Санкт-Петербургский государственный университет информационных технологий, механики и оптики

Кафедра «Компьютерных технологий»

Е.А. Цымбалюк, Р.Г. Зубаиров, А.А. Шалыто

Совместное использование теории компиляторов и Switch-технологии

(на примере разработки интерпретатора сконструированного авторами языка программирования)

Проектная документация

Санкт-Петербург
2006

Оглавление

Введение

В компьютерных технологиях все чаще используются интерпретируемые языки программирования (скриптовые языки). Как правило, они разрабатываются для решения конкретных классов задач. Так, например, язык *JavaScript* [1] – удобен и прост в обращении для написания небольших скриптов для *Web*-страниц, а язык *Lua* [2] используется для написания искусственного интеллекта множества игр [3]. Скриптовые языки удобны для интеграции в приложение и запуска в виде внешних скриптов (*data-driven* программ[1]).

Основное отличие скриптовых языков программирования от языков общего назначения, как *C++* и *Java*, состоит в том, что программа на скриптовом языке не компилируется в исполняемый код (*exe*-файл для *Windows* или *class*-файл для *JVM* (*Java Virtual Machine*), а выполняется сразу после компиляции без создания промежуточного файла.

Практика показывает, что использование скриптовых языков ускоряет и упрощает решение прикладных задач (например, разработка искусственного интеллекта в играх и сценарии в Web-дизайне). Помимо скорости разработки скриптовые языки хороши своей простотой. Человеку не требуется быть профессиональным программистом, чтобы освоить тот или иной скриптовый язык. (Достаточно просто, например, написать скрипт обработки, например, нажатия кнопки *Web*-страницы).

Цель настоящей работы – показать на примере простого языка процесс разработки интерпретатора и подробно изложить все этапы проектирования программ подобного рода.

[1] *Data-driven* – «управляемый данными». Этот термин приобретает все большое распространение в играх. Это концепция выноса различных настроек во внешние файлы, что повышает гибкость программы за счет возможности быстрого изменения данных.

1. Проектирование

1.1. Постановка задачи

Рассмотрим процесс разработки интерпретатора на примере простого языка, чтобы проследить все этапы проектирования, не усложняя в то же время грамматику языка. В основе построения данного интерпретатора положены теории компиляторов и *Switch*-технологии.

Не будем загромождать язык условными переходами и циклами, а также возможностью создания пользовательских процедур. Ограничимся следующими элементами языка:

- переменные для хранения вещественнозначных чисел. Значения переменных интерпретатора хранятся в виде ассоциативного массива, ключом для которого является имя переменной, а значением – значение самой переменной. Если переменная еще не была использована, то для нее создается новая ячейка;
- операции сложения, вычитания, умножения, деления, возведения в степень и скобки. Данные операторы имеют разный приоритет выполнения, что усложняет грамматику, однако в реальных языках программирования уровней приоритетов еще больше;
- вызовы функций. Язык не позволяет пользователю описывать собственные функции, однако предоставляет ряд встроенных: `sqrt` (квадратный корень), `sin` (синус угла), `cos` (косинус угла), `abs` (абсолютное значение числа). На примере функций будут показаны общие принципы вызовов процедур.

На рис. 1 приведено окно программы интерпретатора, созданного в результате выполнения настоящей работы.

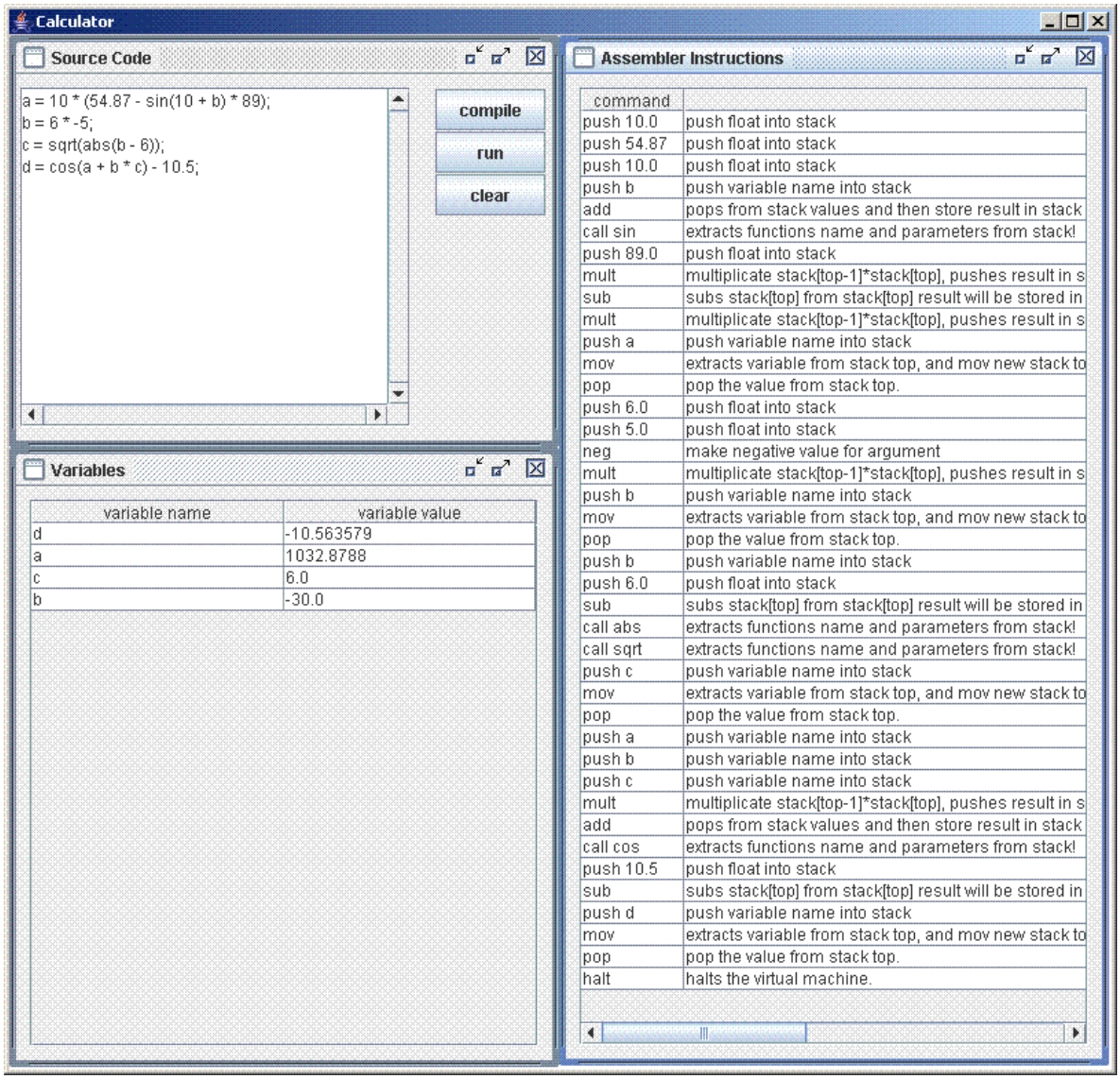

Рис. 1. Окно программы интерпретатора

1.2. Разбиение задачи

Основная идея интерпретатора заключается в генерации байт-кода – набора микрокоманд интерпретатора с последующим их выполнением. Для построения байт-кода используется лексический и синтаксический анализаторы. С помощью лексического анализатора исходный текст программы преобразуется в конечное множество символов, из которых состоит грамматическое описание языка [4]. Далее полученный набор символов обрабатывается синтаксическим анализатором. При этом происходит проверка на соответствие грамматике и генерация инструкций интерпретатора (рис. 2).

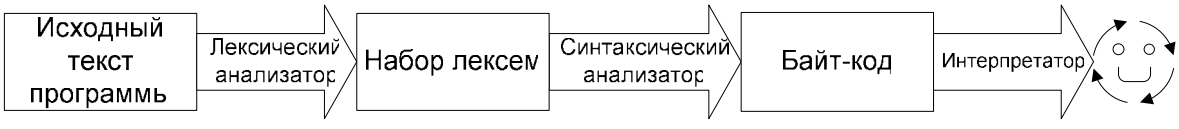

Рис. 2. Процесс работы интерпретатора

Программа проходит все этапы компиляции, как и обычная программа, написанная, например, на языке *C++*. Однако есть важные отличия:

- программа хранится в исходных кодах и преобразуется в байт-код непосредственно перед использованием;[2]
- процесс интеграции с приложением (внешней системой, реализующей низкоуровневый интерфейс скрипта) происходит при непосредственной загрузке скрипта. Таким образом, есть возможность использования ряда объектов и функций, ничего не зная об их реализации и даже не имея представления, будут ли они вообще реализованы в программе;[3]
- роль байт-кода в обычных языках программирования выполняют машинные команды. Однако байт-код скрипта может быть намного шире и абстрактней, чем язык ассемблера компьютера. В частности, на уровне примитивных команд могут поддерживаться объекты и встроенное управление памятью (сборщик мусора), что невозможно на уровне машинных кодов.[4]

1.3. Лексический анализатор

Лексический анализатор разбивает исходный код программы на поток лексем – последовательности символов, воспринимаемые программой, как единое целое.

Для разрабатываемого лексического анализатора важны два класса лексем – символы и слова:

- символы – различные операторы, скобки и запятые: `+ - * / ^ , ; ( )`. Обозначим это множество символом `Y`;
- слова – некоторая последовательность букв (обозначим это множество символом `L`). В данном случае буквой называется любой символ, не являющийся пробельным и символом из множества `Y`. Слово можно описать регулярным выражением «`L+`». Поскольку в выбранном языке допускаются числа с плавающей точкой и символьные переменные, то можно вынести определение типа переменной с синтаксического уровня на семантический. При генерации байт-кода будет происходить проверка выражения – является ли оно числом или словом.

---

[2] На самом деле, например, в играх при выпуске конечного продукта имеет смысл скрипты хранить в виде байт-кода, так как свойство скриптового языка в простой модификации уже не требуется, но необходима высокая скорость загрузки.

[3] На самом деле что-то подобное возможно и в других системах. Например, в *MS Windows* такую функцию выполняют динамические библиотеки. В скриптовых языках подобные связи реализуются более простым и естественным способом.

[4] При использовании современных языков, таких как *Java* и *C#*, также компилируют программу в байт-код, позволяющий программировать на уровне объектов. При запуске таких программ происходит либо преобразование кода в машинные команды, либо интерпретация байт-кода виртуальной машиной.

Итак, регулярные выражения для символов – «`Y`», а для слов – «`L+`». Ниже по этим выражениям строятся детерминированные автоматы [4]. При построении лексического анализатора данные автоматы объединяются в один, и при необходимости устраняется недетерминированность.

На рис. 3 изображен лексический анализатор, способный принимать символы и слова и пропускать пробельные символы.[5]

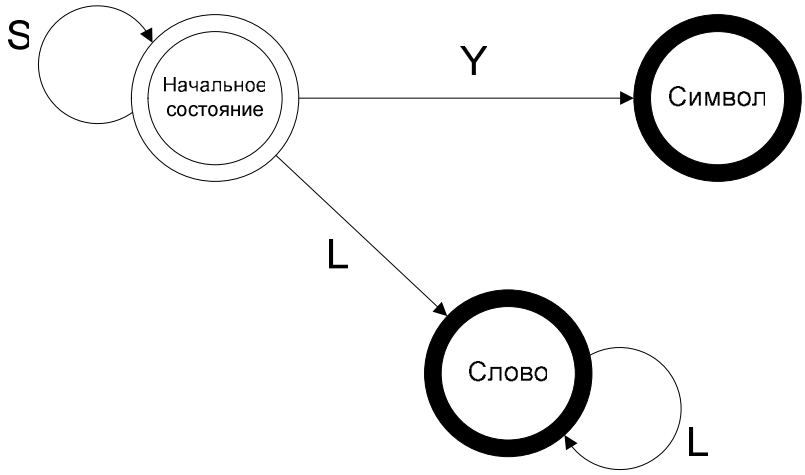

Рис. 3. Построение лексического анализатора. Первый шаг

Обычно в языках программирования есть комментарии. Можно ввести их и в разрабатываемый язык. При этом обработка комментариев (их пропуск) будет выполняться на уровне лексического анализатора. Реализуем два типа комментариев:

- строчный комментарий (комментарий «//» в языке *C++*). Регулярное выражение для него будет выглядеть следующим образом: «`/ / [^\n]* \n`» – последовательность из двух слешей, затем набор символов отличных от перевода строки и перевод строки – окончание комментария;
- блочный комментарий (комментарий «/* */» в языке *C++*). Регулярное выражения для него: «`/ '*' ([^'*']* ('*' [^/])*)* '*' /`». Выражение получилось достаточно громоздким, однако если использовать тот факт, что между началом «/*» и концом «*/» комментария находится набор цепочек, не содержащих последовательность «*/», то можно построить достаточно простой автомат (рис.4).

[5] При построении конечного варианта автомата будет добавлен подсчет строк обработанного исходного кода. Это позволяет выдавать сообщение об ошибках с указанием их места в тексте с точностью до номеров строк.

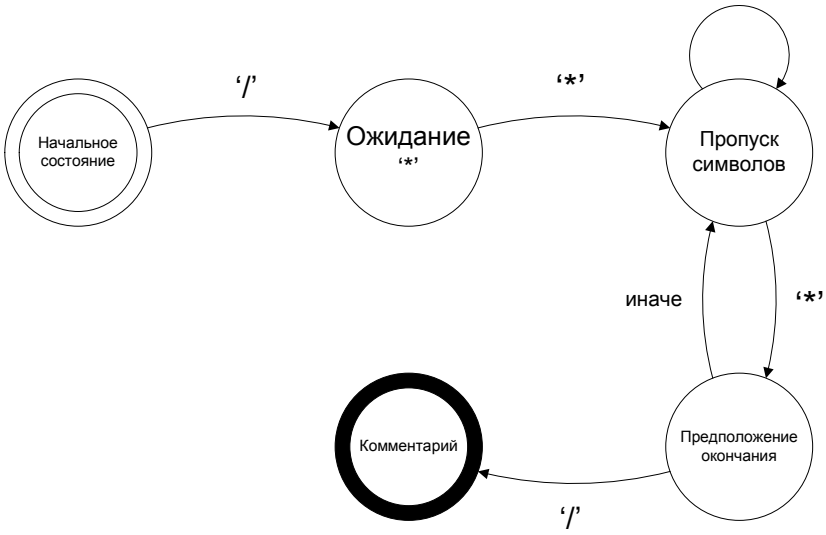

Рис. 4. Построение лексического анализатора. Второй шаг

Объединяя автоматы на рис. 3, 4, построим детерминированный автомат (рис. 5), реализующий лексический анализатор в целом.

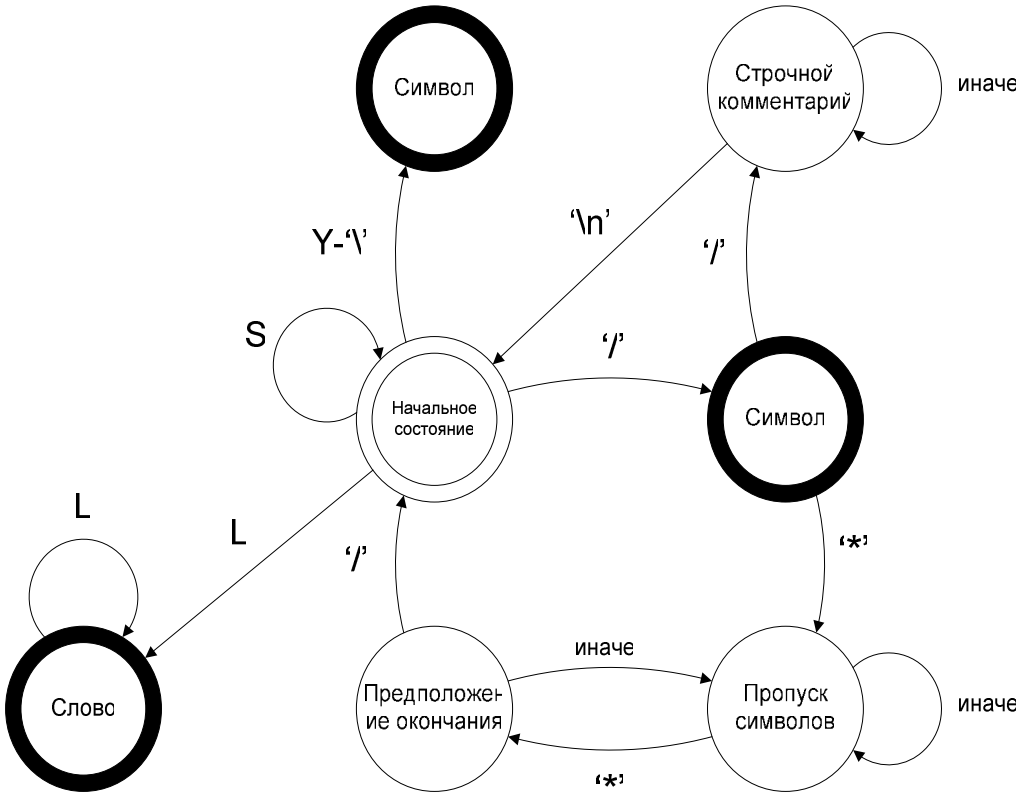

Рис. 5. Граф переходов лексического анализатора

Полученный автомат пропускает пробельные символы и комментарии (строчные и блочные), но допускает слова и символы.

1.4. Синтаксический анализатор

После преобразования текста программы в поток лексем необходимо произвести синтаксический анализ и создать байт-код, соответствующий тексту программы, для дальнейшей интерпретации. Построенный язык может быть описан различными грамматиками. Выбранная грамматика, заданная набором продукций, приведена ниже:

```
Program          →   Program Statement ';'
Program          →   ε
Statement        →   'word' '=' Expression
Statement        →   Expression
Expression       →   Expression '+' Term
Expression       →   Expression '-' Term
Expression       →   Term
Term             →   Term '*' Frac
Term             →   Term '/' Frac
Term             →   Frac
Frac             →   Frac '^' Frac2
Frac             →   Frac2
Frac2            →   '-' Frac2
Frac2            →   'word
Frac2            →   'word' '(' ArgList ')'
Frac2            →   '(' Expression ')'
ArgList          →   ε
ArgList          →   ArgList2
ArgList2         →   ArgList2 ',' Expression
ArgList2         →   Expression
```

В кавычках обозначены терминалы (лексемы, в случае компиляторов это одно и тоже), среди которых использованы все символы разрабатываемого языка. Элементы без кавычек – нетерминалы.

Для разбора этой грамматики построим стандартный *LALR*-анализатор [4]. При разборе выражений входного потока анализатор использует стек и таблицу разбора `T`. Входной поток состоит из последовательности терминалов. Также, необходимо ввести еще од-

ну лексему – конец ввода и дополнительную «нулевую» продукцию, что является требованием построения таблицы разбора:

Program' → *Program*

В начале работы на вершине стека содержится начальное состояние синтаксического анализатора, а затем в зависимости от содержимого таблицы разбор синтаксический анализатор производит следующие действия:

- перенос – в стек заносится очередная лексема (терминал);
- свертка – из стека удаляются символы, количество которых равно количеству элементов (терминалов и нетерминалов) в правой части продукции;[6]
- допуск – состояние окончания ввода, которое достигается, когда программа обработала все лексемы из потока и дошла до конца ввода;
- ошибка – неверный ввод.

Таблица разбора `T` является двумерной – `T[S,I]`, где `S` – текущее состояние, хранящееся на вершине стека, а `I` – терминал или нетерминал. В клетках таблицы `T[S,I]` указаны действия автомата – перенос, свертка, переход в состояние ошибки или допуска.

Таблица приведена в листинге программы. Она не приводится в документации из-за большего размера. Достаточно сказать, что в результате работы алгоритма, изложенного в [5], получено `33` состояния автомата. При девяти нетерминалах и `12` терминалах таблица состоит из `33 * 21 = 693` клеток.

Опишем подробнее работу синтаксического анализатора.

Автомат извлекает из стека текущее состояние разбора, выбирает очередной терминал и проверяет значение в таблице:

- если необходим перенос, в стеке сохраняется новое состояние и осуществляется переход к следующему терминалу;
- если необходимо выполнить свертку (см. [4]), из стека удаляется столько состояний, сколько находится терминалов и нетерминалов в правой части продукции, по которой выполняется свертка. Затем на основе нетерминала, в который произошла свертка, и состояния на вершине стека определяется новое состояние по таблице разбора. Оно сохраняется его в стеке;
- если в соответствующей ячейке таблицы нет записи, произошла ошибка – пользователю выдается сообщение о неверном вводе;
- если значение в соответствующей ячейке – допускающее состояние, то синтаксический анализатор заканчивает работу и за дело берется интерпретатор.

Скелет диаграммы переходов синтаксического анализатора изображен на рис. 6.

[6] Заметим, что в отличие от *LL(1)*-грамматик, значения нетерминалов и терминалов правой части продукции не играют роли – важно лишь их количество.

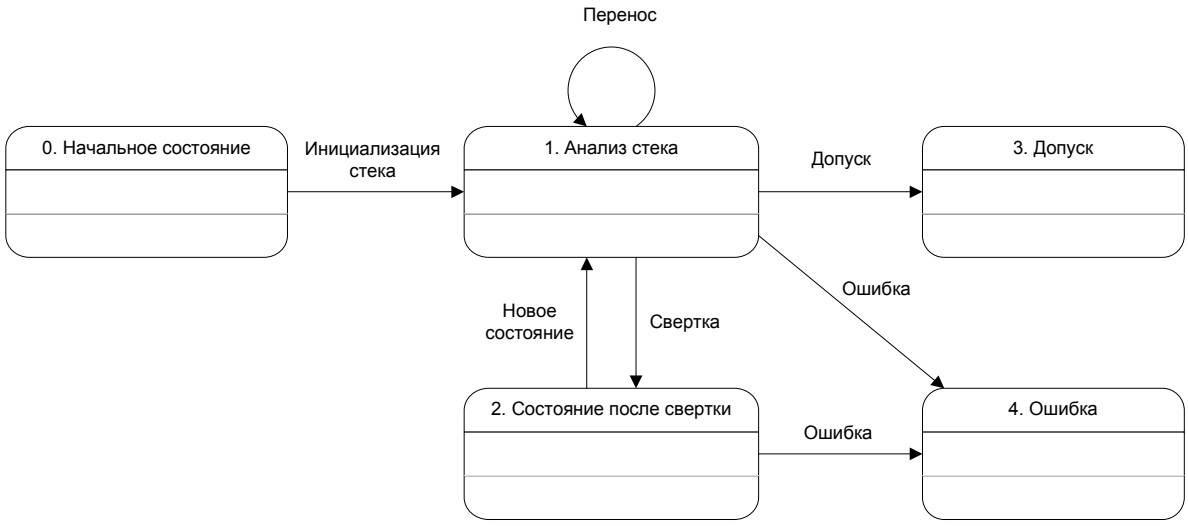

Рис. 6. Граф переходов синтаксического анализатора

Построенный скелет диаграммы переходов синтаксического анализатора не делает главного – не осуществляет генерацию байт-кода. Прелесть *LALR*-компиляторов в том, что процедура, генерирующая код для продукции вызывается при свертке. Поэтому при добавлении генерации кода не придется изменять скелет диаграммы переходов.

Для каждой продукции существует функция, генерирующая соответствующий ей код. Рассмотрим пример генерации кода в момент свертки продукции:

Expression → *Expression* '+' *Term*

Известно, что к моменту свертки данной продукции, уже произошли свертки в нетерминалы `Expression` и `Term`. Поэтому в стеке интерпретатора верхние два числа будут содержать значения, соответствующие этим нетерминалам. Функция данной продукции генерирует команду `ADD`. Таким образом, осуществляется свертка в нетерминал `Expression`.

Диаграмма переходов автомата синтаксического анализатора приведена в разделе 5.

1.5. Интерпретатор байт-кода

Определим язык интерпретатора. Это должен быть набор команд, позволяющий выполнить все необходимые операции построенного языка:

- `MOV A` - запись числа, находящегося на вершине стека в переменную с именем `A`. Имя `A` не может начинаться с цифры, так как иначе оно будет интерпретировано, как число;
- `PUSH A` – извлечение числа из ячейки памяти с именем `A` и размещение его в стеке;
- `PUSHN N` – размещение в стек соответствующего числа `N`;
- `ADD` – сложение двух чисел на вершине стека и размещение результата в стеке;

- `SUB` – вычитание чисел. Число на вершине стека – вычитаемое, а число под ним – уменьшаемое;
- `NEG` – унарный минус. Извлекает число из стека и возвращает обратно уже умноженное на минус единицу;
- `MUL` – умножение чисел. Аналогично сложению;
- `DIV` – деление чисел. Аналогично вычитанию;
- `POW` – возведение числа в степень. Аналогично делению;
- `CALL A N` – вызов встроенного метода `A` с числом параметров, равным `N`. Если подобной функции не существует, либо задано неверное число аргументов – происходит ошибка. Такие ошибки, однако, предупреждаются на уровне синтаксического анализа.

Построим для выражения байт-код:

```
x = -1 * sin(2^t) + min(a,b)
```

```
PUSHN 1
NEG
PUSHN 2
PUSH t
POW
CALL SIN 1
MUL
PUSH a
PUSH b
CALL min 2
ADD
MOV x
```

Автомат интерпретатора байт-кода приведен в разд. 4.

1.6. Взаимодействие модулей программы

Теперь рассмотрим на примере рассмотренной схемы (рис. 2), как модули программы будут взаимодействовать между собой: первоначально запускается модуль синтаксического анализатора, а затем по мере необходимости вызывается лексический анализатор. Результатом работы синтаксического анализатора является байт-код. Этот байт-код обрабатывает интерпретатор.[7] Таким образом, лексический анализатор производит разбиение

[7] В данном случае интерпретацию можно осуществить в процессе синтаксического анализа, так как в построенном языке не присутствуют условные операторы и циклы. Однако такое возможно в очень простых языках. Цель данной работы – показать процесс построения полноценного интерпретатора так, чтобы читатель сам мог расширить грамматику и ассемблер интерпретатора до необходимых ему возможностей.

входного потока на терминалы, которые в свою очередь обрабатываются синтаксическим анализатором (рис. 2).

2. Лексический анализатор

2.1. Описание входных переменных

x1 – Текущий символ C ∈ S (пробельный символ);

x2 – Текущий символ C ∈ Y (один из +, -, *, /, ^, (,), ',', ;);

x3 – Текущий символ C = '/' – слеш;

x4 – Текущий символ C = '*' – звездочка;

x5 – Текущий символ C = '\n' – перенос строки;

x6 – Конец ввода (C = EOF).

2.2. Выходные воздействия

z1 – Чтение очередного символа из потока (C = read);

z2 – Добавление текущего символа в конец результирующего слова (RES += C);

z3 – Очищение результирующей строки (RES = "");

z4 – Увеличение счетчика строк на единицу (++LINE);

z5 – Текущий терминал – слово (TOK = WORD);

z6 – Текущий терминал – символ (TOK = SYMBOL);

z7 – Текущий терминал – конец ввода (TOK = EOF).

2.3. Словесное описание

Лексический анализатор выполняет разбиение входного потока на лексемы. Он записывает тип лексемы и ее значение в переменные своего класса, посредством которых синтаксический анализатор получает информацию о следующей лексеме.

2.4. Схема связей и граф переходов лексического анализатора

Схема связей изображена на рис. 7.

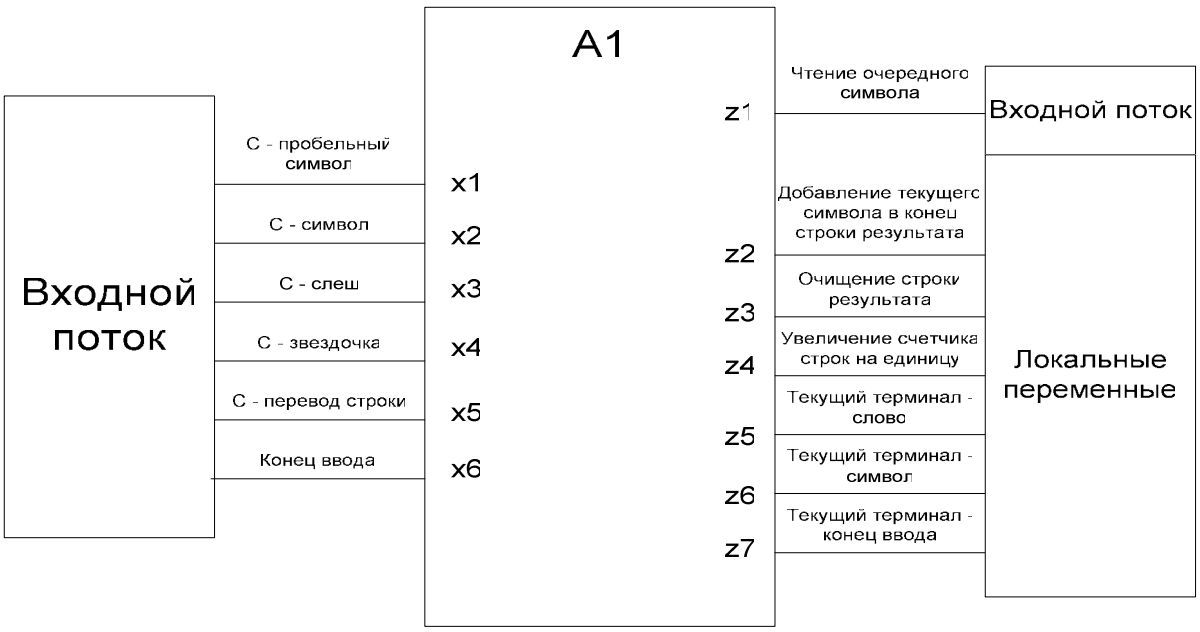

Рис. 7. Схема связей автомата лексического анализатора

Граф переходов изображен на рис. 8.

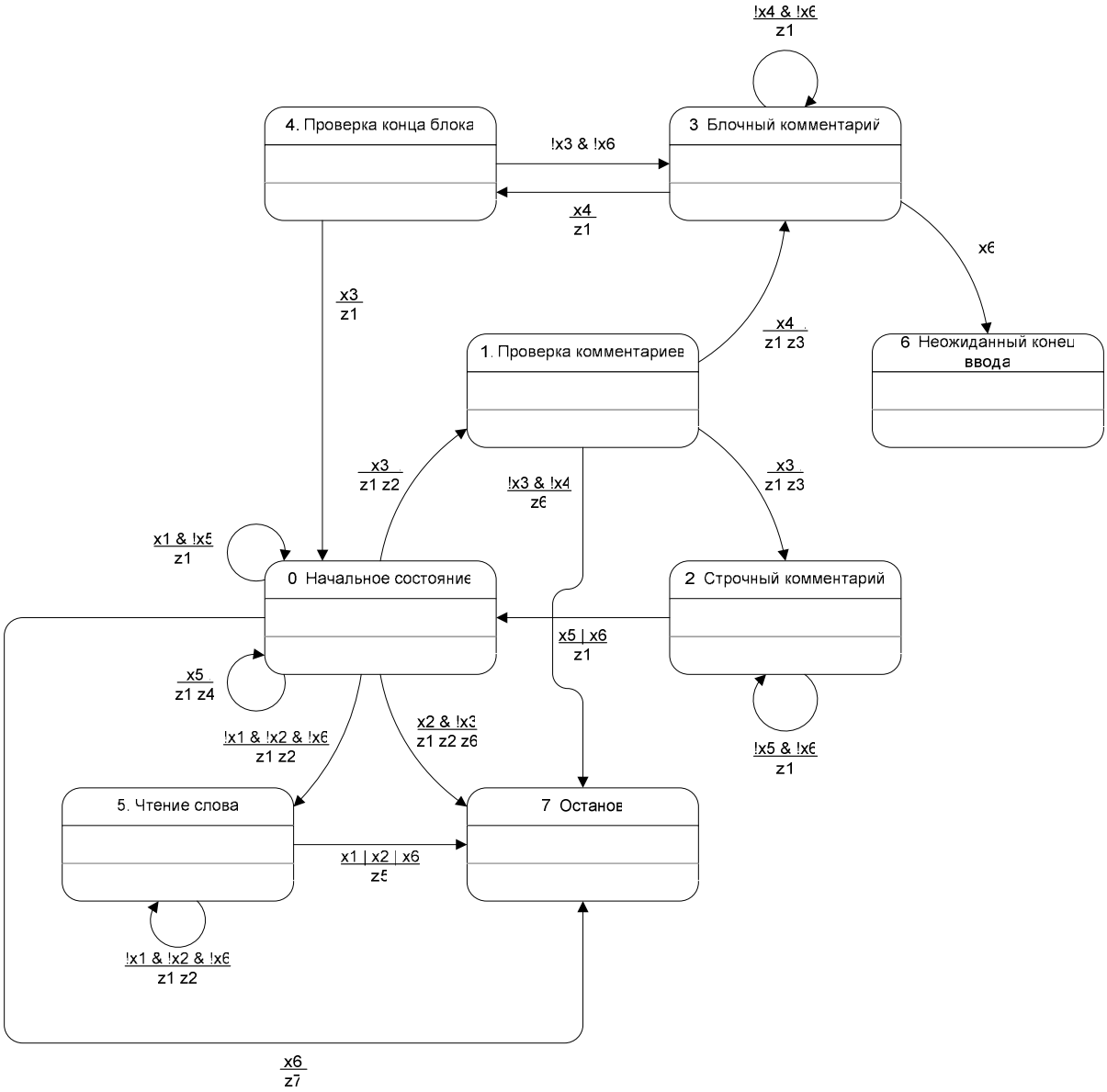

Рис. 8. Граф переходов автомата лексического анализатора

3. Синтаксический анализатор

3.1. Описание входных переменных

`x1` – действие, необходимое для текущего состояния и лексемы – перенос (значение `T[s,t]`, где `s` – текущее состояние, `t` – очередной терминал, а `T` – таблица переходов для данной грамматики, построенная методом, изложенным в [5];

`x2` – действие, необходимое для текущего состояния и лексемы – свертка;

`x3` – действие, необходимое для текущего состояния и лексемы – ошибка (неожиданная лексема);

`x4` – действие, необходимое для текущего состояния и лексемы – допуск (достигли конца ввода и программа синтаксические верна).

3.2. Выходные воздействия

`z1` – задание начального состояния (помещаем на вершину стека начальное состояние);

`z2` – выполнение переноса (помещаем на верину стека новое состояние и вызываем метод, соответствующий данному терминалу);

`z3` – выполнение свертки (удаляем из стека состояния в количестве равном числу элементов в правой части соответствующей свертки, а так же запускает метод, соответствующий заданной свертке);

`z4` – новое состояние после свертки (для получаенного после свертки нетерминала получаем из таблицы переходов новое состояние и опускаем его в стек);

`z5` – запуск автомата `A1` для получение очередного терминала, лексемы.

3.3. Словесное описание

Синтаксический анализатор выполняет обработку потока лексем, полученных в результате работы лексического анализатора. В процессе анализа выполняются свертки и переносы лексем. При этом вызываются необходимые функции генерации байт-кода.

3.4. Схема связей и граф переходов синтаксического анализатора

Схема связей изображена на рис. 9.

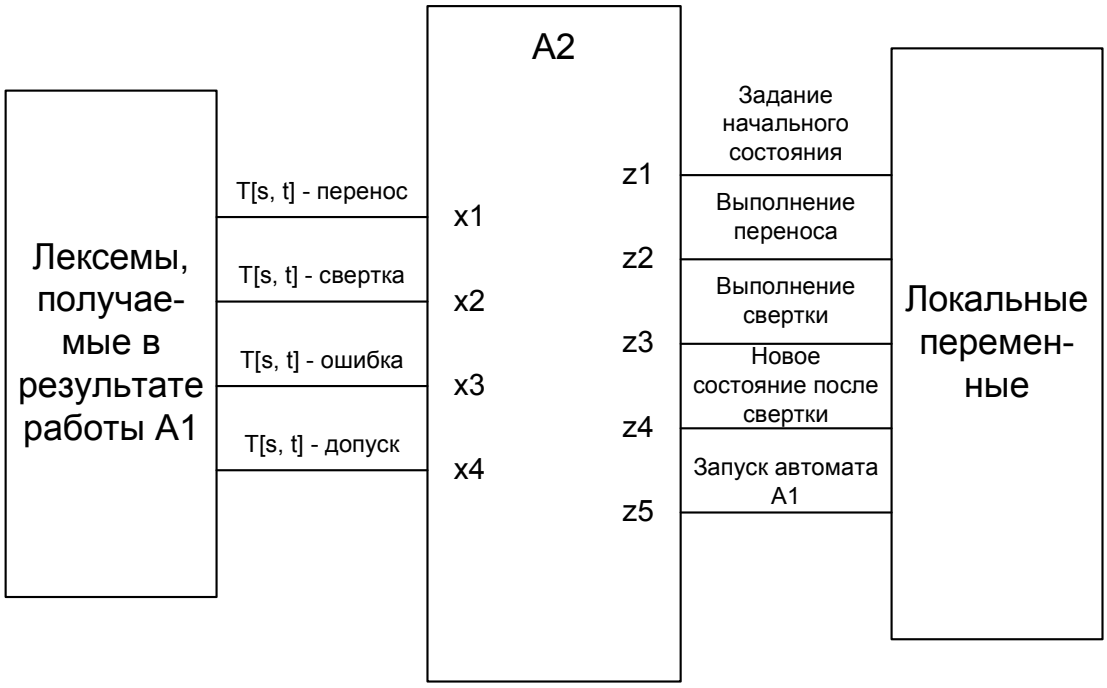

Рис. 9. Схема связей автомата синтаксического анализатора

Граф переходов изображен на рис. 10.

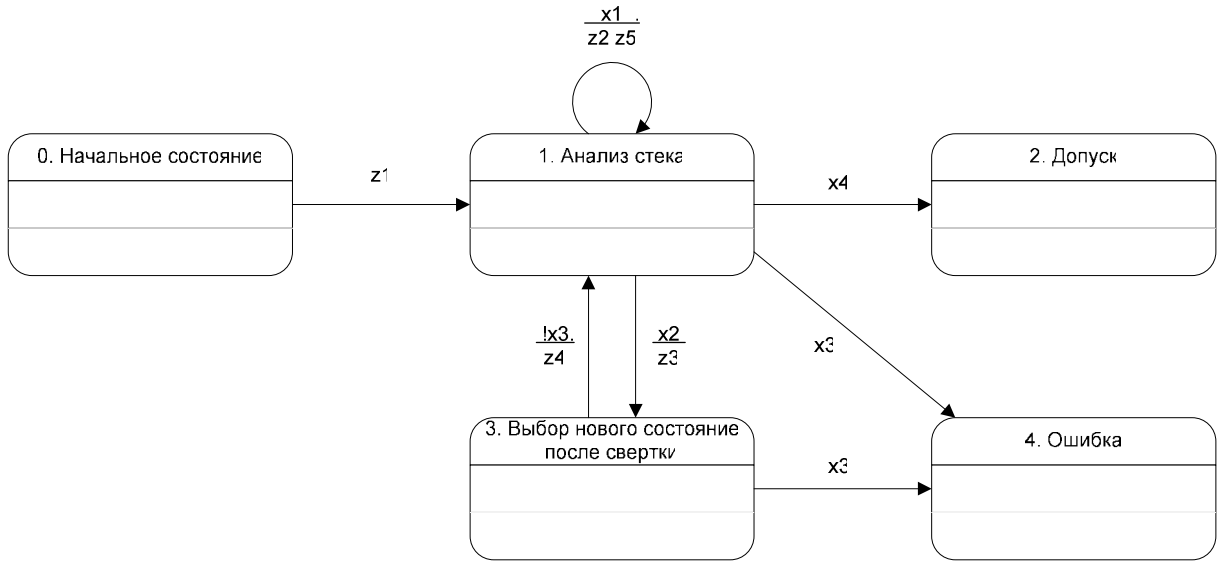

Рис. 10. Граф переходов автомата синтаксического анализатора

4. Интерпретатор

4.1. Описание входных переменных

`x1` – текущая команда в буфере байт-кода одна из команд: `PUSH, PUSHN, MOV, CALL, ADD, SUB, MUL, DIV, NEG, POW`;

`x2` – конец буфера (конец процедуры).

4.2. Выходные воздействия

`z1` – выполнение соответствующей команды ассемблера;

`z2` – переход к следующей команде.

4.3. Словесное описание

Интерпретатор осуществляет исполнение команд байт-кода, сгенерированных при синтаксическом разборе. Он вызывается независимо от синтаксического и лексического анализаторов. Благодаря этому интерпретатор обрабатывает команды не в виде единого потока, а набора самостоятельных инструкций, порядок которых может изменяться в случае условного перехода. На этом основаны условные операторы, циклы и вызовы процедур.

4.4. Схема связей и граф переходов интерпретатора

Схема связей изображена на рис. 11.

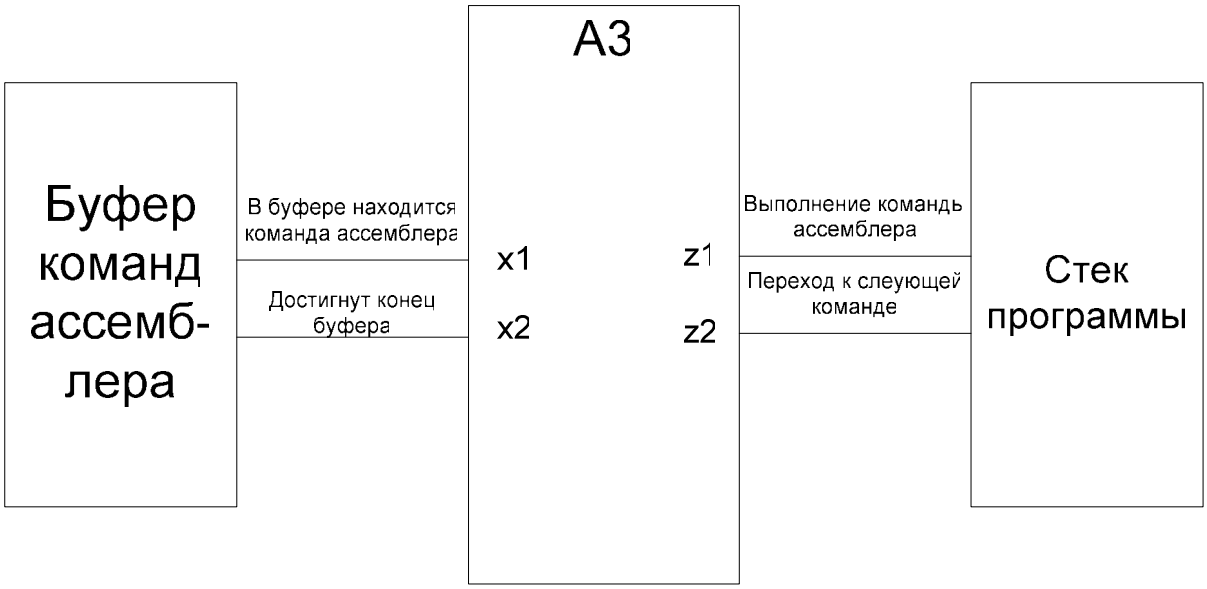

Рис. 11. Схема связей автомата интерпретатора

Граф переходов изображен на рис. 12.

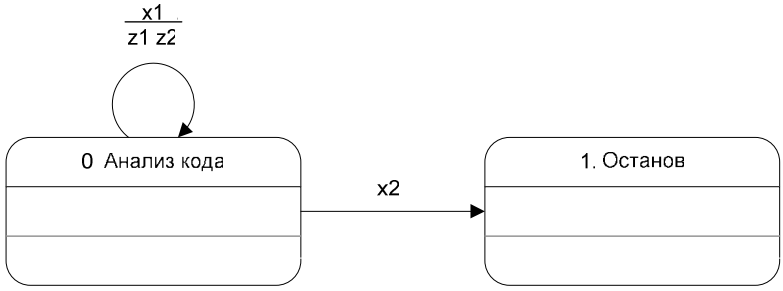

Рис. 12. Граф переходов автомата интерпретатора

5. Реализация

В качестве языка программирования выбран язык *Java*. Для реализации оконного интерфейса используются объекты пакета *Swing*. Для хранения данных применяются стандартные классы коллекций *Java*. Автоматы оформлены в виде классов.

Заключение

Совместное использование теории компиляторов и SWITCH-технологии позволило создать интерпретатор сконструированного авторами языка программирования более строго по сравнению с традиционным подходом.

Источники

1. http://www.javascript.com;
2. http://www.lua.org;
3. http://www.lua.org/uses.html;
4. *Хопкрофт Д., Мотвани Р., Ульман Д.* Введение в теорию автоматов, языков и вычислений. СПб.: Вильямс, 2002;
5. *Ахо А., Сети Р., Ульман Д.* Компиляторы. Принцип, технологии, инструменты. М.: Вильямс, 2001.

Приложения.

П1. Лексический анализатор

```
// file A1.java

package ru.ifmo.calculator.automata;

import java.io.IOException;
import java.io.Reader;

import org.apache.log4j.Logger;

public class A1 {

    private final static Logger log =
        Logger.getLogger(A1.class);

    public static class TokenType {
        public static final int TT_WORD = 0;
        public static final int TT_PLUS = 1;
        public static final int TT_MUNIS = 2;
        public static final int TT_MUL = 3;
        public static final int TT_DIV = 4;
        public static final int TT_POW = 5;
        public static final int TT_LB = 6;
        public static final int TT_RB = 7;
        public static final int TT_COMMA = 8;
        public static final int TT_DCOMMA = 9;
        public static final int TT_EQU = 10;
        public static final int TT_EOF = 11;
    };

    public static class AutomataState {
        public static final byte Auto_START = 0;
        public static final byte Auto_CHECKCOMMENT = 1;
        public static final byte Auto_STRINGCOMMENT = 2;
        public static final byte Auto_BLOCKCOMMENT = 3;
        public static final byte Auto_CHECKEOB = 4;
        public static final byte Auto_READWORD = 5;
        public static final byte Auto_UEOF = 6;
        public static final byte Auto_STOP = 7;
    }
    /**
     * Нетерминалы.
     */
    public static class NONTerminal {
        public static final int NT_Program = 0;
        public static final int NT_Statement = 1;
        public static final int NT_Expression = 2;
        public static final int NT_Term = 3;
        public static final int NT_Frac = 4;
        public static final int NT_Frac2 = 5;
```

```
    public static final int NT_ArgList = 6;
    public static final int NT_ArgList2 = 7;
    public static final int NT_Program_ = 8;
}
/**
 * Терминалы.
 */
public static class Terminal {
    public static final int TR_Word = 0;
    public static final int TR_Plus = 1;
    public static final int TR_Minus = 2;
    public static final int TR_Mul = 3;
    public static final int TR_Div = 4;
    public static final int TR_Pow = 5;
    public static final int TR_Lb = 6;
    public static final int TR_Rb = 7;
    public static final int TR_Comma = 8;
    public static final int TR_DComma = 9;
    public static final int TR_Edu = 10;
    public static final int TR_EOF =11;
}

private byte state;
private Reader input;
private int token;
private char lastChar;
private final static String x2CharSet =
    new String( "+-*/^(),;=:");
private boolean inputEOS = false;
private StringBuffer result;
private long lineCount;

private boolean x1() {
    return ( lastChar==' ' ||
        lastChar == '\t' ||
        lastChar=='\n' ||
        lastChar == '\r' );
}
private boolean x2() {
    return (x2CharSet.indexOf( lastChar )!=-1);
}
private boolean x3() {
    return (lastChar=='/');
}
private boolean x4() {
    return (lastChar=='*');
}
private boolean x5() {
    return (lastChar=='\n');
}
private boolean x6() {
    return isEOS();
}

private void z1() {
```

```
    //log.debug( "reading char from input stream." );

    int u = -1;
    try {
        u = input.read();
    } catch (IOException e) {
        e.printStackTrace();
    }
    if ( u==-1 ) {
        inputEOS = true;
        lastChar = (char)65535;
    }else{
        lastChar = (char)u;
    }
}
private void z2() {
    result.append( lastChar );
}
private void z3() {
    result.delete( 0, result.length() );
}
private void z4() {
    ++lineCount;
}
private void z5() {
    token = TokenType.TT_WORD;
}
private void z6() {
    if ( "+".equals( result.toString() ) ) {
        token = TokenType.TT_PLUS;
    }else if ( "-".equals( result.toString() ) ) {
        token = TokenType.TT_MUNIS;
    }else if ( "*".equals( result.toString() ) ) {
        token = TokenType.TT_MUL;
    }else if ( "/".equals( result.toString() ) ||
            ":".equals( result.toString() ) ) {
        token = TokenType.TT_DIV;
    }else if ( "^".equals( result.toString() ) ) {
        token = TokenType.TT_POW;
    }else if ( "(".equals( result.toString() ) ) {
        token = TokenType.TT_LB;
    }else if ( ")".equals( result.toString() ) ) {
        token = TokenType.TT_RB;
    }else if ( ",".equals( result.toString() ) ) {
        token = TokenType.TT_COMMA;
    }else if ( ";".equals( result.toString() ) ) {
        token = TokenType.TT_DCOMMA;
    }else if ( "=".equals( result.toString() ) ) {
        token = TokenType.TT_EQU;
    };
}
private void z7() {
    token = TokenType.TT_EOF;
}
```

```
public boolean isEOS() {
    return inputEOS;
}
public String getResult() {
    return result.toString();
}
public long getLineCount() {
    return lineCount;
}

/**
 *
 * @return
 */
private boolean isStopped() {
    return ( state==AutomataState.Auto_UEOF ||
            state==AutomataState.Auto_STOP );
}
public void runAutoMata() {
    state = AutomataState.Auto_START;
    result.delete(0, result.length());
    while(!isStopped())
        next();
}
public A1( Reader input ) {
    result = new StringBuffer("");
    init(input);
}

public void init( Reader input ) {
    this.input = input;
    lineCount = 0;
    z1();
}
public int getLastToken() {
    return token;
}

/**
 * ok.
 */
private void stateSTART() {
    if ( x3() ) {
        z2();
        z1();
        state = AutomataState.Auto_CHECKCOMMENT;
    }else if ( x1() && !x5() ) {
        z1();
    }else if ( x5() ) {
        z1();
        z4();
    }else if ( x2() && !x3() ) {
        z2();
        z6();
```

```
            z1();
            state = AutomataState.Auto_STOP;
        }else if ( !x1() && !x2() && !x6() ) {
            z2();
            z1();
            state = AutomataState.Auto_READWORD;
        }else if ( x6() ) {
            z7();
            state = AutomataState.Auto_STOP;
        }
    }
    private void stateCHECKCOMMENT() {
        if ( x3() ) {
            z1();
            z3();
            state = AutomataState.Auto_STRINGCOMMENT;
        }else if ( !x3() && !x4() ) {
            z6();
            state = AutomataState.Auto_STOP;
        }else if ( x4() ) {
            z1();
            z3();
            state = AutomataState.Auto_BLOCKCOMMENT;
        }
    }
    private void stateSTRINGCOMMENT() {
        if ( x5() || x6() ) {
            z1();
            state = AutomataState.Auto_START;
        }else if ( !x5() && !x6() ) {
            z1();
        }
    }
    private void stateBLOCKCOMMENT() {
        if ( x6() ) {
            state = AutomataState.Auto_UEOF;
        } else if ( x4() ) {
            z1();
            state = AutomataState.Auto_CHECKEOB;
        } else if ( !x4() && !x6() ) {
            z1();
        }
    }
    private void stateCHECKEOB() {
        if ( !x3() && !x6() ) {
            state = AutomataState.Auto_STRINGCOMMENT;
        }else if ( x3() ) {
            z1();
            state = AutomataState.Auto_START;
        }
    }
    private void stateREADWORD() {
        if ( !x1() && !x2() && !x6() ) {
            z2();
            z1();
```

```
        }else if ( x1() || x2() || x6() ) {
            z5();
            state = AutomataState.Auto_STOP;
        }
    }
    private void stateUEOF() {}
    private void stateSTOP() {}
    private void next() {
        switch(state) {
            case AutomataState.Auto_BLOCKCOMMENT:
                stateBLOCKCOMMENT();
                break;
            case AutomataState.Auto_CHECKCOMMENT:
                stateCHECKCOMMENT();
                break;
            case AutomataState.Auto_CHECKEOB:
                stateCHECKEOB();
                break;
            case AutomataState.Auto_READWORD:
                stateREADWORD();
                break;
            case AutomataState.Auto_UEOF:
                stateUEOF();
                break;
            case AutomataState.Auto_STRINGCOMMENT:
                stateSTRINGCOMMENT();
                break;
            case AutomataState.Auto_START:
                stateSTART();
                break;
            case AutomataState.Auto_STOP:
                stateSTOP();
                break;
            default:
                System.err.println("Error: State = "
                    + state );
        }
    }
}
```

П2. Синтаксический анализатор

```
// file A2.java
package ru.ifmo.calculator.automata;
import ru.ifmo.calculator.automata.actions.*;
import ru.ifmo.calculator.automata.workers.*;
import ru.ifmo.calculator.il.IILCommand;
import java.util.Stack;
import java.util.ArrayList;
import java.io.FileReader;
import java.io.FileNotFoundException;

import org.apache.log4j.Logger;

/**
 * Автомат для компилятора.
 */
public class A2 {

    private boolean accepted = false;

    private final static Logger log =
        Logger.getLogger( A2.class );

    private String errorString = null;

    /**
     * Состояния автомата.
     */
    public static class AutomataState {
        public static final byte Auto_START = 0;
        public static final byte Auto_STACKANALYS = 1;
        public static final byte Auto_ACCEPT = 2;
        public static final byte Auto_NEWSTATE = 3;
        public static final byte Auto_ERROR = 4;
    }
    /**
     * Массив переносов-сверток.
     */
    public static IAction [][] CompileTable;
    /**
     * Автомат лексического анализатора - A1.
     */
    private A1 lexParser;
    /**
     * Состояние автомата A2.
     */
    private byte state;

    private void stateSTART() {
        z1();
    }
    private void stateASTACK() {
```

```
    if ( x1() ) {
        z2();
        z5();
    } else if ( x2() ) {
        try {
            z3();
            state = AutomataState.Auto_NEWSTATE;
        } catch (Exception e) {
            state = AutomataState.Auto_ERROR;
            errorString = e.getMessage();
        }
    } else if ( x3() ) {
        state = AutomataState.Auto_ERROR;
    } else if ( x4() ) {
        z6();
        state = AutomataState.Auto_ACCEPT;
    }
}
private void stateACCEPT() {}
private void stateNEWSTATE() {
    if ( !x3() ) {
        z4();
        state = AutomataState.Auto_STACKANALYS;
    }else state = AutomataState.Auto_ERROR;
}

public String getErrorString() {
    return errorString;
}

private void stateERROR() {}

private void next() {
    log.info( "Current state " + state
        + " comp state " + compillerState + " A1 "
        + lexParser.getLastToken()
        + " result string '"+lexParser.getResult()+"'" );
    switch( state ) {
        case AutomataState.Auto_START:
            stateSTART();
            break;
        case AutomataState.Auto_ACCEPT:
            stateACCEPT();
            break;
        case AutomataState.Auto_ERROR:
            stateERROR();
            break;
        case AutomataState.Auto_NEWSTATE:
            stateNEWSTATE();
            break;
        case AutomataState.Auto_STACKANALYS:
            stateASTACK();
            break;
    }
}
```

```
public void runAutoMata() {
    while ( state!=AutomataState.Auto_ACCEPT &&
    state!=AutomataState.Auto_ERROR ) {
        next();
    }
    if ( state==AutomataState.Auto_ACCEPT )
        accepted = true;
}

private int compillerState;
private int symbol;
private Stack compillerStateStack;
private Stack variableStack;
private IAction currentAction;

public ArrayList compilledCode;

private boolean x1() {
    return (CompileTable[compillerState][symbol]
        instanceof ShiftAction);
}
private boolean x2() {
    return (CompileTable[compillerState][symbol]
        instanceof ReduceAction);
}
private boolean x3() {
    return (CompileTable[compillerState][symbol] == null );
}
private boolean x4() {
    return (CompileTable[compillerState][symbol]
        instanceof AcceptAction);
}

public A2( String program ) {
    lexParser = new A1(new java.io.StringReader(program));
    compilledCode = new ArrayList();
    symbol = 0;
    compillerStateStack = new Stack();
    variableStack = new Stack();
    state = AutomataState.Auto_START;
}

public A2() throws FileNotFoundException {
    lexParser = new A1(new FileReader("input.txt"));
    compilledCode = new ArrayList();
    symbol = 0;
    compillerStateStack = new Stack();
    variableStack = new Stack();
    state = AutomataState.Auto_START;
}

public A1 getLexParser() {
    return lexParser;
}
```

```
/**
 * Начальное положение - стартовое.
 */
private void z1() {
    state = AutomataState.Auto_STACKANALYS;
    compillerStateStack.push(new Integer(0));

    z5();
}
/**
 * Действия выолняемые при переносе какого либо символа.
 */
private void z2() {
    ShiftAction sa =
        (ShiftAction)CompileTable[compillerState][symbol];
    compillerState = sa.getNewState();
    compillerStateStack.push(new Integer(compillerState));
}
/**
 * выполнение свертки (удаляем из стека состояния
    в количестве равном
 * числу элементов в правой части соответствующей свертки,
    а так же
 * запускает метод, соответствующий заданной свертке)
 */
private void z3() throws Exception {
    currentAction = CompileTable[compillerState][symbol];
    ReduceAction rd = (ReduceAction)currentAction;
    for( int i=0; i<rd.getReduceItemCount(); ++i )
        compillerStateStack.pop();

    rd.performAction( this );
}
private void z4() {
    //взяли последнюю свертку.
    ReduceAction rd =
        (ReduceAction)CompileTable[compillerState][symbol];

    //берем новое состояние
    compillerState =
        ((Integer)compillerStateStack.peek()).intValue();

    symbol = rd.getNonTerminal();
    //все получаем новое состояние и мы довольны!
    //compillerState = sd.getNewState();

    if ( CompileTable[compillerState][symbol]
            instanceof ShiftAction ) {
        ShiftAction sa =
        (ShiftAction) CompileTable[compillerState][symbol];
        compillerState = sa.getNewState();
        compillerStateStack.push(
            new Integer(compillerState));
        symbol = lexParser.getLastToken() + 9;
```

```
    } else {
        //Accept ACTION
    }
}
private void z5() {
    lexParser.runAutoMata();
    symbol = lexParser.getLastToken() + 9;

    if (symbol == A1.Terminal.TR_Word + 9) {
        Object value;
        if (Character.isDigit(
            lexParser.getResult().charAt(0))) {
            value = new Float(Float.parseFloat(
                lexParser.getResult()));
        } else {
            value = lexParser.getResult();
        }
        variableStack.push(value);
    }
}
private void z6() {
    AcceptAction.generateHaltCode( this );
}
public Stack getVariableStack() {
    return variableStack;
}
public long getLineCount() {
    return lexParser.getLineCount();
}
public boolean isAccepted() {
    return accepted;
}
public void setAccepted(boolean accepted) {
    this.accepted = accepted;
}   /**
 * Заполняем массив CompileTable
 */
static {
    CompileTable = new IAction[34][21];
    for( int i=0; i<34; i++ )
        for( int j=0; j<21; j++ )
            CompileTable[i][j] = null;
    CompileTable[0][0] = new ShiftAction( 1 );
    CompileTable[0][8] = new AcceptAction();
    CompileTable[0][9] = new ReduceAction(
        A1.NONTerminal.NT_Program, 1,
        EmptyCodeGeneratorImplementator.getInstance());
    CompileTable[0][11] = new ReduceAction(
        A1.NONTerminal.NT_Program, 1,
        EmptyCodeGeneratorImplementator.getInstance());
    CompileTable[0][15] = new ReduceAction(
        A1.NONTerminal.NT_Program, 1,
        EmptyCodeGeneratorImplementator.getInstance());
    CompileTable[0][20] = new ReduceAction(
        A1.NONTerminal.NT_Program, 1,
```

```
    EmptyCodeGeneratorImplementator.getInstance());

CompileTable[1][1] = new ShiftAction( 5 );
CompileTable[1][2] = new ShiftAction( 6 );
CompileTable[1][3] = new ShiftAction( 7 );
CompileTable[1][4] = new ShiftAction( 8 );
CompileTable[1][5] = new ShiftAction( 9 );
CompileTable[1][9] = new ShiftAction( 2 );
CompileTable[1][11] = new ShiftAction( 3 );
CompileTable[1][15] = new ShiftAction( 4 );
CompileTable[1][20] = new ReduceAction(
    A1.NONTerminal.NT_Program_, 0,
    EmptyCodeGeneratorImplementator.getInstance() );

CompileTable[2][10] = new ReduceAction(
    A1.NONTerminal.NT_Frac2, 1,
    VarPushCodeGeneratorImplementator.getInstance() );
CompileTable[2][11] = new ReduceAction(
    A1.NONTerminal.NT_Frac2, 1,
    VarPushCodeGeneratorImplementator.getInstance() );
CompileTable[2][12] = new ReduceAction(
    A1.NONTerminal.NT_Frac2, 1,
    VarPushCodeGeneratorImplementator.getInstance() );
CompileTable[2][13] = new ReduceAction(
    A1.NONTerminal.NT_Frac2, 1,
    VarPushCodeGeneratorImplementator.getInstance() );
CompileTable[2][14] = new ReduceAction(
    A1.NONTerminal.NT_Frac2, 1,
    VarPushCodeGeneratorImplementator.getInstance() );
CompileTable[2][15] = new ShiftAction( 10 );
CompileTable[2][18] = new ReduceAction(
    A1.NONTerminal.NT_Frac2, 1,
    VarPushCodeGeneratorImplementator.getInstance() );
CompileTable[2][19] = new ShiftAction( 11 );
CompileTable[3][5] = new ShiftAction( 13 );
CompileTable[3][9] = new ShiftAction( 12 );
CompileTable[3][11] = new ShiftAction( 3 );
CompileTable[3][15] = new ShiftAction( 4 );
CompileTable[4][2] = new ShiftAction( 14 );
CompileTable[4][3] = new ShiftAction( 7 );
CompileTable[4][4] = new ShiftAction( 8 );
CompileTable[4][5] = new ShiftAction( 9 );
CompileTable[4][9] = new ShiftAction( 12 );
CompileTable[4][11] = new ShiftAction( 3 );
CompileTable[4][15] = new ShiftAction( 4 );
CompileTable[5][18] = new ShiftAction( 15 );
CompileTable[6][10] = new ShiftAction( 16 );
CompileTable[6][11] = new ShiftAction( 17 );
CompileTable[6][18] = new ReduceAction(
    A1.NONTerminal.NT_Statement, 1,
    EmptyCodeGeneratorImplementator.getInstance() );

CompileTable[7][10] = new ReduceAction(
    A1.NONTerminal.NT_Expression, 2,
    EmptyCodeGeneratorImplementator.getInstance() );
```

```
CompileTable[7][11] = new ReduceAction(
    A1.NONTerminal.NT_Expression, 2,
    EmptyCodeGeneratorImplementator.getInstance() );
CompileTable[7][12] = new ShiftAction( 18 );
CompileTable[7][13] = new ShiftAction( 19 );
CompileTable[7][16] = new ReduceAction(
    A1.NONTerminal.NT_Expression, 2,
    EmptyCodeGeneratorImplementator.getInstance() );
CompileTable[7][17] = new ReduceAction(
    A1.NONTerminal.NT_Expression, 2,
    EmptyCodeGeneratorImplementator.getInstance() );
CompileTable[7][18] = new ReduceAction(
    A1.NONTerminal.NT_Expression, 2,
    EmptyCodeGeneratorImplementator.getInstance() );
CompileTable[8][10] = new ReduceAction(
    A1.NONTerminal.NT_Term, 2,
    EmptyCodeGeneratorImplementator.getInstance() );
CompileTable[8][11] = new ReduceAction(
    A1.NONTerminal.NT_Term, 2,
    EmptyCodeGeneratorImplementator.getInstance() );
CompileTable[8][12] = new ReduceAction(
    A1.NONTerminal.NT_Term, 2,
    EmptyCodeGeneratorImplementator.getInstance() );
CompileTable[8][13] = new ReduceAction(
    A1.NONTerminal.NT_Term, 2,
    EmptyCodeGeneratorImplementator.getInstance() );
CompileTable[8][14] = new ShiftAction( 20 );
CompileTable[8][16] = new ReduceAction(
    A1.NONTerminal.NT_Term, 2,
    EmptyCodeGeneratorImplementator.getInstance() );
CompileTable[8][17] = new ReduceAction(
    A1.NONTerminal.NT_Term, 2,
    EmptyCodeGeneratorImplementator.getInstance() );
CompileTable[8][18] = new ReduceAction(
    A1.NONTerminal.NT_Term, 2,
    EmptyCodeGeneratorImplementator.getInstance() );

CompileTable[10][2] = new ShiftAction( 21 );
CompileTable[10][3] = new ShiftAction( 7 );
CompileTable[10][4] = new ShiftAction( 8 );
CompileTable[10][5] = new ShiftAction( 9 );
CompileTable[10][6] = new ShiftAction( 22 );
CompileTable[10][7] = new ShiftAction( 23 );
//CompileTable[10][4] = new ShiftAction( 8 );
CompileTable[10][9] = new ShiftAction( 12 );
CompileTable[10][11] = new ShiftAction( 3 );
CompileTable[10][15] = new ShiftAction( 4 );
CompileTable[10][16] = new ReduceAction(
    A1.NONTerminal.NT_ArgList, 0,
    EmptyCodeGeneratorImplementator.getInstance() );
CompileTable[11][2] = new ShiftAction( 24 );
CompileTable[11][3] = new ShiftAction( 7 );
CompileTable[11][4] = new ShiftAction( 8 );
CompileTable[11][5] = new ShiftAction( 9 );
CompileTable[11][9] = new ShiftAction( 12 );
```

```
CompileTable[11][11] = new ShiftAction( 3 );
CompileTable[11][15] = new ShiftAction( 4 );
CompileTable[12][10] = new ReduceAction(
    A1.NONTerminal.NT_Frac2, 1,
    VarPushCodeGeneratorImplementator.getInstance() );
CompileTable[12][11] = new ReduceAction(
    A1.NONTerminal.NT_Frac2, 1,
    VarPushCodeGeneratorImplementator.getInstance() );
CompileTable[12][12] = new ReduceAction(
    A1.NONTerminal.NT_Frac2, 1,
    VarPushCodeGeneratorImplementator.getInstance() );
CompileTable[12][13] = new ReduceAction(
    A1.NONTerminal.NT_Frac2, 1,
    VarPushCodeGeneratorImplementator.getInstance() );
CompileTable[13][17] = new ReduceAction(
    A1.NONTerminal.NT_Frac2, 0,
    new NegativateCodeGeneratorImplementator() );
CompileTable[13][18] = new ReduceAction(
    A1.NONTerminal.NT_Frac2, 0,
    new NegativateCodeGeneratorImplementator() );
CompileTable[14][10] = new ShiftAction( 16 );
CompileTable[14][11] = new ShiftAction( 17 );
CompileTable[14][16] = new ShiftAction( 25 );

CompileTable[16][3] = new ShiftAction( 26 );
CompileTable[16][4] = new ShiftAction( 8 );
CompileTable[16][5] = new ShiftAction( 9 );
CompileTable[16][9] = new ShiftAction( 12 );
CompileTable[16][11] = new ShiftAction( 3 );
CompileTable[16][15] = new ShiftAction( 4 );

CompileTable[17][3] = new ShiftAction( 27 );
CompileTable[17][4] = new ShiftAction( 8 );
CompileTable[17][5] = new ShiftAction( 9 );
CompileTable[17][9] = new ShiftAction( 12 );
CompileTable[17][11] = new ShiftAction( 3 );
CompileTable[17][15] = new ShiftAction( 4 );

CompileTable[18][4] = new ShiftAction( 28 );
CompileTable[18][5] = new ShiftAction( 9 );
CompileTable[18][9] = new ShiftAction( 12 );
CompileTable[18][11] = new ShiftAction( 3 );
CompileTable[18][15] = new ShiftAction( 4 );

CompileTable[19][4] = new ShiftAction( 29 );
CompileTable[19][5] = new ShiftAction( 9 );
CompileTable[19][9] = new ShiftAction( 12 );
CompileTable[19][11] = new ShiftAction( 3 );
CompileTable[19][15] = new ShiftAction( 4 );

CompileTable[20][5] = new ShiftAction( 30 );
CompileTable[20][9] = new ShiftAction( 12 );
CompileTable[20][11] = new ShiftAction( 3 );
CompileTable[20][15] = new ShiftAction( 4 );
CompileTable[26][10] = new ReduceAction(
```

```
            A1.NONTerminal.NT_Expression, 0,
            new AddCodeGeneratorImplementator() );
        CompileTable[26][11] = new ReduceAction(
            A1.NONTerminal.NT_Expression, 0,
            new AddCodeGeneratorImplementator() );
        CompileTable[26][12] = new ShiftAction( 18 );
        CompileTable[26][13] = new ShiftAction( 19 );
        CompileTable[26][16] = new ReduceAction(
            A1.NONTerminal.NT_Expression, 0,
            new AddCodeGeneratorImplementator() );
        CompileTable[26][17] = new ReduceAction(
            A1.NONTerminal.NT_Expression, 0,
            new AddCodeGeneratorImplementator() );
        CompileTable[26][18] = new ReduceAction(
            A1.NONTerminal.NT_Expression, 0,
            new AddCodeGeneratorImplementator() );
        CompileTable[30][14] = new ReduceAction(
            A1.NONTerminal.NT_Frac, 0,
            new PowCodeGeneratorImplementator() );
        CompileTable[25][18] = new ReduceAction(
            A1.NONTerminal.NT_Frac2, 3,
            EmptyCodeGeneratorImplementator.getInstance() );
        CompileTable[32][2] = new ShiftAction( 33 );
        CompileTable[32][3] = new ShiftAction( 7 );
        CompileTable[32][4] = new ShiftAction( 8 );
        CompileTable[32][5] = new ShiftAction( 9 );
        CompileTable[32][9] = new ShiftAction( 12 );
        CompileTable[32][11] = new ShiftAction( 3 );
        CompileTable[32][15] = new ShiftAction( 4 );

        CompileTable[33][10] = new ShiftAction( 16 );
        CompileTable[33][11] = new ShiftAction( 17 );
        CompileTable[33][16] = new ReduceAction(
            A1.NONTerminal.NT_Frac2, 0,
            EmptyCodeGeneratorImplementator.getInstance() );
        CompileTable[33][17] = new ReduceAction(
            A1.NONTerminal.NT_Frac2, 0,
            EmptyCodeGeneratorImplementator.getInstance() );
    }
}
```

П3. Интерпретатор

```
package ru.ifmo.calculator.automata;
import java.util.*;
import ru.ifmo.calculator.il.*;
public class A3 {
    private static abstract class Functor {
        public abstract void run(Stack stack, Map variables);
    }
    private static abstract class Functor1ArgFloatRetFloat
            extends Functor {
        public void run(Stack stack, Map vars) {
            Float d = getValue(stack.pop(), vars);
            stack.push(new Float(call(d.floatValue())));
        }
        protected abstract float call(float value);
    }
    public static Map functions;
    static {
        functions = new HashMap();
        functions.put("sin", new Functor1ArgFloatRetFloat() {
            protected float call(float value) {
                return (float) Math.sin(value);
            }
        });
        functions.put("cos", new Functor1ArgFloatRetFloat() {
            protected float call(float value) {
                return (float) Math.cos(value);
            }
        });
        functions.put("abs", new Functor1ArgFloatRetFloat() {
            protected float call(float value) {
                return Math.abs(value);
            }
        });
        functions.put("sqrt", new Functor1ArgFloatRetFloat() {
            protected float call(float value) {
                return (float) Math.sqrt(value);
            }
        });
    }
    public static class AutomataState{
        public static final int Auto_START = 0;
        public static final int Auto_STOP = 1;
    }
    private int state;
    private IILCommand currentCommand;
    private Stack runStack = new Stack();
    private Map varValues = new HashMap();
    private List code;
    private Iterator codeIterator;
    public A3(List code) {
        this.code = new ArrayList(code);
```

```
}
public void clearVariables() {
    varValues.clear();
}
public Map getVariables() {
    return varValues;
}
private Float getVariableValue(String name) {
    return getVariableValue(name, varValues);
}
private static Float getValue(Object o, Map vars) {
    if ( o instanceof Float ) return (Float)o;
    if ( o instanceof String )
        return getVariableValue((String)o, vars);
    return new Float(0);
}
private static Float getVariableValue(String name, Map vars)
{
    Float result = (Float)vars.get(name);
    if (result == null)
        return new Float(0);
    return result;
}

private Float getValue( Object o ) {
    return getValue(o, varValues);
}

private void setVariableValue( String name, Float value ) {
    varValues.put( name, value );
}

private void stateSTART() {
    if (x1()) {
        z1();
    }else if ( x2() ) {
        z2();
        state = AutomataState.Auto_STOP;
    }else if ( x3() ) {
        z3();
    }else if ( x4() ) {
        z4();
    }else if ( x5() ) {
        z5();
    }else if ( x6() ) {
        z6();
    }else if ( x7() ) {
        z7();
    }else if ( x8() ) {
        z8();
    }
}

private void z8() {
    runStack.push(new Float(-getValue(
```

```
            runStack.pop()).floatValue()));
    }
    private void z7() {
        String p = (String)runStack.pop();
        Float f = getValue(runStack.pop());
        setVariableValue( p, f );
        runStack.push(f);
    }
    private boolean x7() {
        return currentCommand instanceof ILMovCommand;
    }
    private void z6() {
        runStack.push(
            ((ILPushFltCommand)currentCommand).getValue());
    }
    private boolean x6() {
        return currentCommand instanceof ILPushFltCommand;
    }
    private boolean x5() {
        return currentCommand instanceof ILPushVarCommand;
    }
    private void z5() {
        runStack.push((
            (ILPushVarCommand)currentCommand).getVarName());
    }
    private boolean x4() {
        return currentCommand instanceof ILCallCommand;
    }
    private boolean x3() {
        return currentCommand instanceof ILPopCommand;
    }
    private boolean x2() {
        return currentCommand instanceof ILHaltCommand;
    }
    private void z4() {
        ILCallCommand ilc = (ILCallCommand)currentCommand;
        Functor func =
            (Functor) functions.get(ilc.getFuncName());
        if (func != null) {
            func.run(runStack, varValues);
            return;
        }
    }
    private void z2() {}
    private void z3() {
        runStack.pop();
    }
    private void z1() {
        Object t0 = runStack.pop();
        Object t1 = runStack.pop();
        Float f = null;
        if (currentCommand instanceof ILAddCommand) {
            f = new Float(getValue(t0).floatValue()
                + getValue(t1).floatValue());
        } else if (currentCommand instanceof ILSubCommand ) {
```

```
            f = new Float(getValue( t1 ).floatValue()
                - getValue( t0 ).floatValue());
        } else if (currentCommand instanceof ILMulCommand ) {
            f = new Float(getValue( t0 ).floatValue()
                * getValue( t1 ).floatValue());
        } else if (currentCommand instanceof ILPowCommand ) {
            float _f = 1.0f;
            int h = getValue( t0 ).intValue();
            for( int i=0; i<Math.abs( h ); i++ ) {
                _f = _f * getValue(t1).floatValue();
            }
            if ( h < 0 )
                _f = 1.0f / _f;
            f = new Float(_f);
        }else if ( currentCommand instanceof ILDivCommand ) {
            //???
            f = new Float(getValue(t1).floatValue()
                / getValue(t0).floatValue());
        };
        runStack.push(f);
    }
    private boolean x1() {
        return currentCommand instanceof ILAddCommand
                    || currentCommand instanceof ILSubCommand
                    || currentCommand instanceof ILDivCommand
                    || currentCommand instanceof ILMulCommand
                    || currentCommand instanceof ILPowCommand;
    }
    private boolean x8() {
        return currentCommand instanceof ILNegativateCommand;
    }
    private void next() {
        switch(state) {
            case AutomataState.Auto_START:
                currentCommand = (IILCommand)codeIterator.next();
                stateSTART();
                break;
            case AutomataState.Auto_STOP:
        }
    }
    public void runAutomata() {
        state = AutomataState.Auto_START;
        codeIterator = code.iterator();
        while ( state!=AutomataState.Auto_STOP
            && codeIterator.hasNext() ) {
            next();
        }
    }
}
```