



Министерство науки и высшего образования Российской Федерации  
Федеральное государственное бюджетное образовательное учреждение  
высшего образования  
«Московский государственный технический университет  
имени Н.Э. Баумана  
(национальный исследовательский университет)»  
(МГТУ им. Н.Э. Баумана)

---

ФАКУЛЬТЕТ Информатика и системы управления

КАФЕДРА Программное обеспечение ЭВМ и информационные технологии

---

**РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА**  
***К КУРСОВОЙ РАБОТЕ***  
***НА ТЕМУ:***

Компилятор языка Pascal

Студент группы ИУ7-21М

\_\_\_\_\_  
(Подпись, дата)

И.Ю. Елгин

\_\_\_\_\_  
(И.О. Фамилия)

Руководитель

\_\_\_\_\_  
(Подпись, дата)

А.А. Ступников

\_\_\_\_\_  
(И.О. Фамилия)

2024 г.

## РЕФЕРАТ

Отчет содержит 29 стр., 5 рис., 3 табл., 11 источн., 2 прил.

Ключевые слова: КОМПИЛЯТОР, MASM, PASCAL, ЛЕКСИЧЕСКИЙ АНАЛИЗ, СИНТАКСИЧЕСКИЙ АНАЛИЗ, ГЕНЕРАЦИЯ КОДА.

Предмет исследования – компилятор. Объект исследования – построение компилятора языка Pascal.

Компилятор – это программное средство, которое преобразует исходный текст программы на определенном языке программирования в машинный код для исполнения на целевой архитектуре Pascal — это язык программирования высокого уровня, созданный в 1968-1969 годах швейцарским ученым Никлаусом Виртом. Исследование привело к разработке концептуальной модели компилятора, Разработанные программы для тестирования компилятора охватывают разнообразные языковые конструкции.

Полученные результаты являются основой для будущей работы над развитием и оптимизацией компилятора Pascal. Было проведено тестирование функциональности компилятора.

## СОДЕРЖАНИЕ

|   |    |
|---|----|
| РЕФЕРАТ.....  | 2  |
| ВВЕДЕНИЕ .....  | 4  |
| 1 Аналитический раздел .....  | 5  |
| 1.1 Принципы компиляции и архитектура компилятора.....                              | 5  |
| 1.1.1 Лексический анализатор .....  | 5  |
| 1.1.2 Синтаксический анализатор .....   | 5  |
| 1.1.3 Семантический анализатор.....   | 7  |
| 1.1.4 Генератор кода .....  | 7  |
| 1.2 MASM.....   | 8  |
| 1.3 Особенности языка Pascal.....   | 8  |
| 1.3.1 Типизация языка Pascal.....   | 8  |
| 1.3.2 Ключевые слова и грамматика языка .....                                       | 9  |
| 2 Конструкторский раздел.....   | 12 |
| 2.1 Концептуальная модель .....   | 12 |
| 2.2 Лексический анализ .....  | 12 |
| 2.3 Синтаксический разбор .....   | 13 |
| 2.4 Генерация кода.....   | 14 |
| 3 Технологический раздел.....   | 15 |
| 3.1 Выбор средств реализации .....  | 15 |
| 3.2 Средства ввода вывода .....   | 15 |
| 3.3 Исчисление выражений .....  | 16 |
| 3.4 Работа с дробными числами.....  | 16 |
| 3.4.1 Основные инструкции работы с дробными числами.....                            | 16 |
| 3.5 Тестирование компилятора .....  | 17 |
| ЗАКЛЮЧЕНИЕ.....   | 20 |
| СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ .....  | 21 |
| ПРИЛОЖЕНИЕ А — Множества FIRST и FOLLOW и таблица синтаксического анализатора ..... | 22 |
| ПРИЛОЖЕНИЕ Б — код ассемблера программы сортировки массива .....                    | 25 |

## ВВЕДЕНИЕ

Компилятор — это специализированная программа, которая преобразует исходный код программы, написанный на одном языке программирования, называемом исходным языком, в эквивалентный код на другом языке программирования, как правило, на машинный код или язык ассемблера, называемом целевым языком, который может быть выполнен на целевом устройстве [1].

Pascal — это язык программирования высокого уровня, созданный в 1968-1969 годах швейцарским ученым Никлаусом Виртом. Язык был назван в честь французского математика и философа Блеза Паскаля. Pascal был разработан как инструмент для обучения программированию и разработке алгоритмов, с акцентом на строгую типизацию и структурированность программного кода [2].

Разработка компиляторов является одной из важнейших задач в области программирования, так как они обеспечивают преобразование исходного кода, написанного на одном языке программирования, в код, исполняемый на конкретной платформе. В данной курсовой работе рассматривается создание прототипа компилятора, который выполняет перевод программы, написанной на языке Pascal, в инструкции ассемблера MASM.

Целью работы является разработка и реализация прототипа компилятора, который выполняет перевод кода программы с языка Pascal в инструкции ассемблера MASM. В рамках работы будут рассмотрены и реализованы основные компоненты компилятора: лексический анализатор, синтаксический анализатор и генератор кода.

Для достижения цели работы необходимо выполнить следующие задачи:

- изучить особенности и грамматику языка Pascal.
- изучить принципы анализа исходных кодов программ и генерации низкоуровневого кода;
- разработать и реализовать прототип компилятора, способного преобразовывать исходный код на языке Pascal в инструкции ассемблера MASM.

# **1 Аналитический раздел**

## **1.1 Принципы компиляции и архитектура компилятора**

Компилятор — это программа, которая переводит исходный код, написанный на одном языке программирования, в объектный код на другом языке (например, машинный код) [1].

Стандартный компилятор включает в себя следующие фазы:

1. предкомпиляция — удаление комментариев из кода программы и подключение встраиваемых файлов;
2. лексический анализ (сканирование) — разбиение исходного кода на лексемы, при этом происходит проверка правильности написания чисел и строк;
3. синтаксический анализ — проверка корректности структуры программы в соответствии с грамматикой языка;
4. семантический анализ — проверка логики и смыслового содержания программы;
5. генерация кода — преобразование промежуточного представления программы в целевой;
6. оптимизация кода.

### **1.1.1 Лексический анализатор**

Лексический анализатор (или сканер) предназначен для разбиения исходного текста программы на лексемы — минимальные единицы смысла, такие как ключевые слова, идентификаторы, операторы и т.д. В процессе анализа исходный код программы произвольной длины разбивается на токены, которые передаются синтаксическому анализатору для дальнейшей обработки.

Лексический анализатор имеет дело с такими объектами, как различного рода константы и идентификаторы (к последним относятся и ключевые слова). Язык констант и идентификаторов в большинстве случаев является регулярным — то есть, может быть описан с помощью регулярных грамматик. Распознавателями для регулярных языков являются конечные автоматы. Существуют правила, с помощью которых для любой регулярной грамматики может быть построен недетерминированный конечный автомат, распознающий цепочки языка, заданного этой грамматикой [3].

Лексический анализатор реализуется с помощью детерминированного конечного автомата.

### **1.1.2 Синтаксический анализатор**

Синтаксический анализатор (или парсер) выполняет проверку структуры исходного кода на соответствие грамматике языка Pascal. Важной задачей данного компонента является контроль выхода за границы таблицы лексем, что позволяет избежать возникновения критических ошибок. Парсер строит дерево разбора, на основе которого затем будет производиться генерация кода. Синтаксический анализатор (парсер) получает поток токенов от лексического анализатора и проверяет правильность их последовательности в соответствии с грамматикой языка программирования. На этом этапе строится синтаксическое дерево (дерево разбора), которое отражает иерархическую структуру программы.

## **Обзор видов синтаксических анализаторов**

Основные типы синтаксических анализаторов включают:

- LL-анализаторы (Left-to-Right, Leftmost derivation);
- LR-анализаторы (Left-to-Right, Rightmost derivation);
- Рекурсивный спуск;
- Парсер на основе операторного предшествования.

LL-анализатор — это метод разбора для определенного подмножества контекстно-свободных грамматик, которые известны как LL-грамматики. Буква L в термине «LL-анализатор» обозначает, что входная строка обрабатывается с начала до конца и генерируется ее левосторонний вывод. Если анализатор использует предварительный просмотр на  $k$  токенов при разборе входных данных, его называют LL( $k$ )-анализатор. LL(1)-анализаторы очень широко распространены, так как они просматривают входные данные только на один шаг вперед, чтобы определить, какое грамматическое правило применить [4].

LR-анализатор читает входной поток слева направо и создает самую правую продукцию контекстно-свободной грамматики. Термин LR( $k$ )-анализатор также используется. Здесь  $k$  означает количество неп прочитанных символов предварительного просмотра во входном потоке, на основе которых принимаются решения в процессе анализа. Синтаксис многих языков программирования определяется грамматикой LR(1) или аналогичной грамматикой. LR-анализатор обрабатывает код сверху вниз, поскольку он пытается создать продукцию верхнего уровня грамматики из листьев [4].

Синтаксический анализатор на основе рекурсивного спуска — это один из самых простых и понятных методов синтаксического анализа. Этот метод заключается в разборе грамматики языка с помощью рекурсивных функций, каждая из которых соответствует одному правилу грамматики. Основная идея заключается в том, что синтаксический анализатор обрабатывает входную строку слева направо, одновременно строя дерево разбора (или синтаксическое дерево) на основе правосторонних продукций грамматики [5].

Парсер операторного предшествования (или анализатор операторного предшествования) используется для анализа выражений в языках программирования, которые содержат операции с различными приоритетами и ассоциативностями. Он строит выражение, начиная с операторов и их операндов, и обрабатывает их в зависимости от их приоритета и ассоциативности [5].

В таблице 1.1 представлен сравнительный анализ синтаксических анализаторов.

Таблица 1.1 — Сравнение генераторов анализаторов исходного кода

| Анализаторы/Критерии | Подход                            | Поддержка синтаксиса языка Pascal | Простота реализации |
|----------------------|-----------------------------------|-----------------------------------|---------------------|
| LL(1)                | Сверху вниз, левосторонний вывод  | Частичный синтаксис               | Прост в реализации  |
| LR(1)                | Снизу вверх, правосторонний вывод | Да, полный синтаксис              | Сложен в реализации |

Продолжение Таблицы 1.1 — Сравнение генераторов анализаторов исходного кода

|                                     |                                    |                     |                    |
|-------------------------------------|------------------------------------|---------------------|--------------------|
| Рекурсивный спуск                   | Рекурсивное разветвление           | Частичный синтаксис | Прост в реализации |
| Парсер операторного предшествования | Обработка операторов по приоритету | Частичный синтаксис | Средняя сложность  |

Как видно из таблицы 1 для реализации синтаксического анализатора для языка pascal лучше использовать LR(1)-анализатор, так как только он способен обработать полный синтаксис языка Pascal.

### 1.1.3 Семантический анализатор

На этапе семантического анализа происходит проверка логической корректности программы, включая проверку типов, диапазонов значений, правильность вызова функций и процедур. Семантический анализатор проверяет соответствие построенного синтаксического дерева правилам и ограничениям языка. На данном этапе также происходит проверка условий и циклов и проверка областей видимости переменных и функций [6].

Для некоторых языков программирования семантический анализатор может быть совмещен с генератором кода. В компиляторной архитектуре это решение известно, как однопроходный компилятор. В однопроходном компиляторе семантический анализ и генерация кода происходят одновременно, что упрощает процесс компиляции и улучшает производительность компилятора. Такой подход хорошо подходит для относительно простых языков программирования, таких как Pascal, особенно для учебных компиляторов. Генерация кода происходит сразу по мере проверки семантики, что позволяет экономить память и время на последующую генерацию [7].

### 1.1.4 Генератор кода

Генератор кода выполняет преобразование дерева разбора в инструкции целевого языка. Этот процесс включает преобразование конструкций языка высокого уровня (таких как циклы, условные операторы и процедуры) в соответствующие им инструкции низкого уровня.

Инструменты генерации исполняемого кода играют ключевую роль в процессе компиляции программ. Они отвечают за трансляцию абстрактного синтаксического дерева (AST-дерева) или промежуточного представления (IR) программы в исполняемый машинный код или байткод, который может быть исполнен на целевой аппаратуре или виртуальной машине [6].

Генерация машинного кода относится к процессу преобразования исходного кода программы в низкоуровневый бинарный код, который может быть исполнен непосредственно аппаратурой процессора. Генерация машинного кода приводит к созданию исполняемых файлов, которые могут быть непосредственно запущены на целевой платформе без дополнительной обработки.

Также компилятор может включать оптимизацию кода и линковку (связывание) — процесс объединения различных объектных файлов и библиотек в один исполняемый файл.

## 1.2 MASM

MASM (Microsoft Macro Assembler) — это ассемблер для процессоров архитектуры x86, разработанный компанией Microsoft. Он используется для разработки низкоуровневого программного обеспечения, такого как драйверы устройств, операционные системы и другие программы, требующие прямого доступа к аппаратным ресурсам [8].

MASM предоставляет средства для работы с макросами, что облегчает написание и поддержку кода, а также предоставляет множество директив (например, `.data`, `.code`, `.stack`, `.model`, `PROC`), которые упрощают организацию кода и управление памятью.

MASM предоставляет поддержку различных типов данных, включая простые типы (`BYTE`, `WORD`, `DWORD`), структуры (`struct`), массивы и даже объектно-ориентированные элементы (прототипы объектов), что позволяет писать более структурированный и понятный код.

MASM позволяет генерировать код для работы в 32 и 64-разрядном режиме процессора, что позволяет компилировать программы под современные операционные системы, такие как Windows 10 и Windows 11 [8].

## 1.3 Особенности языка Pascal

Pascal — это универсальный язык программирования, отличающийся строгой структурой и типизацией переменных, а также интуитивно понятным синтаксисом. Язык Pascal не чувствителен к регистру символов.

Pascal был разработан Никлаусом Виртом в 1970 году, и его стандарты (например, ISO 7185 для стандартного Pascal) строго регламентируют синтаксис и семантику языка. Существуют разные диалекты Pascal (Turbo Pascal, Object Pascal), что может повлиять на особенности реализации компилятора [1].

### 1.3.1 Типизация языка Pascal

Pascal известен своей строгой статической типизацией. Типы данных проверяются во время компиляции, что помогает предотвратить ошибки, связанные с неправильным использованием типов.

Pascal поддерживает широкий спектр базовых типов данных, включая целые числа (`integer`), вещественные числа (`real`), булевы значения (`boolean`), символы (`char`), а также строки (`string`) и массивы [9].

Pascal поддерживает неявное приведение типов, но оно ограничено и осуществляется автоматически компилятором только в некоторых случаях. Например, при выполнении арифметических операций между целыми и вещественными числами (`integer` и `real`) происходит неявное преобразование целого числа в вещественное.

Для явного приведения типов в Pascal используются стандартные функции преобразования типов. Важно отметить, что не все типы могут быть преобразованы друг в друга, и при неправильном приведении типов может возникнуть ошибка во время выполнения программы.



## 1.3.2 Ключевые слова и грамматика языка

### Ключевые слова языка Pascal

В языке Pascal существуют следующие ключевые слова: **and, array, begin, const, div, do, downto, else, end, for, goto, if, label, mod, not, of, or, program, repeat, set, then, to, type, until, var, while.**

Данные слова являются зарезервированными и именовать константы, переменные и функции ими в языке Pascal нельзя [9].

Имена констант переменных и функций должны начинаться с латинской буквы и могут содержать латинские буквы и цифры.

Все переменные и константы должны быть объявлены до их использования в программе в соответствующих разделах `var` и `const`.

### Приоритет операторов

В Pascal существует 5 уровней приоритетов операторов, приведенных в таблице 1.2, и 3 уровня ассоциативности:

- **левоассоциативные** операторы выполняются слева направо;
- **правоассоциативные** операторы выполняются справа налево;
- **неассоциативные** операторы не имеют ассоциативности, что означает, что их нельзя использовать последовательно без использования скобок.

Таблица 1.2 — Приоритеты операторов [10]

| Приоритет  | Оператор                      | Описание                                  | Ассоциативность    |
|------------|-------------------------------|---|--------------------|
| 1 (высший) | @, ^, not                     | Адрес, разыменованье, НЕ                  | Правоассоциативные |
| 2          | *, /, div, mod, and, shl, shr | Умножение, деление, И, сдвиг              | Левоассоциативные  |
| 3          | +, -, or, xor                 | Сложение, вычитание, ИЛИ, исключающее ИЛИ | Левоассоциативные  |
| 4          | =, <>, <, <=, >, >=, in, is   | Операторы сравнения                       | Неассоциативные    |
| 5 (низший) | :=                            | Присваивание                              | Правоассоциативные |

### Грамматика языка

В листинге 1.1 приведена грамматика языка Pascal.

Листинг 1.1 – грамматика языка Pascal [10]

```
START -> program ID; HEADERTYPE .
```

```
HEADERTYPE -> type TYPELIST HEADERCONST | HEADERCONST
```

TYPEID -> integer | real | char | string | boolean | array [ Value . . Value ] of TYPEID

TYPELIST -> ID = TYPEID TYPELIST | ; ID = TYPEID TYPELIST | ;

HEADERCONST -> const CONSTLIST HEADERLABEL | HEADERLABEL

CONSTLIST -> ID = VALUE CONSTLIST | ; ID = VALUE CONSTLIST | ;

HEADERLABEL -> label LABELLIST | HEADERVER

LABELLIST -> ID LABELLIST | ; ID LABELLIST | ;

HEADERVER -> var VARLIST BLOCK

VARLIST -> ID , IDLIST VARLIST | ; ID , IDLIST VARLIST | ID : TYPEID VARLIST | ; ID : TYPEID VARLIST | ;

IDLIST -> ID , IDLIST | ID : TYPEID

BLOCK -> begin OPERATORLIST end

OPERATORLIST -> OPERATOR | OPERATOR OPERATORLIST

OPERATOR -> BLOCK | ID(PARAMETERS) | for ID := VALUE DIRECTION VALUE do OPERATOR | while EXPRESSION do OPERATOR | repeat OPERATOR until EXPRESSION ; | if EXPRESSION then OPERATOR IFTAIL | goto ID | ID := EXPRESSION

IFTAIL -> else OPERATOR | eps

DIRECTION -> to | downto

CONDITION -> ( EXPRESSION = EXPRESSION ) | ( EXPRESSION < EXPRESSION ) | ( EXPRESSION > EXPRESSION ) | ( EXPRESSION <= EXPRESSION ) | ( EXPRESSION >= EXPRESSION ) | ( EXPRESSION <> EXPRESSION )

EXPRESSION -> SimpleExpression | CONDITION

SimpleExpression -> Term | SimpleExpression + Term | SimpleExpression - Term | SimpleExpression or Term

Term -> Factor | Term \* Factor | Term / Factor | Term div Factor | Term and Factor

Factor -> ID | VALUE | ( EXPRESSION ) | not Factor

## **Ввод вывод**

В стандартном Pascal встроены процедуры для работы с вводом и выводом (read, write, readln, writeln), которые компилятор должен поддерживать. Эти операции также включают форматированный вывод, что добавляет сложности в реализации.

## 1.4 Формальная постановка задачи

Необходимо разработать прототип компилятора языка программирования Pascal. На вход компилятор принимает исходный код программы на языке Pascal. В процессе работы компилятор генерирует код на языке MASM. Результатом работы компилятора является исполняемый файл полученный с помощью компилятора MASM. На рисунке 1.1 представлена IDEF0 диаграмма компилятора языка программирования Pascal.

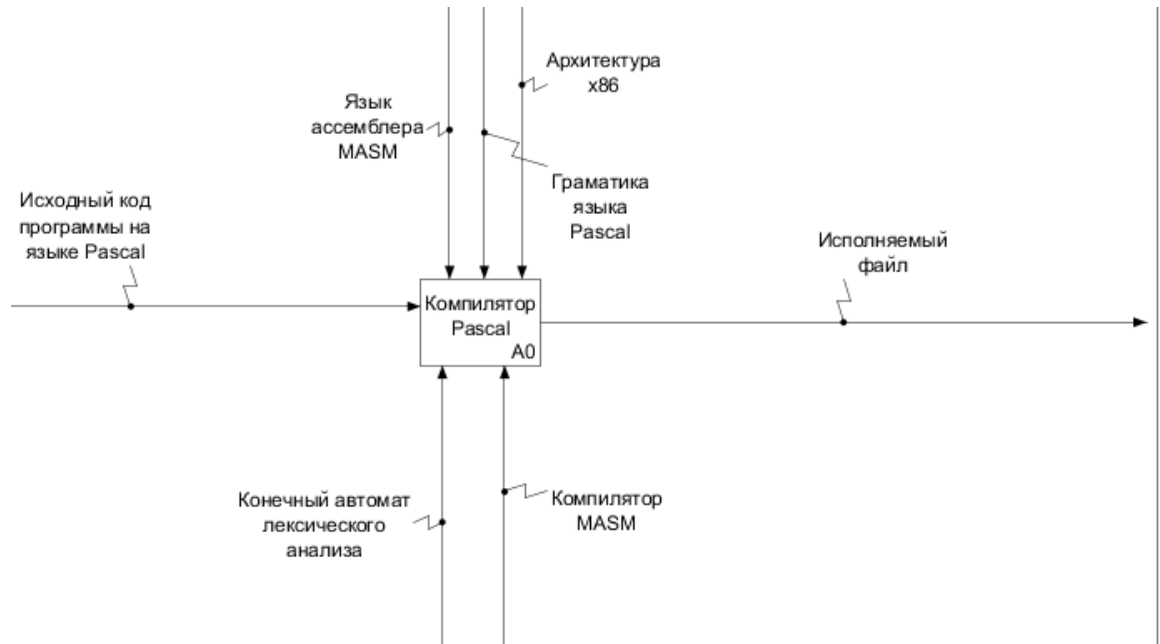


Рисунок 1.1 — IDEF0 диаграмма компилятора языка программирования Pascal

## Вывод

В ходе исследования был рассмотрен процесс построения компилятора, включая сравнительный анализ способов синтаксического анализа. Были выявлены особенности языка программирования Pascal, была сформулирована задача в виде IDEF0 диаграммы компилятора.

## 2 Конструкторский раздел

### 2.1 Концептуальная модель

На рисунке 2.1 представлен схема процесса компиляции языка программирования Pascal.

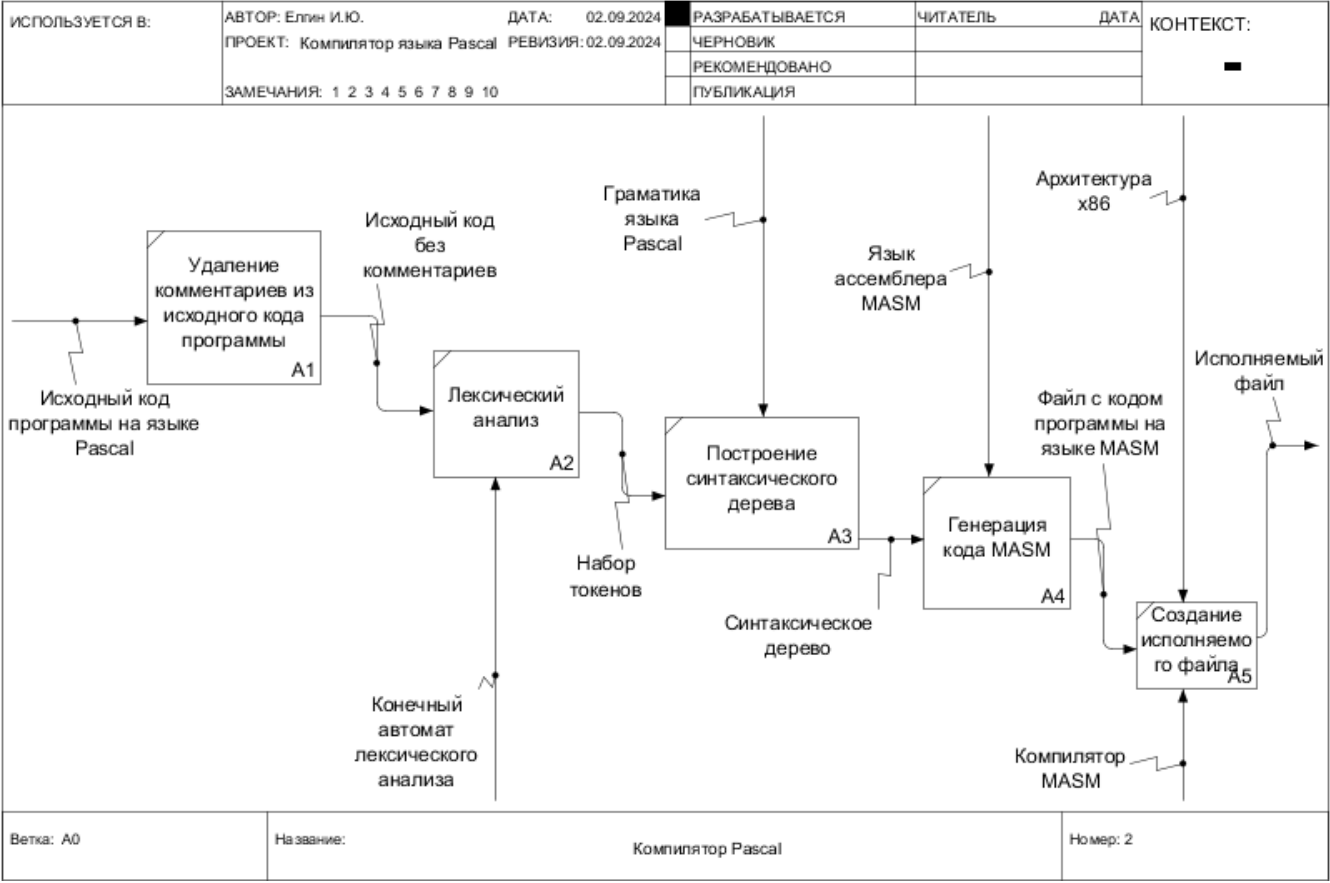


Рисунок 2.1 — Детализированная IDEF0 диаграмма компилятора языка программирования Pascal

Данная диаграмма разделяет компилятор на ряд компонентов. Первый этап (предкомпиляция) удаляет однострочные и многострочные комментарии. Далее код без комментариев передается лексическому анализатору, разделяющему исходный код на лексемы. Далее синтаксический анализатор по последовательности лексем строит AST(синтаксическое)-дерево. По AST-дереву происходит генерация кода MASM. По данному коду компилятор MASM создает исполняемый файл.

### 2.2 Лексический анализ

В листинге 2.1 приведен конечный автомат для лексического анализа программ на языке Pascal. При переходе автомата в конечное состояние происходит выделение лексемы, от того из какого состояния был совершен переход в конечное лексемы подразделяются на: NUMBER — числовые, STRING — строковые, SPECIAL — спецсимволы, WORD — ключевые слова и идентификаторы, так как в Pascal идентификаторы не могут совпадать с ключевыми словами, то определить является ли данная лексема идентификатором можно сравнив ее с ключевыми словами.

```

Q = {S, A, DIGIT, NUMBER, LETTER, WORD, STRING, SPECIAL, SPECIAL2, EOL}

Σ = {DIGIT (0-9), LETTER (a-z), SPECIAL ('(', ')', '+', '-', '*', '/', '<', '>', ':', '=', ';'), SPACE (' ', \n), ' }

q0 = S

F = {EOL}

δ = {
(S) - [DIGIT] -> (A)
(A) - [DIGIT] -> (DIGIT)
(DIGIT) - [DIGIT] -> (NUMBER)
(NUMBER) - [DIGIT] -> (NUMBER)
(NUMBER) - [SPACE] -> (EOL)
(NUMBER) - [SPECIAL] -> (EOL)

(S) - [LETTER] -> (A)
(A) - [LETTER] -> (LETTER)
(LETTER) - [LETTER] -> (WORD)
(LETTER) - [DIGIT] -> (WORD)
(WORD) - [LETTER] -> (WORD)
(WORD) - [DIGIT] -> (WORD)
(WORD) - [SPACE] -> (EOL)
(WORD) - [SPECIAL] -> (EOL)

(S) - [''] -> (STRING)
(STRING) - [LETTER] -> (STRING)
(STRING) - [DIGIT] -> (STRING)
(STRING) - [SPECIAL] -> (STRING)
(STRING) - [SPACE] -> (STRING)
(STRING) - [''] -> (EOL)

(S) - [SPECIAL] -> (SPECIAL)
(SPECIAL) - [< or >] -> (SPECIAL2)
(SPECIAL2) - [=] -> (EOL)
(SPECIAL2) - [SPACE] -> (EOL)
}

```

## 2.3 Синтаксический разбор

Для построения синтаксического дерева используется LR(1)-анализатор. Множества FIRST и FOLLOW, а также символы перехода и правила приведены в приложении А.

По синтаксическому разбору строится синтаксическое дерево в таблице 1.1 приведены узлы этого дерева и их дочерние узлы.

Таблица 2.1 — Узлы синтаксического дерева

| Структурный узел | Дочерние узлы  |
|------------------|--|
| Счетный цикл for | Целочисленная переменная, значение начала счетного цикла, направление цикла, значение окончания счетного цикла, тело цикла |

Продолжение таблицы 2.1 — Узлы синтаксического дерева

|   |  |
|---|--|
| Цикл с предусловием while               | Условие продолжение цикла, тело цикла  |
| Цикл с постусловием repeat              | Условие продолжения цикла, тело цикла  |
| Оператор условного перехода if          | Условие перехода, оператор или блок операторов, исполняемых при выполнении условия, [оператор или блок операторов исполняемых при невыполнении условия (не обязательный узел)] |
| Блок операторов                         | Список операторов  |
| Константы                               | Список констант  |
| Переменные                              | Список переменных с типами   |
| Типы                                    | Список типов   |
| Метки                                   | Список меток   |
| Операторы сравнения <, >, <=, >=, =, <> | Выражение слева, выражение справа  |
| Операторы сложения +, -, or             | Выражение слева, выражение справа  |
| Операторы умножения *, /, div, and      | Выражение слева, выражение справа  |
| Оператор отрицания not                  | Выражение справа   |
| Оператор присваивания :=                | Выражение слева, выражение справа  |
| Функция                                 | Идентификатор функции, список параметров   |
| Выражение                               | Число или (выражение)  |
| Параметр                                | Выражение  |

## 2.4 Генерация кода

Генерация кода совмещена с семантическим анализом, при генерации кода происходит рекурсивный спуск по синтаксическому дереву, при этом проверяется правильность конструкций и соответствие типов в программе.

В начале генерации выделяется место под переменные и массивы в разделе var, константы в разделе const и тексте программы и строки.

### Вывод

В ходе конструирования компилятора была разработана концептуальная модель компилятора языка программирования Pascal. Был приведен детерминированный конечный автомат для лексического анализатора. Рассмотрены вопросы построения синтаксического дерева программы.

## 3 Технологический раздел

### 3.1 Выбор средств реализации

В ходе разработки было решено использовать язык программирования C#. Данный язык позволяет легко реализовать структуру синтаксического дерева и символических таблиц, используя классы и коллекции .NET. Это упрощает управление данными и их обработку на этапах синтаксического и семантического анализа. С использованием строковых функций и шаблонов, доступных в C#, можно эффективно генерировать текстовые представления ассемблерного кода, соответствующие стандартам MASM [11].

### Структура проекта

На рисунке 3.1 представлен граф зависимостей модулей в проекте.

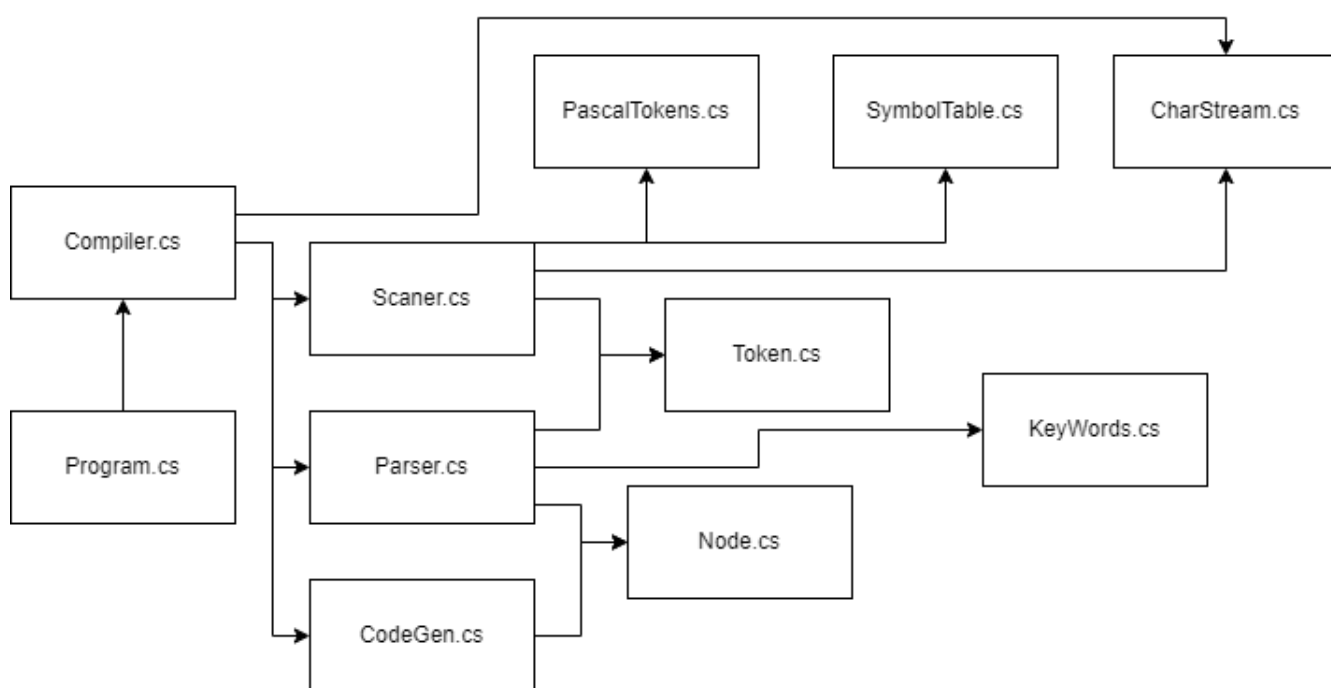


Рисунок 3.1 — Граф зависимостей модулей в проекте

Модули Scanner.cs, Parser.cs и CodeGen.cs реализуют лексический анализатор, синтаксический анализатор и код генератор соответственно. Классы для узлов синтаксического дерева находятся в модуле Node.cs. Класс лексем находится в модуле Token.cs, а ключевые слова языка Pascal в KeyWords.cs

### 3.2 Средства ввода вывода

Для ввода и вывода в программах на Pascal используются функции стандартной библиотеки read, write, writeln. Эти функции принимают неограниченное количество параметров и в соответствии с их типом осуществляют их ввод и вывод.

Для реализации этих функций в MASM воспользуемся библиотекой *C Runtime Library* (CRT) реализующая функции crt\_printf и crt\_scanf [11].

crt\_printf принимает на вход строку формата и параметры для вывода.

crt\_scanf принимает на вход строку формата и адреса переменных ввода.

Каждый параметр передаваемый write или writeln будет отдельно выводиться с помощью crt\_printf, в соответствии с типом для его вывода будет использоваться следующие строки формата:

- целочисленный — «%d»;
- дробный — «%f»;
- символ — «%c»;
- строка — «%s».

Для writeln в конце будет также добавляться символ новой строки.

### 3.3 Исчисление выражений

Для исчисления выражений используется алгоритм простой стековой машины, при рекурсивном спуске по синтаксическому дереву, когда очередным элементом выражения является константа или переменная ее значение помещается в стек если идет обращение к элементу массива, то сначала вычисляется адрес этого элемента с учетом смещения, после чего его значение помещается в стек, если очередной узел это оператор и его дочерние элементы обработаны, то из стека достается два значения и происходит соответствующая операция с ней после чего результат помещается в стек. В случае операторов сравнения результатом является 1 при выполнении условия, 0 в ином случае.

### 3.4 Работа с дробными числами

Для работы с дробными числами необходимо использовать команды сопроцессора.

Microsoft Macro Assembler (MASM) предоставляет программистам возможность работать с дробными числами, используя инструкции сопроцессора x87, который реализует арифметические операции с плавающей запятой. Работа с дробными числами является неотъемлемой частью многих вычислительных задач, особенно в области научных расчетов, графики и других приложений, где требуется высокая точность [8].

Дробные числа в MASM представлены в формате с плавающей запятой, который поддерживается стандартом IEEE 754. Сопроцессор x87 работает с числами в трех основных форматах:

- **Single Precision (32-битное представление):** включает 1 бит для знака, 8 бит для экспоненты и 23 бита для мантиссы;
- **Double Precision (64-битное представление):** включает 1 бит для знака, 11 бит для экспоненты и 52 бита для мантиссы;
- **Extended Precision (80-битное представление):** включает 1 бит для знака, 15 бит для экспоненты и 64 бита для мантиссы.

#### 3.4.1 Основные инструкции работы с дробными числами

MASM поддерживает набор инструкций для выполнения операций с плавающей запятой, таких как сложение, вычитание, умножение, деление и другие. Ниже приведены инструкции, используемые для работы с дробными числами:

- **FADD:** сложение двух дробных чисел;
- **FSUB:** вычитание одного дробного числа из другого;
- **FMUL:** умножение двух дробных чисел;



- **FDIV**: деление одного дробного числа на другое;
- **FLD**: загрузка числа с плавающей запятой в регистр сопроцессора;
- **FST**: сохранение числа с плавающей запятой из регистра в память;
- **FXCH**: обмен значениями двух регистров сопроцессора;

### 3.5 Тестирование компилятора

Были созданы несколько программ, которые соответствуют грамматике языка. Они предназначены для проверки правильности работы различных конструкций языка. Данные программы охватывают следующие сценарии:

- создание переменных и констант;
- ввод и вывод;
- работа с целыми и вещественными числами;
- работа с константными строками;
- создание ветвлений программы;
- создание циклов;
- работа с массивами фиксированного размера;
- логические операции, ленивые логические вычисления.

В листинге 3.1 представлена программа осуществляющая ввод массива и сортирующая его по не возрастанию. В приложении Б представлен сгенерированный код ассемблера MASM этой программы. На рисунке 3.2 приведена работа компилятора и скомпилированного им приложения.

Листинг 3.1 — Пример программы реализующую сортировку пузырьком

```

program BUBBLE_SORT(input, output);
TYPE
  IntArray = array[1..5000] of integer;

VAR
  i, j, n, c, c2, n1: integer;
  numbers : IntArray;

begin
  writeln('Input digit count');
  read(n);
  n1 := n - 1;
  if (n <= 0) then writeln('less then allow')
  else if (n >= 5000) then writeln('too much')
  else
    begin
      i := 1;
      while (i <= n) do
        begin
          read(c);
          numbers[i] := c;
          i := i + 1;
        end;
      i := 0;
    
```

### Продолжение листинга 3.1 — Пример программы реализующую сортировку пузырьком

```

while (i <= n1) do
begin
    i := i + 1;
    j := i + 1;
    while (j <= n) do
    begin
        if numbers[i] < numbers[j] then
        begin
            c := numbers[i];
            numbers[i] := numbers[j];
            numbers[j] := c;
        end;
        j := j + 1;
    end;

    end;
    i := 0;
    writeln('sorted array:');
    while (i < n) do
    begin
        i := i + 1;
        writeln(numbers[i]);
    end;
end;
end.

```

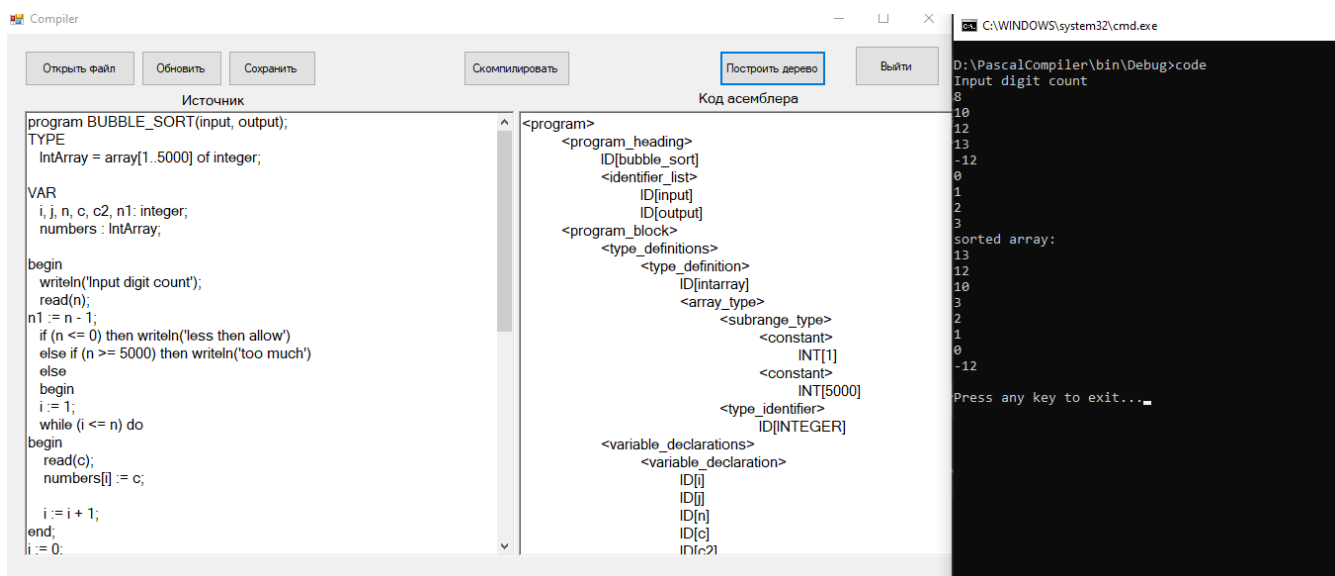


Рисунок 3.2 — Пример работы компилятора и программы

В листинге 3.2 приведен код программы для проверки работы с дробными числами и константами результат работы скомпилированного приложения изображен на рисунке 3.3.

### Листинг 3.2 — Пример проверки работы с дробными числами и константами

```

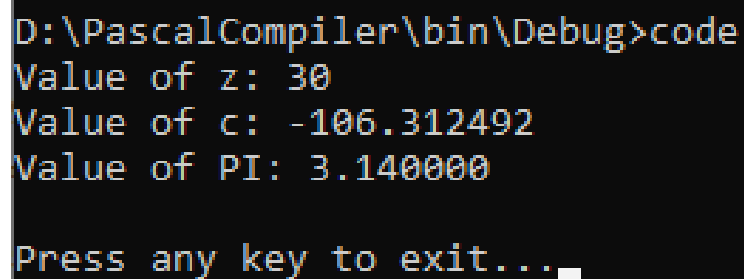
program number(output);
const
    PI = 3.14;

```

```
var
  x, y, z : integer;
  a, b, c : real;
begin
  a := 5.5;
  b := 12.2-a;
  c := a-b * a+b + PI;

  x := -15;
  y := (17+x) * (10-6);
  z := y div x + 15 * 2;
  c := c+ a*x + b/y;

  writeln('Value of z: ',z);
  writeln('Value of c: ',c);
  writeln('Value of PI: ',PI);
end.
```



```
D:\PascalCompiler\bin\Debug>code
Value of z: 30
Value of c: -106.312492
Value of PI: 3.140000

Press any key to exit..._
```

Рисунок 3.3 — Пример работы программы проверки работы с дробными числами и константами

## ЗАКЛЮЧЕНИЕ

В результате проведенной работы был осуществлен анализ предметной области разработки компилятора языка программирования Pascal. Были рассмотрены ключевые аспекты языка Pascal, проведено сравнительное изучение синтаксических анализаторов. Был проведён анализ этапов компиляции кода. Данный анализ позволил сформулировать формальную задачу разработки компилятора Pascal.

В ходе исследования была разработана концептуальная модель компилятора, определены детерминированный конечный автомат для лексического анализатора и LR(1)-анализатор для построения синтаксического дерева.

Был проведен выбор языка программирования C# для реализации проекта, это обусловлено его мощными возможностями и удобством работы со строками. Разработанные программы для тестирования функциональности компилятора охватывают разнообразные языковые конструкции и проведено успешное тестирование компилятора.

Поставленные задачи были решены. Цель работы достигнута: был разработан прототип компилятора Pascal. Полученные результаты являются основой для дальнейшей работы над развитием и оптимизацией данного компилятора.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Компиляторы: принципы, технологии и инструментарий / Ахо А, Лам М, Сети Рави и Ульман Дж // М.: Вильямс. — 2008. — Т. 1. — С. 257.
2. Вирт Никлаус. Алгоритмы + структуры данных = программы. — М.: Мир, 1985. — 368 с.
3. Вирт Никлаус. Проектирование компиляторов (Compiler Construction). — СПб.: Питер, 2005. — 192 с.
4. Ахо А Ульман Дж. Теория синтаксического анализа, перевода и компиляции. Синтаксический анализ. — 1978.
5. Козлов Виктор Александрович. Технология разработки трансляторов. — М.: Издательство МГТУ им. Н. Э. Баумана, 2010. — 672 с.
6. Уотсон Бенджамин Льюис, Хадсон Джеймс М. Компиляторы для современных архитектур. — М.: ДМК Пресс, 2009. — 464 с.
7. Кадомский Андрей Андреевич, Сабинин Олег Юрьевич. ИССЛЕДОВАНИЕ ВОЗМОЖНОСТИ СОЗДАНИЯ УНИВЕРСАЛЬНОГО ЯЗЫКА ПРОГРАММИРОВАНИЯ ВЫСОКОГО УРОВНЯ // Theoretical & Applied Science. — 2018. — № 5. — С. 84–90.
8. MASM32 SDK. [электронный источник] — URL: <https://www.masm32.com/> (дата обращения: 12.06.2024).
9. ISO/IEC 7185:1990. Информационные технологии. Языки программирования. Паскаль. Женева: Международная организация по стандартизации, 1990. 89 с.
10. Глушков Виктор Михайлович, Михеев Александр Алексеевич. Язык Паскаль и его реализации. — М.: Наука, 1983. — 320 с.
11. Руководство по ассемблеру MASM. [электронный источник] — URL: <https://metanit.com/assembler/tutorial/> (дата обращения: 20.06.2024).

## ПРИЛОЖЕНИЕ А — Множества FIRST и FOLLOW и таблица синтаксического анализатора

- **FIRST(START):** {program}
- **FIRST(HEADERTYPE):** {type, const, lable, var}
- **FIRST(TYPEID):** {integer, real, char, string, boolean, array}
- **FIRST(TYPELIST):** {ID, ;}
- **FIRST(HEADERCONST):** {const, lable, var}
- **FIRST(CONSTLIST):** {ID, ;}
- **FIRST(HEADERLABLE):** {lable, var}
- **FIRST(LABLELIST):** {ID, ;}
- **FIRST(HEADERVER):** {var}
- **FIRST(VARLIST):** {ID, ;}
- **FIRST(IDLIST):** {ID}
- **FIRST(BLOCK):** {begin}
- **FIRST(OPERATORLIST):** {begin, ID, for, while, repeat, if, goto, ID}
- **FIRST(OPERATOR):** {begin, ID, for, while, repeat, if, goto, ID}
- **FIRST(IFTAIL):** {else, eps}
- **FIRST(DIRECTION):** {to, downto}
- **FIRST(CONDITION):** { ( }
- **FIRST(EXPRESSION):** {ID, VALUE, (, not}
- **FIRST(SimpleExpression):** {ID, VALUE, (, not}
- **FIRST(Term):** {ID, VALUE, (, not}
- **FIRST(Factor):** {ID, VALUE, (, not}

### FOLLOW множества:

- **FOLLOW(START):** {EOF}
- **FOLLOW(HEADERTYPE):** {EOF}
- **FOLLOW(TYPEID):** {EOF, ;, ID, const, lable, var}
- **FOLLOW(TYPELIST):** {EOF, ID, ;}
- **FOLLOW(HEADERCONST):** {EOF, lable, var}
- **FOLLOW(CONSTLIST):** {EOF, lable, var}
- **FOLLOW(HEADERLABLE):** {EOF}
- **FOLLOW(LABLELIST):** {EOF}
- **FOLLOW(HEADERVER):** {EOF}
- **FOLLOW(VARLIST):** {EOF}
- **FOLLOW(IDLIST):** {EOF, ;}
- **FOLLOW(BLOCK):** {EOF}
- **FOLLOW(OPERATORLIST):** {EOF}
- **FOLLOW(OPERATOR):** {EOF}
- **FOLLOW(IFTAIL):** {EOF}
- **FOLLOW(DIRECTION):** {EOF}
- **FOLLOW(CONDITION):** {EOF, then, do, until}
- **FOLLOW(EXPRESSION):** {EOF, =, <, >, <=, >=, <>, then, do, until}
- **FOLLOW(SimpleExpression):** {EOF, +, -, or, =, <, >, <=, >=, <>, then, do, until}
- **FOLLOW(Term):** {EOF, +, -, or, =, <, >, <=, >=, <>, then, do, until}
- **FOLLOW(Factor):** {EOF, \*, /, div, and, +, -, or, =, <, >, <=, >=, <>, then, do, until}

## Синтаксический анализатор

**START** *program* START → *program* ID ; HEADERTYPE  
**HEADERTYPE** *type* HEADERTYPE → *type* TYPELIST HEADERCONST  
**HEADERTYPE** *const* HEADERTYPE → HEADERCONST  
**HEADERTYPE** *lable* HEADERTYPE → HEADERLABLE  
**HEADERTYPE** *var* HEADERTYPE → HEADERVER  
**TYPEID** *integer* TYPEID → *integer*  
**TYPEID** *real* TYPEID → *real*  
**TYPEID** *char* TYPEID → *char*  
**TYPEID** *string* TYPEID → *string*  
**TYPEID** *boolean* TYPEID → *boolean*  
**TYPEID** *array* TYPEID → *array* [ VALUE .. VALUE ] of TYPEID  
**TYPELIST** ID TYPELIST → ID = TYPEID TYPELIST  
**TYPELIST** ; TYPELIST → ; ID = TYPEID TYPELIST  
**TYPELIST** ; TYPELIST → ;  
**HEADERCONST** *const* HEADERCONST → *const* CONSTLIST HEADERLABLE  
**HEADERCONST** *lable* HEADERCONST → HEADERLABLE  
**CONSTLIST** ID CONSTLIST → ID = VALUE CONSTLIST  
**CONSTLIST** ; CONSTLIST → ; ID = VALUE CONSTLIST  
**CONSTLIST** ; CONSTLIST → ;  
**HEADERLABLE** *lable* HEADERLABLE → *lable* LABELIST  
**HEADERLABLE** *var* HEADERLABLE → HEADERVER  
**LABELIST** ID LABELIST → ID LABELIST  
**LABELIST** ; LABELIST → ; ID LABELIST  
**LABELIST** ; LABELIST → ;  
**HEADERVER** *var* HEADERVER → *var* VARLIST BLOCK  
**VARLIST** ID VARLIST → ID , IDLIST VARLIST  
**VARLIST** ; VARLIST → ; ID , IDLIST VARLIST  
**VARLIST** ID VARLIST → ID : TYPEID VARLIST  
**VARLIST** ; VARLIST → ; ID : TYPEID VARLIST  
**VARLIST** ; VARLIST → ;  
**IDLIST** ID IDLIST → ID , IDLIST  
**IDLIST** ID IDLIST → ID : TYPEID  
**BLOCK** *begin* BLOCK → *begin* OPERATORLIST *end*  
**OPERATORLIST** *begin* OPERATORLIST → OPERATOR OPERATORLIST  
**OPERATORLIST** ID OPERATORLIST → OPERATOR OPERATORLIST  
**OPERATOR** *begin* OPERATOR → BLOCK  
**OPERATOR** ID OPERATOR → ID ( PARAMETRS )  
**OPERATOR** *for* OPERATOR → *for* ID := VALUE DIRECTION VALUE *do* OPERATOR

**OPERATOR** *while* OPERATOR  $\rightarrow$  while EXPRESSION do OPERATOR  
**OPERATOR** *repeat* OPERATOR  $\rightarrow$  repeat OPERATOR until EXPRESSION ;  
**OPERATOR** *if* OPERATOR  $\rightarrow$  if EXPRESSION then OPERATOR IFTAIL  
**OPERATOR** *goto* OPERATOR  $\rightarrow$  goto ID  
**OPERATOR** ID OPERATOR  $\rightarrow$  ID := EXPRESSION  
**IFTAIL** *else* IFTAIL  $\rightarrow$  else OPERATOR  
**IFTAIL** *eps* IFTAIL  $\rightarrow$  eps  
**DIRECTION** *to* DIRECTION  $\rightarrow$  to  
**DIRECTION** *downto* DIRECTION  $\rightarrow$  downto  
**CONDITION** ( CONDITION  $\rightarrow$  ( EXPRESSION = EXPRESSION )  
**CONDITION** ( CONDITION  $\rightarrow$  ( EXPRESSION < EXPRESSION )  
**CONDITION** ( CONDITION  $\rightarrow$  ( EXPRESSION > EXPRESSION )  
**CONDITION** ( CONDITION  $\rightarrow$  ( EXPRESSION <= EXPRESSION )  
**CONDITION** ( CONDITION  $\rightarrow$  ( EXPRESSION >= EXPRESSION )  
**CONDITION** ( CONDITION  $\rightarrow$  ( EXPRESSION <> EXPRESSION )  
**EXPRESSION** ID EXPRESSION  $\rightarrow$  SimpleExpression  
**EXPRESSION** VALUE EXPRESSION  $\rightarrow$  SimpleExpression  
**EXPRESSION** ( EXPRESSION  $\rightarrow$  SimpleExpression  
**EXPRESSION** *not* EXPRESSION  $\rightarrow$  SimpleExpression  
**SIMPLEEXPRESSION** ID SIMPLEEXPRESSION  $\rightarrow$  Term  
**SIMPLEEXPRESSION** VALUE SIMPLEEXPRESSION  $\rightarrow$  Term  
**SIMPLEEXPRESSION** ( SIMPLEEXPRESSION  $\rightarrow$  Term  
**SIMPLEEXPRESSION** *not* SIMPLEEXPRESSION  $\rightarrow$  Term  
**TERM** ID TERM  $\rightarrow$  Factor  
**TERM** VALUE TERM  $\rightarrow$  Factor  
**TERM** ( TERM  $\rightarrow$  Factor  
**TERM** *not* TERM  $\rightarrow$  Factor  
**FACTOR** ID FACTOR  $\rightarrow$  ID  
**FACTOR** VALUE FACTOR  $\rightarrow$  VALUE  
**FACTOR** ( FACTOR  $\rightarrow$  ( EXPRESSION )  
**FACTOR** *not* FACTOR  $\rightarrow$  not Factor



## ПРИЛОЖЕНИЕ Б — код ассемблера программы сортировки массива

```
TITLE BUBBLE_SORT (BUBBLE_SORT.asm)
include masm32rt.inc

.data
;System vairables
format$Char db "%c",0
format$String db "%s",0
format$Int db "%d",0
format$Float db "%f",0
out_real8 real8 ?
temp_real4 real4 ?
temp_sdword sdword ?

;Program vairables
var$_$i SDWORD ?
var$_$j SDWORD ?
var$_$n SDWORD ?
var$_$c SDWORD ?
var$_$c2 SDWORD ?
var$_$n1 SDWORD ?
arr$_$numbers DB 20000 DUP (?)
pStr$1 BYTE "Input digit count",0
pStr$2 BYTE "less then allow",0
pStr$3 BYTE "too much",0
pStr$4 BYTE "sorted array:",0

.code
start:

mov EDX, OFFSET pStr$1
invoke crt_printf, addr format$String, EDX
invoke crt_printf, ADDR format$Char, 10
invoke crt_scanf, addr format$Int, addr var$_$n
MOV EAX, var$_$n
SUB EAX, 1
PUSH EAX

POP var$_$n1

MOV EAX, var$_$n
MOV EBX, 0
CMP EAX, EBX
JLE @@lblCondTrue1
push 0
jmp @@lblCondFalse2

@@lblCondTrue1:
push 1

@@lblCondFalse2:
pop eax
cmp eax, 1
je @@syslbl_3
jmp @@syslbl_4

@@syslbl_3:
mov EDX, OFFSET pStr$2
invoke crt_printf, addr format$String, EDX
invoke crt_printf, ADDR format$Char, 10
jmp @@syslbl_5

@@syslbl_4:
MOV EAX, var$_$n
```

```

MOV EBX, 5000
CMP EAX, EBX
JGE @@lblCondTrue6
push 0
jmp @@lblCondFalse7

@@lblCondTrue6:
push 1

@@lblCondFalse7:
pop eax
cmp eax, 1
je @@syslbl_8
jmp @@syslbl_9

@@syslbl_8:
mov EDX, OFFSET pStr$3
invoke crt_printf, addr format$String, EDX
invoke crt_printf, ADDR format$Char, 10
jmp @@syslbl_10

@@syslbl_9:
MOV var$_$i, 1

mov eax, dword ptr [var$_$i]
mov ebx, dword ptr [var$_$n]
cmp eax, ebx
JLE @@syslbl_wstart11
jmp @@syslbl_wend12

@@syslbl_wstart11:
invoke crt_scanf, addr format$Int, addr var$_$c

push eax
push ebx
mov eax, var$_$i
mov ebx, 1
sub eax, ebx
mov ebx, 4
mul ebx
mov ebx, offset arr$_$numbers
add eax, ebx
mov edi, eax
pop ebx
pop eax
PUSH var$_$c
POP dword ptr [edi]

MOV EAX, var$_$i
ADD EAX, 1
PUSH EAX

POP var$_$i

mov eax, dword ptr [var$_$i]
mov ebx, dword ptr [var$_$n]
cmp eax, ebx
JLE @@syslbl_wstart11
jmp @@syslbl_wend12

@@syslbl_wend12:
MOV var$_$i, 0

mov eax, dword ptr [var$_$i]
mov ebx, dword ptr [var$_$n1]
cmp eax, ebx

```

```

JLE @@syslbl_wstart13
jmp @@syslbl_wend14

@@syslbl_wstart13:
MOV EAX, var$_$i
ADD EAX, 1
PUSH EAX

POP var$_$i

MOV EAX, var$_$i
ADD EAX, 1
PUSH EAX

POP var$_$j

mov eax, dword ptr [var$_$j]
mov ebx, dword ptr [var$_$n]
cmp eax, ebx
JLE @@syslbl_wstart15
jmp @@syslbl_wend16

@@syslbl_wstart15:

push eax
push ebx
mov eax, var$_$i
mov ebx, 1
sub eax, ebx
mov ebx, 4
mul ebx
mov ebx, offset arr$_$numbers
add eax, ebx
mov esi, eax
pop ebx
pop eax
MOV EAX, dword ptr [esi]

push eax
push ebx
mov eax, var$_$j
mov ebx, 1
sub eax, ebx
mov ebx, 4
mul ebx
mov ebx, offset arr$_$numbers
add eax, ebx
mov esi, eax
pop ebx
pop eax
MOV EBX, dword ptr [esi]
CMP EAX, EBX
JL @@lblCondTrue17
push 0
jmp @@lblCondFalse18

@@lblCondTrue17:
push 1

@@lblCondFalse18:
pop eax
cmp eax, 1
je @@syslbl_19
jmp @@syslbl_20

@@syslbl_19:

```

```

push eax
push ebx
mov eax, var$_$i
mov ebx, 1
sub eax, ebx
mov ebx, 4
mul ebx
mov ebx, offset arr$_$numbers
add eax, ebx
mov esi, eax
pop ebx
pop eax
PUSH dword ptr [esi]
POP var$_$c

```

```

push eax
push ebx
mov eax, var$_$i
mov ebx, 1
sub eax, ebx
mov ebx, 4
mul ebx
mov ebx, offset arr$_$numbers
add eax, ebx
mov edi, eax
pop ebx
pop eax

```

```

push eax
push ebx
mov eax, var$_$j
mov ebx, 1
sub eax, ebx
mov ebx, 4
mul ebx
mov ebx, offset arr$_$numbers
add eax, ebx
mov esi, eax
pop ebx
pop eax
PUSH dword ptr [esi]
POP dword ptr [edi]

```

```

push eax
push ebx
mov eax, var$_$j
mov ebx, 1
sub eax, ebx
mov ebx, 4
mul ebx
mov ebx, offset arr$_$numbers
add eax, ebx
mov edi, eax
pop ebx
pop eax
PUSH var$_$c
POP dword ptr [edi]

```

```

jmp @@syslbl_21

```

```

@@syslbl_20:

```

```

@@syslbl_21:
MOV EAX, var$_$j
ADD EAX, 1
PUSH EAX

```

```

POP var$_$j

mov eax, dword ptr [var$_$j]
mov ebx, dword ptr [var$_$n]
cmp eax, ebx
JLE @@syslbl_wstart15
jmp @@syslbl_wend16

@@syslbl_wend16:
mov eax, dword ptr [var$_$i]
mov ebx, dword ptr [var$_$n1]
cmp eax, ebx
JLE @@syslbl_wstart13
jmp @@syslbl_wend14

@@syslbl_wend14:
MOV var$_$i, 0

mov EDX, OFFSET pStr$4
invoke crt_printf, addr format$String, EDX
invoke crt_printf, ADDR format$Char, 10
mov eax, dword ptr [var$_$i]
mov ebx, dword ptr [var$_$n]
cmp eax, ebx
JL @@syslbl_wstart22
jmp @@syslbl_wend23

@@syslbl_wstart22:
MOV EAX, var$_$i
ADD EAX, 1
PUSH EAX

POP var$_$i

push eax
push ebx
mov eax, var$_$i
mov ebx, 1
sub eax, ebx
mov ebx, 4
mul ebx
mov ebx, offset arr$_$numbers
add eax, ebx
mov esi, eax
pop ebx
pop eax
invoke crt_printf, addr format$Int, dword ptr [esi]
invoke crt_printf, ADDR format$Char, 10
mov eax, dword ptr [var$_$i]
mov ebx, dword ptr [var$_$n]
cmp eax, ebx
JL @@syslbl_wstart22
jmp @@syslbl_wend23

@@syslbl_wend23:

@@syslbl_10:

@@syslbl_5:

invoke crt_printf, addr format$Char, 10
inkey "Press any key to exit..."
exit
end start

```

