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

**(БГТУ «ВОЕНМЕХ» им. Д.Ф. Устинова»)**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Факультет |  | И |  | «Информационные управляющие системы» |
|  |  | шифр |  | наименование |
| Кафедра |  | 09.03.01 |  | «Систем управления и компьютерных технологий» |
|  |  | шифр |  | наименование |
| Дисциплина |  | «Системное программное обеспечение» | | |

КУРСОВАЯ РАБОТА

на тему

|  |
| --- |
| «Компилятор с языка Modula 2 на язык Assembler» |
|  |
|  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Выполнил студент группы | | | | |  | И-965 |
| Камнев Егор Олегович | | | | | | |
| Фамилия И.О. | | | | | | |
| **РУКОВОДИТЕЛЬ** | | | | | | |
| Тарасов С.Д. | |  |  | | | |
| Фамилия И.О. Подпись | | | | | | |
| Оценка |  | | |  | | |
| «\_\_\_\_\_» |  | | | 2018\_г. | | |

САНКТ-ПЕТЕРБУРГ

2018 г.

Содержание

|  |  |
| --- | --- |
| Определения, обозначения и сокращения…………………………………... | 3 |
| Введение……………………………………………………………………….. | 4 |
| 1 Теоретические основы……………………………………………………… | 8 |
| 1.1 Описание подмножества языка Modula-2……………………………... | 8 |
| 1.1.1 Лексика языка……………………………………………………….. | 8 |
| 1.1.2 Тип грамматики……………………………………………………... | 9 |
| 1.1.3 Синтаксические диаграммы в формате БНФ……………………... | 9 |
| 1.2 Описание выходного языка(MASM)…………………………………... | 12 |
| 2 Архитектура компилятора………………………………………………….. | 15 |
| 2.1 Лексический анализатор………………………………………………... | 15 |
| 2.1.1 Токены……………………………………………………………….. | 17 |
| 2.2 Синтаксический анализатор……………………………………………. | 17 |
| 2.2.1 Дерево разбора……………………………………………………… | 18 |
| 2.3 Генератор кода………………………………………………………….. | 19 |
| 2.4 Диаграмма классов……………………………………………………… | 20 |
| 3 Тестовые примеры………………………………………………………….. | 21 |
| 3.1 Решение полинома……………………………………………………… | 21 |
| 3.2 Подсчет факториала…………………………………………………….. | 22 |
| Заключение……………………………………………………………………. | 24 |
| Список использованных источников………………………………………... | 25 |
| Листинг программы…………………………………………………………... | 26 |

## ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

**Ассе́мблер** (от англ. *assembler* —сборщик) — транслятор исходного текста программы, написанной на языке ассемблера, в программу на машинном языке.

**Компиля́ция —**трансляция программы, составленной на исходном языке высокого уровня, в эквивалентную программу на низкоуровневом языке, близком машинному коду (абсолютный код, объектный модуль, иногда на язык ассемблера)

**Форма Бэкуса — Наура**(сокр. БНФ, Бэкуса — Наура форма) — формальная система описания синтаксиса, в которой одни синтаксические категории последовательно определяются через другие категории. БНФ используется для описания контекстно-свободных формальных грамматик.

**ПО** — программное обеспечение.

**ОС** — операционная система.

**ЛА** – лексический анализатор.

**СА** – синтаксический анализатор.

**ГК** – генератор кода .

## ВВЕДЕНИЕ

Формально годом рождения теории компиляторов можно считать 1957, когда появился первый компилятор языка Фортран, созданный Бэкусом и дающий достаточно эффективный объектный код. До этого времени создание компиляторов было весьма «творческим» процессом. Лишь появление теории формальных языков и строгих математических моделей позволило перейти от «творчества» к «науке». Именно благодаря этому, стало возможным появление сотен новых языков программирования. Несмотря на то, что к настоящему времени разработаны тысячи различных языков и их компиляторов, процесс создания новых приложений в этой области не прекращается. Это связно как с развитием технологии производства вычислительных систем, так и с необходимостью решения все более сложных прикладных задач. Такая разработка может быть обусловлена различными причинами, в частности, функциональными ограничениями, отсутствием локализации, низкой эффективностью существующих компиляторов. Поэтому основы теории языков и формальных грамматик, а также практические методы разработки компиляторов лежат в фундаменте инженерного образования по информатике и вычислительной технике[1].

Так как в современном мире существует бесчисленное множество языков программирования, то иногда возникает потребность в том, чтобы транслировать код написанный на одном языке программирования в код, который написан на эквивалентном ему другом языке программирования. Эти функции выполняют транслятор, и наиболее существенная их подгруппа - компиляторы.

Современные комплексы программных продуктов достигают миллионы строчек кода, за счет этого миру ПО нужны эффективные трансляторы, чтобы функционировать нормально, а главное быстро. Это особенно важно в системах реального времени, где быстродействие есть главный приоритет.

Цель курсовой работы - написать транслятор для подмножества языка Modula-2 на язык ассемблера.

Основные требования программы:

1. использование С++ в качестве языка программирования;
2. реализация констант;
3. реализация массивов;
4. реализация функций для ввода и вывода информации;
5. реализация одного из циклов;
6. реализация условного оператора;
7. реализация арифметических выражений.

Трансляция программы состоит из следующих этапов:

1. Препроцессорная обработка;
2. Лексический анализ;
3. Синтаксический анализ;
4. Семантический анализ;
5. Оптимизация кода;
6. Создание машинного кода;
7. Постпроцессорная обработка;
8. Линковка(сборка).

В конкретных реализациях компиляторов эти этапы могут быть разделены или, наоборот, совмещены в том или ином виде.

Транслятор в данной курсовой работе состоит из следующих этапов:

1. Лексический анализ. На этом этапе последовательность символов исходного файла преобразуется в последовательность лексем;
2. Синтаксический (грамматический) анализ. Последовательность лексем преобразуется в дерево разбора;
3. Генерация кода. Из промежуточного представления (дерева разбора) порождается код на целевом языке.

Для анализа текста на этапе лексического анализа может использоваться технология конечных автоматов.

**Коне́чный автома́т** — [абстрактный автомат](http://ru-wiki.org/wiki/%D0%90%D0%B1%D1%81%D1%82%D1%80%D0%B0%D0%BA%D1%82%D0%BD%D1%8B%D0%B9_%D0%B0%D0%B2%D1%82%D0%BE%D0%BC%D0%B0%D1%82), число возможных внутренних состояний которого [конечно](http://ru-wiki.org/wiki/%D0%9A%D0%BE%D0%BD%D0%B5%D1%87%D0%BD%D0%BE%D0%B5_%D0%BC%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE).

Существуют различные способы задания алгоритма функционирования конечного автомата. Например, конечный автомат может быть задан в виде упорядоченной пятерки элементов некоторых [множеств](http://ru-wiki.org/wiki/%D0%9C%D0%BD%D0%BE%D0%B6%D0%B5%D1%81%D1%82%D0%B2%D0%BE_(%D0%BC%D0%B0%D1%82%D0%B5%D0%BC%D0%B0%D1%82%D0%B8%D0%BA%D0%B0)):

* входной алфавит (конечное множество входных символов), из которого формируются входные слова, воспринимаемые конечным автоматом;
* множество внутренних состояний;
* начальное состояние;
* множество заключительных, или конечных состояний;
* функция переходов, определенная как отображение, то есть значение функции переходов на упорядоченной паре (состояние, входной символ или пустая цепочка) есть множество всех состояний, в которые из данного состояния возможен переход по данному входному символу или пустой цепочке (ε).

Принято полагать, что конечный автомат начинает работу в состоянии, последовательно считывая по одному символу входного слова (цепочки входных символов). Считанный символ переводит автомат в новое состояние в соответствии с функцией переходов.

Читая входную цепочку символов и делая переходы из состояния в состояние, автомат после прочтения последнего символа входного слова окажется в некотором состоянии.

Если это состояние является заключительным, то говорят, что автомат допустил слово.

В данной работе будет использоваться детерминированный конечный автомат, так как считывание текста происходит посимвольно, значит, переход автомата из одного состояния в другое будет однобуквенным, и каждой паре <состояние, символ> соответствует только одно состояние для перехода.

На этапе синтаксического анализа будет использоваться метод рекурсивного спуска[2], такая технология применима для анализа контекстно-свободной грамматики. Контекстная свободность подмножества языка Modula-2 доказывается далее.

Задачи:

1. Написать компилятор;
2. Написать и откомпилировать программу решения квадратного уравнения;
3. Написать и откомпилировать программу подсчёта факториала.

Полные текстовые примеры поставленных задач и полученные результаты представлены в разделе №3 отчета.

## 

## Раздел 1. Теоретические основы

### 1.1 Описание подмножества языка Modula-2

Modula-2 — язык программирования общего назначения, потомок и прямой наследник языка Паскаль, разработанный Никлаусом Виртом в 1978 году. Язык Modula-2 — структурный, модульный язык программирования, с синтаксисом, основанным на языке Паскаль, но заметно переработанным и улучшенным. Его отличия от языка Паскаль приведены ниже.[3]

1. Язык представляет собой набор модулей — самостоятельных единиц компиляции, которые могут компилироваться раздельно. Программный модуль может быть разделён на две части: модуль определений и модуль реализации. Модуль определений — это внешний интерфейс модуля, то есть набор экспортируемых им имён констант, переменных, типов, заголовков процедур и функций, которые доступны внешним модулям. Модуль реализации содержит программный код, в частности, конкретизацию описаний всего, что перечислено в модуле определений.

2. Более систематизированный синтаксис языка.

3. Процесс – как ключевое понятие мультипрограммирования.

4. Средства программирования низкого уровня, позволяющие ослабить жесткий контроль типов и отображать данные, имеющие структуру Модулы-2, на память, не обладающую внутренней структурой.

5. Процедурный тип - позволяет динамически присваивать процедуры переменным.

#### 1.1.1 Лексика языка

<идентификатор> → <буква> {<буква> | <цифра>}

<число> → <цифра> {<цифра>}

<символ> → <буква> | <цифра> | <разделитель>

<буква> → "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" | "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" | "U" | "V" | "W" | "X" | "Y" | "Z" | "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" | "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" | "u" | "v" | "w" | "x" | "y" | "z"

<цифра> → "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

<разделитель> → **`;`** | **`.`** | **`,`** | **`:`** | **`{`** | **`}`** |  **`(`** | **`)`** |  **`[`** | `**]`** |  **`>`** | **`<`** |  **`=>`** |  **`=<`**| **`=`**| **`<>`**| **`+`**| **`-`**| **`\*`** |  **`/`** | **`:=`**.

Для обработки комментариев, нельзя использовать детерминированный конечный автомат, поэтому, перед передачей очередного символа в автомат, проверяется наличие символов начала комментария, всё, что следует после этих символов до пары символов, обозначающих конец комментария, отбрасывается.

<комментарий> → “(\*” {<символ>} “\*)

#### 1.1.2 Тип грамматики

Подмножество языка Modula-2 – относится к языкам с контекстно-свободной грамматикой, что доказывается далее.

Контекстно-свободная грамматика[5] — частный случай формальной грамматики , у которой левые части всех продукций являются одиночными нетерминалами - объектами, обозначающими какую-либо сущность языка (например: формула, арифметическое выражение, команда) и не имеющими конкретного символьного значения). Смысл термина «контекстно-свободная» заключается в том, что есть возможность применить продукцию к нетерминалу, в отличие от общего случая неограниченной грамматики Хомского, не зависит от контекста этого нетерминала.

Терминальные символы: **`;`** , **`.`** , **`,`** , **`:`**, **`{`**,  **`}`**,  **`(`**, **`)`**,  **`[`**, `**]`**,  **`>`**,  **`<`**,  **`=>`**,  **`=<`**,  **`=`**,  **`<>`**,  **`+`**,  **`-`**,  **`\*`**,  **`/`**,  **`:=`**, заглавные и прописные латинские буквы, цифры.

Ключевые слова: MODULE, END, VAR, INTEGER, REAL, CONST, BOOLEAN, BEGIN, WHILE, DO, CHAR, AND, OR, IF, THEN, ELSE, ARRAY, OF, POINTER, TO, PROCEDURE, DIV, MOD, ELSIF.

#### 1.1.3 Синтаксические диаграммы в формате БНФ

S → S1 S2

S1 → “MODULE” <идентификатор> “;” {<раздел описаний>}

S2 → “BEGIN” <раздел операторов> “END” <идентификатор> “.”

<раздел описаний> → {<раздел констант> <раздел переменных> <раздел процедур>}

<раздел констант> → “CONST” <описание константы> “;”

<раздел переменных> → “VAR” <описание переменной> “;”

<описание переменной> → <список имен переменных> “:” <тип>

<список имен переменных> → <идентификатор> {“,” <идентификатор>}

<раздел процедур> → <описание процедуры> “;” [<раздел описаний>] “BEGIN” [<раздел операторов>] “END” <идентификатор> “;”

<раздел операторов> → <основной оператор> | <производный оператор>

<основной оператор> → <оператор присваивания> | <оператор процедуры>

<оператор присваивания> → <переменная> “:=” <выражение>

<оператор процедуры> → <идентификатор> [“(“ <фактический параметр>   
{ “,” <фактический параметр> } “)”] “;”

<фактический параметр> → <переменная> | <число>

<производный оператор> → <оператор цикла> | <условный оператор>

<условный оператор> → “IF” <логическое выражение> “THEN”   
<раздел операторов> [<иначе>] “END”

<иначе> → “ELSEIF” <логическое выражение> ”THEN” <раздел операторов> <иначе> ”END” | “ELSE” <раздел операторов> “THEN” <раздел операторов> “END”

<оператор цикла> → “WHILE” <логическое выражение> “DO”   
<раздел операторов> “END”

<выражение> → {<арифметическое выражение>} | {<логическое выражение>} | {<литерное выражение>} | {<ссылочное выражение>}

<арифметическое выражение> → {<слагаемое>} [<операция сложение>]

<операция сложения> → “+” | “-“

<слагаемое> → <множитель> [<операция умножения>]

<операция умножения> = “\*” | “/” | “MOD” | “DIV”

<множитель> → <вещественное число> | <целое число> |   
“(“ <арифметическое выражение> “)” | <переменная>

<литерное выражение> → <литерная константа> | <литерная переменная>

<ссылочное выражение> = “NIL” | <ссылочная переменная>

<логическое выражение> → <простое логическое выражение> |   
<скалярное отношение>

<простое логическое выражение> → <логическое слагаемое> {“OR” <логическое слагаемое>}

<логическое слагаемое> → <логический множитель> {”AND”   
<логический множитель>}

<логический множитель> → <логическая константа> |   
<логическая переменная> | “(“ <логическое выражение> “)”

<скалярное отношение> → <арифметическое выражение> <операция сравнения> <арифметическое выражение> | <простое логическое выражение> <операция сравнения> <простое логическое выражение> | <литерное выражение> <операция сравнения> <литерное выражение> | <ссылочное выражение> ”=” <ссылочное выражение> | <ссылочное выражение> ”<>” <ссылочное выражение>

<операция сравнения> → “=” | “<>” | “<=” | “>=” | “<” | “>”

<индекс> → <арифметическое выражение>

<переменная> → <идентификатор>

<описание константы> → <идентификатор> ”=” <константа>

<константа> → {“-”}<число> | <литерная константа>

<список идентификаторов> → <идентификатор> {,<идентификатор>}

<описание типа> → <идентификатор> ”=” <тип>

<тип> = <ключевой тип> | <регулярный тип> | <ссылочный тип>

<ключевой тип> → “INTEGER” | “REAL” | “CHAR”

<ссылочный тип> → “POINTER” “TO” <тип>

<регулярный тип> → “ARRAY” “[“ <индекс>{“,” <индекс>} “]” “OF” <тип>

<описание процедуры> → “PROCEDURE” <идентификатор> [<формальный параметр>]

<формальный параметр> → “(“ <список параметров>   
{“;”<список параметров>} “)”

<список параметров> → <параметр значение> | <параметр переменная>

<параметр значение> → <число>

<параметр переменная> → “VAR” <список идентификаторов> “:” <тип>

<число> → <целое> | <вещественное>

<целое> → <цифра> {<цифра>}

<вещественные> → <целое> ”.” <целое>

Правила в синтаксических диаграммах соответствуют виду A→𝛽, где A-нетерминал, -цепочка из терминальных и нетерминальных символов, из чего следует, что грамматика подмножества языка Modula-2 является контекстно-свободной, значит и язык задаваемый этой грамматикой так же является контекстно-свободным. Поэтому применим метод рекурсивного спуска для анализа данной грамматики.

### 1.2 Описание выходного языка(MASM)

Алфавит языка:

1. ‘A’ – ‘Z’, ‘a’ – ‘z’;

2. Цифры ‘0’ – ‘9’;

3. Пробел;

4. Прочие символы: ? @ \_ $ : . ( ) ; , ` ".

Типы данных:

Обозначение Количество байт

* byte или db 1;
* word или dw 2;
* dword или dd 4;
* qword или dq 8;
* tword или dt 10.

Команды процессора:

* Команды пересылки данных - mov
* Арифметические команды- add, sub, imul
* Логические и побитовые операции - or, and, xor, shr
* Команды управления ходом выполнения программы - jmp, loop, ret

Команды математического сопроцессора:

* finit – инициализация стека сопроцессора;
* fld, fild – загрузка операнда в вершину стека;
* fst, fist – сохранение вершины стека в память;
* fadd, fsub, fmul, fdiv – сложение, вычитание, умножение и деление вещественных чисел;
* fsqrt – вычисление квадратного корня.

Регистры общего назначения

**Регистр** - это сверхбыстрая память, которая расположена в процессоре. Так как регистр по сути определенная память, то было бы логично предположить, что он хранит какие-то данные полученные при работе программы, но не надо полагать, что все так просто, так как регистры бывают разные, но в основном это хранение данные, которое реализуют регистры общего назначения[4].

Эти регистры, как говорилось выше, служат для хранения промежуточных данных. В основном их размер 32 байта.

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

## 

## Раздел 2. Архитектура компилятора

На рисунке 1 представлен принцип работы компилятора и взаимосвязь отдельных его элементов.

На вход лексическому анализатору подается исходный текст программы, затем он, по запросу синтаксического анализатора передает токен (лексему). Синтаксический анализатор в свою очередь разбирает поступившую цепочку лексем и вызывает соответствующие методы генератора кода.

Лексический и синтаксический анализаторы на протяжении работы программы взаимодействуют с таблицами ключевых слов, идентификаторов, операторов и разделителей.

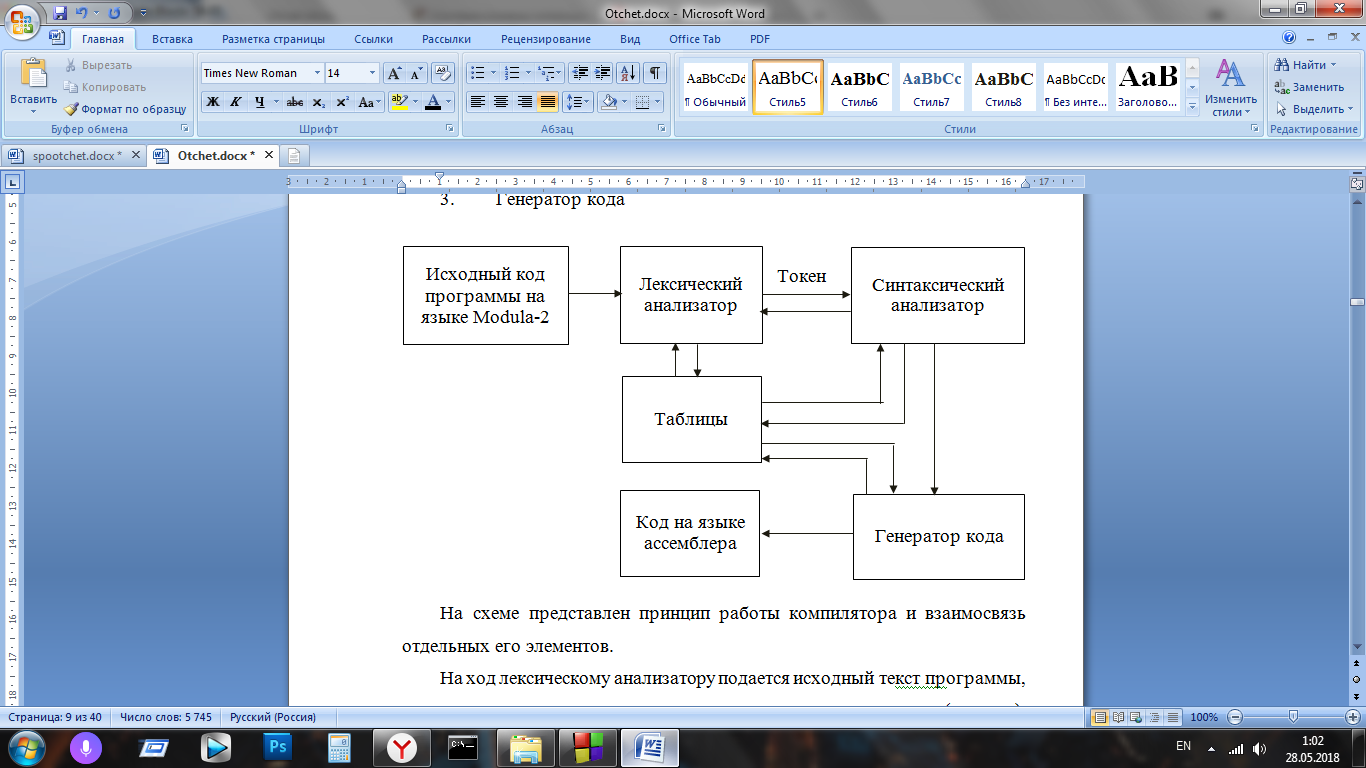


Рисунок 1 - принцип работы компилятора

### 2.1 Лексический анализатор

Лексический анализатор (ЛА) – считывает и обрабатывает текст программы посимвольно и формирует из них лексемы, которые передаются синтаксическому анализатору. Лексема(token) – минимальная неделимая лексическая единица языка. Также лексический анализатор обрабатывает пробелы и удаляет комментарии.

В данном трансляторе используется ЛА на базе детерминированного конечного автомата (ДКА) с магазинной памятью, реализованного в виде класса. Ключевые (зарезервированные) слова занесены в специальную таблицу: ( “ключевое слово” | “тип токена” ). При считывании идентификаторы проверяются по это таблице и если такое слово есть, токену присваивается определенное значение из таблицы.

На выходе из этого модуля получается список токенов, но токены в отличии от лексем несут в себе добавочную информацию.

В классе ЛА реализованы следующие методы:

|  |  |
| --- | --- |
| sym | является ли символ «символом» |
| add1 | добавление в таблицу переменных, констант, ключевых слов |
| add2 | добавление в таблицу операций |
| numtostr | перевод числа в строку |
| num | является ли символ цифрой |
| oper | является ли символ знаком операции |
| separ | является ли символ разделителем |
| mpclear | очистка магазина |
| InsertWord | добавление ключевого слова или переменной в список токенов |
| InsertNumber | добавление числовой константы в список токенов |
| InsertOper | добавление операции в список токенов |
| Insert | добавление токена в зависимости от его типа |
| automat | функция, реализующая действие автомата |
| printLex | печать токенов в файл |
| getsym | получение очередного символа |

Каждому символу соответствует событие, которое может быть к примеру: FlagString, FlagNumber. После того, как детерминированный автомат пришел в состояние конца, совершается создание токена и добавление его в список токенов, который будет использован в дальнейшем другими компонентами программы. При считывании символа из файла, определенная переменная проходится по циклу, где уже определяет в соответствии с состояниями дальнейшее состояние системы конечного автомата. Так же немаловажным фактором конечного автомата является его детерминированность, что означает, что автомат не может возвращаться в предыдущие состояния и не имеет возможность перехода в новое состояние по двум разным цепочкам.

#### 2.1.1 Токены

Токены являются важным звеном в транслирования программ, так как они являются структурным элементом грамматики программ, за счет чего используются в синтаксическом анализаторе. Токен в отличии от лексемы несет в себе дополнительную информацию:

— тип токена;

— значение токена;

— номер в таблице идентификаторов;

Лексический анализатор после того, как конечный автомат пришел в конечное состояние создает соответствующий токен, а токен уже в свою очередь работает с таблицами идентификаторов, разделителей, а также ключевых слов.

### 2.2 Синтаксический анализатор

Перед синтаксическим анализатором стоят две основные задачи: проверить правильность конструкций программы, которая представляется в виде уже выделенных слов входного языка, и преобразовать ее в вид, удобный для дальнейшей генерации кода. Одним из таких способов представления является дерево синтаксического разбора.

Синтаксически анализатор напрямую работает с цепочкой токенов, выданных лексическим анализатором и проверяет их пригодность определенным правилам языка. 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 укорачивающие контекстно-свободные грамматики вида A → α.

Для описания правил грамматики используется форма Бэкуса – Наура. В данной работе используется метод рекурсивного спуска для обработки языковых конструкций языка.

В классе СА реализованы следующие методы:

|  |  |
| --- | --- |
| scancode | Возвращает следующий токен от ЛА |
| Synanalys | Производит синтаксический анализ программы |

А также методы обработки каждого нетерминального символа такие, как:

|  |  |
| --- | --- |
| IFELSE | Обработка условного оператора |
| CIKL | Обработка оператора цикла |
| ARIFM | Обработка арифметических операций |

И так далее.

#### 2.2.1 Дерево разбора

В ходе синтаксического анализа исходный текст преобразуется в структуру данных, обычно — в дерево, которое отражает синтаксическую структуру входной последовательности и хорошо подходит для дальнейшей обработки. Как правило, результатом синтаксического анализа является синтаксическое строение предложения, представленное в виде дерева. В данной работе используется нисходящий разбор.

**Нисходящий разбор**заключается в построении дерева разбора, начиная от корневой вершины. Разбор заключается в заполнении промежутка между начальным нетерминалом и символами входной цепочки правилами, выводимыми из начального нетерминала. Подстановка основывается на том факторе, что корневая вершина является узлом, состоящим из листьев, являющихся цепочкой терминалов и нетерминалов одного из альтернативных правил, порождаемых начальным нетерминалом. Вместо новых нетерминальных вершин осуществляется подстановка выводимых из них правил. Процесс протекает до тех пор, пока не будут установлены все связи дерева, соединяющие корневую вершину и символы входной цепочки, или пока не будут перебраны все возможные комбинации правил. В последнем случае входная цепочка отвергается. Построение дерева разбора подтверждает принадлежность входной цепочки данному языку.

Класс ASTnode состоит из следующих методов:

|  |  |
| --- | --- |
| AddChild | добавление дочернего узла |
| settext  setparent  settype | установка значений узла |
| printtext  rettype  rettext  curpar  childtext | печать значений узла дерева |
| retchild | возвращение указателя на дочерний узел |
| RemoveChild | удаление узла |
| ASTremove | удаление дерева |
| printnode | вывод дерева на экран |
| childscount | возврат количества дочерних узлов у узла |

Для каждых конструкций языка были созданы соответствующие функции из-за чего такой подход позволил полноценно управлять трансляцией и совершать обходы по дереву в нужном порядке. В узле дерева находится вектор указателей на детей дерева, ну и соответственно информация в виде структуры. Данная структура хранит в себе такие поля как: значение элемента, тип элемента, указатель на родительский узел.

### 2.3 Генератор кода

Генерация кода порождает результирующую объектную программу на языке ассемблера. Внутреннее представление программы может иметь любую структуру в зависимости от реализации компилятора, в то время как результирующая программа всегда представляет собой линейную последовательность команд. Поэтому генерация объектного кода (объектной программы) в любом случае должна выполнять действия, связанные с преобразованием сложных синтаксических структур в линейные цепочки.

Генерация кода является последней стадией разработки компилятора в данной курсовой работе. Конечным языком является ассемблер, что усложняет работу и требует знаний этого языка. Следует отметить, что в ассемблере используется повсеместно 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.

В классе ГК реализованы следующие методы:

|  |  |
| --- | --- |
| ifcode | печать кода условного оператора |
| while\_cikle | печать кода оператора цикла |
| ProvCikl | обход синтаксического дерева |
| setindex | Установка соответствия между регистрами и идентификаторами |
| printdata | возвращение указателя на дочерний узел |
| opproc | удаление узла |
| prisv | удаление дерева |
| GenerCode | Запуск ГК |

Полный текст компилятора представлен в приложении.

**2.4 Диаграмма классов.**

Схема работы классов программы представлена на рисунке 2.

|  |
| --- |
| **класс GENcode** |
| +ifcode(ASTnode \*): void |
| +while\_cikle(ASTnode \*): void |
| +ProvCikl(ASTnode \*): void |
| +setindex(ASTnode \*): void |
| +printdata(): void |
| +opproc(ASTnode\*): void |
| +prisv(ASTnode\*): void |
| +GenerCode(): int |

**класс SYNTAX**

|  |
| --- |
| -\*root: ASTnode |
| -code: fstream |
| -mod: ofstream |
| -word, t, nam, page, currentproc: string |
| -tableid: struct |
| -torindex: struct |
| -fi: vector <forindex> |
| -metka: int |
| +Syntax(); |
| +idstringtype(string): string |
| +scancode(): void |
| +Synanalys(): void |
| +getindex(string): string |

**класс LEXIC**

|  |
| --- |
| -cin\_code: ifstream |
| -mainwords: ifstream cin\_ |
| -Lexems: vector <string> |
| -mp: string |
| -flagS, flagN, flagO, flagL: bool |
| -ch: char |
| -list1: struct |
| -list2: struct |
| -numstroke: int |
| +Lexic() |
| +sym: bool |
| +add1(list1 \*&head, string w, int n): void |
| +add2(list2 \*&head, string w, char id, int n): void |
| +numtostr(int): int |
| +num(): bool |
| +oper(): bool |
| +separ(): bool |
| +mpclear(): void |
| +InsertWord(): void |
| +InsertNumber(): void |
| +InsertOper(): void |
| +Insert(): void |
| +automat(char get): void |
| +printLex(): void |
| +getsym(): char |

**1..n**

**1**

**класс ASTnode**

|  |  |
| --- | --- |
| -Type: int |  |
| -Text: string |
| -\*parent: ASTnode |
| -childs: vector <ASTnode> |
| +ASTnode(int type, string text, ASTnode \*child1, ASTnode \*child2) |  |
| +AddChild (ASTnode \*child): int |  |
| +settext (string qwe): void |  |
| +setparent (ASTnode \*qwe): void |
| +settype (int qwe): void |
| +printtext(): void |  |
| +retype(): int |
| +rettext(): string |
| +curpar(): int |
| +childtext(int): string |
| +\*retchild(int): ASTnode |  |
| +RemoveChild (ASTnode \*child): int |  |
| +ASTremove (ASTnode \*node): void |  |
| +printnode (ASTnode \*node): void |  |
| +childscount(): int |  |

**1**

**1**

**1**

**1**

Рисунок 2 - Диаграмма классов

## Раздел 3. Тестовые примеры

### 3.1 Расчет суммы ряда рекурсивным методом

|  |  |
| --- | --- |
| Код на языке Modula-2 | Код на языке Ассемблер |
| MODULE modula2;  VAR a:INTEGER;  VAR b:INTEGER;  PROCEDURE Sum(VAR k:INTEGER, VAR d:INTEGER);  BEGIN  IF d>0 THEN  k:=k+d;  d:=d-1;  Sum(k, d);  ELSE  a:=k;  END  END Sum  BEGIN  printstring("Vvedite\_N:");  scanint(b);  a:=0;  Sum(a, b);  printstring("Summa\_ryada=");  printint(a);  END modula2. | .386  .model flat,stdcall  include \masm32\include\masm32rt.inc  include \masm32\include\msvcrt.inc  include \masm32\include\kernel32.inc  includelib \masm32\lib\masm32.lib  includelib \masm32\lib\msvcrt.lib  includelib \masm32\lib\kernel32.lib  .data  prnt\_scan\_real DB "%f",0  OutputFormat db "%i",0  a DD ?  b DD ?  var\_1 DB "Vvedite\_N:",0  var\_2 DB "Summa\_ryada=",0  .code  Sum proc k:DWORD, d:DWORD  MOV edx,d  CMP edx,0  JLE L0  MOV eax,k  MOV ebx,d  ADD eax,ebx  MOV k,eax  MOV eax,d  SUB eax,1  MOV d,eax  invoke Sum, k,d  JMP L1  L0:  MOV eax,k  MOV a,eax  L1:  ret  Sum ENDP  start:  FINIT  printf("Vvedite\_N:")  invoke crt\_scanf, ADDR OutputFormat, ADDR b  MOV eax,0  MOV a,eax  invoke Sum, a,b  printf("Summa\_ryada=")  printf("%i",a)  invoke crt\_\_getch  invoke ExitProcess,0  end start  END |

Результат работы программы представлен на рисунке 2.

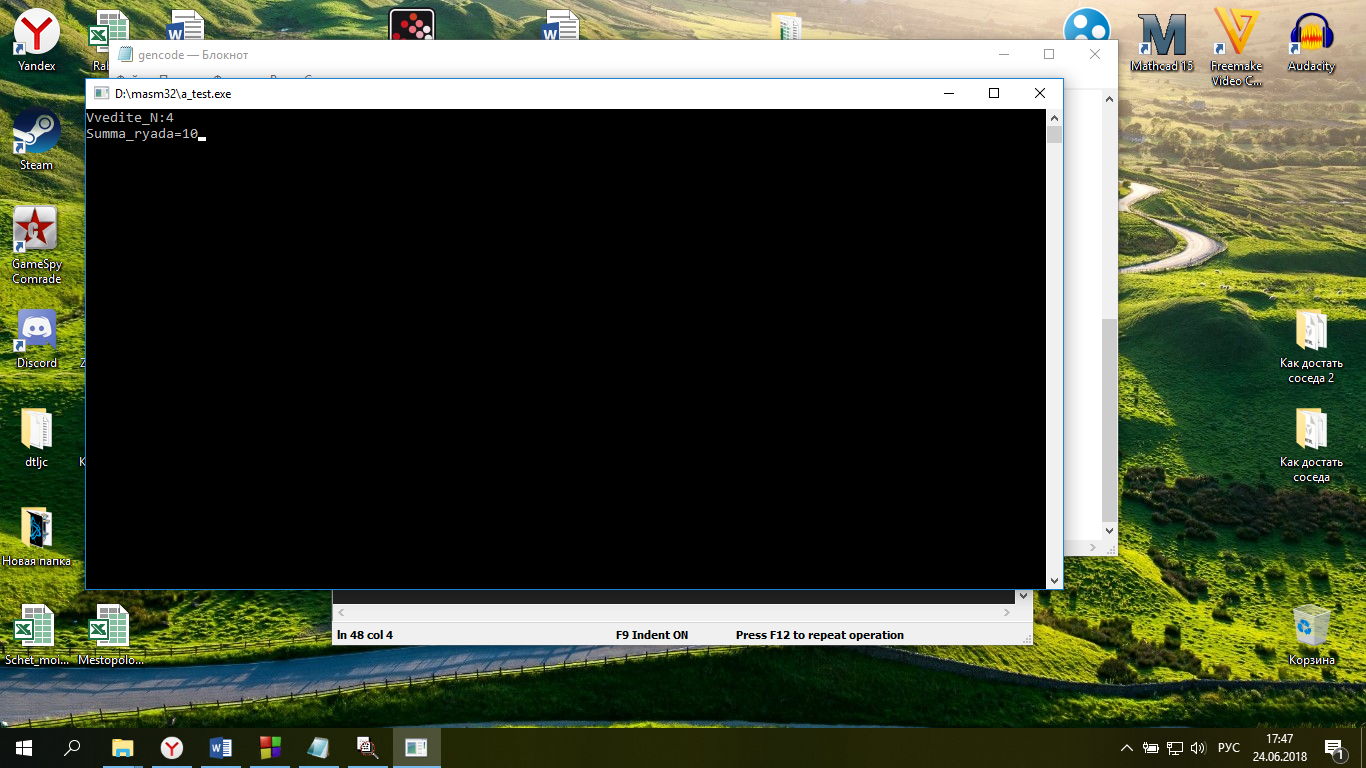


Рисунок 2 – расчет полинома

### 3.2 Подсчёт факториала циклическим методом

|  |  |
| --- | --- |
| Код на языке Modula-2 | Код на языке Ассемблер |
| MODULE modula2;  VAR n:INTEGER;  VAR i:INTEGER;  VAR res:INTEGER;  BEGIN  printstring("Vvedite\_chislo: ");  scanint(n);  res:=1;  i:=1;  WHILE i<=n DO  res:=res\*i;  i:=i+1;  END  printstring("Factorial=");  printint(res);  END modula2. | .386  .model flat,stdcall  include \masm32\include\masm32rt.inc  include \masm32\include\msvcrt.inc  include \masm32\include\kernel32.inc  includelib \masm32\lib\masm32.lib  includelib \masm32\lib\msvcrt.lib  includelib \masm32\lib\kernel32.lib  .data  prnt\_scan\_real DB "%f",0  OutputFormat db "%i",0  n DD ?  i DD ?  res DD ?  var\_1 DB "Vvedite\_chislo:",0  var\_2 DB "Factorial=",0  .code  start:  FINIT  printf("Vvedite\_chislo:")  invoke crt\_scanf, ADDR OutputFormat, ADDR n  MOV eax,1  MOV res,eax  MOV eax,1  MOV i,eax  MOV edx,i  L0: CMP edx,n  JG L1  MOV eax,res  MOV ebx,i  IMUL eax,ebx  MOV res,eax  MOV eax,i  ADD eax,1  MOV i,eax  MOV edx,i  JMP L0  L1:  printf("Factorial=")  printf("%i",res)  invoke crt\_\_getch  invoke ExitProcess,0  end start  END |

Результат работы программы представлен на рисунке 3.

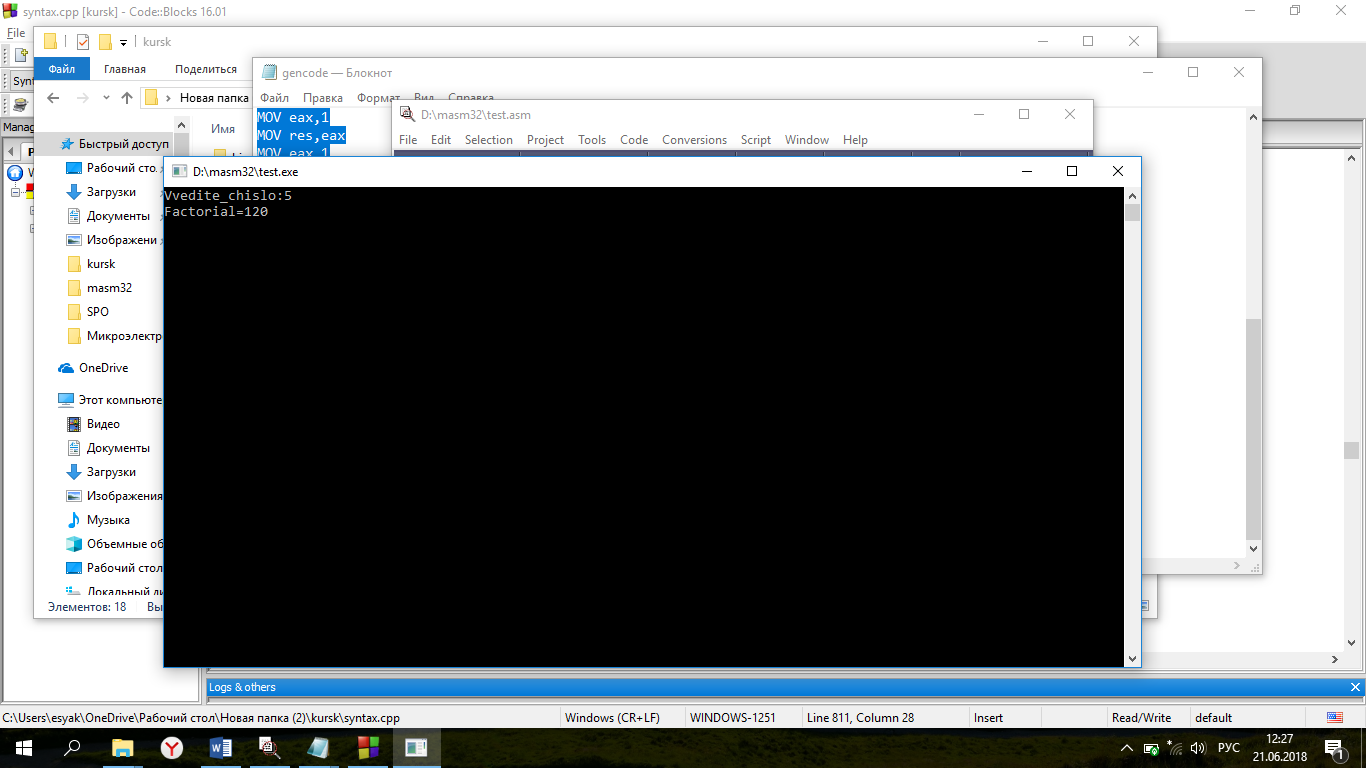


Рисунок 3 - подсчёт факториала

## 

## ЗАКЛЮЧЕНИЕ

В результате выполнения курсовой работы для заданного входного языка были построены отдельные части компилятора. В первой части работы была разработана программа, которая получает на входе файл на подмножестве языка Modula-2, организует таблицу идентификаторов методом упорядоченного списка, позволяет осуществить многократный поиск идентификатора в этой таблице. Во второй части работы была написана программа, которая выполняет лексический анализ входного текста и порождает таблицу лексем с указанием их типов и значений. Третья часть курсовой работы была посвящена разработке программы, которая порождает таблицу лексем и выполняет синтаксический разбор текста с построением дерева разбора и дальнейшей генерацией кода. Отдельные части компилятора, разработанные в данной курсовой работе, дают представление о технике и методах, лежащих в основе построения компиляторов. Программа была написана на языке программирования C++.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Вирт Н. Построение компиляторов. — М.: ДМК Пресс, 2010

2. Ахо А., Сети Р., Ульман Дж. Д. Компиляторы: принципы, технологии и инструменты, 2-е изд. [Текст]/ М.: Вильямс, 2008. – 1157 с.

3. Документация языка Подмножество Modula-2 [Электронный ресурс]. URL: http://www.modula2.org/ (Дата обращения 20.03.2018)

4. Зубков С. В. Assembler для DOS, Windows и UNIX, 3-е изд. [Текст]/ – М. : ДМК Пресс ; СПб. : Питер, 2004. – 608 с.

5. Вирт Н. Программирование на языке Модула-2. М.: Мир, 1987;

Листинг программы

*lexix.h:*

#ifndef LEXIC\_H

#define LEXIC\_H

#include <fstream>

#include <string> //подключение библиотек

#include <iostream>

#include <vector>

using namespace std;

class Lexic

{

ifstream cin\_code; //файл с исходной программой

ifstream cin\_mainwords; //файл с ключевыми словами исходного языка

ofstream out\_code;

vector <string> Lexems; //массив лексем

string mp; //автомат

bool flagS, flagN, flagO, flagL; //флаги накопления слова, числа и оператора соответственно

char ch; //входящий символ

struct list1 //список ключевых слов и т.д.

{

string word; //идентификатор

int num; //номер в списке

list1 \*next; //ссылка на следующий

} \*variables, \*constants, \*keywords; // списки переменных, констант и ключевых слов

struct list2 //список операций

{

string word; //идентификатор

char id; //символ типа операции

int num; //номер операции среди операций того же типа

list2 \*next; //ссылка на следующий элемент

} \*operations; //список операций

int numstroke;

public:

Lexic();

int add0(list1 \*&head, string w); //добавление в список 1 типа при неизвестном номере

void add1(list1 \*&head, string w, int n); //добавление в очередь 1 типа при известном номере

void add2(list2 \*&head, string w, char id, int n); //добавление в список 2 типа

bool sym(); //является ли символ «символом»

string numtostr(int);

void openfile(char \*);

bool num(); //является ли символ цифрой

bool oper(); //является ли символ знаком операции

bool separ(); //является ли символ разделителем

void mpclear(); //очистка магазина

void InsertWord(); //добавление ключевого слова или переменной

void InsertNumber(); //добавление числовой константы

void InsertOper(); //добавление операции

void Insert(); //добавление токена в зависимости от его типа

void Error(); //обработчик ошибок

void automat(char get); //функция, реализующая действие автомата

void printLex();

bool endfile();

char getsym();

int Lexanalys();

void closefiles();

};

#endif

*suntax.h:*

#ifndef SYNTAX\_H

#define SYNTAX\_H

#include <stack>

#include <vector>

#include <iostream>

#include <clocale>

#include <fstream>

#include <string> //подключение библиотек

using namespace std;

class ASTnode

{

int Type;

string Text;

ASTnode \*parent;

vector <ASTnode\*> childs;

public:

ASTnode(int type, string text, ASTnode \*child1, ASTnode \*child2);

void printtext();

int AddChild(ASTnode \*child);

void settext(string qwe);

int curpar();

ASTnode \*retchild(int);

string childtext(int);

void setparent(ASTnode \*qwe);

void settype(int qwe);

int rettype();

string rettext();

int RemoveChild(ASTnode \*child);

void ASTremove(ASTnode \*node);

void printnode(ASTnode \*node);

int childscount();

};

class Syntax

{

ASTnode \*root;

fstream code;

ofstream mod;

string word, t, nam, page, currentproc;

struct tableid

{

string var, type, proc, zn;

int mas, kol;

}rty;

struct forindex

{

string jk1, jk2;

}jok;

vector <forindex> fi;

vector <tableid> TI;

int realvar;

int metka;

int varnum;

int joknum;

int

UNKNOWN,

ADD,

SUB,

MUL,

DIV,

MOD,

STR,

ASSIGN,

KEYWORD,

SEPARATOR,

IDENT,

CONSTANT,

PROGRAMM,

PROCEDURE,

ERROR,

NONTERMINAL,

OR,

AND,

EQ,

NE,

NOT,

SRAV,

OPCIKLE,

OPIF,

OPELSE,

OPELSIF,

TERMINAL,

fparam;

public:

Syntax();

~Syntax();

void scancode();

string idstringtype(string);

void while\_cikle(ASTnode \*);

int GSTPS(string);

string HelpStr(string);

int TIcheck(string);

int Synanalys();

void ifcode(ASTnode \*);

void ProvCikl(ASTnode \*);

void printjok();

int ischislo(string);

//int typeof(string);

ASTnode\* START();

ASTnode\* MODULE();

ASTnode\* NAME();

ASTnode \*O();

ASTnode \*P();

void setindex(ASTnode \*);

string getindex(string);

void printdata();

ASTnode \*PO();

void opproc(ASTnode\*);

ASTnode \*IFELSE();

ASTnode \*CIKL();

ASTnode \*OO();

ASTnode \*A();

void printid();

int check(string w);

ASTnode \*BOOL();

ASTnode \*JOIN();

ASTnode \*EQUA();

ASTnode \*REL();

ASTnode \*ARIFM();

ASTnode \*term();

ASTnode \*unary();

ASTnode \*factor();

ASTnode \*offset(string qwe);

string sidtype(string w);

int idtype(string w);

void FACTPAR(ASTnode \*);

ASTnode \*F();

void prisv(ASTnode\*);

ASTnode \*FORMALPAR();

ASTnode \*C();

ASTnode \*C1();

ASTnode\* CHISLO();

ASTnode\* V();

ASTnode\* TYPE();

int GenerCode();

};

#endif

*lexic.cpp:*

#include "lexic.h"

#include <fstream>

#include <string> //подключение библиотек

#include <iostream>

#include <vector>

#include <stdlib.h>

using namespace std;

Lexic::Lexic()

{

//cin\_code.open("input.mod");

cin\_mainwords.open("mainwords.txt");

out\_code.open("temp.txt");

variables = NULL;

operations = NULL;

constants = NULL;

numstroke=1;

keywords = NULL; //обнуление указателей на начала списков

flagL=flagN=flagO=flagS=false; //установка начальных состояний автомата

int k;

char q; //символ типа операции

cin\_mainwords >> k >> q;

string s;

for (int i = 0; i < k; i++) //чтение ключевых слов из файла и помещение в список

{

cin\_mainwords >> s;

add1(keywords, s, i);

}

//print2(keywords, "k");

cin\_mainwords >> k;

for (int i = 0; i < k; i++) //чтение операций из файла и помещение в список

{

int n;

cin\_mainwords >> n >> q;

for (int j = 0; j < n; j++) //чтение операций типа ch

{

cin\_mainwords >> s;

add2(operations, s, q, j);

}

}

}

void Lexic::openfile(char \*s)

{

cin\_code.open(s);

}

int Lexic::Lexanalys()

{

while (!endfile()) //пока не достигнут конец файла

{

automat(getsym()/\*читаем символ\*/); //запускаем автомат

}

automat(' '); //запускаем автомат с пробельным символом для обработки содержимого магазина на последнем шаге

printLex(); //вывод списков

closefiles();

return 1;

}

void Lexic::closefiles()

{

cin\_code.close();

cin\_mainwords.close();

out\_code.close();

}

int Lexic::add0(list1 \*&head, string w) //добавление в список 1 типа при неизвестном номере

{

list1 \*temp = new list1; //создание нового элемента

temp -> next = NULL;

temp -> word = w; //присвоение новому элементу идентификатора w

if (head) //если список не пуст

{

list1 \*t;

for (t=head; t -> next != NULL; t = t -> next);

temp -> num = t -> num + 1;

t -> next = temp; //добавление в хвост очереди

}

else

{

temp -> num = 0;

head = temp;

}

return temp -> num; //возвращаем номер добавленного элемента

}

void Lexic::add1(list1 \*&head, string w, int n) //добавление в очередь 1 типа при известном номере

{

list1 \*temp = new list1; //аналогично add0

temp -> word = w;

temp -> next = NULL;

temp -> num = n;

if (head)

{

list1 \*t;

for (t=head; t -> next != NULL; t = t -> next);

t -> next = temp;

}

else

{

head = temp;

}

}

void Lexic::add2(list2 \*&head, string w, char id, int n) //добавление в список 2 типа

{

list2 \*temp = new list2;

temp -> next = NULL;

temp -> id = id;

temp -> word = w;

temp -> num = n;

if (head)

{

list2 \*t;

for (t=head; t -> next != NULL; t = t -> next);

t -> next = temp;

}

else

{

head = temp;

}

}

bool Lexic::sym() //является ли символ «символом»

{

return ((ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z') || (ch == '\_')) ? true : false;

}

bool Lexic::num() //является ли символ цифрой

{

return (ch >= '0' && ch <= '9') ? true : false;

}

bool Lexic::oper() //является ли символ знаком операции

{

return (ch =='"' || ch==':' || ch== ',' || ch=='.' || ch == '+' || ch == '-' || ch == '\*' || ch == '/' || ch == '>' || ch == '<' || ch == '&' || ch == '|' || ch == '=' || ch == ';' || /\*ch == '(' || ch == ')' ||\*/ ch == '{' || ch == '}') ? true : false;

}

bool Lexic::separ() //является ли символ разделителем

{

return (ch == '"' || ch=='[' || ch==']' || ch== ',' || ch=='.' || ch == ' ' || ch == '\n' || ch == ')' || ch == '(' || ch == ';' || ch == '}' || ch == '{' || (!flagO && mp != "" && oper())) ? true : false;

}

void Lexic::mpclear() //очистка магазина

{

mp = "";

}

void Lexic::InsertWord() //добавление ключевого слова или переменной

{

//char ch1;

string w;

list1 \*temp;

for (temp=keywords; temp!=NULL; temp = temp -> next) //поиск в списке констант

{

if (temp -> word == mp)

{

w = "k";

/\*ch1 = '0' + temp -> num;

w += ch1;\*/

w += " " + temp -> word + " " + numtostr(numstroke);

Lexems.push\_back(w);

return;

}

}

for (temp=variables; temp!=NULL; temp = temp -> next) //поиск в списке переменных

{

if (temp -> word == mp)

{

w = "j";

w += " " + temp -> word + " " + numtostr(numstroke);

Lexems.push\_back(w);

return;

}

}

//int nnum = add0(variables, mp); //добавление новой переменной

w = "j";

w += " " + mp + " " + numtostr(numstroke);

Lexems.push\_back(w);

//cout << w << endl;

}

void Lexic::InsertNumber() //добавление числовой константы

{

//char ch1;

string w;

list1 \*temp;

for (temp=constants; temp!=NULL; temp = temp -> next) //поиск в списке констант

if (temp -> word == mp)

{

w = "c";

w += " " + temp -> word + " " + numtostr(numstroke);

Lexems.push\_back(w);

return;

}

//int nnum = add0(constants, mp); //добавление новой константы

w = "c";

w += " " + mp + " " + numtostr(numstroke);

Lexems.push\_back(w);

}

void Lexic::InsertOper() //добавление операции

{

string w;

list2 \*temp;

for (temp=operations; temp != NULL; temp = temp -> next)

//поиск в списке операций

if (temp -> word == mp)

{

w = temp -> id;

w += " " + temp -> word + " " + numtostr(numstroke);

Lexems.push\_back(w);

return;

}

}

string Lexic::numtostr(int num)

{

char buf[5];

string temp="";

itoa(num, buf, 10);

for(int i=0 ; i<5 && buf[i]!='\0'; i++) temp+=buf[i];

//temp+="\0";

return temp;

}

void Lexic::Insert() //добавление токена в зависимости от его типа

{

string w="";

if (flagS) //если токен – ключевое слово или имя переменной

InsertWord();

if (flagN) //если токен – числовая константа

InsertNumber();

if (flagO) //если токен - операция

InsertOper();

if(flagL)

{

w = "l";

w += " " + mp + " " + numtostr(numstroke);

Lexems.push\_back(w);

}

mpclear(); //очистка магазина

}

void Lexic::Error() //обработчик ошибок

{

Lexems.push\_back("Error"); //вывод сообщения об ошибке

while (!separ() && cin\_code.peek()!=EOF) //пропуск символов до следующего разделителя

cin\_code >> ch;

mpclear();

}

void Lexic::automat(char get) //функция, реализующая действие автомата

{

string w;

static char last=' ';

ch=get;

int h=0;

static int q=0;

if(q==1)

{

numstroke++;

q=0;

}

if(ch=='\n')

{

q=1;

}

if(ch=='"')

{

//Insert();

ch=getsym();

mp="\"";

while(ch!='"')

{

mp+=ch;

ch=getsym();

}

mp+=ch;

flagL=true;

flagS = flagN = flagO = false;

Insert();

flagL=false;

mp=")";

flagO=true;

Insert();

return;

}

if (mp.length() == 0) //если магазин пуст

{

last=ch;

if (!(ch == ' ' || ch == '\n')) //если символ не пробел и не маркер конца строки

{

flagS = sym();

flagN = num();

flagO = oper(); //определяем тип символа

mp += ch; //добавляем символ в магазин

return;

}

else return;

}

if(ch=='\*' && last=='(')

{

while(h!=1)

{

do

{

ch=getsym();

}while(ch!='\*');

if(getsym()!=')') continue;

else h=1;

}

h=0;

return;

}

if (!separ() || (flagN && ch=='.')) //если символ - не разделитель

{

if ((sym() || num()) && flagS)

//если накапливаем «слово» и текущий символ – «символ» или цифра

mp += ch;

else if ((num() || ch=='.') && flagN)

mp += ch;

//если накапливаем число и текущий символ - цифра или точка

else if (oper() && flagO)

//если накапливаем операцию и текущий символ – символ операции

mp += ch;

else if (flagN && sym())

//если накапливаем число, а текущий символ – «символ», то ошибка

Error();

else

{

if(mp=="[")

{

w="i ";

w += mp + " " + numtostr(numstroke);

Lexems.push\_back(w);

}

Insert();

flagS = sym();

flagN = num();

flagO = oper();

mp += ch; //добавляем символ в магазин

}

last = ch;

return;

}

if (separ())

{

if (oper()) //если разделитель – символ операции

{

Insert();

flagS = sym();

flagN = num();

flagO = oper();

mp = ch;

}

else

{

Insert();

if(ch=='(' || ch==')')

{

w="s ";

w += ch;

w+= " " + numtostr(numstroke);

Lexems.push\_back(w);

}

else if(ch=='[' || ch==']')

{

w="i ";

w += ch;

w+= " " + numtostr(numstroke);

Lexems.push\_back(w);

}

}

last = ch;

}

}

void Lexic::printLex()

{

for(int i=0 ; i<(int)Lexems.size() ; i++)

out\_code << Lexems[i] << endl;

}

bool Lexic::endfile()

{

return cin\_code.peek()==EOF ? true : false;

}

char Lexic::getsym()

{

char q;

q=cin\_code.get();

return q;

}

*syntax.cpp:*

#include "syntax.h"

#include <stack>

#include <vector>

#include <iostream>

#include <stdlib.h>

#include <clocale>

#include <fstream>

#include <string> //подключение библиотек

using namespace std;

ASTnode::ASTnode(int type, string text, ASTnode \*child1, ASTnode \*child2)

{

parent = NULL;

Type = type;

Text = text;

if(child1 != NULL) AddChild(child1);

if(child2 != NULL) AddChild(child2);

}

void ASTnode::printtext()

{

cout << this->Text;

}

ASTnode\* ASTnode::retchild(int i)

{

return this->childs[i];

}

string ASTnode::rettext()

{

return this->Text;

}

string ASTnode::childtext(int i)

{

return this->childs[i]->Text;

}

int ASTnode::AddChild(ASTnode \*child)

{

if(child==NULL) return 0;

childs.push\_back(child);

child->parent = this;

return 1;

}

void ASTnode::settext(string qwe)

{

Text=qwe;

}

int ASTnode::curpar()

{

if(this->parent==NULL) return 0;

else return 1;

}

int ASTnode::rettype()

{

return this->Type;

}

void ASTnode::setparent(ASTnode \*qwe)

{

parent=qwe;

}

void ASTnode::settype(int qwe)

{

Type=qwe;

}

int ASTnode::RemoveChild(ASTnode \*child)

{

if (child->parent == this) child->parent = NULL;

return 2;

}

void ASTnode::ASTremove(ASTnode \*node)

{

if(node==NULL) return;

for(int i=0 ; i<(int)node->childs.size() ; i++)

{

ASTremove(node->childs[i]);

}

node->childs.clear();

delete node;

}

void ASTnode::printnode(ASTnode \*node)

{

static int g=0;

if(node==NULL) return;

cout << node->Text <<endl;

for(int i=0 ; i<(int)node->childs.size() ; i++)

{

for(int j=0 ; j<g ; j++) cout << " ";

cout << "|";

g++;

printnode(node->childs[i]);

}

g--;

}

int ASTnode::childscount()

{

return this->childs.size();

}

Syntax::Syntax()

{

mod.open("gencode.asm");

varnum=1;

ERROR = 404;

NONTERMINAL = 101;

UNKNOWN = 0;

KEYWORD=1;

CONSTANT = 2;

IDENT = 3;

ADD=11;

SUB=12;

MUL = 13;

DIV = 14;

MOD = 15;

ASSIGN = 16;

STR=71;

SEPARATOR = 52;

OR = 17;

AND=18;

EQ = 19;

NE=20;

NOT=21;

PROCEDURE=200;

SRAV=22;

joknum=0;

realvar=0;

OPCIKLE=30;

OPIF=31;

OPELSE=32;

OPELSIF=33;

TERMINAL=100;

root = NULL;

fparam=0;

metka=0;

code.open("temp.txt", ios::in);

currentproc="glob";

}

Syntax::~Syntax()

{

mod.close();

code.close();

root->ASTremove(root);

}

void Syntax::printdata()

{

mod << ".386" << endl;

mod << ".model flat,stdcall" << endl;

mod << "include \\masm32\\include\\masm32rt.inc" << endl;

mod << "include \\masm32\\include\\msvcrt.inc" << endl;

mod << "include \\masm32\\include\\kernel32.inc" << endl;

mod << "includelib \\masm32\\lib\\masm32.lib" << endl;

mod << "includelib \\masm32\\lib\\msvcrt.lib" << endl;

mod << "includelib \\masm32\\lib\\kernel32.lib" << endl;

mod << ".data" << endl;

mod << "prnt\_scan\_real DB \"%f\",0" << endl;

mod << "OutputFormat db \"%i\",0" << endl;

for(int i =0 ; i<(int)TI.size() ; i++)

{

if(TI[i].proc!="glob" && TI[i].type!="PROCEDURE")

{

mod << "proc\_" << TI[i].proc << " " << TI[i].var << " DD " << TI[i].zn << endl;

}

else if(TI[i].proc=="glob" && TI[i].type!="PROCEDURE")

{

if(TI[i].type=="string")

{

mod << "var\_" << TI[i].kol << " DB " << TI[i].var << ",0" << endl;

}

else

{

switch(idtype(TI[i].var))

{

case 4:

mod << TI[i].var << " DD " << TI[i].zn << endl;

break;

}

}

}

}

mod << ".code" << endl;

mod << "start:" << endl;

mod << "FINIT" << endl;

}

void Syntax::printjok()

{

for(int i=0 ; i<(int)fi.size() ; i++)

{

if(ischislo(fi[i].jk1)==0)

{

mod << "MOV " << fi[i].jk2 << "," << fi[i].jk1 << endl;

}

}

}

void Syntax::ProvCikl(ASTnode \*current)

{

for(int j=0 ; j<current->childscount() ; j++)

{

if(current->retchild(j)->rettext()=="WHILE")

{

while\_cikle(current->retchild(j));

}

else if(current->retchild(j)->rettext()=="IF")

{

ifcode(current->retchild(j));

}

else if(current->retchild(j)->rettext()!=":=")

{

opproc(current->retchild(j));

}

else

{

if(current->retchild(j)->retchild(1)->childscount()==0)

{

mod << "MOV eax," << current->retchild(j)->retchild(1)->rettext() << endl;

mod << "MOV " << current->retchild(j)->retchild(0)->rettext() << ",eax" << endl;

}

else

{

setindex(current->retchild(j)->retchild(1));

printjok();

prisv(current->retchild(j));

fi.clear();

joknum=0;

realvar=0;

}

}

}

}

int Syntax::GenerCode()

{

ASTnode \*current=NULL;

int i=root->childscount();

printdata();

for(int j=2 ; j<i-1 ; j++)

{

if(root->childtext(j)=="NACHINKA")

{

current=root->retchild(j);

ProvCikl(current);

}

}

mod << "invoke crt\_\_getch" << endl;

mod << "invoke ExitProcess,0" << endl;

mod << "end start" << endl;

mod << "END" << endl;

return 1;

}

void Syntax::while\_cikle(ASTnode \*temp)

{

if(temp->retchild(0)->rettext()=="<")

{

mod << "MOV edx," << temp->retchild(0)->retchild(0)->rettext() << endl;

mod << "L" << metka << ": CMP edx," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JGE " << "L" << metka+1 << endl;

}

else if(temp->retchild(0)->rettext()==">")

{

mod << "MOV edx," << temp->retchild(0)->retchild(0)->rettext() << endl;

mod << "L" << metka << ": CMP edx," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JLE " << "L" << metka+1 << endl;

}

else if(temp->retchild(0)->rettext()=="<=")

{

mod << "MOV edx," << temp->retchild(0)->retchild(0)->rettext() << endl;

mod << "L" << metka << ": CMP edx," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JG " << "L" << metka+1 << endl;

}

else if(temp->retchild(0)->rettext()==">=")

{

mod << "MOV edx," << temp->retchild(0)->retchild(0)->rettext() << endl;

mod << "L" << metka << ": CMP edx," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JL " << "L" << metka+1 << endl;

}

else if(temp->retchild(0)->rettext()=="=")

{

mod << "MOV edx," << temp->retchild(0)->retchild(0)->rettext() << endl;

mod << "L" << metka << ": CMP edx," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JNE " << "L" << metka+1 << endl;

}

else if(temp->retchild(0)->rettext()=="<>")

{

mod << "MOV edx," << temp->retchild(0)->retchild(0)->rettext() << endl;

mod << "L" << metka << ": CMP edx," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JE " << "L" << metka+1 << endl;

}

ProvCikl(temp->retchild(1));

mod << "MOV edx," << temp->retchild(0)->retchild(0)->rettext() << endl;

mod << "JMP L" << metka << endl;

metka++;

mod << "L" << metka << ":" << endl;

}

void Syntax::ifcode(ASTnode \*temp)

{

if(temp->retchild(0)->rettext()=="<")

{

mod << "MOV edx," << temp->retchild(0)->retchild(0)->rettext() << endl;

mod << "CMP edx," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JGE " << "L" << metka << endl;

}

else if(temp->retchild(0)->rettext()==">")

{

mod << "MOV edx," << temp->retchild(0)->retchild(0)->rettext() << endl;

mod << "CMP edx," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JLE " << "L" << metka << endl;

}

else if(temp->retchild(0)->rettext()=="<=")

{

mod << "MOV edx," << temp->retchild(0)->retchild(0)->rettext() << endl;

mod << "CMP edx," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JG " << "L" << metka << endl;

}

else if(temp->retchild(0)->rettext()==">=")

{

mod << "MOV edx," << temp->retchild(0)->retchild(0)->rettext() << endl;

mod << "CMP edx," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JL " << "L" << metka << endl;

}

else if(temp->retchild(0)->rettext()=="=")

{

mod << "MOV edx," << temp->retchild(0)->retchild(0)->rettext() << endl;

mod << "CMP edx," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JNE " << "L" << metka << endl;

}

else if(temp->retchild(0)->rettext()=="<>")

{

mod << "MOV edx," << temp->retchild(0)->retchild(0)->rettext() << endl;

mod << "CMP edx," << temp->retchild(0)->retchild(1)->rettext()<< endl;

mod << "JE " << "L" << metka << endl;

}

if(temp->childscount()>2 && temp->retchild(2)->rettext()=="ELSE")

{

ProvCikl(temp->retchild(1));

mod << "JMP L" << metka+1 << endl;

mod << "L" << metka << ":" << endl;

ProvCikl(temp->retchild(3));

mod << "L" << ++metka << ":" << endl;

}

else

{

ProvCikl(temp->retchild(1));

mod << "MOV edx," << temp->retchild(0)->retchild(0)->rettext() << endl;

mod << "L" << metka << ":" << endl;

metka++;

}

}

void Syntax::setindex(ASTnode \*temp)

{

static int n=1;

if(temp->childscount()==1)

{

jok.jk1=temp->retchild(0)->rettext();

jok.jk2="eax";

fi.push\_back(jok);

}

else

for(int i=0 ; i<temp->childscount() ; i++)

{

setindex(temp->retchild(i));

if(temp->retchild(i)->rettype()==IDENT && idstringtype(temp->retchild(i)->rettext())=="REAL")

{

realvar=1;

}

if(temp->retchild(i)->rettype()==IDENT)

{

jok.jk1=temp->retchild(i)->rettext();

for(int j=0 ; j<(int)fi.size(); j++)

{

if(fi[j].jk1==temp->retchild(i)->rettext())

{

n=0;

break;

}

}

if(n)

{

switch(joknum)

{

case 0:

jok.jk2="eax";

break;

case 1:

jok.jk2="ebx";

break;

case 2:

jok.jk2="ecx";

break;

case 3:

jok.jk2="edx";

break;

}

fi.push\_back(jok);

joknum++;

}

n=1;

}

else if(temp->retchild(i)->rettype()==CONSTANT)

{

jok.jk1=jok.jk2=temp->retchild(i)->rettext();

fi.push\_back(jok);

}

}

}

string Syntax::getindex(string s)

{

for(int i=0 ; i<(int)fi.size() ; i++)

{

if(fi[i].jk1==s) return fi[i].jk2;

}

return "";

}

void Syntax::opproc(ASTnode \*temp)

{

if(temp->rettext()=="sqrt")

{

if(temp->retchild(0)->rettype()==IDENT && idstringtype(temp->retchild(0)->rettext())=="REAL")

{

realvar=1;

}

if(realvar==1)

{

mod << "MOV eax," << temp->retchild(0)->rettext() << endl;

mod << "FLD eax" << endl;

mod << "FSQRT" << endl;

mod << "FSTP eax" << getindex(temp->retchild(0)->rettext()) << endl;

mod << "MOV " << temp->retchild(0)->rettext() << ","<< "eax" << endl;

}

else

{

mod << "MOV eax," << temp->retchild(0)->rettext() << endl;

mod << "FILD eax" << endl;

mod << "FSQRT" << endl;

mod << "FISTP eax" << getindex(temp->retchild(0)->rettext()) << endl;

mod << "MOV " << temp->retchild(0)->rettext() << ","<< "eax" << endl;

}

realvar=0;

}

else if(temp->rettext()=="printreal")

{

mod << "printf(\"%f\"," << temp->retchild(0)->rettext() << ")" << endl;

}

else if(temp->rettext()=="printint")

{

mod << "printf(\"%i\"," << temp->retchild(0)->rettext() << ")" << endl;

}

else if(temp->rettext()=="scanint")

{

mod << "invoke crt\_scanf, ADDR OutputFormat, ADDR " << temp->retchild(0)->rettext() << endl;

}

else if(temp->rettext()=="scanreal")

{

mod << "invoke crt\_scanf, ADDR OutputFormat, ADDR " << temp->retchild(0)->rettext() << endl;

}

else if(temp->rettext()=="printstring")

{

mod << "printf("<< temp->retchild(0)->rettext() << ")" << endl;

}

}

int Syntax::GSTPS(string s)

{

for(int i=0 ; i < (int)TI.size() ; i++)

{

if(TI[i].var==s)

{

return TI[i].kol;

}

}

return 0;

}

void Syntax::prisv(ASTnode \*temp)

{

if(temp->retchild(0)->rettype()!=IDENT && temp->retchild(0)->rettype()!=CONSTANT)

{

prisv(temp->retchild(0));

}

if(temp->retchild(1) && temp->retchild(1)->rettype()!=IDENT && temp->retchild(1)->rettype()!=CONSTANT)

{

prisv(temp->retchild(1));

}

if(realvar==1)

{

if(temp->rettext()=="-")

{

if(temp->childscount()==1)

{

temp->settext(temp->retchild(0)->rettext());

mod << "NEG " << getindex(temp->retchild(0)->rettext()) << endl;

}

else

{

temp->settext(temp->retchild(0)->rettext());

mod << "FSUB " << getindex(temp->retchild(0)->rettext()) << "," << getindex(temp->retchild(1)->rettext()) << endl;

}

}

else if(temp->rettext()=="+")

{

temp->settext(temp->retchild(0)->rettext());

mod << "FADD " << getindex(temp->retchild(0)->rettext()) << "," << getindex(temp->retchild(1)->rettext()) << endl;

}

else if(temp->rettext()=="\*")

{

temp->settext(temp->retchild(0)->rettext());

mod << "FMUL " << getindex(temp->retchild(0)->rettext()) << "," << getindex(temp->retchild(1)->rettext()) << endl;

}

else if(temp->rettext()=="/")

{

temp->settext(temp->retchild(0)->rettext());

mod << "FDIV " << getindex(temp->retchild(0)->rettext()) << "," << getindex(temp->retchild(1)->rettext()) << endl;

}

else if(temp->rettext()==":=")

{

mod << "MOV " << temp->retchild(0)->rettext() << "," << getindex(temp->retchild(1)->rettext()) << endl;

return;

}

}

else

{

if(temp->rettext()=="-")

{

if(temp->childscount()==1)

{

temp->settext(temp->retchild(0)->rettext());

mod << "NEG " << getindex(temp->retchild(0)->rettext()) << endl;

}

else

{

temp->settext(temp->retchild(0)->rettext());

mod << "SUB " << getindex(temp->retchild(0)->rettext()) << "," << getindex(temp->retchild(1)->rettext()) << endl;

}

}

else if(temp->rettext()=="+")

{

temp->settext(temp->retchild(0)->rettext());

mod << "ADD " << getindex(temp->retchild(0)->rettext()) << "," << getindex(temp->retchild(1)->rettext()) << endl;

}

else if(temp->rettext()=="\*")

{

temp->settext(temp->retchild(0)->rettext());

mod << "IMUL " << getindex(temp->retchild(0)->rettext()) << "," << getindex(temp->retchild(1)->rettext()) << endl;

}

else if(temp->rettext()=="/")

{

temp->settext(temp->retchild(0)->rettext());

mod << "IDIV " << getindex(temp->retchild(0)->rettext()) << "," << getindex(temp->retchild(1)->rettext()) << endl;

}

else if(temp->rettext()==":=")

{

mod << "MOV " << temp->retchild(0)->rettext() << "," << getindex(temp->retchild(1)->rettext()) << endl;

return;

}

}

}

void Syntax::scancode()

{

code >> t >> word >> page;

}

int Syntax::ischislo(string s)

{

for (int i = 0; i < (int)s.length(); i++)

{

if(s[i]=='.') continue;

if ((s[i] < '0') || (s[i] > '9')) return 0;

}

return 1;

}

int Syntax::Synanalys()

{

if(START()!=NULL)

{

return 1;

}

else return 0;

}

ASTnode\* Syntax::START()

{

ASTnode \*prog = new ASTnode(NONTERMINAL, "PROGRAMM", NULL, NULL);

root=prog;

ASTnode \*qwe = NULL;

while(!code.eof())

{

try

{

qwe=MODULE();

if(qwe==NULL)

{

//cout << word << endl<< "Syntax Error" << endl;

return NULL;

}

prog->AddChild(qwe);

scancode();

if(word!=";") throw 10;

qwe=O();

prog->AddChild(qwe);

if(word!="BEGIN")

{

throw 20;

root->ASTremove(root);

return NULL;

}

prog->AddChild(P());

if(word!="END")

{

throw 21;

root->ASTremove(root);

return NULL;

}

prog->AddChild(new ASTnode(TERMINAL, word, NAME(), NULL));

scancode();

if(word!=".")

{

throw 4;

root->ASTremove(root);

return NULL;

}

code >> t;

}

catch(int i)

{

switch(i)

{

case 20: cout << "Expected BEGIN"; break;

case 21: cout << "Expected END"; break;

case 4: cout << "Expected ."; break;

case 10: cout << "Expected ;"; break;

case 12: cout << "Undeclared identifier"; break;

case 2: cout << "Syntax error"; break;

case 3: cout << "Expected number"; break;

case 5: cout << "Expected OF"; break;

case 6: cout << "Expected type of identifier"; break;

case 7: cout << "Expected name"; break;

default: break;

}

root->ASTremove(root);

cout <<" String: " << page;

return NULL;

}

}

prog->printnode(prog);

return prog;

}

ASTnode\* Syntax::MODULE()

{

ASTnode \*name=NULL;

scancode();

if(word!="MODULE") return NULL;

else

{

name = NAME();

if(name==NULL) return NULL;

return new ASTnode(KEYWORD, "MODULE", name, NULL);

}

}

ASTnode\* Syntax::NAME()

{

scancode();

nam=word;

if(t!="j") throw 7;

return new ASTnode(IDENT, word, NULL, NULL);

}

ASTnode\* Syntax::O()

{

string temp;

ASTnode \*ob=NULL;

ob=new ASTnode(NONTERMINAL, "DECLARATION", NULL, NULL);

scancode();

while(word=="CONST" || word=="VAR" || word=="PROCEDURE")

{

if(word=="CONST") ob->AddChild(C());

if(word=="VAR") ob->AddChild(V());

if(word=="PROCEDURE") ob->AddChild(F());

scancode();

}

if(ob->childscount()==0)

{

delete ob;

return NULL;

}

return ob;

}

ASTnode\* Syntax::P()

{

ASTnode \*temp=NULL;

temp=new ASTnode(NONTERMINAL, "NACHINKA", NULL, NULL);

scancode();

while(t=="j" || word=="WHILE" || word=="IF")

{

if(t=="j") temp->AddChild(OO());

else if(word=="WHILE" || word=="IF") temp->AddChild(PO());

scancode();

}

if(temp->childscount()==0) return NULL;

return temp;

}

ASTnode\* Syntax::PO()

{

if(word=="IF") return IFELSE();

else if(word=="WHILE") return CIKL();

else return NULL;

}

ASTnode\* Syntax::IFELSE()

{

ASTnode \*temp=NULL;

temp=new ASTnode(OPIF, "IF", NULL, NULL);

scancode();

temp->AddChild(BOOL());

if(word!="THEN")

{

cout << "Error: Expected THEN";

root->ASTremove(root);

return NULL;

}

temp->AddChild(P());

if(word=="ELSE")

{

temp->AddChild(new ASTnode(OPELSE, word, NULL, NULL));

temp->AddChild(P());

}

else if(word=="ELSEIF")

{

while(word=="ELSEIF")

{

temp->AddChild(new ASTnode(OPELSIF, word, NULL, NULL));

scancode();

temp->AddChild(BOOL());

if(word!="THEN")

{

cout << "Error: Expected THEN";

root->ASTremove(root);

return NULL;

}

temp->AddChild(P());

}

if(word=="ELSE")

{

temp->AddChild(new ASTnode(OPELSE, word, NULL, NULL));

temp->AddChild(P());

}

}

if(word!="END")

{

cout << "Error: Expected END";

root->ASTremove(root);

return NULL;

}

return temp;

}

ASTnode\* Syntax::CIKL()

{

ASTnode \*temp=NULL;

temp=new ASTnode(OPCIKLE, "WHILE", NULL, NULL);

scancode();

temp->AddChild(BOOL());

if(word!="DO")

{

cout << "Error: Expected DO";

root->ASTremove(root);

return NULL;

}

temp->AddChild(P());

if(word!="END")

{

cout << "Error: Expected END";

root->ASTremove(root);

return NULL;

}

return temp;

}

ASTnode\* Syntax::OO()

{

ASTnode \*temp=NULL;

string ident;

ident=word;

scancode();

if(word=="(")

{

temp=new ASTnode(PROCEDURE, ident, NULL, NULL);

FACTPAR(temp);

if(word!=")")

{

temp->ASTremove(temp);

cout << "Error";

return NULL;

}

scancode();

if(word!=";")

{

temp->ASTremove(temp);

cout << "Error";

return NULL;

}

}

else if(word==":=") temp=new ASTnode(ASSIGN, word, new ASTnode(IDENT, ident, NULL, NULL), A());

if(word!=";") throw 10;

return temp;

}

ASTnode\* Syntax::A()

{

ASTnode \*temp=NULL;

scancode();

temp=ARIFM();

return temp;

}

void Syntax::printid()

{

for(int i=0 ; i<(int)TI.size() ; i++)

cout << TI[i].var << " " << TI[i].type << " " << TI[i].kol << " " << TI[i].mas<< endl;

}

int Syntax::check(string w)

{

if(w=="OR") return 17;

if(w=="AND") return 18;

if(w=="+") return 11;

if(w=="-") return 12;

if(w=="\*") return 13;

if(w=="/" || w=="DIV") return 14;

if(w=="MOD") return 15;

if(w=="j") return 3;

if(w=="c") return 2;

return 0;

}

ASTnode\* Syntax::BOOL()

{

ASTnode \*temp=NULL;

string w;

temp=JOIN();

while(word=="OR")

{

w=word;

scancode();

temp=new ASTnode(OR, w, temp, JOIN());

}

return temp;

}

ASTnode\* Syntax::JOIN()

{

ASTnode \*temp=NULL;

string w;

temp=EQUA();

while(word=="AND")

{

w=word;

scancode();

temp=new ASTnode(AND, w, temp, EQUA());

}

return temp;

}

ASTnode\* Syntax::EQUA()

{

ASTnode \*temp=NULL;

string w;

temp=REL();

while(word=="=" || word=="<>")

{

w=word;

scancode();

temp=new ASTnode(AND, w, temp, REL());

}

return temp;

}

ASTnode\* Syntax::REL()

{

ASTnode \*temp=NULL;

string w;

temp=ARIFM();

if(word=="<" || word==">" || word==">=" || word=="<=")

{

w=word;

scancode();

return new ASTnode(SRAV, w, temp, ARIFM());

}

else return temp;

}

ASTnode\* Syntax::ARIFM()

{

ASTnode \*temp=NULL;

string w;

temp=term();

while(word=="+" || word=="-")

{

w=word;

scancode();

temp=new ASTnode(check(w), w, temp, term());

}

return temp;

}

ASTnode\* Syntax::term()

{

ASTnode \*temp=NULL;

string w;

temp=unary();

while(word=="\*" || word=="/" || word=="MOD" || word=="DIV")

{

w=word;

scancode();

temp=new ASTnode(check(w), w, temp, unary());

}

return temp;

}

ASTnode\* Syntax::unary()

{

if(word=="-")

{

scancode();

return new ASTnode(SUB, "-", unary(), NULL);

}

else if(word=="NOT")

{

scancode();

return new ASTnode(NOT, "NOT", unary(), NULL);

}

else return factor();

}

ASTnode\* Syntax::factor()

{

ASTnode \*temp=NULL;

string w;

if(t=="c")

{

temp = new ASTnode(CONSTANT, word, NULL, NULL);

scancode();

return temp;

}

else if(t=="j")

{

w=word;

scancode();

if(idtype(w)==-1) throw 12;

if(idtype(w)!=55) return new ASTnode(IDENT, w, NULL, NULL);

else return offset(w);

}

else if(word=="(")

{

scancode();

temp=BOOL();

if(word==")")

{

scancode();

return temp;

}

}

return NULL;

}

ASTnode\* Syntax::offset(string qwe)

{

ASTnode \*temp=NULL, \*t1=NULL;

//cout << word;

if(word!="[")

{

//cout << "yes";

throw 2;

return NULL;

}

scancode();

t1=new ASTnode(MUL, "\*", new ASTnode(CONSTANT, word, NULL, NULL), new ASTnode(CONSTANT, sidtype(qwe), NULL, NULL));

temp=new ASTnode(ADD, "+", new ASTnode(IDENT, qwe, NULL, NULL), t1);

scancode();

if(word!="]")

{

throw 2;

return NULL;

}

scancode();

return temp;

}

string Syntax::sidtype(string w)

{

for(int i=0 ; i<(int)TI.size() ; i++)

{

if(TI[i].var==w)

{

if(TI[i].type=="INTEGER" || TI[i].type=="CARDINAL" || TI[i].type=="REAL")

{

return "4";

}

else if(TI[i].type=="CHAR") return "1";

}

}

return " ";

}

int Syntax::idtype(string w)

{

for(int i=0 ; i<(int)TI.size() ; i++)

{

if(TI[i].var==w)

{

if(TI[i].mas==1) return 55;

else if(TI[i].type=="INTEGER" || TI[i].type=="CARDINAL" || TI[i].type=="REAL" || TI[i].type=="CONST")

{

return 4;

}

else if(TI[i].type=="CHAR") return 1;

}

}

return -1;

}

string Syntax::idstringtype(string w)

{

for(int i=0 ; i<(int)TI.size() ; i++)

{

if(TI[i].var==w)

{

return TI[i].type;

}

}

return "-1";

}

void Syntax::FACTPAR(ASTnode \*temp)

{

int str=0;

do

{

scancode();

if(t=="l") str=1;

else if(t!="j" && t!="c")

{

temp->ASTremove(temp);

cout << "Error";

return;

}

if(str==1)

{

temp->AddChild(new ASTnode(STR, word, NULL, NULL));

str=0;

if(TIcheck(word))

{

rty.kol=varnum;

varnum++;

rty.mas=0;

rty.proc="glob";

rty.var=HelpStr(word);

rty.type="string";

TI.push\_back(rty);

}

}

else if(t=="j") temp->AddChild(new ASTnode(IDENT, word, NULL, NULL));

else if(t=="c") temp->AddChild(new ASTnode(CONSTANT, word, NULL, NULL));

scancode();

}

while(word==",");

}

int Syntax::TIcheck(string s)

{

for(int i=0 ; i<(int)TI.size() ; i++)

{

if(TI[i].var==s) return 0;

}

return 1;

}

string Syntax::HelpStr(string s)

{

if(s=="endl")

{

return "\\n";

}

else return s;

}

ASTnode\* Syntax::F()

{

ASTnode \*temp=NULL, \*name=NULL, \*parameters=NULL, \*temp2=NULL, \*endname=NULL;

if(word!="PROCEDURE") return NULL;

temp=new ASTnode(KEYWORD, "PROCEDURE", NULL, NULL);

name=NAME();

rty.proc=currentproc;

currentproc=word;

rty.type="PROCEDURE";

rty.mas=0;

rty.kol=0;

rty.var=nam;

TI.push\_back(rty);

if(name==NULL)

{

delete name;

delete temp;

return NULL;

}

scancode();

if(word=="(")

{

parameters=FORMALPAR();

if(parameters==NULL)

{

delete name;

delete temp;

return NULL;

}

name->AddChild(parameters);

scancode();

}

if(word!=";")

{

delete name;

delete temp;

delete parameters;

return NULL;

}

temp2=O();

name->AddChild(temp2);

if(word!="BEGIN")

{

delete name;

delete temp;

delete parameters;

delete temp2;

return NULL;

}

name->AddChild(P());

if(word!="END")

{

delete name;

delete temp;

delete parameters;

delete temp2;

return NULL;

}

endname=NAME();

if(endname==NULL)

{

delete name;

delete endname;

delete temp;

delete parameters;

delete temp2;

return NULL;

}

delete endname;

/\*scancode();

if(word!=";")

{

delete name;

delete endname;

delete temp;

delete parameters;

delete temp2;

return NULL;

}\*/

temp->AddChild(name);

currentproc="glob";

return temp;

}

ASTnode\* Syntax::FORMALPAR()

{

ASTnode \*temp=NULL, \*temp2=NULL, \*par=NULL;

temp=new ASTnode(NONTERMINAL, "PARAMETERS", NULL, NULL);

fparam=1;

do

{

scancode();

if(word=="VAR")

{

scancode();

par=new ASTnode(KEYWORD, word, NULL, NULL);

temp->AddChild(par);

}

else

{

temp->AddChild(new ASTnode(KEYWORD, word, NULL, NULL));

}

scancode();

if(word!=":")

{

delete temp;

delete temp2;

return NULL;

}

temp2=TYPE();

if(temp2==NULL)

{

delete temp;

delete temp2;

return NULL;

}

par->AddChild(temp2);

scancode();

}

while(word!=")");

fparam=0;

return temp;

}

ASTnode\* Syntax::C()

{

ASTnode \*temp=NULL, \*name=NULL, \*value=NULL, \*temp2=NULL;

if(word!="CONST") return NULL;

temp=new ASTnode(KEYWORD, "CONST", NULL, NULL);

name=NAME();

rty.proc=currentproc;

rty.var=nam;

rty.type="CONST";

rty.mas=0;

rty.kol=1;

if(name==NULL)

{

delete name;

delete temp;

return NULL;

}

scancode();

if(word!="=")

{

delete name;

delete temp;

return NULL;

}

value=C1();

rty.zn=value->rettext();

temp2=new ASTnode(ASSIGN, "=", name, value);

temp->AddChild(temp2);

scancode();

if(word!=";")

{

delete name;

delete temp;

delete temp2;

delete value;

return NULL;

}

TI.push\_back(rty);

return temp;

}

ASTnode\* Syntax::C1()

{

ASTnode \*temp=NULL;

scancode();

if(word=="-")

{

scancode();

temp=new ASTnode(SUB, "-", CHISLO(), NULL);

return temp;

}

else return CHISLO();

}

ASTnode\* Syntax::CHISLO()

{

if(!ischislo(word)) return NULL;

return new ASTnode(CONSTANT, word, NULL, NULL);

}

ASTnode\* Syntax::V()

{

ASTnode \*temp=NULL, \*name=NULL;

string ide;

if(word!="VAR") return NULL;

temp=new ASTnode(KEYWORD, "VAR", NULL, NULL);

name=NAME();

if(name==NULL) return NULL;

temp->AddChild(name);

scancode();

/\*while(word==",")

{

name=NAME();

if(name==NULL)

{

delete temp;

return NULL;

}

temp->AddChild(name);

code >> t >> word;

}\*/

if(word!=":") return NULL;

temp->AddChild(TYPE());

scancode();

if(word!=";")

{

delete name;

delete temp;

return NULL;

}

return temp;

}

ASTnode\* Syntax::TYPE()

{

scancode();

if(word!="INTEGER" && word!="REAL" && word!="CHAR" && word!="BOOLEAN" && word!="ARRAY") throw 6;

if(word=="ARRAY")

{

if(fparam)

{

scancode();

scancode();

word="ARRAY OF " + word;

}

else

{

rty.var=nam;

rty.mas=1;

scancode();

if(word!="[") throw 2;

scancode();

if(word!="0") throw 3;

scancode();

if(word!=".") throw 4;

scancode();

if(word!=".") throw 4;

scancode();

if(!ischislo(word)) throw 3;

else rty.kol=atoi(word.c\_str())+1;

scancode();

if(word!="]") throw 2;

scancode();

if(word!="OF") throw 5;

scancode();

if(t!="k") throw 6;

else rty.type=word;

word="ARRAY OF " + word;

rty.proc=currentproc;

rty.zn="?";

}

}

else

{

rty.proc=currentproc;

rty.var=nam;

rty.type=word;

rty.mas=0;

rty.kol=1;

rty.zn="?";

}

if(fparam==0) TI.push\_back(rty);

return new ASTnode(KEYWORD, word, NULL, NULL);

}

*main.cpp:*

#include <iostream>

#include "lexic.h"

#include "syntax.h"

using namespace std;

int main(int argc, char\* argv[])

{

Lexic L;

Syntax S;

int a;

setlocale(0, "rus");

cout << "Какую программу компилировать(1 или 2)?\n";

cin >> a;

if (a == 1)// если передаем аргументы, то argc будет больше 1(в зависимости от кол-ва аргументов)

{

L.openfile("input.mod");// вывод второй строки из массива указателей на строки(нумерация в строках начинается с 0 )

}

else if(a==2)

{

L.openfile("input2.mod");

}

if(L.Lexanalys())

{

if(S.Synanalys())

{

if(S.GenerCode())

{

cout << "Компиляция завершена!!!" << endl;

}

}

}

return 0;

}