ;  
  
unary\_expression  
 : postfix\_expression  
 | unary\_operator cast\_expression  
 ;  
  
unary\_operator  
 : '-'  
 | '!'  
 ;  
  
multiplicative\_expression  
 : unary\_expression  
 | multiplicative\_expression '\*' unary\_expression  
 | multiplicative\_expression '/' unary\_expression  
 ;  
  
additive\_expression  
 : multiplicative\_expression  
 | additive\_expression '+' multiplicative\_expression  
 | additive\_expression '-' multiplicative\_expression  
 ;  
  
relational\_expression  
 : additive\_expression  
 | relational\_expression '<' additive\_expression  
 | relational\_expression '>' additive\_expression  
 | relational\_expression "<=" additive\_expression  
 | relational\_expression ">=" additive\_expression  
 ;  
  
equality\_expression  
 : relational\_expression  
 | equality\_expression "!=" relational\_expression  
 | equality\_expression "==" relational\_expression  
 ;  
  
logical\_and\_expression  
 : equality\_expression  
 | logical\_and\_expression "&&" equality\_expression  
 ;  
  
logical\_or\_expression  
 : logical\_and\_expression  
 | logical\_or\_expression "||" logical\_and\_expression  
 ;  
  
assignment\_expression  
 : logical\_or\_expression  
 | unary\_expression assignment\_operator assignment\_expression  
 ;  
  
assignment\_operator  
 : '='  
 | MUL\_ASSIGN  
 | DIV\_ASSIGN  
 | MOD\_ASSIGN  
 | ADD\_ASSIGN  
 | SUB\_ASSIGN  
 ;  
  
expression  
 : assignment\_expression  
 | expression ',' assignment\_expression  
 ;  
  
constant\_expression  
 : conditional\_expression  
 ;  
   
identifier\_list  
 : IDENTIFIER  
 | identifier\_list ',' IDENTIFIER  
 ;  
   
   
initializer\_list  
 : initializer  
 | initializer\_list ',' initializer  
 ;  
  
statement  
 : compound\_statement  
 | expression\_statement  
 | selection\_statement  
 | iteration\_statement  
 ;  
   
   
compound\_statement  
 : '{' '}'  
 | '{' statement\_list '}'  
 ;  
   
statement\_list  
 : statement  
 | statement\_list statement  
 ;  
  
expression\_statement  
 : ';'  
 | expression ';'  
 ;  
  
selection\_statement  
 : "if" '(' expression ')' statement  
 | "if" '(' expression ')' statement "else" statement  
 ;  
  
iteration\_statement  
 : "while" '(' expression ')' statement  
 | "do" statement "while" '(' expression ')' ';'  
 | "for" '(' expression\_statement expression\_statement expression ')' statement  
 ;  
  
function\_definition  
 : "function" IDENTIFIER '(' direct\_declarator\_list ')' ':' type\_specifier  
 compound\_statement  
 ;

Для синтаксического анализа, в данной работе, используется метод рекурсивного спуска. Суть метода, заключается в создании ряда функций с именами, как у продукций грамматики. Процесс разбора начинается с главной продукции, в процессе её разбора, происходят переходы между правилами, а значит и функциями обработчиками, каждая из которых разбирает свою часть правила.

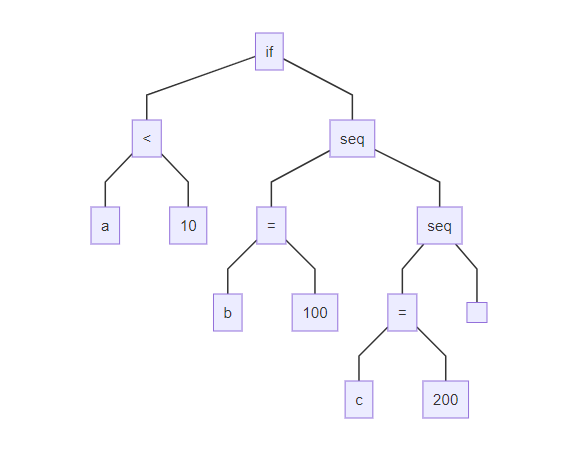
Задача синтаксического анализа состоит в том, чтобы определить имеет ли цепочка лексем конструкцию, заданную синтаксисом языка. Этот этап является основным для поиска ошибок в написанной программе.

Результатом синтаксического анализа является AST. AST (Abstract Syntax Tree, абстрактное синтаксическое дерево) — это дерево, которое в абстрактном виде представляет структуру программы. AST содержит полную синтаксическую модель программы без лишних деталей (таких, как пробельные символы или комментарии).

AST состоит из узлов, каждый узел задается с помощью класса Node. Каждый узел имеет значение, тип, а также 4 потомка. Все возможные типы описаны в перечислении NodeType, а также представлены списком ниже:

1. TERMINAL
2. VARIABLE\_DECLARATION
3. USING\_VARIABLE
4. VARIABLE\_TYPE
5. CONSTANT\_DECLARATION
6. USING\_CONSTANT
7. NUMBER\_CONST
8. BOOLEAN\_CONST
9. STRING\_CONST
10. ADD
11. SUB
12. MUL
13. DIV
14. EXPONENTIATION
15. BEFORE\_INC
16. BEFORE\_DEC
17. AFTER\_INC
18. AFTER\_DEC
19. UNARY\_PLUS
20. UNARY\_MINUS
21. UNARY\_EXCLAMATION
22. LOGICAL\_AND
23. LOGICAL\_OR
24. SET
25. LESS
26. GREATER
27. EQUAL
28. NOT\_EQUAL
29. LESS\_EQUAL
30. GREATER\_EQUAL
31. DO\_WHILE
32. FOR
33. WHILE
34. BREAK
35. CONTINUE
36. IF
37. IF\_ELSE
38. INDEX\_CAPTURE
39. FUNCTION\_CALL
40. FUNCTION\_ARGS
41. FUNCTION\_ARG
42. RETURN
43. FUNCTION\_IMPLEMENTATION
44. FUNCTION\_IMPLEMENTATION\_ARG
45. FUNCTION\_IMPLEMENTATION\_ARGS
46. FUNCTION\_IMPLEMENTATION\_RETURN\_TYPE
47. EXPRESSION
48. CONST\_EXPRESSION
49. STATEMENT
50. SEQ\_STATEMENT
51. STATEMENT\_LIST
52. INITIALIZER
53. INITIALIZER\_LIST
54. NEW
55. PROGRAM

Типичное дерево разбора выглядит следующим образом:



Также каждый узел может содержать указатель на таблицу переменных. Он используется, чтобы явно задать для каждого узла описывающего блок вида {...} свою таблицу переменных, для дальнейшего анализа и генерации кода.

Также каждый узел может иметь идентификатор блока, в котором он расположен. Это используется для новых переменных, чтобы явно знать, в каком блоке она была объявлена. Это позволяет избежать ситуаций, когда используется глобальная переменная с таким же именем, а не локальная.

## Реализация

Основной класс реализующий синтаксический анализ — это класс Parser. В нем, для каждой продукции грамматики, создана отдельная функция. Для каждого метода после тире написано, что он разбирает

1. Node\* primary\_expression(); — числовые, логические и строковые константы, идентификаторы. В зависимости от текущего токена может переходить в parenthesized\_expression, initializer, declaration\_statement, operator\_statement;
2. Node\* parenthesized\_expression(); — выражения в скобках, переходит в expression;
3. Node\* postfix\_expression(); — взятие индекса, вызов функции, постфиксный инкремент и декремент;
4. Node\* argument\_expression\_list(); — аргументы для вызова функции;
5. Node\* unary\_expression(); — префиксный инкремент и декремент, унарный минус;
6. Node\* multiplicative\_expression(); — выражения умножения;
7. Node\* additive\_expression(); — выражения сложения;
8. Node\* relational\_expression(); — выражения сравнения: больше, меньше, больше или равно, меньше или равно;
9. Node\* equality\_expression(); — выражения сравнения: равно, неравно;
10. Node\* logical\_and\_expression(); — выражения с логическим И;
11. Node\* logical\_or\_expression(); — выражения с логическим ИЛИ;
12. Node\* assignment\_expression(); — выражения присваивания, в том числе и присваивание с умножением, делением, сложение и вычитание;
13. Node\* expression(); — выражения
14. Node\* statement(); — конструкции;
15. Node\* compound\_statement(); — конструкции{...};
16. Node\* statement\_list(); — конструкции{...}, вспомогательный метод для compound\_statement;
17. Node\* expression\_statement(); — конструкция с выражением;
18. Node\* selection\_statement(); — конструкция if else;
19. Node\* iteration\_statement(); — конструкции: while, do while, for;
20. Node\* declaration\_type(); — вспомогательный метод для declaration\_statement. Разбирает тип переменной;
21. Node\* declaration\_statement(); — объявления переменных и констант;
22. Node\* initializer(); — список инициализаторов для массива;
23. Node\* initializer\_list(); — вспомогательный метод для initializer:
24. Node\* function\_argument(); — аргумент функции при ее объявлении;
25. Node\* function\_argument\_list(); — список аргументов при объявлении функции;
26. Node\* function\_statement(); — объявления функций;
27. Node\* operator\_statement(); — операторы return, new.

Класс имеет несколько полей:

1. Lexer\* m\_lex — лексический анализатор;
2. Ast\* m\_tree — абстрактное синтаксическое дерево;
3. Asm\* m\_asm — генератор ассемблерного кода.

У класса есть две основных функции:

1. void parse(); — метод реализующий синтаксический анализ;
2. void generate(); — метод реализующий генерацию кода;

Также для упрощения вызова ошибки есть функция error:

В результате работы анализатора в консоль выводится абстрактное синтаксическое дерево.

## Результат работы

Результатом работы синтаксического анализатора является абстрактное синтаксическое дерево, в котором в абстрактном виде описаны все конструкции кода.

# Семантический анализ

## Описание

Семантический анализ — обычно заключается в проверке правильности типов данных, используемых в программе. Кроме того, на этом этапе компилятор должен также проверить, соблюдаются ли определенные контекстные условия входного языка. В современных языка программирования одним из примеров контекстных условий может служить обязательность описания переменных: для каждого использующего вхождения идентификатора должно существовать единственное определяющее вхождение. Другой пример контекстного условия: число и атрибуты фактических параметров вызова процедуры должны быть согласованы с определением этой процедуры. Такие контекстные условия не всегда могут быть проверены во время синтаксического анализа и потому обычно выделяются в отдельную фазу.

## Реализация

За семантический анализ отвечает класс Ast, который, помимо хранения абстрактного синтаксического дерева, занимается и его обработкой.

Основными переменными, которые хранит класс, являются следующие переменные:

1. size\_t m\_count\_blocks — количество областей видимости для переменных;
2. VariableTable m\_all\_variables — таблица переменных;
3. vector<Variable\*> m\_global\_variables — массив глобальных переменных;
4. vector<VariableTable\*> m\_variable\_tables — массив таблиц переменных для каждой области видимости;
5. FunctionTable m\_functions — таблица функций;
6. GlobalFunctions m\_global\_functions — таблица глобальных, предопределенных функций.

Это количество блоков, таблица переменных, массив глобальных переменных, массив таблиц переменных для каждого блока, таблица функций и таблица глобальных, предопределенных функций.

Основными функция являются следующие функции (2 блока):

1 блок:

1. void designate\_blocks(); — выделение областей видимости переменных;
2. void designate\_variables(); — выделение переменных;
3. void designate\_global\_variables(); — выделение глобальных переменных;
4. void designate\_functions(); — выделение функций;
5. void designate\_arrays(); — выделение массивов.

2 блок:

1. void check\_const(); — проверка констант;
2. void check\_array(); — проверка массив и их инциализаторов;
3. void check\_functions\_call(); — проверка вызовов функций;
4. void check\_expression(); — проверка выражений.

Анализатор умеет проверять следующие семантические ошибки:

1. Присваивание константам после инициализации;
2. Проверка вызовов функций, проверка нужного количества аргументов с нужным типом;
3. Проверка корректности присваивания, например, если присваиваются переменные разных типов, то выбрасывается ошибка;
4. Проверка инициализации массива, проверка корректности типов в списке инициализаторов.

Также класс реализует функцию void print(Node\* sub, size\_t level) для печати дерева, которая является рекурсивной.

## Результат работы

Результатом работы семантического анализатора являются таблицы переменных, а также функций. Также косвенным результатом является гарантия синтаксической корректности программы. Помимо этого, семантический анализатор добавляет новую информацию в AST, такую как идентификаторы блоков, идентификаторы блоков, где объявлена переменная и т.п.

# Генерация кода

## Описание

Генерация кода является последней стадией процесса компиляции. Для данной курсовой работы конечный язык является ассемблер, что усложняет работу и требует знаний этого языка. Следует отметить, что в ассемблере используется стек. На нем удобно считать как арифметические операции, так и операции сравнения. Ассемблер предоставляет возможности для трансляции регулярных языков, но, чтобы использовать контекстно-свободный язык, надо предварительно его видоизменить, что, собственно, и сделано при обходе дерева разбора.

Ассемблирование может быть не первым и не последним этапом на пути получения исполнимого модуля программы. Так, многие компиляторы с языков программирования высокого уровня выдают результат в виде программы на языке ассемблера, которую в дальнейшем обрабатывает ассемблер. Также результатом ассемблирования может быть не исполняемый, а объектный модуль, содержащий разрозненные блоки машинного кода и данных программы, из которого (или из нескольких объектных модулей) в дальнейшем с помощью редактора связей может быть получен исполняемый файл.

В данной работе использовался синтаксис ассемблера MASM, так как разработка велась на Windows. Этот синтаксис имеет как свои недостатки, так и плюсы. Надо учитывать, что ассемблер в основном работает с самым низким уровнем иерархии данных в компьютере и из-за чего у него иногда есть проблемы с совместимостью на разных ОС. Дело усложняется тем, что на разных процессорах пишутся по-разному те или иные команды, к примеру команды на Intel могут существенно отличаться от команд на AMD. К примеру, в некоторых процессорах нет команд деления, которые «вшиты».

Регистр — это сверхбыстрая память, которая расположена в процессоре.

Регистры общего назначения служат для хранения промежуточных данных. В основном их размер 32 бита.

1. eax — применяется для хранения промежуточных данных, то есть он используется в качестве универсального аккумулятора значений.
2. ebx — базовый регистр, который используется для хранения адреса на некоторый блок в памяти
3. ecx — счетчик, который применяется в основном для циклов
4. edx — регистр данных, используется для хранения промежуточных вычислений
5. esp — указатель стека. Содержит адрес вершины стека
6. esi — индекс источника, в цепочечных операциях содержит указатель на текущий элемент-источник
7. edi — индекс приёмника, в цепочечных операциях содержит указатель на текущий элемент-приёмник
8. ebp — указатель базы кадра стека

## Реализация

Для генерации кода используется класс Asm. Класс содержит следующие основные поля:

1. ofstream m\_file — файл, в который будет записан результирующий код;
2. Ast\* m\_ast — синтаксическое дерево, полученное после семантического анализа;

Для удобной генерации ассемблерного кода, созданы 4 переменные, каждая из которых описывает свой блок ассемблерного кода:

1. string m\_data — блок данных;
2. string m\_before\_main — блок до стартовой метки;
3. string m\_function\_implementations — блок функций;
4. string m\_main — блок основного кода.

Основными функция для генерации кода являются следующие методы:

1. void blocks\_recursive(Node\* current\_node) — метод для генерации основных блоков (if, while и т.д);
2. void expression\_recursive(Node\* current\_node) — метод для генерации кода выражений (арифметических, логических и т.д.). Генерация кода для выражений осуществлялась с помощью стека, что очень похоже на обратную польскую запись.

Для генерации ассемблерного кода были созданы функции с именами, как у команд ассемблера, а также с количеством аргументов, как у команды. Это позволило проще работать с командами, а также увеличило читаемость кода, для генерации.

Команды ассемблера здесь добавляются с помощью методов класса. Далее приведены все реализованные команды:

1. void raw(const string& value)
2. void push(const string& value)
3. void pop(const string& value)
4. void add(const string& value1, const string& value2)
5. void sub(const string& value1, const string& value2)
6. void imul(const string& value1, const string& value2)
7. void mov(const string& value1, const string& value2)
8. void logical\_or(const string& value1, const string& value2)
9. void logical\_and(const string& value1, const string& value2)
10. void logical\_xor(const string& value1, const string& value2)
11. void cmp(const string& value1, const string& value2)
12. void jmp(const string& value)
13. void je(const string& value)
14. void jne(const string& value)
15. void jl(const string& value)
16. void jle(const string& value)
17. void jg(const string& value)
18. void jge(const string& value)
19. void label(const string& value)
20. void call(const string& value)
21. void ret(const string& value = "")
22. void proc(const string& value)
23. void endp(const string& value)
24. void procedure\_prolog(size\_t level, size\_t size\_local\_variable)
25. void procedure\_epilogue()
26. void finit()
27. void fild(const string& value)
28. void fdiv(const string& value1, const string& value2)
29. void fist(const string& value)
30. void fsqrt()
31. void comment(const string& value)

Был написан код для функций ввода, вывода, и квадратного корня. За их генерацию отвечают следующие методы:

1. void init\_input\_function(); — функция ввода числа;
2. void init\_print\_function(); — функция вывода числа;
3. void init\_println\_function(); — функция вывода строки;
4. void init\_sqrt\_function(); — функция квадратного корня.

Каждая из которых вызывается только в том случае, если функция определена в исходном коде.

Функция set\_place\_for\_writing позволяет задавать, куда будут писать все функции для команд ассемблера.

Для деления используется вспомогательный процессор FPU. Для его использования создаются две вспомогательные переменные для делителя и делимого.

## Результат работы

Результатом работы генератора кода, является корректный код на языке ассемблера, который компилируется в верную по функционалу программу.

# Заключение

Были получены знания и навыки о поэтапной разработке компилятора. Навык составления таблицы лексем для распознавания и формирования лексем языка. Разработано программное средство, реализующее лексический анализ текста программы на входном языке. Реализован синтаксический анализатор методом рекурсивного спуска, выполняющий построение дерева разбора. Разработан функционал генерации ассемблерного кода.

Разработка происходила на языке программирования C++ в среде JetBrains CLion 2020.1 с использованием компилятора MVSC. Был реализован компилятор для языка высокого уровня – TypeScript на язык низкого уровня – ассемблер, включающий в себя этапы лексического, синтаксического, семантического анализов, а также генератор кода. Программа была протестирована на различных примерах.

# Тестовый пример

Бинарная рекурсивная сортировка:

Реализация на TypeScript:

function print(n: number): void;

function println(n: string): void;

function input(): number;

function sqrt(a: number): number;

function qs(s\_arr: number[], first: number, last: number): void

{

if (first < last)

{

let left: number = first;

let right: number = last;

let middle: number = s\_arr[(left + right) / 2];

do

{

while (s\_arr[left] < middle)

{

left += 1;

}

while (s\_arr[right] > middle)

{

right -= 1;

}

if (left <= right)

{

let tmp: number = s\_arr[left];

s\_arr[left] = s\_arr[right];

s\_arr[right] = tmp;

left += 1;

right -= 1;

}

} while (left <= right);

qs(s\_arr, first, right);

qs(s\_arr, left, last);

}

}

let arr: number[] = [10, 4, 2, 14, 67, 2, 11, 33, 1, 15];

for (let i: number = 0; i < 10; i += 1)

{

print(arr[i]);

}

println(" ");

qs(arr, 0, 9);

for (let k: number = 0; k < 10; k += 1)

{

print(arr[k]);

}

println(" ");

Выходной код на ассемблере:

.586

.model flat, stdcall

include <\masm32\include\msvcrt.inc>

include <\masm32\include\kernel32.inc>

includelib <\masm32\lib\msvcrt.lib>

includelib <\masm32\lib\kernel32.lib>

data segment

string\_const\_0 db " ",13,10,0

string\_const\_1 db " ",13,10,0

div\_operand\_1 dd 0

div\_operand\_2 dd 0

; Global variable START

i27 dd 0

k31 dd 0

s\_arr10\_arr dd 0 dup (0)

s\_arr10 dd offset s\_arr10\_arr

arr1\_arr dd 10,4,2,14,67,2,11,33,1,15

arr1 dd offset arr1\_arr

; Global variable END

print\_format db "%d ", 0

println\_format db "%s", 0

input\_format db "%d", 0

input\_result dd 0

sqrt\_result dd 0

data ends

text segment

left14 = -4 ; size = 4

right14 = -8 ; size = 4

middle14 = -12 ; size = 4

tmp25 = -16 ; size = 4

arg\_n2 = 8 ; size = 4

arg\_n4 = 8 ; size = 4

arg\_a8 = 8 ; size = 4

arg\_s\_arr10 = 8 ; size = 4

arg\_first10 = 12 ; size = 4

arg\_last10 = 16 ; size = 4

print PROC

enter 0, 0

mov eax, [ebp + 8]

invoke crt\_printf, offset print\_format, eax

leave

ret 4

print ENDP

println PROC

enter 0, 0

mov eax, [ebp + 8]

invoke crt\_printf, offset println\_format, eax

leave

ret 4

println ENDP

input PROC

enter 0, 0

invoke crt\_scanf, offset input\_format, offset input\_result

mov eax, input\_result

leave

ret

input ENDP

sqrt PROC

enter 0, 0

mov eax, [ebp + 8]

mov sqrt\_result, eax

finit

fild sqrt\_result

fsqrt

fist sqrt\_result

mov eax, sqrt\_result

leave

ret 4

sqrt ENDP

qs PROC

enter 0, 16

push arg\_first10[ebp]

pop ecx

push arg\_last10[ebp]

pop edx

cmp ecx, edx

jge \_compare\_not\_equal428025419349334

push 1

jmp \_compare\_end428025419349334

\_compare\_not\_equal428025419349334:

push 0

\_compare\_end428025419349334:

pop eax

cmp eax, 0

je \_if\_end\_14

\_if\_start\_14:

push arg\_first10[ebp]

pop eax

mov left14[ebp], eax

push arg\_last10[ebp]

pop eax

mov right14[ebp], eax

push left14[ebp]

push right14[ebp]

pop eax

pop ebx

add eax, ebx

push eax

push 2

pop ebx

pop eax

mov div\_operand\_1, eax

mov div\_operand\_2, ebx

finit

fild div\_operand\_2

fild div\_operand\_1

fdiv st(0), st(1)

fist div\_operand\_1

push div\_operand\_1

pop edx

imul edx, 4

mov esi, arg\_s\_arr10[ebp]

push [esi[edx]]

pop eax

mov middle14[ebp], eax

\_loop\_start\_17:

\_loop\_aftereffects\_17:

\_loop\_start\_20:

\_loop\_aftereffects\_20:

push left14[ebp]

pop edx

imul edx, 4

mov esi, arg\_s\_arr10[ebp]

push [esi[edx]]

pop ecx

push middle14[ebp]

pop edx

cmp ecx, edx

jge \_compare\_not\_equal428025420293164

push 1

jmp \_compare\_end428025420293164

\_compare\_not\_equal428025420293164:

push 0

\_compare\_end428025420293164:

pop eax

cmp eax, 0

je \_loop\_end\_20

push left14[ebp]

push 1

pop eax

pop ebx

add eax, ebx

push eax

pop eax

mov left14[ebp], eax

jmp \_loop\_start\_20

\_loop\_end\_20:

\_loop\_start\_23:

\_loop\_aftereffects\_23:

push right14[ebp]

pop edx

imul edx, 4

mov esi, arg\_s\_arr10[ebp]

push [esi[edx]]

pop ecx

push middle14[ebp]

pop edx

cmp ecx, edx

jle \_compare\_not\_equal428025420742628

push 1

jmp \_compare\_end428025420742628

\_compare\_not\_equal428025420742628:

push 0

\_compare\_end428025420742628:

pop eax

cmp eax, 0

je \_loop\_end\_23

push right14[ebp]

push 1

pop ebx

pop eax

sub eax, ebx

push eax

pop eax

mov right14[ebp], eax

jmp \_loop\_start\_23

\_loop\_end\_23:

push left14[ebp]

pop ecx

push right14[ebp]

pop edx

cmp ecx, edx

jg \_compare\_not\_equal428025421155740

push 1

jmp \_compare\_end428025421155740

\_compare\_not\_equal428025421155740:

push 0

\_compare\_end428025421155740:

pop eax

cmp eax, 0

je \_if\_end\_25

\_if\_start\_25:

push left14[ebp]

pop edx

imul edx, 4

mov esi, arg\_s\_arr10[ebp]

push [esi[edx]]

pop eax

mov tmp25[ebp], eax

push right14[ebp]

pop edx

imul edx, 4

mov esi, arg\_s\_arr10[ebp]

push [esi[edx]]

pop edx

push left14[ebp]

pop ecx

imul ecx, 4

mov esi, arg\_s\_arr10[ebp]

add esi, ecx

mov [esi], edx

push tmp25[ebp]

pop edx

push right14[ebp]

pop ecx

imul ecx, 4

mov esi, arg\_s\_arr10[ebp]

add esi, ecx

mov [esi], edx

push left14[ebp]

push 1

pop eax

pop ebx

add eax, ebx

push eax

pop eax

mov left14[ebp], eax

push right14[ebp]

push 1

pop ebx

pop eax

sub eax, ebx

push eax

pop eax

mov right14[ebp], eax

\_if\_end\_25:

push left14[ebp]

pop ecx

push right14[ebp]

pop edx

cmp ecx, edx

jg \_compare\_not\_equal428025422646967

push 1

jmp \_compare\_end428025422646967

\_compare\_not\_equal428025422646967:

push 0

\_compare\_end428025422646967:

pop eax

cmp eax, 0

je \_loop\_end\_17

jmp \_loop\_start\_17

\_loop\_end\_17:

; init stack for qs

push right14[ebp]

push arg\_first10[ebp]

push arg\_s\_arr10[ebp]

; call qs

call qs

; init stack for qs

push arg\_last10[ebp]

push left14[ebp]

push arg\_s\_arr10[ebp]

; call qs

call qs

\_if\_end\_14:

leave

ret 12

qs ENDP

\_\_start:

enter 0, 0

push 0

pop eax

mov i27, eax

\_loop\_start\_29:

push i27

pop ecx

push 10

pop edx

cmp ecx, edx

jge \_compare\_not\_equal428025423689585

push 1

jmp \_compare\_end428025423689585

\_compare\_not\_equal428025423689585:

push 0

\_compare\_end428025423689585:

pop eax

cmp eax, 0

je \_loop\_end\_29

; init stack for print

push i27

pop edx

imul edx, 4

mov esi, arr1

push [esi[edx]]

; call print

call print

\_loop\_aftereffects\_29:

push i27

push 1

pop eax

pop ebx

add eax, ebx

push eax

pop eax

mov i27, eax

jmp \_loop\_start\_29

\_loop\_end\_29:

; init stack for println

push offset string\_const\_0

; call println

call println

; init stack for qs

push 9

push 0

push arr1

; call qs

call qs

push 0

pop eax

mov k31, eax

\_loop\_start\_33:

push k31

pop ecx

push 10

pop edx

cmp ecx, edx

jge \_compare\_not\_equal428025424410180

push 1

jmp \_compare\_end428025424410180

\_compare\_not\_equal428025424410180:

push 0

\_compare\_end428025424410180:

pop eax

cmp eax, 0

je \_loop\_end\_33

; init stack for print

push k31

pop edx

imul edx, 4

mov esi, arr1

push [esi[edx]]

; call print

call print

\_loop\_aftereffects\_33:

push k31

push 1

pop eax

pop ebx

add eax, ebx

push eax

pop eax

mov k31, eax

jmp \_loop\_start\_33

\_loop\_end\_33:

; init stack for println

push offset string\_const\_1

; call println

call println

leave

ret

text ends

end \_\_start

# СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Е. Н. Ишакова. Разработка компиляторов: Методические указания к курсовой работе. — Ориенбург: ГОУ ОГУ, 2005
2. А. Е. Пентус, М. Р. Пентус. Теория формальных языков: Учебное пособие — М.:ЦПИ, 2004
3. А. Ю. Молчанов. Системное программное обеспечение. — СПб.: Питер, 2003
4. Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман. Компиляторы: принципы, технологии и инструментарий — 2 изд. — М.: Вильямс, 2008
5. Вирт Н. Построение компиляторов. — М.: ДМК Пресс, 2010

# ****ПРИЛОЖЕНИЕ****

Текст программы на прилагаемом компакт-диске.