**Министерство науки и высшего образования РФ**

**ФГБОУ ВО  
«БРЯНСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»**

**ФАКУЛЬТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ**

**Кафедра «Информатика и программное обеспечение»**

**Расчетно-графическая работа**

**по дисциплине «Теория автоматов и формальных языков»**

направление подготовки 09.03.04 «Программная инженерия»

на тему:

**Основы разработки компилятора**

Выполнил студ. гр. О-18-ПРИ-рпс-Б:

**Лядов В. С.**

Проверил:

к.т.н., доц. **Трубаков Е. О.**

Брянск 2021

# СОДЕРЖАНИЕ

[СОДЕРЖАНИЕ 2](#_Toc59748346)

[ВВЕДЕНИЕ 3](#_Toc59748347)

[1. ТЕКСТОВОЕ ОПИСАНИЕ ЯЗЫКА 4](#_Toc59748348)

[2. ОПИСАНИЕ ЯЗЫКА НА ОСНОВЕ РЕГУЛЯРНЫХ ВЫРАЖЕНИЙ 6](#_Toc59748349)

[3. КОНТЕКСТНО - СВОБОДНАЯ ГРАММАТИКА 7](#_Toc59748350)

[4. АВТОМАТНАЯ ГРАММАТИКА 8](#_Toc59748351)

[5. НЕДЕТЕРМИНИРОВАННЫЙ АВТОМАТ 9](#_Toc59748352)

[6. ДЕТЕРМИНИРОВАННЫЙ АВТОМАТ 11](#_Toc59748353)

[7. МИНИМАЛЬНЫЙ ДЕТЕРМИНИРОВАННЫЙ АВТОМАТ 12](#_Toc59748354)

[8. ЛЕКСИЧЕСКИЙ АНАЛИЗ 14](#_Toc59748355)

[9. ТАБЛИЦА КОМПИЛЯТОРА 19](#_Toc59748356)

[10. ДЕРЕВО РАЗБОРА 21](#_Toc59748357)

[11. ГЕНЕРАЦИЯ КОДА 25](#_Toc59748358)

[12. ОПТИМИЗАЦИЯ КОДА 28](#_Toc59748359)

[ЗАКЛЮЧЕНИЕ 31](#_Toc59748360)

[СПИСОК ЛИТЕРАТУРЫ 32](#_Toc59748361)

# Введение

В расчетно-графической работе была разработана программа, выполняющая генерацию кода для языка, который содержит оператор ветвления арифметические выражения, разделённые символом «;» (точка с запятой). Арифметические выражения состоят из идентификаторов, шестнадцатеричных чисел, знака присваивания (:=), знаков операций +, –, \*, / и круглых скобок. Шестнадцатеричными числами считать последовательность цифр и символов a, b, c, d, e, f, начинающуюся с цифры (например, 89, 45ac, 0abc).

Данная пояснительная записка содержит подробное описание этапов разработки расчетно-графической работы. Их можно разделить на две группы – теоретические и практические.

Теоретические этапы можно разделить на стадии. Первая стадии – текстовое описание языка. Здесь был разработан лексический анализ. Вторая стадия – это описание языка на основе регулярных выражений и построение контекстно-свободной и автоматной грамматики. Третья стадия – построение конечного автомата. Сначала был построен недетерминированный конечный автомат, впоследствии он был преобразован в детерминированный, а после была произведена минимизация детерминированного.

Практические этапы заключались в создании компилятора. Сначала был написан код, выполняющий лексический анализ и составляющий таблицу имен. После этого был выполнен синтаксический анализ, построено дерево разбора, на основе которого генерировался псевдоассемблерный код. Заключительный этап состоял в оптимизации данного кода.

В работе каждая стадия будет рассмотрена подробнее.

# Текстовое описание языка

Первый этап – лексический анализ и построение таблицы имен. Входом компилятора, является строка символов некоторого алфавита.

Работа лексического анализатора состоит в том, чтобы сгруппировать определенные терминальные символы в единые синтаксические объекты, называемые лексемами*.* Лексема – это строка символов, с которой связывается лексическая структура, состоящая из пары вида (тип лексемы, некоторые данные). Первой компонентой пары является синтаксическая категория, такая, как «константа» или «идентификатор», а вторая – указатель: в ней указывается адрес ячейки, хранящей информацию об этой конкретной лексеме. Таким образом, лексический анализатор – это транслятор, входом которого служит строка символов, представляющая исходную программу, а выходом — последовательность лексем. Этот выход образует вход синтаксического анализатора.

Мною был выполнен лексический анализа математического выражения *z = 2x2 – 2y2 + 5*. В результате была получена последовательность лексем и таблица идентификаторов.

*Таблица 1*

|  |  |  |
| --- | --- | --- |
| Номер элемента | Идентификатор | Тип |
| 1 | z | Переменная |
| 2 | x | Переменная |
| 3 | y | Переменная |
| 4 | 2 | Целочисленная Константа |
| 5 | 5 | Целочисленная Константа |

*Таблица 1. Таблица идентификаторов*

Также в результате фазы лексического анализа была получена следующая последовательность лексем.

<ид>1 = <ид>4 \* <ид>2 ^ <ид>4 - <ид>4 \* <ид>3 ^ <ид>4 + <ид>5

# Описание языка на основе регулярных выражений

Регулярные выражения представляют собой один из способов определения регулярных языков. Они позволяют определять допустимые цепочки декларативным способом. Поэтому регулярные выражения используются в качестве входного языка во многих системах, обрабатывающих цепочки символов.

Алгебра регулярных выражений состоит из констант (ε, ∅) и переменных для обозначения языков и операторов для задания трех операций: объединение ( + или | ), конкатенация ( . ) и итерация ( \* ).

На рисунке 1 представлено описание языка при помощи регулярных выражений.

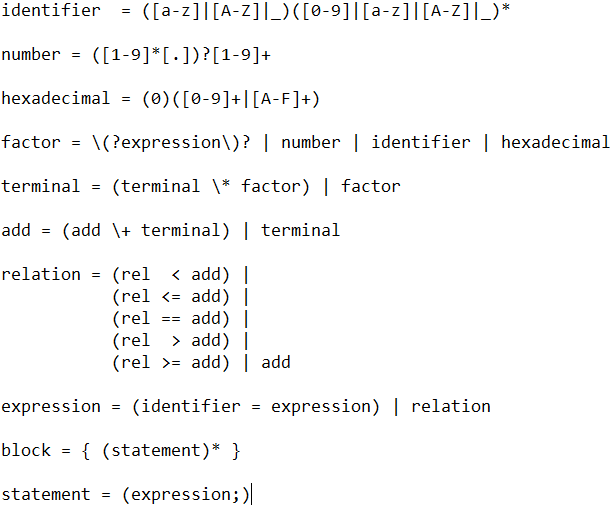


Рис. 1. Описание языка при помощи регулярных выражений

# Контекстно - свободная грамматика

Задача этапа – построение контекстно-свободной грамматики.

Грамматика – это описание способа построения предложений некоторого языка. Она относится к способу определения языков – порождению цепочек символов. Грамматику языка можно описать различными способами.

Формально грамматика G определяется как четвёрка G (VT, VN, R, S), где:

VT – множество терминальных символов, или алфавит терминальных символов;

VN – множество нетерминальных символов, или алфавит нетерминальных символов;

R – множество правил (продукций) грамматики вида α → β

S – целевой (начальный) символ грамматики.

С учетом всех особенностей заданного языка мною была построена следующая контекстно-свободная грамматика (рис. 2).

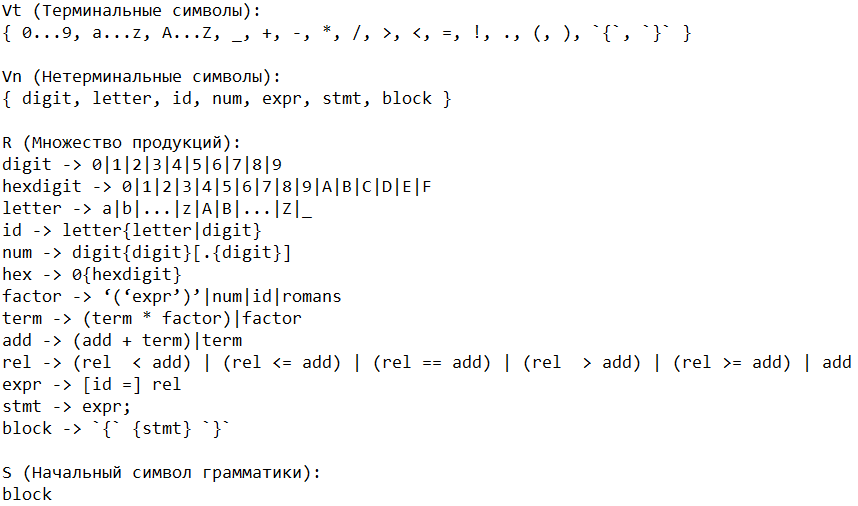


Рис. 2. Описание языка с помощью КС-грамматики

# Автоматная грамматика

Из четырех типов грамматик контекстно-свободные грамматики являются наиболее важными с точки зрения приложений к языкам программирования и компиляции. С помощью КС-грамматики можно определить большую часть структуры языка программирования. При построении грамматик, задающих конструкции языков программирования, часто приходится прибегать к их преобразованию, чтобы порождаемый язык приобрел нужную структуру.

На этом этапе необходимо построить автоматную грамматику, путем преобразования праволинейной.

Перейти от праволинейной грамматики к автоматной можно расширением нетерминального словаря следующим образом: S→x1S1 ; S1→x2S2 ; S2→x3А и т. д., оставляя в правой части правила один терминальный и один нетерминальный символы или один терминальный.

Мною была построена автоматная грамматика (рис. 3).

**Vt (Терминальные символы):**

{ 0...9, a...z, A...Z, \_, +, -, \*, /, >, <, =, !, ., (, ), ‘{‘, ‘}’}

**id ->** [a..z] **id**

**id ->** [a..z]

**int ->** [0..9] **int**

**int ->** [0..9]

**op ->** [+,-,\*,/]

**Vn (Нетерминальные символы):**

{ ifelse,ifpart,cond\_op,cond\_right,ifblock,asgn,expr,right\_h,end }

**R (Множество продукций):**

**ifelse -> ‘if’ ifpart**

**ifpart ->** [id|int] **cond\_op**

**ifpart ->** [id|int] **ifblock**

**cond\_op -> op cond\_right**

**cond\_right ->** [id|int] **ifblock**

**ifblock ->** { id **asgn** }

**asgn ->** ‘:=’ **expr**

**expr ->** [id|int] **right\_h**

**right\_h ->** op **expr**

**right\_h ->** op **end**

**end ->** [id|int]

# Недетерминированный автомат

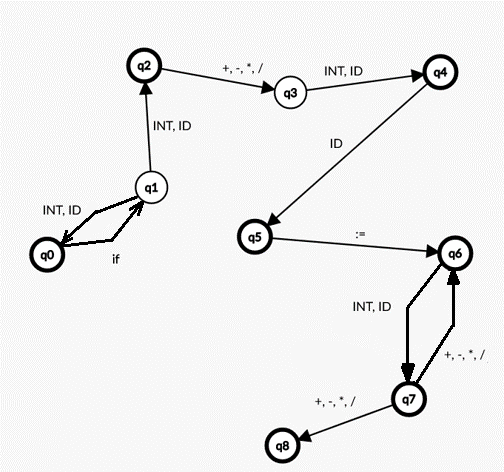
Недетерминированный конечный автомат определяется выражением *A*=(*Q, V, М, S, Z*), где *Q* – множество (алфавит) внутренних состояний; *V* – входной алфавит; *М* – функция переходов, представляющая отображение вида: *V\*Q →P*(*Q*); *P*(*Q*) – множество подмножеств из *Q*; *S∈Q* – множество начальных состояний; *Z∈Q* – множество заключительных состояний; *S≠Z*. Для автоматной грамматики из прошлого пункта была построена таблица состояний, а затем с ее помощью построена таблица недетерминированного конечного автомата:

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ifelse | ifpart | cond\_op | cond\_right | ifblock | asgn | expr | right\_h | end |
| q0 | q1 | q2 | q3 | q4 | q5 | q6 | q7 | q8 |

Таблица 2. Таблица состояний недетерминированного автомата

Таблица 3. Таблица переходов недетерминированного конечного автомата

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | INT | ID | if | else | + | - | \* | / | := |
| q0 |  |  | q1 |  |  |  |  |  |  |
| q1 | q0, q2 | q0, q2 |  |  |  |  |  |  |  |
| q2 |  |  |  |  | q3 | q3 | q3 | q3 |  |
| q3 | q4 | q4 |  |  |  |  |  |  |  |
| q4 |  | q5 |  |  |  |  |  |  |  |
| q5 |  |  |  |  |  |  |  |  | q6 |
| q6 | q7 | q7 |  |  |  |  |  |  |  |
| q7 |  |  |  |  | q6, q8 | q6, q8 | q6, q8 | q6, q8 |  |
| q8 |  |  |  |  |  |  |  |  |  |



# Детерминированный автомат

Детерминированный конечный автомат определяется выражением *А*= (*Q, V, М, S, Z*), где *Q* – алфавит состояний; *V* – входной алфавит; *М* – функция переходов вида: *Q\*V→Р*(*Q*); *S* – начальное состояние; *Z* – множество заключительных состояний; Для недетерминированного автомата из предыдущего пункта был построен детерминированный автомат.

Таблица 3. Таблица переходов недетерминированного конечного автомата

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | INT | ID | if | else | + | - | \* | / | := |
| q0 |  |  | q1 |  |  |  |  |  |  |
| q1 | q0, q2 | q0, q2 |  |  |  |  |  |  |  |
| q2 |  |  |  |  | q3 | q3 | q3 | q3 |  |
| q3 | q4 | q4 |  |  |  |  |  |  |  |
| q4 |  | q5 |  |  |  |  |  |  |  |
| q5 |  |  |  |  |  |  |  |  | q6 |
| q6 | q7 | q7 |  |  |  |  |  |  |  |
| q7 |  |  |  |  | q6, q8 | q6, q8 | q6, q8 | q6, q8 |  |
| q8 |  |  |  |  |  |  |  |  |  |

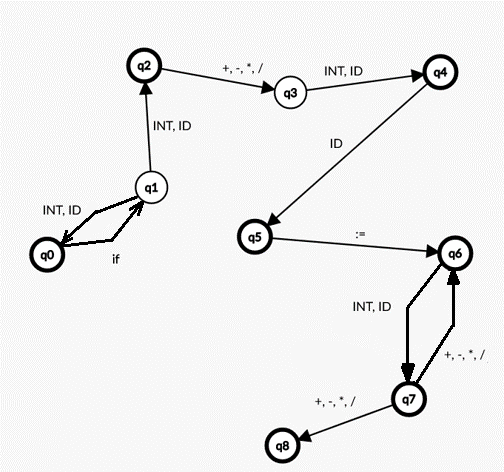


Рис. 3. Граф детерминированного конечного автомата

# Минимальный детерминированный автомат

Минимизация автомата, эквивалентного полученному в предыдущем разделе полностью определенному автомату, проводится в два этапа. Сначала множество состояний автомата разбивается на классы эквивалентности, а затем строится минимальный (приведенный) автомат.

Была построена треугольная таблица эквивалентности (рис. 9), клетки которой соответствуют всем парам рабочих состояний.

Не вычеркнутые клетки результирующей таблицы соответствуют всем парам эквивалентных состояний.



Рис. 4. Таблица эквивалентности

Исходя из полученной таблицы были получены следующие классы эквивалентности:

r0 = {q0}; r3 = {q1, q3, q8} r2 = {q2}; r4 = {q4}; r5 = {q5};

r6 = {q6}; r7 = {q7}.

Таблица 4. Таблица переходов минимального автомата.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | INT | ID | if | else | + | - | \* | / | := |
| q0 |  |  | q138 |  |  |  |  |  |  |
| q2 |  |  |  |  | q138 | q138 | q138 | q138 |  |
| q4 |  | q5 |  |  |  |  |  |  |  |
| q5 |  |  |  |  |  |  |  |  | q6 |
| q6 | q7 | q7 |  |  |  |  |  |  |  |
| q7 |  |  |  |  | q6, q138 | q6, q138 | q6, q138 | q6, q138 |  |
| q138 | q0, q2, q4 | q0, q2, q4 |  |  |  |  |  |  |  |

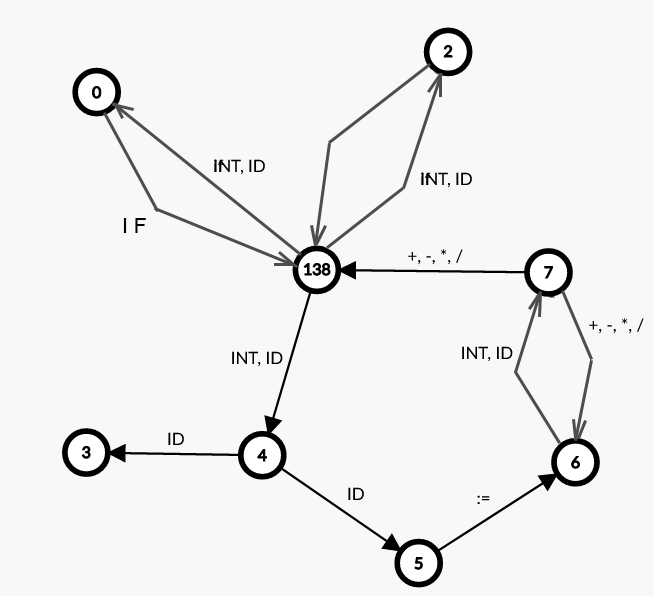


Рис. 5. Граф минимального автомата

# Лексический анализ

Первая стадия разработки проекта – лексический анализ и построение таблицы имен. На этом этапе программа принимала строку символов некоторого алфавита, а далее лексический анализатор группировал определенные терминальные символы в единые синтаксические объекты – лексемы.

На листинге 1 представлено описание класса Lexer, отвечающего за разбиение входных данных на лексемы. На листинге 2 представлен год основной функции класса – scan ( );

Листинг 1

class Lexer {

private:

    char peek = ' ';

    std::ifstream \*input;

    std::map<std::string, Token\*> words;

    void reserve(Word \*w);

    void readch();

    bool readch(char c);

public:

    static int line;

    bool iseof = false;

    std::string outp = "";

    Lexer(std::ifstream \*inp);

    Token\* scan();

    void printTable();

};

Листинг 2

Token\* Lexer::scan() {

    for ( ; !iseof; readch() ) {

        if (peek == ' ' || peek == '\t') continue;

        else if (peek == '\n') line = line + 1;

        else break;

    }

    switch (peek)

    {

    case ':':

        if (readch('&')) { outp += ":="; return (Token\*)&(Word::\_asgn); }

        else             { outp += ":"; return new Token((Tag)':'); }

    case '&':

        if (readch('&')) { outp += "&&"; return (Token\*)&(Word::\_and); }

        else             { outp += "&"; return new Token((Tag)'&'); }

    case '|':

        if (readch('|')) { outp += "||"; return (Token\*)&(Word::\_or); }

        else             { outp += "|"; return new Token((Tag)'|'); }

    case '=':

        if (readch('=')) { outp += "=="; return (Token\*)&(Word::\_eq); }

        else             { outp += "="; return new Token((Tag)'='); }

    case '!':

        if (readch('=')) { outp += "!="; return (Token\*)&(Word::\_ne); }

        else             { outp += "!"; return new Token((Tag)'!'); }

    case '<':

        if (readch('=')) { outp += "<="; return (Token\*)&(Word::\_le); }

        else             { outp += "<"; return new Token((Tag)'<'); }

    case '>':

        if (readch('=')) { outp += ">="; return (Token\*)&(Word::\_ge); }

        else             { outp += ">"; return new Token((Tag)'>'); }

    }

    if (isdigit(peek)) {

        int v = 0;

        do {

            v = 10\*v + (peek - '0');

            readch();

        } while ( isdigit(peek) );

        if (peek != '.') {

            if (words.find(std::to\_string(v)) != words.end()) {

                outp += "\_" + std::to\_string(v) + "\_";

                return words[std::to\_string(v)];

            }

            Num \*n = new Num(v);

            words.insert({std::to\_string(v), n});

            outp += "\_" + n->toString() + "\_";

            return n;

        }

        float x = v;

        float d = 10;

        for (;;) {

            readch();

            if (!isdigit(peek)) break;

            x = x + (peek - '0') / d;

            d = d \* 10;

        }

        outp += std::to\_string(x);

        if (words.find(std::to\_string(v)) != words.end()) {

            outp += "\_" + std::to\_string(v) + "\_";

            return words[std::to\_string(v)];

        }

        Real \*n = new Real(v);

        words.insert({std::to\_string(v), n});

        outp += "\_" + n->toString() + "\_";

        return n;

    }

    if (isalpha(peek)) {

        std::string s;

        do {

            s += peek; readch();

        } while ( isalnum(peek) );

        if (words.find(s) != words.end()) {

            outp += "\_" + words[s]->toString() + "\_";

            return words[s];

        }

        Word \*w = new Word(s, Tag::ID);

        words.insert({s, w});

        outp += "\_" + words[s]->toString() + "\_";

        return w;

    }

    Token \*tok = new Token((Tag)peek);

    outp += std::string(1, peek);

    peek = ' ';

    return tok;

}

# Таблица компилятора

Таблица идентификаторов (символов) – это структура данных, которая используется компилятором для хранения информации о различных элементах программы, таких как переменные, константы, имена функций или процедур, т.е. идентификаторах.

Рассмотрим таблицу компилятора, построенную для примера исходного выражения из прошлого пункта. Таблица идентификаторов формируется с помощью метода открытой адресации.

Листинг 3

/\*\*

 \*       OPEN ADDRESSING HASH TABLE

 \*       WITH QUADRATIC PROBING

 \*/

struct Pair {

    bool is\_del = false;

    std::string key;

    Token \* value;

    Pair(std::string k, Token \*val) {

        key = k;

        value = val;

    }

};

class Hashtable {

private:

    int size;

    Pair\*\* arr;

public:

    int hash(std::string key);

    Hashtable();

    Hashtable(int sz);

    Token\* find(std::string key);

    void   insert(std::string key, Token\* value);

    void   remove(std::string key);

    void   print();

};

На листинге 4 представлен функция вставки значения в таблицу

Листинг 4

void Hashtable::insert(std::string key, Token \*value)

{

    int h = hash(key);

    int pos = h;

    do

    {

        if (!arr[pos] || arr[pos]->is\_del)

        {

            arr[pos] = new Pair(key, value);

            return;

        }

        else

        {

            if (arr[pos]->key == key)

                return;

            pos = (pos + 1) % size;

        }

    } while (pos != h); // if pos == h again, the array is full

    throw("hashtable is full");

}

# Дерево разбора

Синтаксический анализ, или разбор, как его еще называют — это процесс, в котором исследуется строка лексем и устанавливается, удовлетворяет ли она структурным условиям, явно сформулированным в определении синтаксиса языка.

Выходом анализатора служит дерево, которое представляет синтаксическую структуру, присущую исходной программе.

Для выполнения стадии построения дерева разбора сначала была разработана структура самого дерева, в частности его узлов (листинг 5). Для примера ниже представлены три самых базовых узла дерева – Node, Expr и Id.

Листинг 5

class Node {

public:

    int lexline = 0;

    static int labels;

    Node() {

        lexline = Lexer::line;

    }

    void error(std::string s) {

        throw ("near line " + std::to\_string(lexline) + ": " + s);

    }

    int newLabel() {

        return ++labels;

    }

    void emitlabel(int i) {

        std::cout << "L" << std::to\_string(i) << ":";

    }

    void emit(std::string s) {

        std::cout << "\t" << s << std::endl;

    }

    virtual std::string toString() { return ""; }

};

class Id: public Expr {

public:

    int offset; // relative address

    Id() { }

    Id(Word \*id, Type \*p, int b): Expr(id, p) {

        offset = b;

    }

};

class Expr: public Node {

public:

    Token \*op;

    Type \*type;

    Expr() {};

    Expr(Token \*tok, Type \*p) {

        op = tok;

        type = p;

    }

    virtual Expr\* gen() { return this; }

    virtual Expr\* reduce() { return this; }

    virtual void jumping(int t, int f) {

        emitjumps(toString(), t, f);

    }

    void emitjumps(std::string test, int t, int f) {

        if (t != 0 && f != 0) {

            emit("if " + test + " goto L" + std::to\_string(t));

            emit("goto L" + std::to\_string(f));

        }

        else if (t != 0)

            emit("if " + test + " goto L" + std::to\_string(t));

        else if (f != 0)

            emit("iffalse " + test + " goto L" + std::to\_string(f));

        else ; // nothing since both t and f fall through

    }

    virtual std::string toString() {

        return op->toString();

    }

};

На листинге 6 представлена часть кода парсера, отвечающего за построение дерева разбора:

Листинг 6

Stmt \* Parser::assign() {

    Stmt \*stmt;

    Token \*t = look;

    match(Tag::ID);

    Id \*id = top->get(t);

    if (id == nullptr)

        error(t->toString() + " undeclared");

    if ((int)look->tag == '=') {                // S -> id = E ;

        move();

        stmt = new Set(id, boolean());

    }

    else {                                      // S -> L = E ;

        Access \*x = offset(id);

        match('=');

        stmt = new SetElem(x, boolean());

    }

    match(';');

    return stmt;

}

Expr \*Parser::boolean()

{

    Expr\* x = join();

    while( look->tag == Tag::OR ) {

        Token \*tok = look;

        move();

        x = new Or(tok, x, join());

    }

    return x;

}

Expr \*Parser::join()

{

    Expr\* x = equality();

    while( look->tag == Tag::AND ) {

        Token \*tok = look;

        move();

        x = new And(tok, x, equality());

    }

    return x;

}

Expr \*Parser::equality()

{

    Expr\* x = rel();

    while( look->tag == Tag::EQ || look->tag == Tag::NE) {

        Token \*tok = look;

        move();

        x = new Rel(tok, x, rel());

    }

    return x;

}

# Генерация кода

Дерево, построенное синтаксическим анализатором, используется для того, чтобы получить перевод входной программы. Этот перевод может быть программой в машинном языке, но чаще он бывает программой в промежуточном языке, например, язык ассемблера. Выходом синтаксического анализатора служит дерево, с помощью этого дерева и информации, хранящейся в таблице имен, можно построить объектный код (листинг 7). В моем случае парсер переводит программу в промежуточный three-address код, который затем транслируется специальным классом в код, подобный ассемблеру.

Листинг 7

void \_if (std::istringstream &iss) {

    try {

        std::string left, op, right, \_goto, label, jmp;

        if (left != "true")

            iss >> op >> right;

        iss >> \_goto >> label;

        if (left == "true") jmp = "JMP";

        else if (op == ">=") jmp = "JGE";

        else if (op == "==") jmp = "JE";

        else if (op == "<=") jmp = "JLE";

        else if (op == "!=") jmp = "JNE";

        else if (op == ">")  jmp = "JG";

        else if (op == "<")  jmp = "JL";

        std::cout << "CMP " << left << ", " << right << "\n";

        std::cout << jmp << " " << label << "\n";

    } catch (std::exception e) {};

}

void \_iffalse (std::istringstream &iss) {

    try {

        std::string left, op, right, \_goto, label, jmp;

        iss >> left;

        if (left != "false")

            iss >> op >> right;

        iss >> \_goto >> label;

        if (left == "false") jmp = "JMP";

        else if (op == ">=") jmp = "JLE";

        else if (op == "==") jmp = "JNE";

        else if (op == "<=") jmp = "JGE";

        else if (op == "!=") jmp = "JE";

        else if (op == ">")  jmp = "JL";

        else if (op == "<")  jmp = "JG";

        std::cout << "CMP " << left << ", " << right << "\n";

        std::cout << jmp << " " << label << "\n";

    } catch (std::exception e) {};

}

void threeAddress(std::string left, std::istringstream &iss)

{

    std::string asgn, first, op, second;

    iss >> asgn >> first >> op >> second;

    std::cout << "LOAD " << first << "\n";

    if (op != "") {

        if      (op == "\*") std::cout << "MPY " << second << "\n";

        else if (op == "/") std::cout << "DIV " << second << "\n";

        else if (op == "+") std::cout << "ADD " << second << "\n";

        else if (op == "-") std::cout << "SUB " << second << "\n";

    }

    std::cout << "STORE " << left << "\n";

}

bool islabel(std::string s) {

    return s[s.length() - 1] == ':';

}

Листинг 8

L1:

  LOAD 32.649998

  STORE a

LOAD abc

L3:

  LOAD 453

  STORE b

L4:

  CMP a, b

  JLE L6

L5:

  LOAD 25

  MPY 748

  STORE t1

  LOAD 54.669998

  ADD t1

  STORE z

  JMP L2

L6:

  LOAD 748

  STORE z

L2:

# Оптимизация кода

Последний этап – оптимизация сгенерированного кода. Для оптимизации мы используем правила, код функций которых представлен на листинге 8. Для примера представлена функция, удаляющая неиспользуемые переменные из кода Результат – оптимизированный код – представлен ниже.

Листинг 9

void removeUnused() {

    std::set<Word\*> unused;

    std::map<std::string, Token\*>::iterator it;

    Word \*w;

    Token \*tok;

    for (std::string line: \*lines) {

        lex->nextline(line);

        while (!lex->iseof) {

            tok = lex->scan();

        }

    }

    for (it = lex->words.begin(); it != lex->words.end(); it++) {

        if (it->second && it->second->tag == Tag::ID) {

            w = (Word\*) it->second;

            if (w->use\_count <= 1 && !isLabel(w))

                unused.insert(w);

        }

    }

    std::cout << "Unused vars removed:";

    for (auto w: unused) std::cout << w->toString() << " ";

    std::cout << "\n";

    std::vector<std::string>::iterator list\_i;

    std::set<Word\*>::iterator set\_i;

    for (list\_i = lines->begin(); list\_i != lines->end(); list\_i++) {

        lex->nextline(\*list\_i);

        while (!lex->iseof) {

            tok = lex->scan();

            // check this token among unused variables

            // and delete line if found

            Word \*tmp = (Word\*) tok;

            set\_i = unused.find(tmp);

            if (set\_i != unused.end()) {

                lines->erase(list\_i);

                break;

            }

        }

    }

    // join list of strings4

    std::ostringstream os;

    std::copy(lines->begin(), lines->end() - 1,

        std::ostream\_iterator<std::string>(os, "\n"));

    os << \*lines->rbegin();

    result = os.str();

}

Оптимизированный код:

L1:

  LOAD 32.649998

  STORE a

L3:

  LOAD 453

  STORE b

L4:

  CMP a, b

  JLE L6

L5:

  LOAD 25

  MPY 748

  STORE t1

  LOAD 54.669998

  ADD t1

  STORE z

  JMP L2

L6:

  LOAD 748

  STORE z

L2:

# ЗАКЛЮЧЕНИЕ

В ходе выполнения данной расчётно-графической работы была создан язык, описанный при помощи регулярных выражений, КС – грамматик, конечных автоматов, а также разработана программа, позволяющая компилировать конструкции разработанного языка в объектный код псевдо-ассемблера для последующей оптимизации.

Были пройдены все стадии разработки компилятора, такие как: лексический анализ входных данных, создание таблицы идентификаторов и последовательности лексем, построение дерева разбора, генерация и оптимизация объектного кода.

За время разработки и тестирования было выяснено, что программа работает без критических ошибок и полностью соответствует требованиям.

В результате проделанной работы были получены важные навыки работы с регулярными выражениями, навыки построения КС – грамматик и конечных автоматов, а также улучшены навыки алгоритмического программирования.

# СПИСОК ЛИТЕРАТУРЫ

1. Князьков В.С., Волченская Т.В. - Введение в теорию автоматов - Национальный Открытый Университет "ИНТУИТ", 2016 - 89с.
2. Алымова, Е. В. Конечные автоматы и формальные языки : учебник / Е. В. Алымова. В. М. Деундяк. А. М. Пеленнцын ; Южный федеральный университет. - Ростов-на-Дону : Таганрог : Издательство Южного федерального университета. 2018. - 292 с.
3. Наоми Седер, Python – Экспресс – курс / Седер Наоми – 3-е изд. – Спб.: Питер, 2019. – 480 с.
4. Саломаа А. Жемчужины теории формальных языков. М.: Мир, 2016.
5. Хопкрофт Дж. Э., Мотвани Р., Ульман Дж. Д. Введение в теорию автоматов, языков и вычислений, 4-е изд. М.: Вильямс, 2015.