Санкт-Петербургский государственный институт
точной механики и оптики (технический университет)

Кафедра «Компьютерные технологии»

А.А. Штучкин, А.А. Шалыто

Совместное использование теории построения компиляторов и SWITCH-технологии (на примере построения калькулятора)

Проектная документация

Проект создан в рамках
«Движения за открытую проектную документацию»
http://is.ifmo.ru

Санкт-Петербург
2003

Содержание

1. Введение

В работе [1] достаточно подробно (главы 6-9) рассмотрен вопрос о программной реализации калькулятора.

Авторы, конечно, понимают, что в этой книге построение калькулятора описывается для демонстрации возможностей языка, а не методов проектирования калькуляторов, но все-таки…

Почти полностью заимствовав описание калькулятора из работы [1], авторы использовали теорию построения компиляторов [2] для его проектирования.

Однако при реализации калькулятора, спроектированного на основе работы [2], возникает ряд проблем, которые могут быть устранены при совместном применении указанной теории и SWITCH-технологии [3, 4], которая базируется на использовании конечных автоматов.

Перечислим недостатки, которые позволяет устранить указанная технология.

В работе [2], как и в других работах по теории компиляторов [5], излагаются основы их проектирования, но переход от проекта компилятора к его программной реализации недостаточно формализован.

Еще один недостаток известного подхода состоит в том, что некоторые из используемых алгоритмов описываются традиционным путем с помощью псевдоязыка, в то время как это может быть сделано с помощью автоматов. Это позволяет на всех этапах построения компиляторов применять автоматы, описывая их графами переходов.

Изложенный подход позволяет устранить указанные недостатки, а также упростить и сделать более понятным текст программы, реализующей калькулятор.

Этот пример был выбран по нескольким причинам:

- любой калькулятор является мини-интерпретатором со своим синтаксисом и семантикой и при желании может быть достроен до полноценного компилятора;
- именно калькулятор неоднократно применялся для демонстрации возможностей технологий и языков программирования, например в работе [1]. Это позволяет провести сравнение нашего проекта с уже написанным;
- построение компиляторов/интерпретаторов тесно связано с теорией автоматов. Поэтому построение калькулятора с использованием SWITCH-технологии, также основанной на теории автоматов, достаточно естественно.

Процесс разработки компиляторов подробно рассмотрен в работе [2]. По мере изложения данного проекта во многом повторяется путь, описанный в этой книге. Это позволит специалистам в теории компиляторов легче сравнить классический подход с предлагаемым. Главное отличие описываемого подхода состоит в том, что в проекте вся логика представлена, реализована и документирована в терминах автоматов. Это позволяет сделать ее более прозрачной и удобной в понимании даже для человека, не знакомого с этой областью программирования.

Используя подход, изложенный в работе [2], построим калькулятор по частям и изложим все этапы проектирования программ этого класса. При этом на этапе проектирования строятся (в виде графов переходов автоматов) все основные алгоритмы, составляющие проект. Разрабатывается проектная документация, являющаяся неотъемлемой частью SWITCH-технологии [3]. В ней приводятся схемы связей, определяющие интерфейс автоматов, графы переходов автоматов, а также листинги модулей программы. Приведен пример протокола работы программы, описывающий ее поведение в терминах автоматов.

При реализации данного проекта применяется язык C++ (без использования классов) из-за его распространенности и удобной библиотеки.

2. Проектирование

2.1. Декомпозиция задачи

В теории построения компиляторов [2] в большинстве случаев применяется следующее разбиение на подзадачи (блоки):

1. Лексический анализатор.
2. Синтаксический анализатор.
3. Генератор машинного кода.

Так как в данной работе рассматривается не компилятор, а интерпретатор, то «Генератор машинного кода» заменяется на «Эмулятор». Для калькулятора в качестве эмулятора применяется стековая машина. Блоки 1-3 в программе выполняются последовательно в том порядке, в котором они указаны в списке. Схема взаимодействия этих модулей представлена на рис. 1.

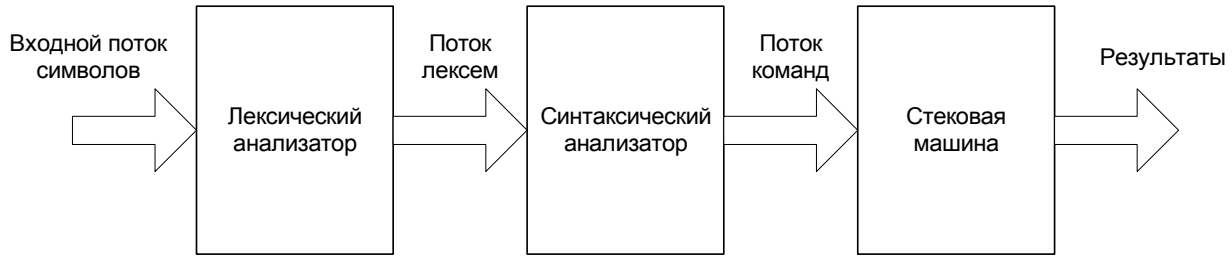

Рис. 1. Схема взаимодействия модулей программы

В представленном проекте внутренняя структура программы соответствует логической структуре. Каждому логическому блоку на рис. 1 соответствует отдельный модуль программы.

2.2. Лексический анализатор

Лексический анализатор производит предварительную обработку текста, разбивая его на *лексемы*.

Лексемой называется последовательность символов, воспринимаемая остальной программой как единое целое. В рассматриваемом проекте применяются только весьма простые лексемы – число (возможно, с десятичной точкой), символы арифметических операций (+, -, *, /), круглые скобки, точка с запятой (символ конца выражения). Также на этапе лексического анализа формируются атрибуты лексем, не влияющие на синтаксический разбор, но требующиеся в дальнейшем при вычислении выражения.

Атрибутом называется некоторая дополнительная информация (кроме типа) о каждой конкретной лексеме. Например, пусть **N** (от слова Number) – лексема, соответствующая числу. Во входном потоке она представлена последовательностью цифр и, возможно, десятичной точкой. Тогда значение этого числа во внутреннем представлении может передаваться в качестве атрибута данной лексемы. Этот атрибут будет в дальнейшем использован стековой машиной. Если представить лексему как пару <тип, атрибут>, то, например, входная строка

```
(2 + 3)*10.2;
```

трансформируется в последовательность лексем

```
<(, > <N, 2> <+, > <N, 3> <), > <*, > <N, 10.2> <;, >
```

Другие лексемы данного проекта атрибутов не имеют.

Для упрощения передачи лексемы синтаксическому анализатору, в данном проекте используются две глобальные переменные – *LexemeType* и *LexemeNum*, соответствующие типу и атрибуту текущей лексемы. Так как типов лексем немного, то для наглядности и простоты отладки переменная *LexemeType* имеет тип *char* (символ) и в ней хранится либо

сама лексема, если она односимвольная, либо символ ‘N’, если тип лексемы – число.

В простых случаях, к которым можно отнести и рассматриваемый пример, лексический анализатор может быть реализован «примитивно» так, как это показано в листинге 1.

Листинг 1. «Примитивная» **реализация лексического анализатора**

```
// Глобальные переменные для передачи лексемы синтаксическому анализатору
char LexemeType;
double LexemeNum;

// Последний считанный символ
char t = ' ';

void lex(void)
{
	// Удаление пробелов, табуляций и переводов строк
	while (t == ' ' || t == '\t' || t == '\n')
		t = getchar();
	LexemeNum = 0;
	if (isdigit(t))
	{
		// Обработка числа
		while (isdigit(t))
		{
			LexemeNum = LexemeNum * 10 + t - '0';
			t = getchar();
		}

		// Обработка дробной части числа
		double f = 1;
		if (t == '.')
		{
			t = getchar();
			while (isdigit(t))
			{
				f = f / 10;
				LexemeNum = LexemeNum + f * (t - '0');
				t = getchar();
			}
		}
		LexemeType = 'N';
		return;
	}
	else if (t == '+' || t == '-' || t == '*' || t == '/'
	      || t == '(' || t == ')' || t == ';')
	{
		// Обработка остальных лексем
		LexemeType = t;
		return;
	}
	else error();
}
```

Однако такая реализация обладает рядом недостатков. Например, в ней смешивается логика работы и особенности реализации. Другой ее недостаток состоит в том, что лексический анализатор, написанный таким образом, не имеет формальной связи с языком лексем, который он разбирает. Для того чтобы ее установить, необходимо обратиться к теории регулярных выражений [2, 5, 6].

Приведем регулярные выражения для лексем данного проекта:

1. **N** → digit digit* (‘.’ digit*)?
2. **+** → ‘+’
3. **-** → ‘-’
4. ***** → ‘*’
5. **/** → ‘/’
6. **(** → ‘(’
7. **)** → ‘)’
8. **;** → ‘;’

Символ, заключенный в кавычки, означает сам этот символ, digit – одна из цифр 0-9, знак «*» после выражения обозначает, что это выражение может появиться ноль или более раз, знак «?» – что выражение может появиться ноль или один раз.

Из работы [6] известно, что по регулярным выражениям можно построить конечный автомат, который разбирает язык, задаваемый этими выражениями. Для каждого регулярного выражения построим диаграмму переходов, воспользовавшись стандартными методами построения, изложенными в работе [2]. Диаграмма переходов – это стилизованный граф переходов автомата, распознающего данную лексему. В общем случае автомат может быть недетерминированным. В данном графе переходов начальное состояние обозначается стрелкой с надписью *start*, а конечное – жирным кружком. Например, лексема «+» выразится диаграммой, показанной на рис. 2.

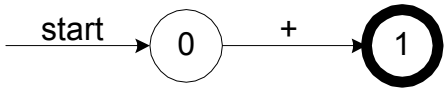

Рис. 2. Диаграмма переходов для лексемы «+»

Аналогичным образом могут быть построены и остальные диаграммы переходов для односимвольных лексем. Единственной сложной лексемой в данном случае является лексема **N**. Диаграмма переходов, построенная для нее, используя формальный метод из работы [2], приведена на рис. 3.

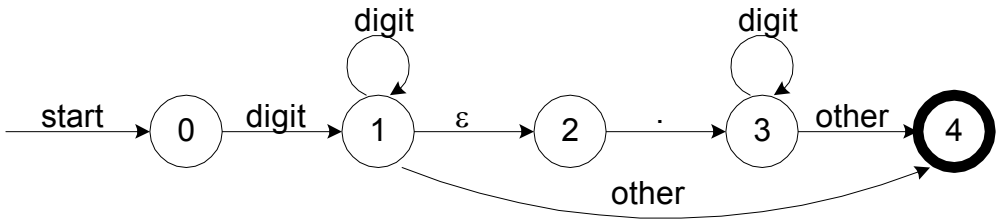

Рис. 3. Диаграмма переходов для числа. Недетерминированный автомат

Диаграмма переходов для данной лексемы содержит «пустой» символ ε, что обозначает недетерминированность автомата. Удалив лишнее состояние 2 и соответствующую дугу, получим эквивалентный детерминированный автомат (рис. 4).

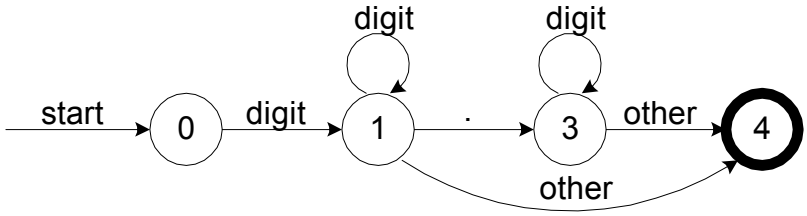

Рис. 4. Диаграмма переходов для числа. Детерминированный автомат

Объединив диаграммы переходов всех лексем, получим общую диаграмму переходов лексического анализатора (рис. 5). Каждое конечное состояние в этой диаграмме будет соответствовать одной лексеме.

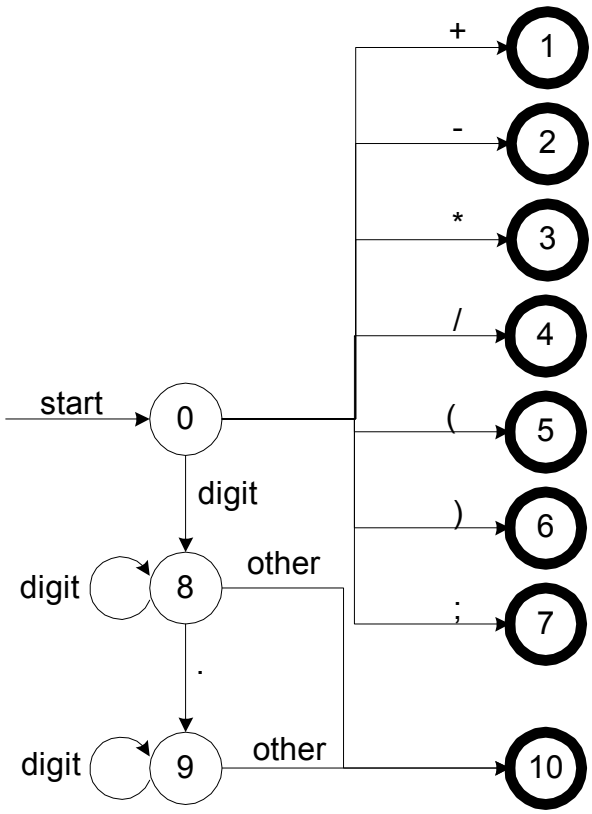

Рис. 5. Лексический анализатор. Диаграмма переходов

В работе [2] для реализации диаграмм переходов предлагается использовать конструкцию языка С, очень похожую на шаблон автомата в SWITCH-технологии. Однако, эта конструкция сложнее для понимания и нет метода ее формального построения по диаграмме переходов. В данной же работе предлагается преобразовать диаграмму переходов в граф переходов автомата, который строится и реализуется в рамках SWITCH-технологии. Для этого все конечные состояния заменим одним (состояние 0). Состояниям 0, 8, 9 в диаграмме переходов соответствуют состояния 1, 2, 3. Объединим переходы, соответствующие односимвольным лексемам, в переход 1-0. Также добавим петлю у состояния 1 для того, чтобы обрабатывать пробелы между символами. Граф переходов получившегося распознающего автомата приведен на рис. 6.

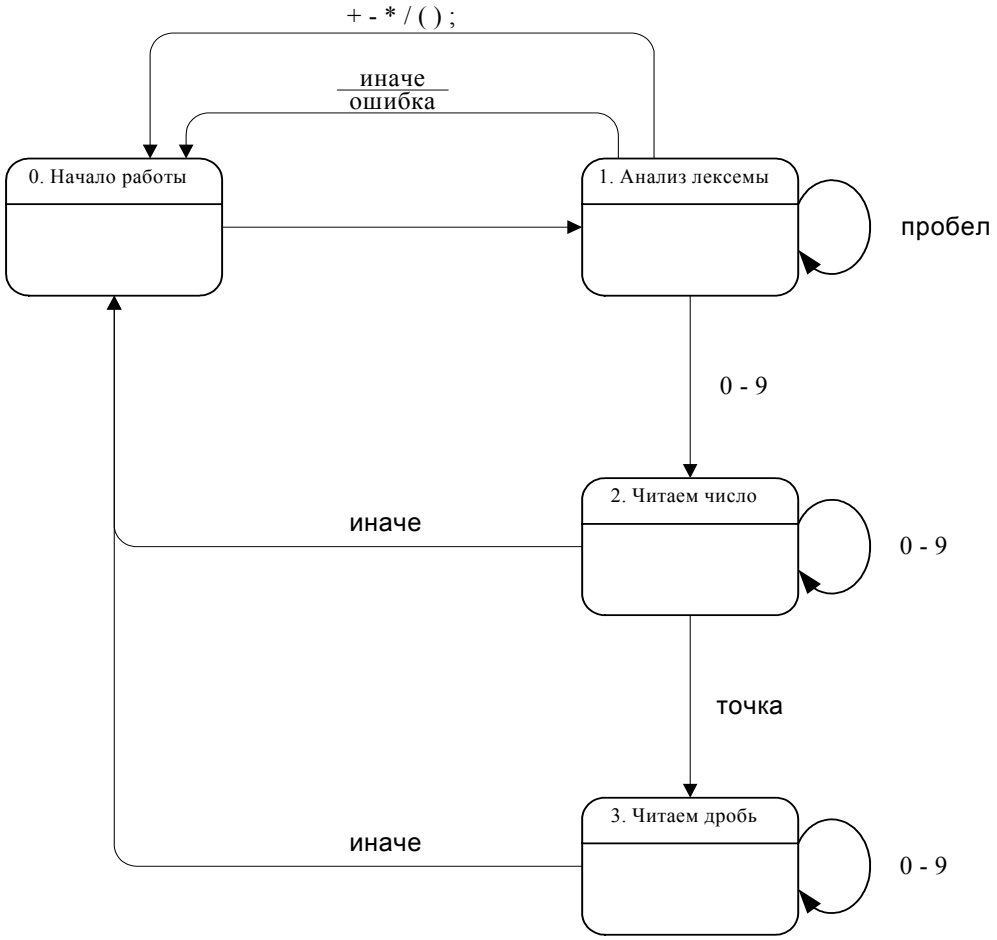

Рис. 6. Лексический анализатор. Граф переходов распознающего автомата

Преобразуем построенный автомат, введя в него выходные воздействия:

- инициализация переменных;
- установка типа лексемы и ее атрибута;
- обновление атрибута лексемы текущей цифрой;
- чтение следующего символа;

Для обозначения конца потока формируется фиктивная лексема, которую в дальнейшем будем обозначать символом «$». Преобразованный автомат представлен на рис. 7.

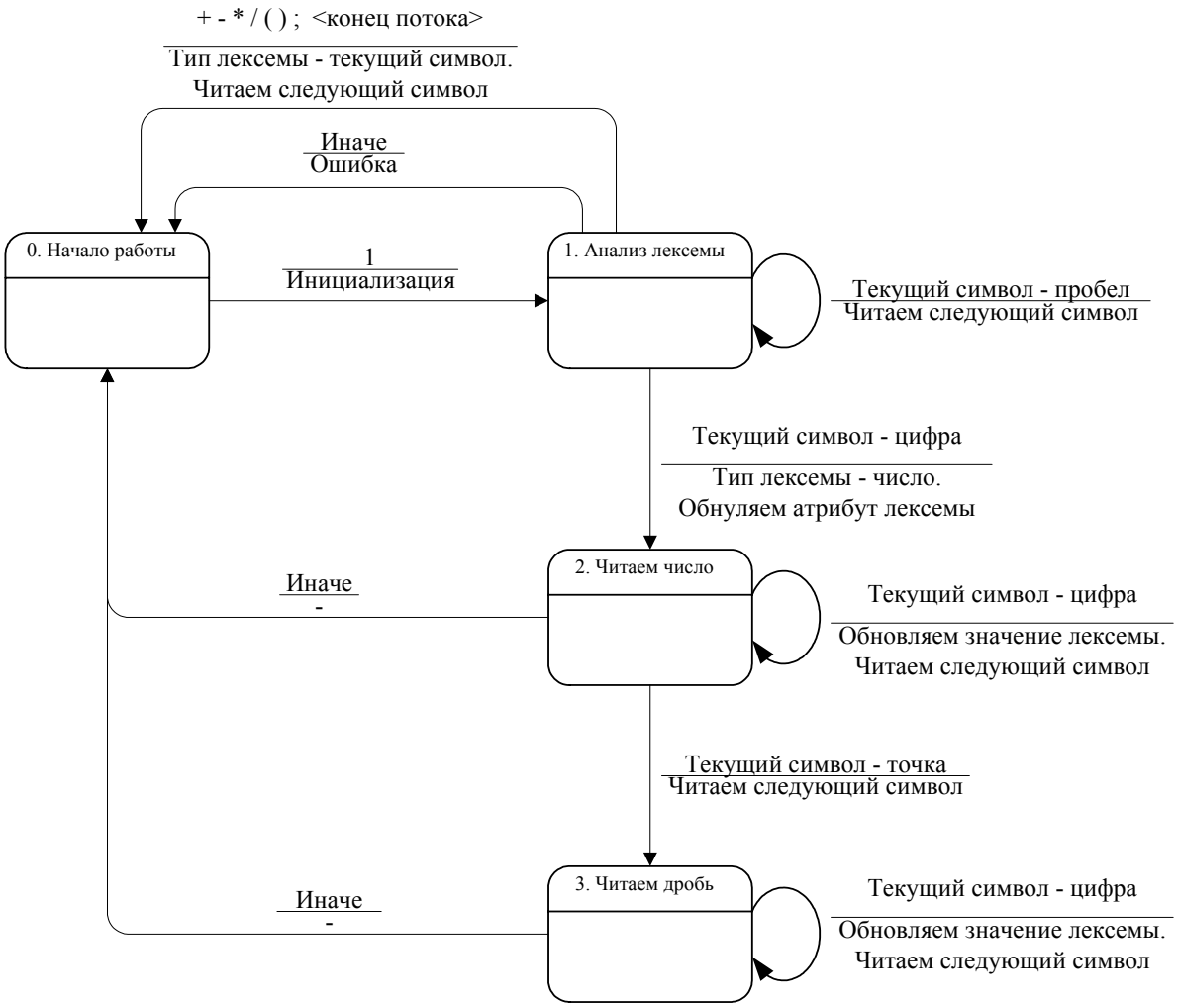

Рис. 7. Лексический анализатор. Граф переходов преобразованного автомата

Таким образом, проектирование лексического анализатора указанным выше методом и реализация получившихся диаграмм переходов с помощью SWITCH-технологии обладает рядом преимуществ перед классическим методом. Во-первых, устранены недостатки, указанные выше. Во-вторых, работа лексического анализатора стала понятнее благодаря графическому отображению графа переходов автомата. В-третьих, реализация автомата с помощью шаблона функции (в рамках SWITCH-технологии) позволяет избежать многих ошибок кодирования программы.

Формализация графа переходов выполнена в разделе 4.

2.3. Синтаксический анализатор

После разбиения входного потока символов на лексемы необходимо произвести синтаксический анализ и сформировать команды для модуля «Эмулятор». Согласно работе [2], для этого построим синтаксический анализатор, разбирающий грамматику G, определяющую арифметические выражения:

1. S → E **;** S
2. S → ε
3. E → T + E
4. E → T **-** E
5. T → F * T
6. T → F / T
7. F → **N**
8. F → **(** E **)**

В описании данной грамматики заглавными буквами (S, E, T, F) обозначены нетерминалы, жирными(**;**, +, **-**, *, /, **N**, **(**, **)**) – терминалы (или лексемы), ε – пустой символ. Эта грамматика порождает арифметические выражения, построенные по обычным правилам и разделенные символом «;». Для облегчения разбора данной грамматики, как изложено в работе [2], преобразуем ее в эквивалентную LL(1)-грамматику G':

1. S → E **;** S
2. S → ε
3. E → T E’
4. E’ → + T E’
5. E’ → **-** T E’
6. E’ → ε
7. T → F T’
8. T’ → * F T’
9. T’ → / F T’
10. T’ → ε
11. F → **N**
12. F → **(** E **)**

Для ее разбора построим стандартный нерекурсивный LL(1)-анализатор. При разборе выражений из входного потока такой анализатор использует стек и таблицу разбора M. Входной поток состоит из последовательности терминалов (в данном случае лексем), заканчивающейся маркером конца. Стек используется для хранения терминалов и нетерминалов, которые еще осталось разобрать, а также специального маркера дна. В начале работы стек содержит начальный символ грамматики (в данном случае S) над маркером дна. Схема синтаксического анализатора и его «окружения» приведена на рис.8.

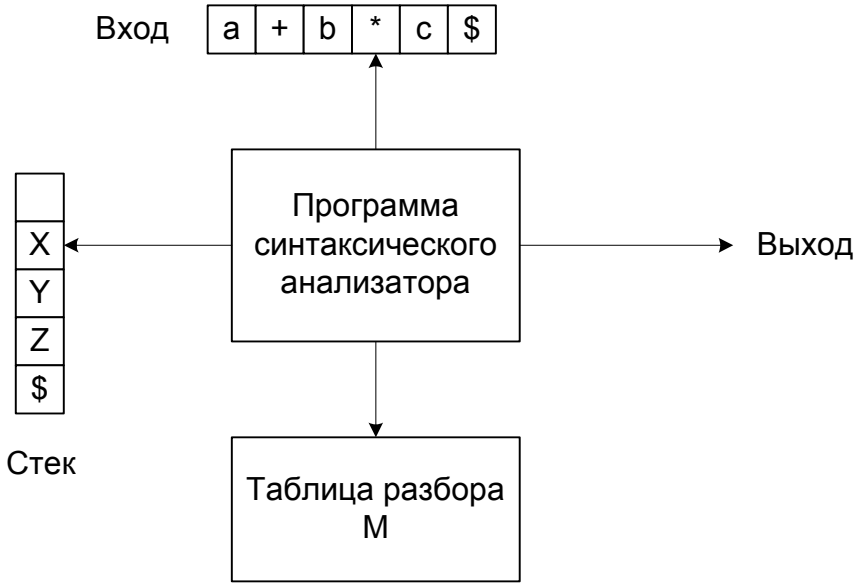

Рис. 8. Схема синтаксического анализатора и его «окружения»

Таблица разбора M представляет собой двухмерный массив записей *M[X, a]*, где *X* – нетерминал, а *a* – терминал (лексема) или символ $. Значением элемента этой таблицы является номер правила вывода, необходимого для разбора, или указание об ошибке. Таблица строится по грамматике еще до начала работы анализатора и остается постоянной при его работе. Это единственная часть анализатора, зависящая от грамматики. Поэтому при изменении грамматики (например, при усложнении выражений) требуется исправить только таблицу, а сам алгоритм работы анализатора останется прежним (при условии, что грамматика все еще будет относиться к классу LL(1)). Метод построения таблицы разбора по грамматике рассмотрен в книге [2]. В данной работе приведен результат применения этого метода к грамматике G' (табл. 1).

Таблица 1. Таблица разбора M грамматики G'

	N	+	-	*	/	(	)	;	$
S	1	ошибка	ошибка	ошибка	ошибка	1	ошибка	ошибка	2
E	3	ошибка	ошибка	ошибка	ошибка	3	ошибка	ошибка	ошибка
E’	ошибка	4	5	ошибка	ошибка	ошибка	6	6	ошибка
T	7	ошибка	ошибка	ошибка	ошибка	7	ошибка	ошибка	ошибка
T’	ошибка	10	10	8	9	ошибка	10	10	ошибка
F	11	ошибка	ошибка	ошибка	ошибка	12	ошибка	ошибка	ошибка

Анализатор управляется программой, которая работает следующим образом [2]. Рассматриваются символ на вершине стека *X* и текущий входной символ *a*. Эти символы определяют действия синтаксического анализатора. Имеются следующие варианты действий:

1. Если $X = a = \$$, то синтаксический анализатор прекращает работу и сообщает об успешном разборе выражения.
2. Если $X = a \neq \$$, то анализатор снимает с вершины стека символ *X* и принимает из входного потока следующий символ.
3. Если $X \neq a$, и *X* – терминал, то выдается сообщение об ошибке.
4. Если же *X* – нетерминал, то проверяется запись *M[X, a]* из таблицы разбора *M*. В ней записан либо номер правила грамматики, либо запись об ошибке. Например, если *M[X, a]* = 5, а в грамматике под номером 5 есть правило $U \rightarrow XYZ$, то синтаксический анализатор замещает U на вершине стека на ZYX (с X на вершине стека). Если же *M[X, a] = «ошибка»*, то анализатор выдает сообщение об ошибке.

На основе изложенного построим алгоритм работы синтаксического анализатора, который можно представить на псевдоязыке следующем образом.

Алгоритм 1. Синтаксический анализатор

Вход: строка *w* и таблица разбора *M*.

Выход: если строка *w* выводится заданной грамматикой, то последовательность правил вывода, иначе – сообщение об ошибке.

Начальные условия: первоначально синтаксический анализатор находится в следующей конфигурации - в стеке находятся символы $S (на его вершине стартовый символ S), во входном потоке – последовательность символов w$.

```
Прочитать символ из входного потока
repeat
    Обозначим через a последний прочитанный из входного потока символ,
    а через X - символ на вершине стека
    if X – терминал или $ then
        if X = a then
            Снять с вершины стека X и прочитать следующий символ
        else error()
    else    /* X – нетерминал */
        if M[X,a] – не ошибка then begin
            Взять правило вывода X→ Y1Y2 …Yk, номер которого равен M[X,a]
            Снять X с вершины стека
            Поместить в стек Yk,Yk-1, … ,Y1
        end
        else error()
until X = $   /* Стек пуст  */
```

Данный алгоритм можно также представить в виде автомата, что не было сделано в работе [2]. Представление автомата в виде графа переходов более наглядно и удобно для описания данного алгоритма по сравнению с его записью на псевдоязыке.

Граф переходов автомата, построенный словесному описанию алгоритма, приведен на рис.9.

Рис. 9. Граф переходов автомата синтаксического анализатора

Таким образом, нерекурсивный LL(1)-анализатор построен. Однако, как и первая версия лексического анализатора (рис.6), он делает лишь половину работы – определяет, является ли данное выражение арифметическим. Калькулятор же должен вычислять

значение выражения. Для этого в алфавит грамматики введем операционные символы (команды). Работают они следующим образом – команды добавляются в правила грамматики и при разборе аналогично терминалам и нетерминалам помещаются в стек. Далее, если на вершине стека появляется команда, то алгоритм посылает ее следующему модулю – эмулятору, формируя таким образом поток команд (рис. 1). Грамматика, дополненная командами, называется атрибутивной грамматикой (АТ-грамматикой).

Команды эмулятору будем обозначать следующим образом:

- @+, @-, @*, @/ - команды ‘+’, ‘-‘, ‘*’, ‘/’ соответственно;
- @N – команда «Поместить в стек последнее число»;
- @P – команда «Вывести результат в выходной поток».

Для правильного вычисления арифметических выражений в грамматику G' добавлены операционные символы таким образом, чтобы обеспечить порядок выдачи команд эмулятору, соответствующий обратной польской записи. Это позволяет правильно расставлять приоритеты операций и корректно вычислять выражения. Заметим, что при добавлении команд таблица разбора M не изменяется.

Преобразованная АТ-грамматика G'' имеет вид:

1. $S \rightarrow E$ **;** @P S
2. $S \rightarrow \varepsilon$
3. $E \rightarrow T\ E'$
4. $E' \rightarrow$ **+** T @+ E'
5. $E' \rightarrow$ **-** T @- E'
6. $E' \rightarrow \varepsilon$
7. $T \rightarrow F\ T'$
8. $T' \rightarrow$ ***** F @* T'
9. $T' \rightarrow$ **/** F @/ T'
10. $T' \rightarrow \varepsilon$
11. $F \rightarrow$ **N** @N
12. $F \rightarrow$ **(** E **)**

Для обработки команд в текстовое описание алгоритма добавим еще один вариант действий:

5. Если X – операционный символ (команда), то снять его с вершины стека и выдать в выходной поток (выполнить).

На псевдоязыке преобразованный алгоритм описывается следующим образом (вход, выход и начальные условия остаются такими же, как и в алгоритме 1):

Алгоритм 2. Синтаксический анализатор АТ-грамматик

```
Прочитать символ из входного потока
repeat
    Обозначим через a последний прочитанный из входного потока символ,
    а через X - символ на вершине стека
    if X – терминал или $ then
        if X = a then
            Снять с вершины стека X и прочитать следующий символ
        else error()
    else
        if X – нетерминал then
            if M[X,a] – не ошибка then begin
                Взять правило вывода X→ Y1Y2 …Yk,
                      номер которого равен M[X,a]
```

```
                Снять X с вершины стека
                Поместить в стек Yk,Yk-1, … ,Y1
            end
            else error()
        else  /* X – команда */
            Выполнить X
            Снять X с вершины стека

until X = $   /* Стек пуст  */
```

Заметим, что добавление команд усложнило алгоритм по сравнению с предыдущей версией. При использовании автомата изменений потребуется гораздо меньше – добавится лишь одна петля в состоянии 1(рис. 10).

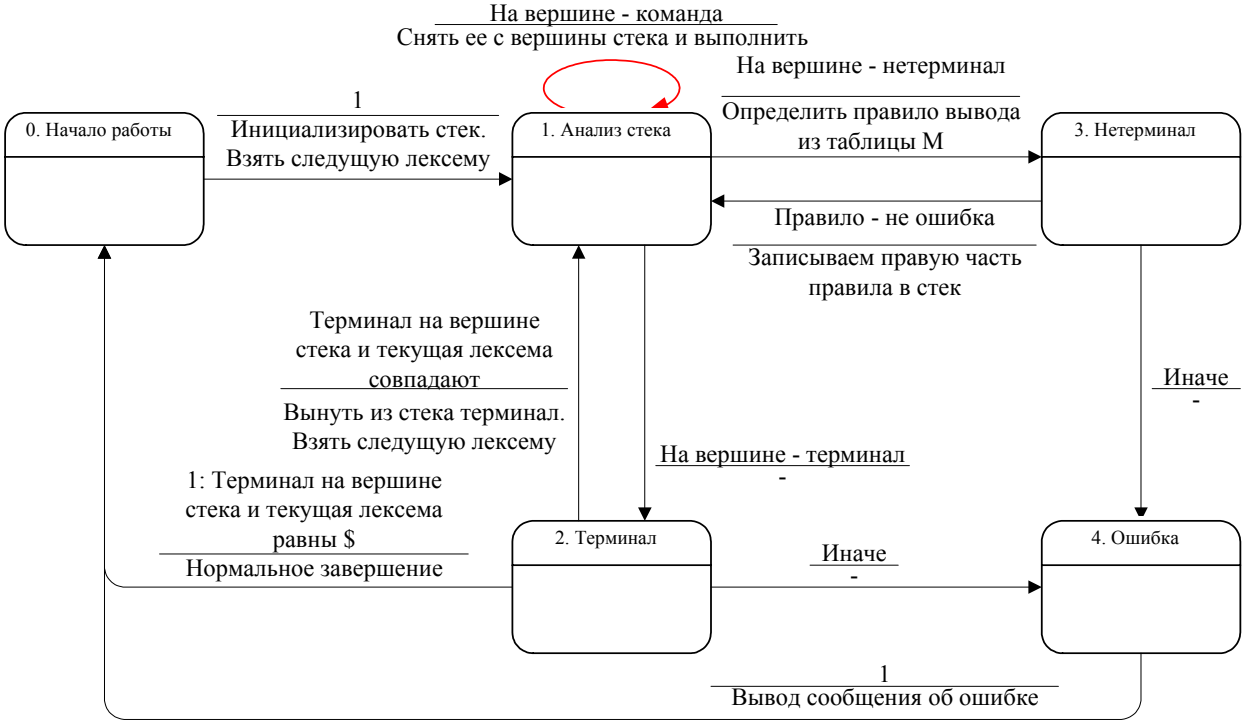

Рис. 10. Граф переходов преобразованного автомата синтаксического анализатора

Формализация графа переходов автомата синтаксического анализатора выполнена в разделе 4.

2.4. Стековая машина

Во многих компиляторах после синтаксического анализатора генерируется промежуточный код, который можно оптимизировать и превратить в машинный код. В качестве абстрактной машины для генерации промежуточного кода часто выбирается абстрактная стековая машина [2]. В данном проекте стековая машина выбрана в качестве «Эмулятора», так как для вычисления результата выражения достаточно преобразовать его в обратную польскую запись (чем занимается синтаксический анализатор) и подать его на вход стековой машины.

Эмулятор будет содержать стек чисел и выполнять с ним заданный набор операций:

- ‘+’ – взять два числа из стека, сложить их и положить результат обратно в стек;
- ‘-’, ‘*’, ‘/’ – выполняются аналогично сложению;
- ‘N’ – положить на вершину стека последнее обработанное число из синтаксического анализатора (атрибут).
- ‘P’ – взять из стека число и выдать его в выходной поток (на экран) в качестве результата вычисления очередного выражения.

На рис. 11 в качестве примера приведена схема работы команды «*» .

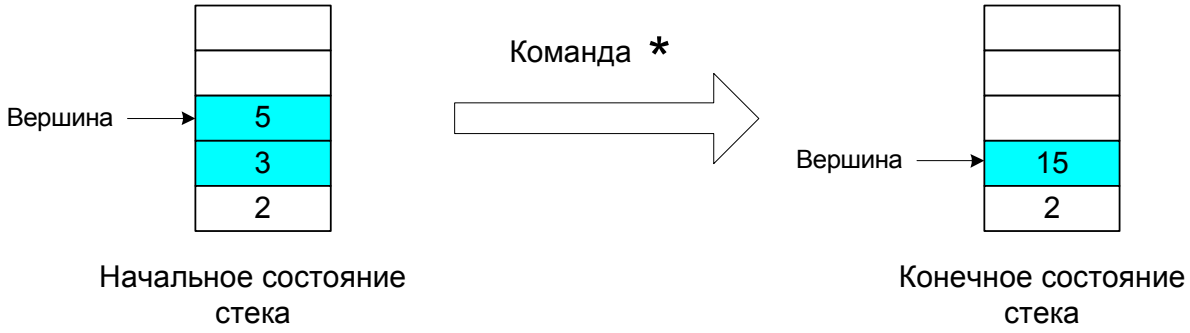

Рис. 11. Схема работы команды «*» в стековой машине

Укажем для выражения 2+3*5; последовательность команд, формируемую при использовании предлагаемого подхода:

N(2) N(3) N(5) * + P

Здесь запись N(2) обозначает передачу стековой машине команды N с атрибутом, равным двум. В смысле логики это самый «бедный» модуль нашей программы и он будет оформлен в виде одной процедуры, а не автомата.

2.5. Главный модуль и взаимодействие модулей программы

Теперь, когда вся основная логика модулей описана, необходимо собрать все модули в одну программу. Для этого определим интерфейсы взаимодействия всех модулей и порядок их запуска. В теории построения компиляторов [2, 5] есть несколько стандартных схем взаимодействия модулей, различающихся, например, в одно- и двухпроходных компиляторах. Одной из стандартных является и схема, реализуемая в данном проекте. В этой схеме в начале запускается синтаксический анализатор. Далее он по мере надобности запускает и лексический анализатор, и стековую машину. По данным синтаксический и лексический анализаторы взаимодействуют, используя текущую лексему (набор глобальных переменных), а синтаксический анализатор и стековая машина – используя текущую команду (также глобальная переменная).

Для удобства запуска программы и инициализации описанных модулей создан главный модуль программы. Он является самым маленьким по объему кода и запускает синтаксический анализатор.

В данном проекте модули достаточно независимы, и поэтому при необходимости их можно заменить более сложными или более подходящими под новые требования к программе. Например, если потребуется разработать калькулятор, вычисляющий более сложные выражения (не поддающиеся LL(1)-разбору), то можно заменить LL(1)-анализатор, например на LR-анализатор, практически без изменения остальных модулей.

3. Реализация

Для реализации выбран один из самых распространенных языков программирования – C++. Каждый модуль расположен в отдельном файле: *calc.cpp*, *lex.cpp*, *syn.cpp* и *stack.cpp*. При этом системозависимая и системонезависимая части каждого модуля не разделены на отдельные файлы, так как исходный текст программы достаточно мал. Заголовочный файл в данном проекте один – *calc.h*. В нем объявлены используемые библиотеки и интерфейсы модулей. Взаимодействие модулей осуществляется через глобальные переменные: *LexemeType* представляет тип лексемы; *LexemeNum* – значение лексемы (атрибут), если ее тип – число; *Command* – тип команды для стековой машины. В программе используется стандартная библиотека *STL* языка C++ для реализации стеков и ввода/вывода. Для отладки введен еще один модуль – *log.cpp* и *log.h*, который обеспечивает ведение протокола. Исходный текст данного модуля не приведен в работе, так как не имеет прямого отношения к задаче и без ущерба для понимания опущен.

4. Техническая документация

4.1. Описание входных переменных

x100 – Входной поток не инициализирован.
x101 – Текущий символ c – один из символов +, -, *, /, (,), ;.
x102 – Текущий символ c – пробел.
x103 – Текущий символ c – цифра.
x104 – Текущий символ c – точка.
x105 – Текущий символ c – $ (конец потока).

x201 – На вершине стека – команда.
x202 – На вершине стека – терминал.
x203 – На вершине стека – нетерминал.
x204 – Терминал на вершине стека совпадает с текущей лексемой.
x205 – Текущая лексема – $ и вершина стека - $.
x206 – Текущая лексема – ошибка.
x207 – Текущее правило вывода существует (не ошибка).

4.2. Описание выходных воздействий

z101 – Прочитать следующий символ.
z102 – Поместить текущий символ c в тип лексемы.
z103 – Установить тип лексемы – число и инициализировать значение лексемы.
z104 – Обновить значение атрибута лексемы.
z105 – Обновить дробную часть значения атрибута лексемы.
z106 – Установить тип лексемы – $ (конец потока).
z107 – Установить тип лексемы – ошибка.

z200 – Инициализировать стек (поместить в него $S).
z201 – Вынуть команду из стека для исполнения.
z202 – Вынуть терминал из стека.
z203 – Определить правило вывода.
z204 – Поместить в стек правило вывода.
z205 – Вывести сообщение об ошибке.

z301 – Исполнить команду стековой машиной.

4.3. Лексический анализатор

4.3.1. Словесное описание

Лексический анализатор выполняет разбиение входного потока на лексемы и вычисляет значение лексем, являющихся числами. Он вызывается из синтаксического анализатора и помещает результат – текущую лексему, а, возможно, и атрибут – в глобальные переменные.

4.3.2. Схема связей и граф переходов

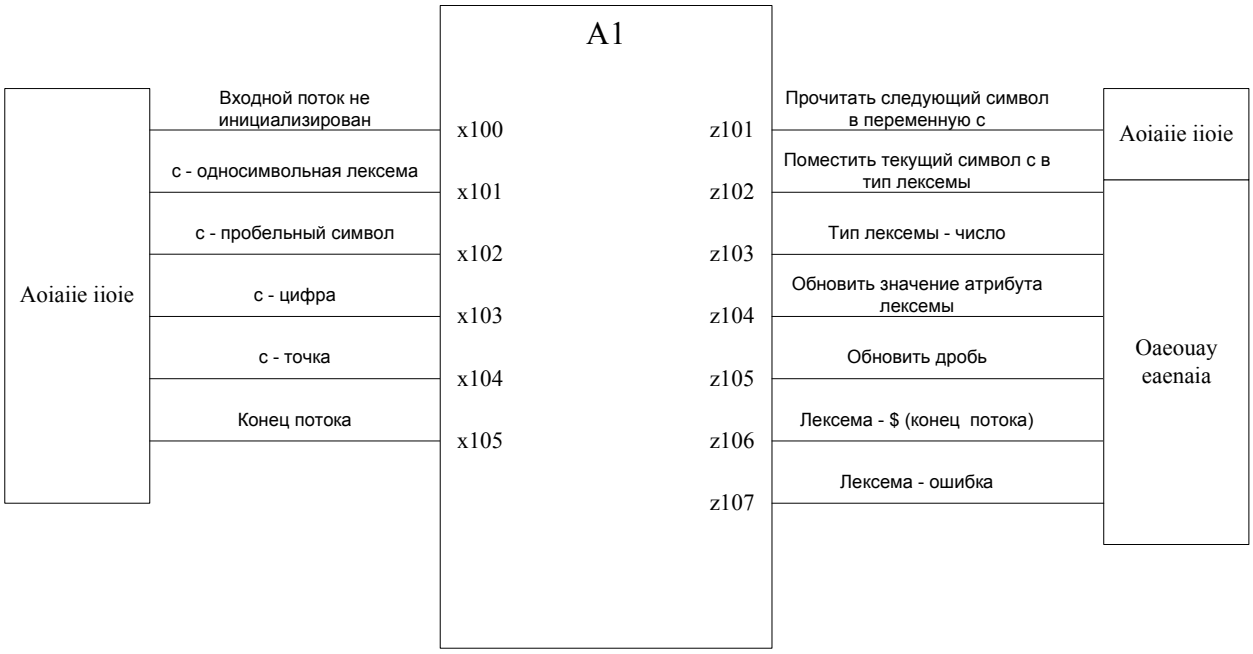

Рис. 12. Схема связей автомата лексического анализатора

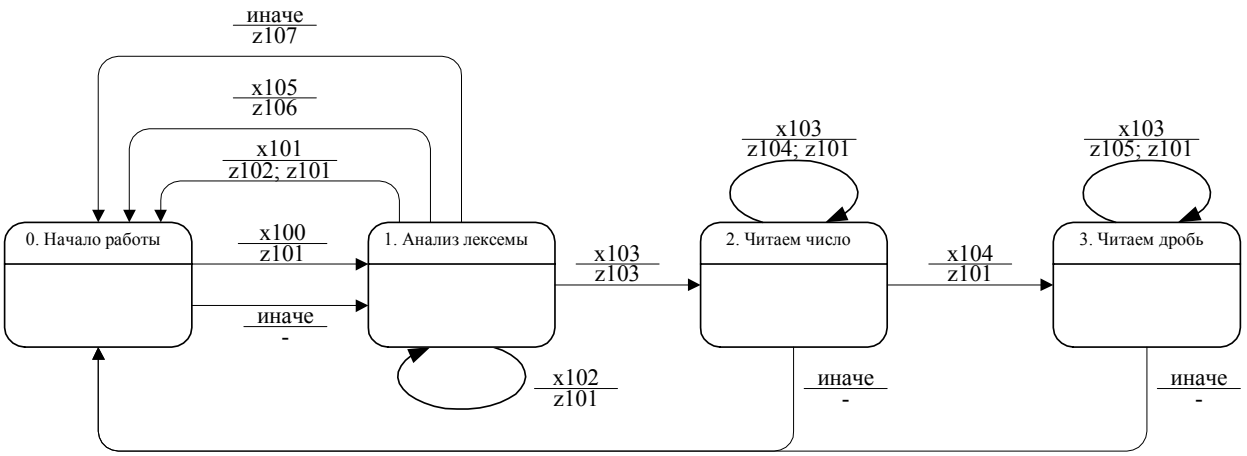

Рис. 13. Граф переходов автомата лексического анализатора

4.3.3. Исходный текст модуля *lex.cpp*

```
// Модуль "Лексический анализатор"

#include "calc.h"

// Переменные, доступные извне
char LexemeType;
double LexemeNum;

// Множества символов, определяемые анализатором

const char *symbols = "+-*/();";
const char *digits  = "0123456789";
const char *whites  = " \n\r\t";

// Внутренние переменные

char c = 0;
double t;

// Входные переменные

int x100()
{
    int res = (c == 0);
    log_input("x100 – Входной поток не инициализирован", res);
    return res;
}

int x101()
{
    int res = (strchr(symbols, c) != NULL);
    log_input("x101 - Текущий символ c - один из: +,-,*,/,(,),;.", res);
    return res;
}

int x102()
{
    int res = (strchr(whites, c) != NULL);
    log_input("x102 - Текущий символ - пробел", res);
    return res;
}

int x103()
{
    int res = (strchr(digits, c) != NULL);
    log_input("x103 - Текущий символ - цифра", res);
    return res;
}

int x104()
{
    int res = (c == '.');
    log_input("x104 - Текущий символ - точка", res);
    return res;
}

int x105()
{
    int res = cin.eof();
    log_input("x105 - Текущий символ - $ (конец потока)", res);
    return res;
}

// Выходные воздействия

void z101()
{
    log_output("z101 - Прочитать следующий символ");
    cin.get(c);

    if (cin.eof())
        c = '\xFF';
}

void z102()
{
    log_output("z102 - Поместить текущий символ c в тип лексемы");
    LexemeType = c;
```

```
}

void z103()
{
    log_output("z103 - Установить тип лексемы - число и инициализировать значение лексемы");
    LexemeType = 'N';
    LexemeNum = 0;
    t = 1;
}

void z104()
{
    log_output("z104 - Обновить значение атрибута лексемы");
    int digit = c - '0';
    LexemeNum *= 10;
    LexemeNum += digit;
}

void z105()
{
    log_output("z105 - Обновить дробную часть значения атрибута лексемы");
    int digit = c - '0';
    t /= 10;
    LexemeNum += digit*t;
}

void z106()
{
    log_output("z106 - Установить тип лексемы - $ (конец потока)");
    LexemeType = '$';
}

void z107()
{
    log_output("z107 - Установить тип лексемы - ошибка");
    LexemeType = 'E';
}

// Автомат лексического анализатора

void A1()
{
    int y1 = 0;
    log_automata_started("A1", y1);

    do {
        switch (y1)
        {
            case 0:
            {
                log_state_changed("A1", y1, "Начало работы");
                if (x100()) {   z101();                         y1 = 1; }
                else
                            {                                   y1 = 1; }
            }
            break;

            case 1:
            {
                log_state_changed("A1", y1, "Анализ лексемы");
                if (x101()) {   z102(); z101();                 y1 = 0; }
                else
                if (x102()) {   z101();                                 }
                else
                if (x103()) {   z103();                         y1 = 2; }
                else
                if (x105()) {   z106();                         y1 = 0; }
                else
                            {   z107();                         y1 = 0; }
            }
            break;
            case 2:
            {
                log_state_changed("A1", y1, "Читаем число");
                if (x103()) {   z104(); z101();                         }
                else
                if (x104()) {   z101();                         y1 = 3; }
                else
                            {                                   y1 = 0; }
            }
            break;
```

```
            case 3:
            {
                log_state_changed("A1", y1, "Читаем дробь");
                if (x103()) {   z105(); z101();                         }
                else
                            {                                   y1 = 0; }
            }
            break;
        }
    } while (y1);

    log_automata_ended("A1", y1);
}
```

4.4. Синтаксический анализатор

4.4.1. Словесное описание

Синтаксический анализатор разбирает LL(1)-грамматику G арифметических выражений и формирует поток команд для стековой машины. Запускается в начале работы программы. Вызывает лексический анализатор и стековую машину. Так как в языке C++ все массивы начинаются с индекса 0, то добавлено фиктивное правило вывода номер «0», которое не влияет на работу остальной программы. Также для удобства написания программы нетерминалы «E'» и «T'» заменены символами «e» и «t». Для обозначения ошибки в таблице разбора, приведенной в программе, используется число 0.

4.4.2. Схема связей и граф переходов

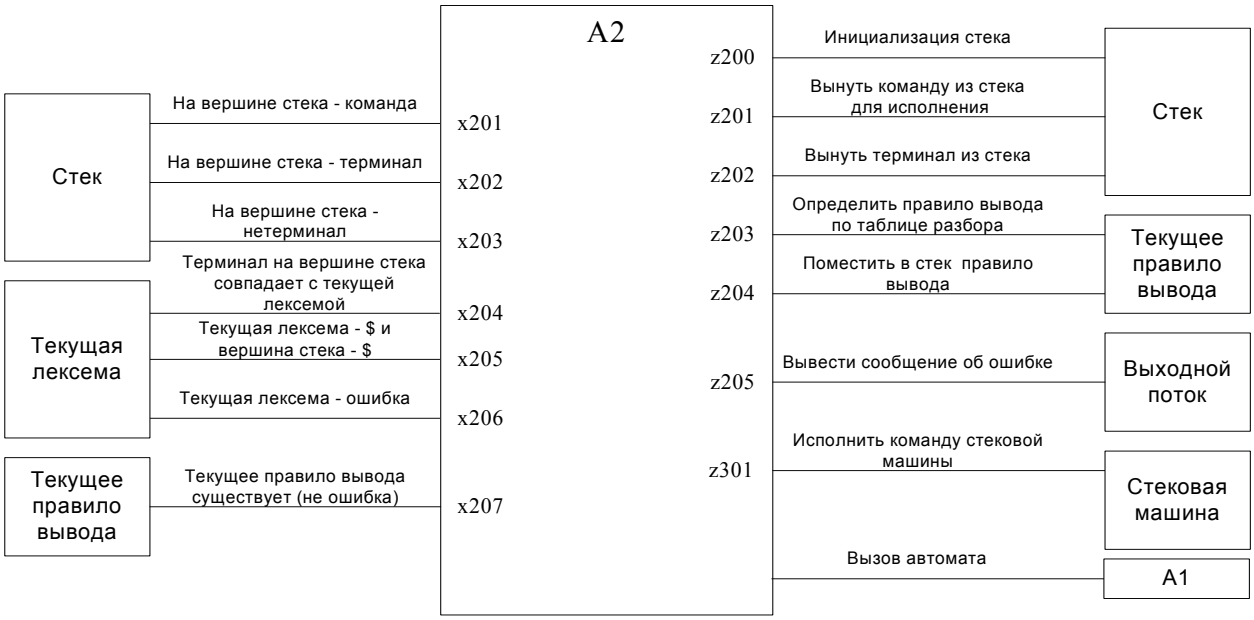

Рис. 10. Схема связей автомата синтаксического анализатора

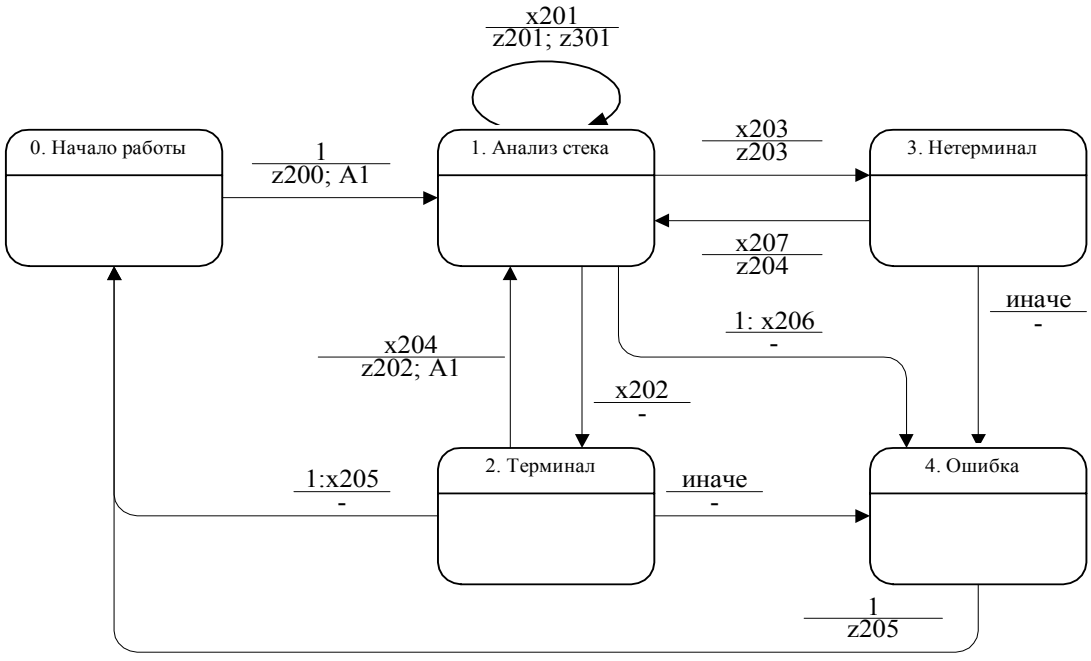

Рис. 11. Граф переходов автомата синтаксического анализатора

4.4.3. Исходный текст модуля *syn.cpp*

```
// Модуль "Синтаксический анализатор"

#include "calc.h"
#include <stack>

// Внутренние пременные - стек и текущее правило
stack<char> Stack;
const char *rule;


// Константы, задающие грамматику

// Терминалы и нетерминалы
const char terminals[]    = "N+-*/();$";
const char nonterminals[] = "SEeTtF";

// Правила вывода (с атрибутами)
const char *R[]      = {"",                  //0 - фиктивное правило
                        "S>E@P;S",           //1
                        "S>",                //2
                        "E>Te",              //3
                        "e>+T@+e",           //4
                        "e>-T@-e",           //5
                        "e>",                //6
                        "T>Ft",              //7
                        "t>*F@*t",           //8
                        "t>/F@/t",           //9
                        "t>",                //10
                        "F>N@N",             //11
                        "F>(E)"              //12
                        };


// Таблица разбора для LL(1) грамматики
const int M[6][9]  =
//         N  +  -  *  /  (  )  ;  $
/* S */ {{ 1, 0, 0, 0, 0, 1, 0, 0, 2},
/* E */  { 3, 0, 0, 0, 0, 3, 0, 0, 0},
/* e */  { 0, 4, 5, 0, 0, 0, 6, 6, 0},
/* T */  { 7, 0, 0, 0, 0, 7, 0, 0, 0},
/* t */  { 0,10,10, 8, 9, 0,10,10, 0},
/* F */  {11, 0, 0, 0, 0,12, 0, 0, 0}};

// Входные переменные

int x201()
{
    int res = (Stack.top() == '@');
    log_input("x201 - На вершине стека - команда", res);
    return res;
}

int x202()
{
    int res = (strchr(terminals, Stack.top()) != NULL);
    log_input("x202 - На вершине стека - терминал", res);
    return res;
}

int x203()
{
    int res = (strchr(nonterminals, Stack.top()) != NULL);
    log_input("x203 - На вершине стека - нетерминал", res);
    return res;
}

int x204()
{
    int res = (Stack.top() == LexemeType);
    log_input("x204 - Терминал на вершине стека совпадает с текущей лексемой", res);
    return res;
}

int x205()
{
    int res = (LexemeType == '$') && (Stack.top() == '$');
    log_input("x205 - Текущая лексема - $ и вершина стека - $", res);
    return res;
}
```

```
int x206()
{
    int res = (LexemeType == 'E');
    log_input("x206 - Текущая лексема - ошибка", res);
    return res;
}

int x207()
{
    int res = (strcmp(rule, R[0]) != 0);
    log_input("x207 - Текущее правило вывода существует (не ошибка)", res);
    return res;
}
```

// Выходные воздействия

```
void z200()
{
    log_output("z200 - Инициализировать стек (поместить в него $S)");
    Stack.push('$');
    Stack.push(nonterminals[0]);
}

void z201()
{
    log_output("z201 - Вынуть команду из стека для исполнения");
    Stack.pop();
    Command = Stack.top();
    Stack.pop();
}

void z202()
{
    log_output("z202 - Вынуть терминал из стека");
    Stack.pop();
}

void z203()
{
    log_output("z203 - Определить правило вывода");
    int  t = strchr(terminals, LexemeType) - terminals;
    int nt = strchr(nonterminals, Stack.top()) - nonterminals;

    int rulenum = M[nt][t];

    rule = R[rulenum];

    Stack.pop();
}

void z204()
{
    log_output("z204 - Поместить в стек правило вывода");
    for (const char *c = rule + strlen(rule) - 1; *c != '>'; c--)
        Stack.push(*c);
}

void z205()
{
    log_output("z205 - Вывести сообщение об ошибке");
    cout << "Ошибка в выражении." << endl;
}

void z301()
{
    log_output("z301 - Исполнить команду стековой машиной");
    StackMachine();
}
```

// Автомат синтаксического анализатора

```
void A2()
{
    int y2 = 0;

    log_automata_started("A2", y2);
    do {
        switch (y2)
        {
            case 0:
            {
                log_state_changed("A2", y2, "Начало работы");
                                z200(); A1();                   y2 = 1;
            }
            break;
            case 1:
            {
                log_state_changed("A2", y2, "Анализ стека");
                if (x206()) {                                   y2 = 4; }
                else
                if (x201()) {   z201(); z301();                         }
                else
                if (x202()) {                                   y2 = 2; }
                else
                if (x203()) {   z203();                         y2 = 3; }
            }
            break;
            case 2:
            {
                log_state_changed("A2", y2, "Терминал");
                if (x205()) {                                   y2 = 0; }
                else
                if (x204())
                            {   z202(); A1();                   y2 = 1; }
                else
                            {                                   y2 = 4; }
            }
            break;
            case 3:
            {
                log_state_changed("A2", y2, "Нетерминал");
                if (x207()) {   z204();                         y2 = 1; }
                else
                            {                                   y2 = 4; }
            }
            break;
            case 4:
            {
                log_state_changed("A2", y2, "Ошибка");
                                z205();                         y2 = 0;
            }
            break;
        }
    } while (y2);

    log_automata_ended("A2", y2);
}
```

4.5. Стековая машина

4.5.1. Словесное описание

Стековая машина имеет стек чисел и выполняет арифметические операции над его элементами. Вызывается из синтаксического анализатора. Выводит в выходной поток результаты вычислений выражений. Модуль выполнен в виде одной функции *StackMachine()*.

4.5.2. Исходный текст модуля *stack.cpp*

```
// Модуль "Стековая Машина"

#include "calc.h"
#include <stack>

// Переменная, доступная извне для передачи данных
char Command;

// Стек чисел. Внутренняя переменная
stack<double> Stack;


// Функция, реализующая стековую машину
void StackMachine()
{
    switch(Command)
    {
        case '+':                       // Выполнить команду "+"
        {
            double arg1 = Stack.top();
            Stack.pop();

            double arg2 = Stack.top();
            Stack.pop();

            double res = arg2 + arg1;

            Stack.push( res );
        }
        break;
        case '-':                       // Выполнить команду "-"
        {
            double arg1 = Stack.top();
            Stack.pop();

            double arg2 = Stack.top();
            Stack.pop();

            double res = arg2 - arg1;

            Stack.push( res );
        }
        break;
        case '*':                       // Выполнить команду "*"
        {
            double arg1 = Stack.top();
            Stack.pop();

            double arg2 = Stack.top();
            Stack.pop();

            double res = arg2 * arg1;

            Stack.push( res );
        }
        break;
        case '/':                       // Выполнить команду "/"
        {
            double arg1 = Stack.top();
            Stack.pop();

            double arg2 = Stack.top();
            Stack.pop();

            double res = arg2 / arg1;

            Stack.push( res );
```

```
        }
        break;
        case 'N':                   // Помещаем в стек значение
                                    // глобальной переменной LexemeNum
        {
            double res = LexemeNum;

            Stack.push( res );
        }
        break;
        case 'P':                   // Выбираем из стека результат
                                    // и выводим его в выходной поток
        {
            double arg = Stack.top();
            Stack.pop();

            cout << arg << endl;
        }
        break;
    }
}
```

4.6. Главный модуль и заголовочный файл

4.6.1. Исходный текст модуля *calc.cpp*

```
// Главный модуль программы

#include "calc.h"

// Начало
void main(void)
{
    log_begin(LOG_COMPLETE);

    // Запуск синтаксического анализатора
    A2();

    log_end();
}
```

4.6.2. Исходный текст заголовочного файла *calc.h*

```
// Заголовочный файл для всех модулей.

// Библиотека ввода/вывода
#include <iostream>
using namespace std;
#include "log.h"


// Модули, представленные в программе
void A1();
void A2();
void StackMachine();

// Данные, через которые идет взаимодействие модулей.
extern char   LexemeType;
extern double LexemeNum;

extern char   Command;
```

5. Протоколирование

Приведем пример выражения, которое можно вычислить с помощью разработанной программы и протокол ее работы.

Выражение: **2 + 2 ;**

Результат: **4**

Протокол:

```
! Начало работы программы.
{ A2 начал работу в состоянии 0
# A2 перешел в состояние 0 : Начало работы
* z200 - Инициализировать стек (поместить в него $S)
{ A1 начал работу в состоянии 0
# A1 перешел в состояние 0 : Начало работы
+ x100 – Входной поток не инициализирован. Значение - истина
* z101 - Прочитать следующий символ
# A1 перешел в состояние 1 : Анализ лексемы
+ x101 - Текущий символ c - один из: +,-,*,/,(,),;.. Значение - ложь
+ x102 - Текущий символ - пробел. Значение - ложь
+ x103 - Текущий символ - цифра. Значение - истина
* z103 - Установить тип лексемы - число и инициализировать значение лексемы
# A1 перешел в состояние 2 : Читаем число
+ x103 - Текущий символ - цифра. Значение - истина
* z104 - Обновить значение атрибута лексемы
* z101 - Прочитать следующий символ
# A1 перешел в состояние 2 : Читаем число
+ x103 - Текущий символ - цифра. Значение - ложь
+ x104 - Текущий символ - точка. Значение - ложь
} A1 закончил работу в состоянии 0
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - ложь
+ x202 - На вершине стека - терминал. Значение - ложь
+ x203 - На вершине стека - нетерминал. Значение - истина
* z203 - Определить правило вывода
# A2 перешел в состояние 3 : Нетерминал
+ x207 - Текущее правило вывода существует (не ошибка). Значение - истина
* z204 - Поместить в стек правило вывода
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - ложь
+ x202 - На вершине стека - терминал. Значение - ложь
+ x203 - На вершине стека - нетерминал. Значение - истина
* z203 - Определить правило вывода
# A2 перешел в состояние 3 : Нетерминал
+ x207 - Текущее правило вывода существует (не ошибка). Значение - истина
* z204 - Поместить в стек правило вывода
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - ложь
+ x202 - На вершине стека - терминал. Значение - ложь
+ x203 - На вершине стека - нетерминал. Значение - истина
* z203 - Определить правило вывода
# A2 перешел в состояние 3 : Нетерминал
+ x207 - Текущее правило вывода существует (не ошибка). Значение - истина
* z204 - Поместить в стек правило вывода
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - ложь
+ x202 - На вершине стека - терминал. Значение - ложь
+ x203 - На вершине стека - нетерминал. Значение - истина
* z203 - Определить правило вывода
# A2 перешел в состояние 3 : Нетерминал
+ x207 - Текущее правило вывода существует (не ошибка). Значение - истина
* z204 - Поместить в стек правило вывода
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - ложь
```

```
+ x202 - На вершине стека - терминал. Значение - истина
# A2 перешел в состояние 2 : Терминал
+ x205 - Текущая лексема - $ и вершина стека - $. Значение - ложь
+ x204 - Терминал на вершине стека совпадает с текущей лексемой. Значение - истина
* z202 - Вынуть терминал из стека
{ A1 начал работу в состоянии 0
# A1 перешел в состояние 0 : Начало работы
+ x100 - Входной поток не инициализирован. Значение - ложь
# A1 перешел в состояние 1 : Анализ лексемы
+ x101 - Текущий символ с - один из: +,-,*,/,(,),;.. Значение - истина
* z102 - Поместить текущий символ с в тип лексемы
* z101 - Прочитать следующий символ
} A1 закончил работу в состоянии 0
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - истина
* z201 - Вынуть команду из стека для исполнения
* z301 - Исполнить команду стековой машиной
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - ложь
+ x202 - На вершине стека - терминал. Значение - ложь
+ x203 - На вершине стека - нетерминал. Значение - истина
* z203 - Определить правило вывода
# A2 перешел в состояние 3 : Нетерминал
+ x207 - Текущее правило вывода существует (не ошибка). Значение - истина
* z204 - Поместить в стек правило вывода
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - ложь
+ x202 - На вершине стека - терминал. Значение - ложь
+ x203 - На вершине стека - нетерминал. Значение - истина
* z203 - Определить правило вывода
# A2 перешел в состояние 3 : Нетерминал
+ x207 - Текущее правило вывода существует (не ошибка). Значение - истина
* z204 - Поместить в стек правило вывода
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - ложь
+ x202 - На вершине стека - терминал. Значение - истина
# A2 перешел в состояние 2 : Терминал
+ x205 - Текущая лексема - $ и вершина стека - $. Значение - ложь
+ x204 - Терминал на вершине стека совпадает с текущей лексемой. Значение - истина
* z202 - Вынуть терминал из стека
{ A1 начал работу в состоянии 0
# A1 перешел в состояние 0 : Начало работы
+ x100 - Входной поток не инициализирован. Значение - ложь
# A1 перешел в состояние 1 : Анализ лексемы
+ x101 - Текущий символ с - один из: +,-,*,/,(,),;.. Значение - ложь
+ x102 - Текущий символ - пробел. Значение - ложь
+ x103 - Текущий символ - цифра. Значение - истина
* z103 - Установить тип лексемы - число и инициализировать значение лексемы
# A1 перешел в состояние 2 : Читаем число
+ x103 - Текущий символ - цифра. Значение - истина
* z104 - Обновить значение атрибута лексемы
* z101 - Прочитать следующий символ
# A1 перешел в состояние 2 : Читаем число
+ x103 - Текущий символ - цифра. Значение - ложь
+ x104 - Текущий символ - точка. Значение - ложь
} A1 закончил работу в состоянии 0
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - ложь
+ x202 - На вершине стека - терминал. Значение - ложь
+ x203 - На вершине стека - нетерминал. Значение - истина
* z203 - Определить правило вывода
# A2 перешел в состояние 3 : Нетерминал
+ x207 - Текущее правило вывода существует (не ошибка). Значение - истина
* z204 - Поместить в стек правило вывода
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - ложь
+ x202 - На вершине стека - терминал. Значение - ложь
```

```
+ x203 - На вершине стека - нетерминал. Значение - истина
* z203 - Определить правило вывода
# A2 перешел в состояние 3 : Нетерминал
+ x207 - Текущее правило вывода существует (не ошибка). Значение - истина
* z204 - Поместить в стек правило вывода
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - ложь
+ x202 - На вершине стека - терминал. Значение - истина
# A2 перешел в состояние 2 : Терминал
+ x205 - Текущая лексема - $ и вершина стека - $. Значение - ложь
+ x204 - Терминал на вершине стека совпадает с текущей лексемой. Значение - истина
* z202 - Вынуть терминал из стека
{ A1 начал работу в состоянии 0
# A1 перешел в состояние 0 : Начало работы
+ x100 – Входной поток не инициализирован. Значение - ложь
# A1 перешел в состояние 1 : Анализ лексемы
+ x101 - Текущий символ с - один из: +,-,*,/,(,),;.. Значение - истина
* z102 - Поместить текущий символ с в тип лексемы
* z101 - Прочитать следующий символ
} A1 закончил работу в состоянии 0
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - истина
* z201 - Вынуть команду из стека для исполнения
* z301 - Исполнить команду стековой машиной
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - ложь
+ x202 - На вершине стека - терминал. Значение - ложь
+ x203 - На вершине стека - нетерминал. Значение - истина
* z203 - Определить правило вывода
# A2 перешел в состояние 3 : Нетерминал
+ x207 - Текущее правило вывода существует (не ошибка). Значение - истина
* z204 - Поместить в стек правило вывода
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - истина
* z201 - Вынуть команду из стека для исполнения
* z301 - Исполнить команду стековой машиной
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - ложь
+ x202 - На вершине стека - терминал. Значение - ложь
+ x203 - На вершине стека - нетерминал. Значение - истина
* z203 - Определить правило вывода
# A2 перешел в состояние 3 : Нетерминал
+ x207 - Текущее правило вывода существует (не ошибка). Значение - истина
* z204 - Поместить в стек правило вывода
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - истина
* z201 - Вынуть команду из стека для исполнения
* z301 - Исполнить команду стековой машиной
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - ложь
+ x202 - На вершине стека - терминал. Значение - истина
# A2 перешел в состояние 2 : Терминал
+ x205 - Текущая лексема - $ и вершина стека - $. Значение - ложь
+ x204 - Терминал на вершине стека совпадает с текущей лексемой. Значение - истина
* z202 - Вынуть терминал из стека
{ A1 начал работу в состоянии 0
# A1 перешел в состояние 0 : Начало работы
+ x100 – Входной поток не инициализирован. Значение - ложь
# A1 перешел в состояние 1 : Анализ лексемы
+ x101 - Текущий символ с - один из: +,-,*,/,(,),;.. Значение - ложь
+ x102 - Текущий символ - пробел. Значение - ложь
+ x103 - Текущий символ - цифра. Значение - ложь
+ x105 - Текущий символ - $ (конец потока). Значение - истина
* z106 - Установить тип лексемы - $ (конец потока)
} A1 закончил работу в состоянии 0
# A2 перешел в состояние 1 : Анализ стека
```

```
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - ложь
+ x202 - На вершине стека - терминал. Значение - ложь
+ x203 - На вершине стека - нетерминал. Значение - истина
* z203 - Определить правило вывода
# A2 перешел в состояние 3 : Нетерминал
+ x207 - Текущее правило вывода существует (не ошибка). Значение - истина
* z204 - Поместить в стек правило вывода
# A2 перешел в состояние 1 : Анализ стека
+ x206 - Текущая лексема - ошибка. Значение - ложь
+ x201 - На вершине стека - команда. Значение - ложь
+ x202 - На вершине стека - терминал. Значение - истина
# A2 перешел в состояние 2 : Терминал
+ x205 - Текущая лексема - $ и вершина стека - $. Значение - истина
} A2 закончил работу в состоянии 0
! Конец работы программы.
```

6. Заключение

На основе выполненного проекта можно сделать следующие выводы:

- совместное применение теории построения компиляторов и SWITCH-технологии позволяет объединить математическую строгость проектирования этого класса программ с формальностью их реализации;
- на всех этапах создания компилятора используются конечные автоматы в форме графов переходов;
- разработка калькулятора полностью документирована (кроме формальных преобразований из теории компиляторов);
- получаемый исходный код весьма прост и понятен, так как структура каждого из его основных модулей изоморфна структуре схемы связей автомата и его графа переходов;
- если автоматическое построение лексических и синтаксических анализаторов целесообразно выполнять с помощью генераторов Lex и Yacc соответственно (или их аналогов) [2], то ручное проектирование – на основе предлагаемого подхода.

Литература

1. *Страуструп Б.* Язык программирования C++. М.: Бином, СПб.: Невский диалект, 2001.
2. *Ахо А., Сети Р., Ульман Д.* Компиляторы. Принципы, технологии, инструменты. М.: Вильямс, 2001.
3. *Шалыто А.А.* SWITCH-технология. Алгоритмизация и программирование задач логического управления. СПб.: Наука, 1998.
4. *Шалыто А.А., Туккель Н.И.* От тьюрингова программирования к автоматному. // Мир ПК, 2002, №2.
5. *Легалов А.И.* Трансляторы. Методы разработки. http://www.softcraft.ru/translat.shtml
6. *Хопкрофт Д., Мотвани Р., Ульман Д.* Введение в теорию автоматов, языков и вычислений. СПб.: Вильямс, 2002.