

методическое пособие

# ОБРАБОТКА ТЕКСТОВЫХ ФОРМАТОВ ДАННЫХ И РЕАЛИЗАЦИЯ КОМПЬЮТЕРНЫХ ЯЗЫКОВ на Flex/Bison/C++/LLVM Java/C#/ANTLR

GitHub: <https://github.com/ponyatov/lexman>

© <[dponyatov@gmail.com](mailto:dponyatov@gmail.com)>

28 декабря 2015 г.

# Оглавление

Применение . . . . .	2
Необходимое программное обеспечение . . . . .	3
<b>1 Структура компилятора</b>	<b>4</b>
1.1 Термины . . . . .	4
1.2 Структура типового компилятора . . . . .	6
1.3 Архитектура LLVM . . . . .	7
<b>2 Типичная структура проекта</b>	<b>8</b>
2.1 README.md . . . . .	9
2.2 Makefile . . . . .	9
2.3 bat.bat . . . . .	11
2.4 rc.rc . . . . .	12
<b>3 Лексер и утилита flex</b>	<b>13</b>
3.1 Структура файла описания лексера . . . . .	14
3.2 Запуск flex . . . . .	15
3.2.1 Запуск в варианте для старого lex . . . . .	15

3.2.2	<b>flex</b> и генерация лексера на $C^{++}$	17
3.3	Регулярные выражения	18
3.4	Примеры самостоятельного применения	20
3.4.1	<b>Pij2D</b> : загрузка файла числовых данных	21
3.5	Схема файлов для связки <b>flex/bison</b>	26
3.6	Лексер для языка Ы: script/lpp.lpp	28
<b>4</b>	<b>Синтаксис и реализация языка Ы</b>	<b>30</b>
4.1	Комментарий	30
4.2	Скалярные типы	30
4.2.1	Строка	30
<b>5</b>	<b>ANTLR</b>	<b>32</b>
	<b>Литература</b>	<b>33</b>
	Основы компиляторов	33
	LLVM	34
	Java/antlr	34
	Утилиты	34
	L <sup>A</sup> T <sub>E</sub> X: система верстки для научных публикаций	34

## Применение

- обработка текстовых форматов данных  
файлы САПР, исходные данные для расчетных программ
- командный интерфейс для устройств на микроконтроллерах  
управление человеко-читаемыми командами, передача пакетов данных любой структуры и типов

- реализация специализированных скриптовых языков
- обработка исходных текстов программ  
модификация, трансляция на другие языки программирования,  
универсальный язык шаблонов для ЯП с ограниченными или отсутствующими макросами

## Необходимое программное обеспечение

- Windows

MinGW	<a href="http://www.mingw.org/">http://www.mingw.org/</a>	пакет компилятора и утилит C++ GNU GCC toolchain (g++, flex, bison, make)
git-scm	<a href="https://git-scm.com/">https://git-scm.com/</a>	git-клиент
gvim	<a href="http://www.vim.org/download.php#pc">http://www.vim.org/download.php#pc</a>	минималистичный редактор кода с самой простой подсветкой синтаксиса (на регулярках 3.3)
clang	<a href="http://llvm.org">http://llvm.org</a>	компилятор C/C++ на базе LLVM
llvm	???	сама библиотека LLVM

- Linux (суперкластер СГАУ "Сергей Королев")

g++, flex, bison, make, git, llvm(-3.5), clang

# Глава 1

## Структура компилятора

### 1.1 Термины

**исходный код** , исходник: текстовое представление программы, предназначенное для чтения и написания человеком. Формат определяется синтаксисом используемого языка программирования или описания данных

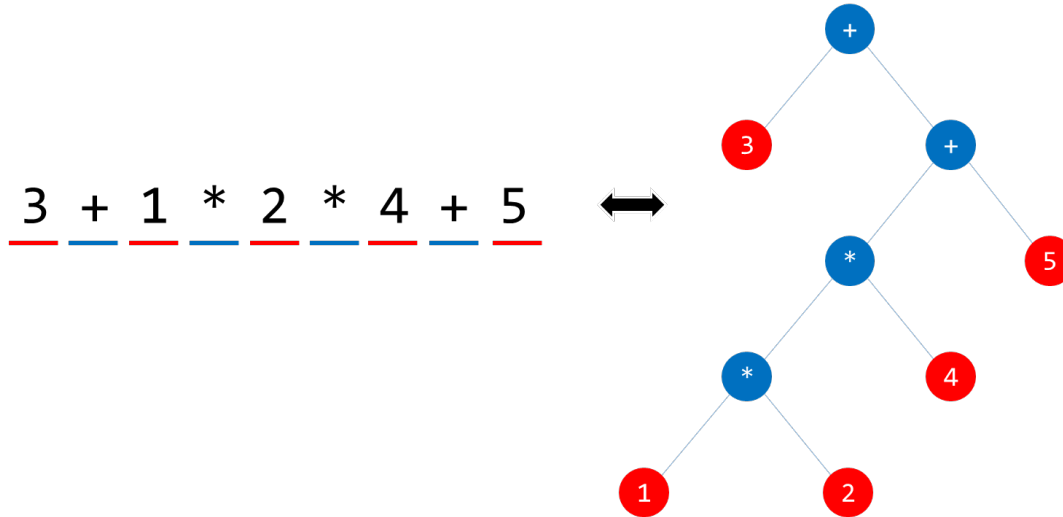
**лексер** **3** программный компонент, выполняющий выделение синтаксических элементов (токенов) из входного потока символов.

**токен** объект, содержащий выделенный из исходного кода текст, имя файла/строку/столбец исходника, маркер типа данных (число, строка, оператор), и т.п.

**парсер** **??** компонент, выполняющий анализ структуры текстового файла данных, с учетом вложенных скобок, синтаксических блоков типа begin/end, условных конструкций, описаний числовых матриц и векторов, и т.п.

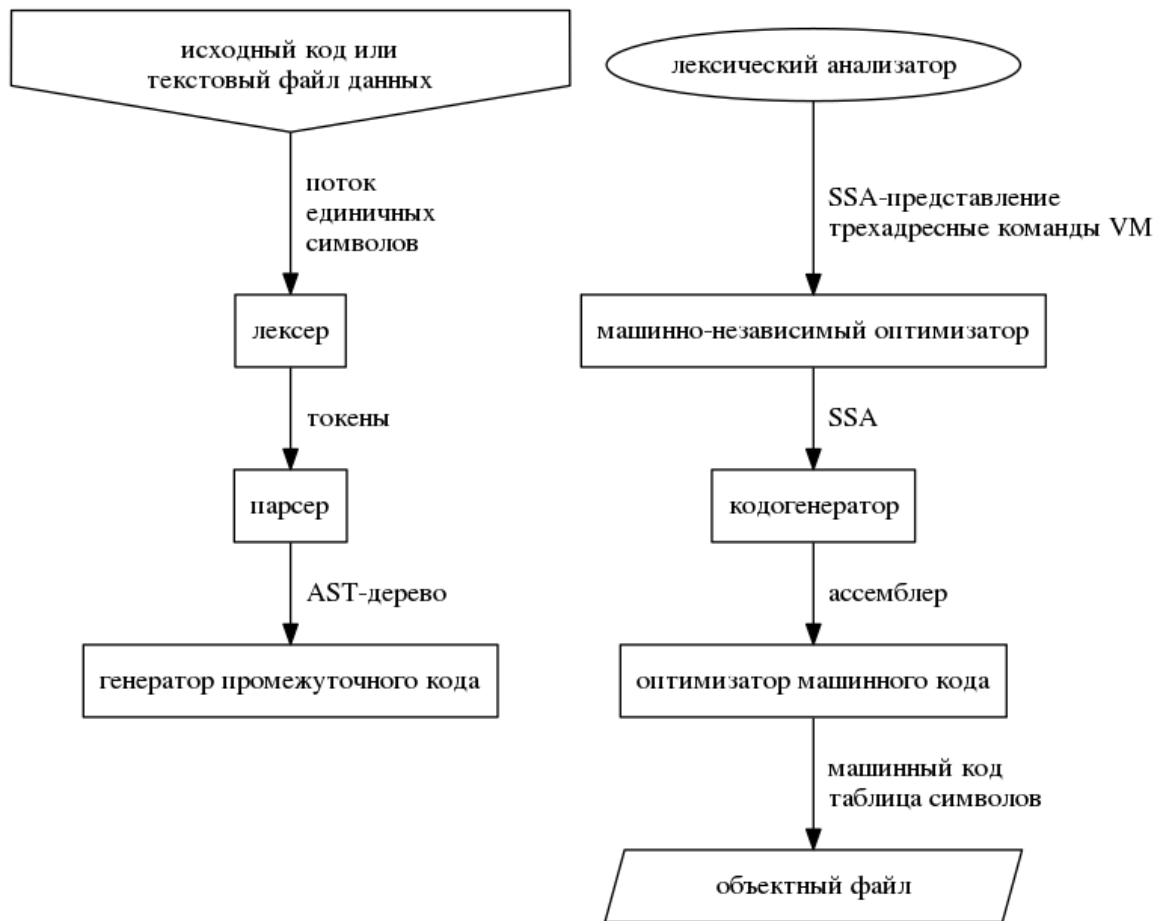
**AST** [A]bstract [S]yntax [T]ree, абстрактное синтаксическое дерево

вложенная структура данных, состоящая из синтаксических объектов: терминалы (целые, строки, символы,...) и нетерминалы (операторы ссылающиеся на операнды, блоки кода содержащие списки операций,...). AST хранит информацию о вложенности конструкций, порядке вычислений выражений, подчиненности элементов и т.п.



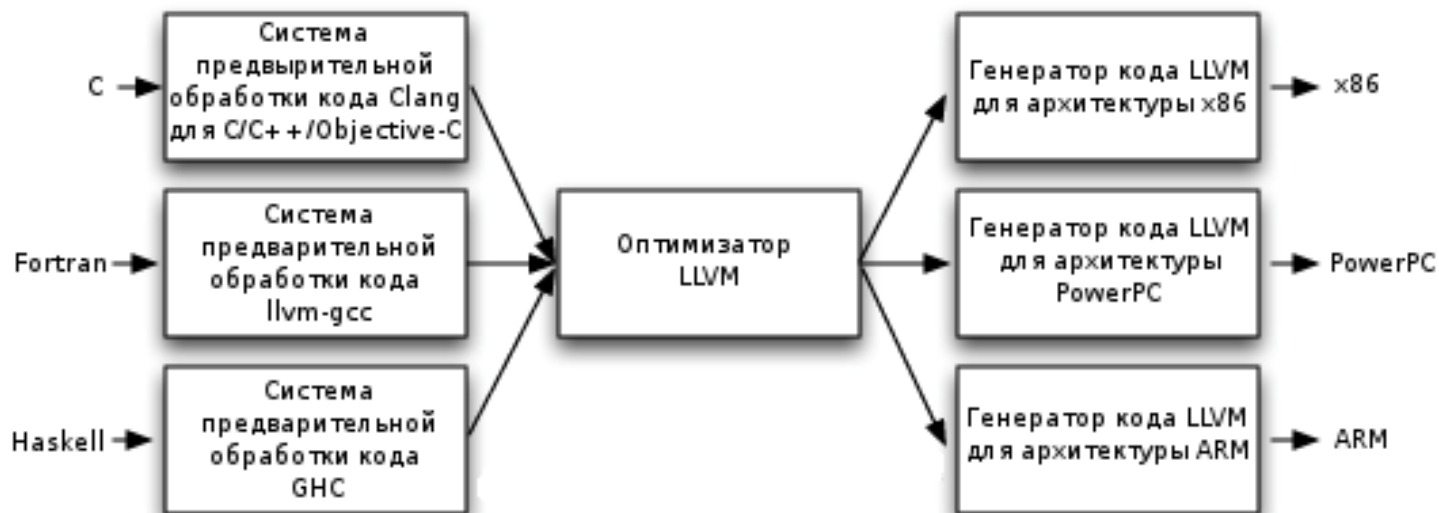
**SSA** ?? [S]ingle [S]tate [A]ssignment, однократное назначение: промежуточное представление, в котором каждой переменной значение присваивается лишь единожды. Переменные исходной программы разбиваются на версии, обычно с помощью добавления суффикса, таким образом, что каждое присваивание осуществляется уникальной версии переменной. В SSA используются [машинно-независимые трехадресные команды](#) абстрактной виртуальной машины.

## 1.2 Структура типового компилятора





## 1.3 Архитектура LLVM





# Глава 2

## Типичная структура проекта

README.md	2.1	github	описание проекта на <a href="https://github.com/">https://github.com/</a>
Makefile	2.2	make	зависимости между файлами и команды сборки
lpp.lpp	??	flex	лексер 3
ypp.ypp	??	bison	парсер ??
hpp.hpp	??	g++/clang++	заголовочные файлы C++
cpp.cpp	??	g++/clang++	C++-код: ядро интерпретатора, компилятор, реализация динамических типов, пользовательский код
bat.bat	2.3	win32	запускалка gvim
rc.rc	??	windres	описание ресурсов: иконки приложения, меню,..
logo.ico		windres	логотип в .ico формате
logo.png			логотип в .png (для github README)
filetype.vim	??	(g)vim	привязка расширения файлов скриптов
syntax.vim	??	(g)vim	синтаксическая подсветка для скриптов
.gitignore	??	git	список временных и производных файлов

## 2.1 README.md

```
# <логотип> <название>
(с) <имя> <email>
<лицензия>
<ссылка на проект на GitHub>
### <ссылки, дополнительная информация>
```

README.md

```
1 # ![logo](logo.png) Mega script language
2
3 (с) Vasya Pupkin <pupkin@gmail.com>, all rights reserved
4
5 license: http://www.gnu.org/copyleft/lesser.html
6
7 GitHub: https://github.com/pupkin/megascript
```

## 2.2 Makefile

Опции сборки (win32|linux):

EXE    суффикс исполняемого файла  
RES    имя объектного файла ресурсов win32.exe  
TAIL   опция команды **tail** число последних строк **MODULE.log**

Makefile

```
1 # EXE = .exe |
2 # RES = res.res |
3 # TAIL = -n17|-n7
```

Модуль заполняется автоматически по имени текущего каталога:

Makefile

```
1 MODULE = $(notdir $(CURDIR))
```

Цель команды **make** по умолчанию: сборка и интерпретация тестового файла

Makefile

```
1 .PHONY: exes
2 exes: ./$(MODULE)$(EXE)
3     ./$(MODULE)$(EXE) < $(MODULE).bi > $(MODULE).blog && tail $(TAIL) $(MODULE).blog
```

Вторая (стандартная) цель **clean**: удаление временных и рабочих файлов

Makefile

```
1 .PHONY: clean
2 clean:
3     rm -rf *~ *.~ *.exe *.elf *.log ypp.tab.?pp lex.yy.c $(RES)
```

Сборка C++ части

Makefile

```
1 C = cpp.cpp ypp.tab.cpp lex.yy.c
2 H = hpp.hpp ypp.tab.hpp
3 # CXX = clang++
4 CXXFLAGS += -I. -std=gnu++11
```

```
5 ./$(MODULE)$(EXE): $(C) $(H) $(RES)
6     $(CXX) $(CXXFLAGS) -o $@ $(C) $(RES)
```

Генерация кода парсера

Makefile

```
1 ypp.tab.cpp: ypp.ypp
2     bison $<
```

Генерация кода лексера

Makefile

```
1 lex.yy.c: lpp.lpp
2     flex $<
```

Компиляция файла ресурсов (win32)

Makefile

```
1 res.res: rc.rc
2     windres $< -O coff -o $@
```

## 2.3 bat.bat

bat.bat

```
1 @start .
2 @gvim -c "colorscheme darkblue" -p lexman.scr lexman.log \
3     ypp.ypp lpp.lpp hpp.hpp cpp.cpp \
4     Makefile
```

## 2.4 rc.rc

rc.rc

```
1 logo ICON "logo.ico"
```



# Глава 3

## Лексер и утилита flex

**Лексер** выполняет разбор входного потока единичных символов, выделяя из него группы символов. Код лексера генерируется с помощью утилиты **flex**, из набора правил, состоящих из двух частей:

1. регулярное выражение **3.3**, задающее шаблон для выделения группы символов, и
2. блок произвольного кода на  $C^{++}$ , выполняющего с найденным текстом нужные действия.

Для простых применений вы можете прописать нужные вам действия непосредственно с заданным текстом (запись в отдельный файл, преобразования, ...).

В случае использования лексера в составе транслятора/компилятора, лексер выполняет **токенизацию**: первичное преобразование найденных блоков исходного текста в **токены**.

## 3.1 Структура файла описания лексера

Для утилиты [flex](#)<sup>1</sup> используется файл с расширением `.l/.lex/.lpp` [5]:

```
// секция определений
%{
    // заголовочный C++ код
    #include "hpp.hpp"
    #include "parser.tab.hpp"
    std::string StringParseBuffer;
}%
// опции
%option ...
// дополнительные состояния лексера
%x state1
%s state2
// секция правил
%%
%%
// секция подпрограмм
```

Минимальный вариант `.lex`-файла:

```
%option main    // добавить автоматическую функцию main()
%%
...    // правила
```

---

<sup>1</sup> или ее предшественника [lex](#)

```
\n      {<код для конца строки>} или {}  
.  
%%      {<код для нераспознанного символа>}, {}  
%%
```

## 3.2 Запуск flex

### 3.2.1 Запуск в варианте для старого lex

Для начала рассмотрим вариант использования для старой версии лексического генератора **lex**, который вы внезапно встретите в какой-нибудь старой коммерческой UNIX-системе. Подробно отличия рассмотрены в [6].

empty.l

```
1%option main  
2%%  
3%%
```

```
lex empty.l  
cc -o empty.exe lex.yy.c  
./empty.exe < lex.yy.c > empty.log
```

На новых UNIXах аналогичного результата можно добиться командами, включающими режим совместимости со старыми версиями ПО:

```
flex -l empty.l  
gcc -std=c89 -Wpedantic -o empty.exe lex.yy.c  
./empty.exe < lex.yy.c > empty.log
```



После выполнения команды `lex` будет создан файл `lex.yy.c`, содержащий [чисто синтый](#) код лексера, который можно откомпилировать любым ANSI-совместимым компилятором Си для любого микроконтроллера, или отечественной ВПКшной поделки типа [КР1878ВЕ1](#).

Полученная программа читает символы с `stdin`, и выводит все нераспознанные символы на `stdout`.

Сравнив файлы `lex.yy.c` и `empty.log`, вы увидите что они полностью совпадают. Чтобы сделать что-то типа полезное, добавим несколько правил, и получим список команд препроцессора, характерных для языка Си:

В конец набора правил добавим удаление пробельных символов и нераспознанных символов:

empty.l

```
1%option main
2%%
3[ \t\r\n]+ {} /* spaces */
4. {} /* undetected chars */
5%%
```

empty.log

В итоге мы получили пустой файл, так как были удалены все символы. Теперь пользуясь справочником по языку Си, добавим [в начало списка](#) правило, используя [регулярное выражение](#)<sup>2</sup> для команд препроцессора:

empty.l

```
1#.+\\n { printf("%s",yytext); }
```

---

<sup>2</sup> подробно рассмотрены далее [3.3](#)

```

1#line 3 "lex.yy.c"
2#define YY_INT_ALIGNED short int
3#define FLEX_SCANNER
4#define YY_FLEX_MAJOR_VERSION 2
5#define YY_FLEX_MINOR_VERSION 5

```

В результате на выходе мы получили все части строк от символа `#` до конца строки `\n`, между которыми находится 1+ любых символов `.+`.

### 3.2.2 flex и генерация лексера на C++

Если вы пишете лексический анализатор для компьютера, а не микроконтроллера, это удобнее делать на C++. Если вы пишете на C++, лучше использовать расширения файлов `.lpp`. Это расширение также укажет на то, что полученный генератор ограниченно применим для микроконтроллера: код на C++ для МК требует очень аккуратной работы с динамической памятью из-за малого объема ОЗУ.

Современный генератор анализаторов flex поддерживает два варианта генерации кода, совместимого с C++:

1. использовать традиционный запуск `flex empty.lpp`, но компилировать полученный `lex.yy.c` компилятором `g++`: в этом случае вы можете свободно использовать в правилах код на C++, но весь ввод/вывод будет работать через файлы `FILE* stdin, stdout`, а не через потоки.

```

1 %{
2 #include <iostream>
3 using namespace std;
4 string StringParseBuffer;

```

```

5 %}
6 %option main
7 %%
8 #.+\\n      { cout<<yytext; }
9 [ \\t\\r\\n]+ { } /* spaces */
10 .          { } /* undetected chars */
11 %%

```

2. запускать `flex -+ empty.lpp`, `flex++ empty.lpp` или с `%option c++` в `.lpp` файле: анализатор будет сгенерирован в файл `lex.yy.cc`, и требует от вас создания файла `FlexLexer.h`, содержащего определения пары служебных классов для лексера. Детали использования `flex++` рассмотрены в ??.

## 3.3 Регулярные выражения

**Регулярное выражение** [4], или **regex** — текстовая строка, используемая в качестве шаблона для проверки другой строки на совпадение, или поиска подстрок по шаблону.

Большинство букв и символов соответствуют сами себе. Например, регулярное выражение `test` будет в точности соответствовать строке `test`. Некоторые символы это специальные **метасимволы**, и сами себе не соответствуют:

[ ] используются для определения набора символов, в виде отдельных символов или диапазона, например `regex [0-9A-F]` задает одну цифру шестнадцатеричного числа; набор `[abcd]` можно заменить на диапазон `[a-d]`.

\ (обратная косая черта) используется для **экранирования** специальных символов, для представления как текстовых символов самих по себе

. (точка) обозначает любой символ, кроме конца строки

^ начало строки

\n конец строки

\t символ табуляции

( ) скобки используются для определения области действия

| вертикальная черта разделяет допустимые варианты, часто используется вместе со скобками: `gr(a|e)y`  
описывают строку `gray` или `grey`

**квантификатор** после символа, символьного класса или группы определяет, сколько раз предшествующее выражение может встречаться

{n} n раз

{n,m} от n до m раз

{n,} не менее n раз

{,m} не более m раз

? {0,1} необязательный элемент

\* {0,} 0+ раз

+ {1,} 1+ раз

## Жадная и ленивая квантификация

Квантификаторам в регулярных выражениях соответствует максимально длинная строка из возможных (квантификаторы являются **жадными** (greedy)).

Это может оказаться значительной проблемой. Например, часто ожидают, что выражение `(<.*>)` найдёт в тексте теги HTML. Однако, если в тексте есть более одного HTML-тега, то этому выражению соответствует целиком строка, содержащая множество тегов.

Эту проблему можно решить двумя способами:

1. Учитывать символы, не соответствующие желаемому образцу, через отрицание в наборе символов: `<[<^>]*>`.
2. Определить квантификатор как **ленивый** (lazy) — большинство реализаций обработчиков регулярных выражений позволяют это сделать, добавив после квантификатора знак вопроса: `<.*?>`

Использование ленивых квантификаторов может повлечь за собой обратную проблему, когда выражению соответствует слишком короткая, в частности, пустая строка.

## 3.4 Примеры самостоятельного применения

Лексер может быть использован как самостоятельный инструмент, если не требуется анализ синтаксиса, и достаточно выполнять заданный  $C^{++}$  код при срабатывании одного из регулярных выражений.

### 3.4.1 Pij2D: загрузка файла числовых данных

Формат файла:

- число строк матрицы  $\text{max}=\text{Rmax}$
- число элементов в строке  $\text{max}=\text{Xmax}$
- данные построчно

Fi.dat

1	20.0000												
2	90.0000												
3	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	...
4	1.0060	1.0060	1.0060	1.0060	1.0059	1.0059	1.0058	1.0058	1.0057	1.0056	1.0054	1.0053	...
5	1.0049	1.0049	1.0049	1.0048	1.0047	1.0046	1.0045	1.0043	1.0042	1.0040	1.0037	1.0034	...
6	...												
7	0.9936	0.9936	0.9935	0.9935	0.9934	0.9933	0.9932	0.9931	0.9929	0.9927	0.9925	0.9922	...
8	0.9932	0.9932	0.9932	0.9932	0.9931	0.9931	0.9930	0.9930	0.9929	0.9928	0.9927	0.9926	...
9	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	1.0000	

Pij2D.lpp

```
1 %{\n2 #include "hpp.hpp"\n3\n4 int  item=0;\n5 int  R=0,Rlimit;\n6 int  X=0,Xlimit;\n7\n8 double  Fi[Rmax][Xmax];\n9
```

```

10}%}
11%option noyywrap
12S [\+\-]?
13N [0-9]+
14%%
15{S}{N}(\.{N})?      {
16    item++;
17    if (item==1) { Rlimit=atoi(yytext); cout << "R limit:\t" << Rlimit <<"\n"; }
18    if (item==2) { Xlimit=atoi(yytext); cout << "X limit:\t" << Xlimit <<"\n";
19        X=R=0; assert(Rlimit<Rmax); assert(Xlimit<Xmax);
20        cout << "\nFi[] phield data:"; }
21    if (item>2) { Fi[R][X++] = atof(yytext); }
22    }
23
24[\r\n]+      { if (item>2) { X=0; R++; } }
25
26<<EOF>> {
27    for (int r=0;r<=Rlimit;r++) {
28        cout << "\n\n" << r <<": ";
29        for (int x=0;x<Xlimit;x++) {
30            cout << Fi[r][x] << " ";
31        }
32        yyterminate();
33    }
34
35    {}
36}%}

```

```
1 #ifndef _H_PIJ2D
2 #define _H_PIJ2D
3
4 #define TE "te.log"
5
6 #define Rmax 20+1
7 #define Xmax 90+1
8
9 // #include <mpi.h>
10
11 #include <iostream>
12 #include <fstream>
13 #include <iomanip>
14 #include <cmath>
15 #include <cstdio>
16 #include <cstdlib>
17 #include <cstdio>
18 #include <cassert>
19 using namespace std;
20
21 extern int doit();
22
23 extern int ylex();
24
25 extern double Fi[Rmax][Xmax];
26
27 #endif // _H_PIJ2D
```



```

1 int main (int argc, char *argv[]) {
2     // command line processing
3     assert (argc==4+1); // pij.exe <V> <Qm> <Alpha> <r>
4     V = atof(argv[1]); assert(V >0); cout << "V:\t\t\t" << V << "\n";
5     Qm = atof(argv[2]); assert(Qm >0); cout << "Qm:\t\t\t" << Qm << "\n";
6     Alpha = atof(argv[3]); assert(Alpha>0); cout << "Alpha:\t\t" << Alpha << "\n";
7     r = atof(argv[4])/1000; assert(r >0); cout << "r:\t\t\t" << r << "\n";
8     // Fi.txt fielf data parsing
9     while (yylex()); // Fi.txt parser loop from stdin
10    // compute tracks
11    return doit();
12}

```

Rmax // строк, не более чем

Xmax // столбцов, не более чем

double Fi[Rmax][Xmax] // массив под данные

int item // общий счетчик прочитанных чисел

argc, argv // часть исходных данных задается с командной строки

doit() // функция обработки данных

while (yylex()); // цикл опроса лексера,

yylex() // на каждый вызов возвращается один токен

item используется для определения, какой тип имеет текущее прочитанное число: Rlimit, Xlimit или данные.

Конец строки в обработке не участвует, факт перехода на следующую строку

## Компиляция

```
cd pij/pij2d && make
```

### Makefile

```
1
2 #MPICH2 = C:\MinGW\MPICH2
3
4 MODULE = pij2d
5
6 .PHONY: exec
7 exec: Fi.txt
8 Fi.txt: ./$(MODULE)$(EXE)
9     ./$(MODULE)$(EXE) 1000 1 0.1 2 < Fi.txt > $(MODULE).log
10
11 C = cpp.cpp lex.yy.c
12 H = hpp.hpp
13 CXXFLAGS += -I. -std=gnu++11
14 #-I$(MPICH2)/include -L$(MPICH2)/lib
15 ./$(MODULE)$(EXE): $(C) $(H)
16     $(CXX) $(CXXFLAGS) -o $@ $(C)
17 # -lmpi
18 lex.yy.c: lpp.lpp
19     flex $<
20
21 NOW = $(shell date +%Y%m%d%H%M)
22 .PHONY: rar
23 rar: $(NOW).rar
24 $(NOW).rar: test.files Fi.txt
```

```
rar a -m5 -o- -r $@ @test.files
```

```
verb|cd pij/pij2d|
```

## 3.5 Схема файлов для связки flex/bison

Сложные техники работы с текстовыми данными в этой книге будут далее рассматриваться на примере скриптового языка Ы. В этом разделе рассмотрен вариант лексера, работающего в связке с генератором синтаксических анализаторов **bison** ?? . Такая связка — типичная схема построения синтаксического анализатора, способного разбирать многоуровневые синтаксические конструкции.

Если вам требуется разбирать вложенные выражения, начиная от арифметических выражений с инфиксными операторами и скобками, типа  $(1+2*3)/\sin(x)$ , вам необходимо использовать связку **flex+bison** (**lex+yacc**)

в верхней части (до линии) перечислены определенные в файле объекты,  
после линии указаны используемые объекты из других файлов.

**урр.урр**

int	yyparse()	запуск парсера
union	yylval { int i; float f; std::string *s; sym*o; }	структура для одного синтаксического узла
void	yyerror(std::string)	функция вызывается парсером при возникновении синтаксической ошибки

## lpp.lpp

int	yylex()	функция лексера, выделяет <b>один</b> токен в <code>yylval</code> и возвращает код токена, определенный в <code>ypp.ypp</code> .
char*	yytext	указатель на текст токена, выделенный лексером
int	yylength	длина выделенного текста
int	yylineno	номер текущей строки, требует <code>%option yylineno</code> , используется в <code>yyperror()</code>

## hpp.hpp

class	sym	??	базовый виртуальный класс для символьных типов данных <u>скалярные типы данных:</u>
sym	Sym	??	символ
sym	Str	??	строка
sym	Int	??	целое число
sym	Num	??	число с плавающей точкой
			<u>функциональные типы данных:</u>
sym	Op	??	оператор
#define	TOC(C,X)	??	макрос, создающий объект класса C для токена X

## cpp.cpp

int	main(int argc, char *argv[ ])		
void	yyperror(std::string)	??	функция аварийного завершения по ошибке
	yyparse()		

## 3.6 Лексер для языка Ы: script/lpp.lpp

Так как используется модульная компиляция, в файле **hpp.hpp** вынесены объявления, которые нам нужно подключить:

lpp.lpp

```
1 %{\n2 #include "hpp.hpp"
```

Для компиляции и вывода номера строк в синтаксических ошибках нужно включить пару опций:

lpp.lpp

```
1 %option noyywrap\n2 %option yylineno
```

### Комментарий

lpp.lpp

```
1 #[^\n]*          {}          /* line comment */
```

### Строка

Для разбора строк будет использоваться специальное состояние лексера, и отдельный буфер разбора:

lpp.lpp

```
1 std::string StringParseBuffer;\n2 %}
```

```
3%x  stringstate
4%%
5 '.....{BEGIN(stringstate);_StringParseBuffer="";}...../*_string_*/
6<stringstate>' {BEGIN(INITIAL);
7                yylval.o = new Str(StringParseBuffer); return STR;}
8<stringstate>. {StringParseBuffer+=yytext;}
```

# Глава 4

## Синтаксис и реализация языка Ы

### 4.1 Комментарий

lpp.lpp

```
1 #[^\n]*      {}      /* line comment */
```

### 4.2 Скалярные типы

#### 4.2.1 Строка

Работа лексера по разбору строк описана в [3.6](#)

lpp.lpp

```
1 std::string  StringParseBuffer ;
2 %}
```



# Глава 5

## ANTLR

# Литература

## Основы компиляторов

- [1] [Книга Дракона \(Dragon Book\)](#):  
Альфред Ахо, Моника С. Лам, Рави Сети, Джеффри Ульман  
**Компиляторы: принципы, технологии и инструментарий**
- [2] Habrahabr: [Компиляция. 1: лексер](#)
- [3] Habrahabr: [Компиляция. 2: грамматики](#)
- [4] [Бкипедия](#) [Регулярные выражения](#)
- [5] [OpenNet.ru](#): [Генератор лексических анализаторов lex](#)
- [6] [Converting from old lex & yacc to flex & bison](#)

## LLVM

- [7] [LLVM tutorial](#)

## Java/ANTLR

- [8] [Грамматика арифметики или пишем калькулятор на ANTLR](#)

## Утилиты

- [9] Emden Gansner and Eleftherios Koutsofios and Stephen North  
[Drawing graphs with dot](#)

## Л<sup>A</sup>T<sub>E</sub>X: система верстки для научных публикаций

- [10] Википедия: [система верстки Л<sup>A</sup>T<sub>E</sub>X](#)

- [11] Котельников И. А., Чеботаев П. З.

**Л<sup>A</sup>T<sub>E</sub>X по-русски.**

— СПб.: «Корона-Век», 2011. — 496 с. — 2000 экз. — ISBN 978-5-7931-0878-2.

[12] Львовский С. М.

**Набор и верстка в системе LaTeX.**

— М.: МЦНМО, 2006. — С. 448. — ISBN 5-94057-091-7.

[13] Балдин Е. М.

**Компьютерная типография LaTeX.**

— «БХВ-Петербург», 2008. — 304 с. — 2000 экз. — ISBN 978-5-9775-0230-6. Книга доступна в электронном виде на сайте CTAN под лицензией CC-BY-SA.

[14] Столяров А. В.

**Сверстай диплом красиво: LaTeX за три дня.**

— Москва: МАКС Пресс, 2010. — 100 с. — 200 экз. — ISBN 978-5-317-03440-5.