

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>

30 декабря 2015 г.

Оглавление

Применение	3
Необходимое программное обеспечение	3
1 Структура компилятора	5
1.1 Термины	5
1.2 Структура типового компилятора	7
1.3 Архитектура LLVM	8
2 Типичная структура проекта	9
2.1 README.md	10
2.2 Makefile	10
2.3 bat.bat	12
2.4 rc.rc	13
3 Лексер и утилита flex	14
3.1 Структура файла описания лексера	15
3.2 Запуск flex	16
3.2.1 Запуск в варианте для старого lex	16

3.2.2	flex и генерация лексера на C++	18
3.3	Регулярные выражения	19
3.4	Примеры самостоятельного применения	21
3.4.1	Pij2D : загрузка файла числовых данных	22
3.5	Схема файлов для связки flex/bison	27
3.6	Лексер для языка Ы: script/lpp.lpp	29
4	Парсер, утилита bison	33
4.1	Парсер языка Ы	33
5	Синтаксис и реализация языка Ы	36
5.1	Комментарий	36
5.2	AST: абстрактный символьный тип	37
5.3	Скалярные типы	39
5.3.1	Символ <sym:symbol>	39
5.3.2	Строка <str:'строка'>	40
5.3.3	Числа	41
5.3.4	Int: целое число <int:1234>	41
5.3.5	Hex: машинное шестнадцатеричное <hex:0x12AF>	42
5.3.6	Bin: машинное двоичное <bin:0b1101>	43
5.3.7	Num: число с плавающей точкой <num:1.23> <num:-3e+5>	43
5.4	Композитные типы	44
5.4.1	List: список <[:]>	44
5.4.2	Vector: вектор <:>	44
5.4.3	Pair: пара <x:y>	44
5.5	Функциональные типы	44
5.5.1	Op: оператор <op:+>	44

5.5.2	Fn: встроенная функция <fn:sin>	44
5.5.3	Lambda: лямбда-функция <_:_>	44

6 ANTLR 45

Литература 46

Основы компиляторов	46
LLVM	47
Java/ANTLR	47
Утилиты	47
L ^A T _E X: система верстки для научных публикаций	47

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 <http://www.mingw.org/>

git-scm <https://git-scm.com/>

gvim <http://www.vim.org/download.php#pc>

clang <http://llvm.org>

llvm ???

пакет компилятора и утилит C++

GNU GCC toolchain (g++, flex, bison, make)

git-клиент

минималистичный редактор кода

с самой простой подсветкой синтаксиса

(на регулярках 3.3)

компилятор C/C++ на базе 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

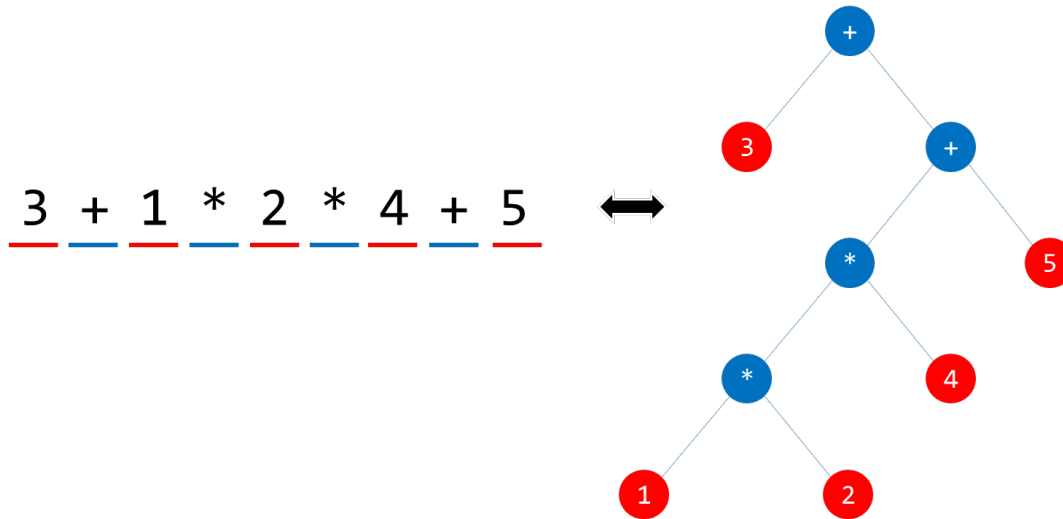
лексер **3** программный компонент, выполняющий выделение синтаксических элементов (токенов) из входного потока символов.

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

парсер **4** 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.

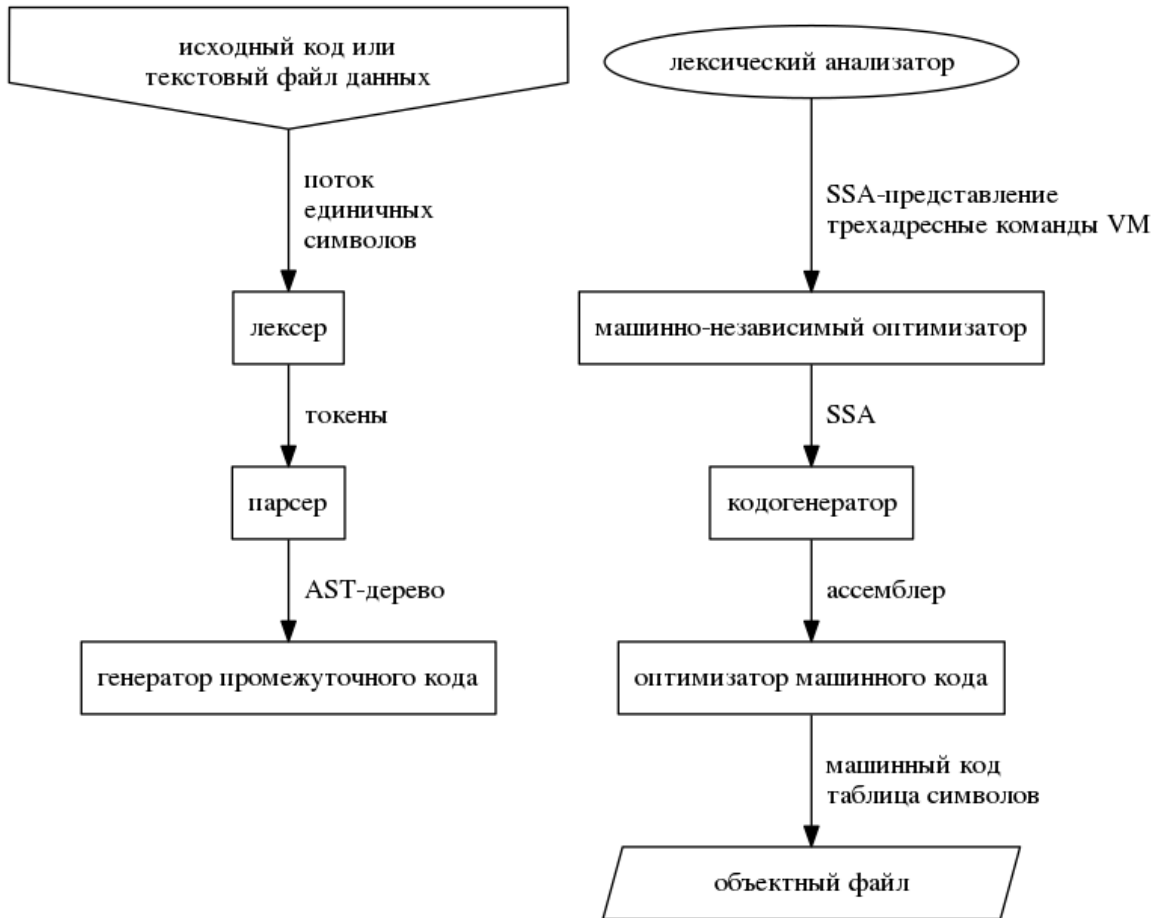
AST [A]bstract [S]yntax [T]ree, абстрактное синтаксическое дерево

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.



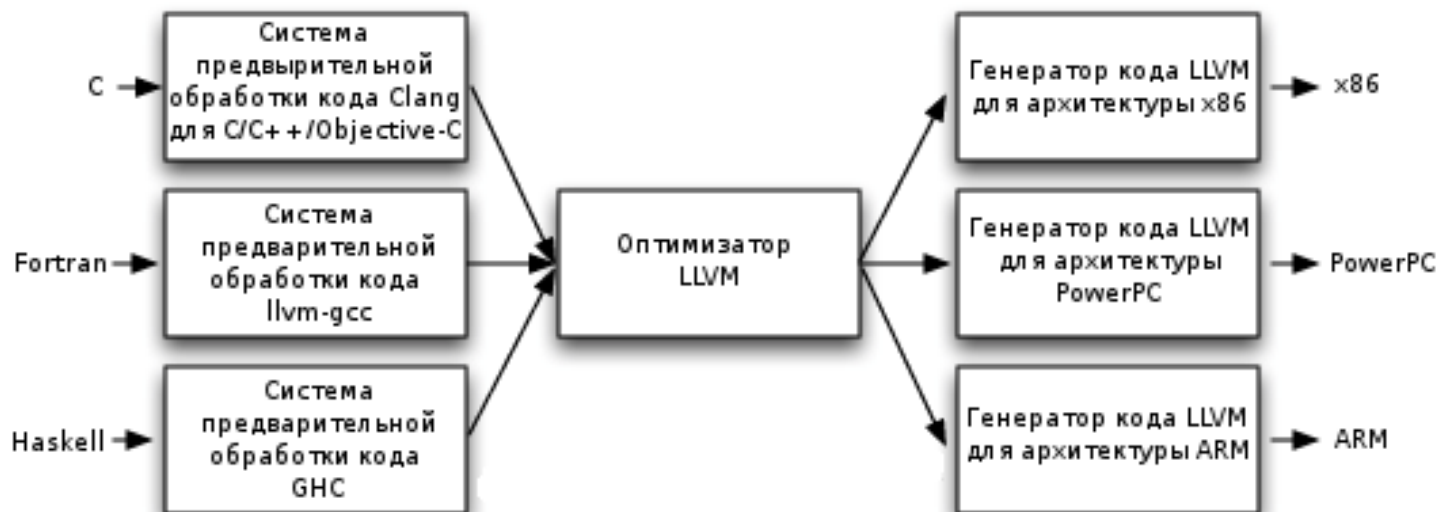
SSA ?? [S]ingle [S]tate [A]ssignment, однократное назначение: промежуточное представление, в котором каждой переменной значение присваивается лишь единожды. Переменные исходной программы разбиваются на версии, обычно с помощью добавления суффикса, таким образом, что каждое присваивание осуществляется 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 https://github.com/
Makefile	2.2	make	зависимости между файлами и команды сборки
lpp.lpp	??	flex	лексер 3
ypp.ypp	??	bison	парсер 4
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>
(c) <имя> <email>
<лицензия>
<ссылка на проект на GitHub>
### <ссылки, дополнительная информация>
```

README.md

```
1 # ![logo](logo.png) Mega script language
2
3 (c) 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: exec
2 exec: ./$(MODULE)$(EXE)
3      tail $(TAIL) $(MODULE).blog && echo
```

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 += -l. -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 script.bl script.blog ypp.ypp lpp.lpp hpp.hpp cpp.cpp
```

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](#)¹ 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
```

¹ 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 сишный** код лексера, который можно откомпилировать любым ANSI-совместимым компилятором Си для любого микроконтроллера, или отечественной ВПКшной поделки типа **KP1878BE1**.

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**² для команд препроцессора:

`empty.l`

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

² подробно рассмотрены далее **3.3**

empty.log

```
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 Си FILE* stdin, stdout, а не через потоки.

empty.lpp

```
1%{
2#include <iostream>
3using namespace std;
4string 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 %{
2 #include "hpp.hpp"
3
4 int item=0;
5 int R=0, Rlimit;
6 int X=0, Xlimit;
7
8 double Fi[Rmax][Xmax];
9
```

```

10 %}
11 %option noyywrap
12 S [\+\-]?
13 N [0-9]+
14 %%
15 {S}{N}(\.{N})?      {
16     item++;
17     if (item==1) { Rlimit=atoi(yytext); cout << "Rlimit:\t" << Rlimit << "\n"; }
18     if (item==2) { Xlimit=atoi(yytext); cout << "Xlimit:\t" << Xlimit << "\n";
19         X=R=0; assert(Rlimit<Rmax); assert(Xlimit<Xmax);
20         cout << "\nFi[]_uphi_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 yylex();
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\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 > $(MODULE).log
10
11 C = cpp.cpp lex.yy.c
12 H = hpp.hpp
13 CXXFLAGS += -l. -std=gnu++11
14 #-I$(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
```

3.5 Схема файлов для связки flex/bison

Сложные техники работы с текстовыми данными в этой книге будут далее рассматриваться на примере скриптового языка Bl. В этом разделе рассмотрен вариант лексера, работающего в связке с генератором синтаксических анализаторов **bison** 4. 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.

ypp.ypp

int	yyparse()	запуск парсера
union	yyval { 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 один токен в yylval и возвращает код токена, определенный в упр.урр.
char*	yytext	указатель на текст токена, выделенный лексером
int	yylength	длина выделенного текста
int	yylineno	номер текущей строки, требует %option yylineno, используется в yyerror()

hpp.hpp

class	sym	5.2	базовый виртуальный класс для символьных типов данных <u>скалярные типы данных:</u>
sym	Sym	5.3.1	символ
sym	Str	5.3.2	строка
sym	Int	5.3.4	целое число
sym	Num	5.3.7	число с плавающей точкой
			<u>функциональные типы данных:</u>
sym	Op	??	оператор
#define	TOC(C,X)	3.6	макрос, создающий объект класса C для токена X

cpp.cpp

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

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 %{  
2 #include "hpp.hpp"
```

Макрос `TOC` выполняет токенизацию символьного объекта [5.2](#), возвращая ссылку на объект и код токена в парсер:

hpp.hpp

```
1 #define TOC(C,X) { yyval.o = new C(yytext); return X; }
```

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

lpp.lpp

```
1 %option noyywrap  
2 %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;  
2 %}  
3 %x stringstate  
4 %%  
5 ' ' {BEGIN(stringstate); StringParseBuffer="";} /* string */  
6 <stringstate>' {BEGIN(INITIAL);  
7           yyval.o = new Str(StringParseBuffer); return STR;}  
8 <stringstate>. {StringParseBuffer+=yytext;}
```

В области определений задана строка — буфер для накопления символов строки при ее разборе.

Через %x создано **состояние лексера**, в области правил для этого состояния через <состояние> задаются специальные **правила, срабатывающие только для этого состояния**.

Состояния лексера переключаются макросом BEGIN(), состояние по умолчанию — INITIAL.

' переключает состояние лексера, и обнуляет накопительный буфер

<stringstate>' выключает состояние stringstate, и реализует работу макроса TOC(C,X) особым образом, создавая объект Str из содержимого буфера, а не строки *yytext

<stringstate>. добавляет в буфер (.)=любой символ

Числа

lpp.lpp

```
1 S [\+ \-]?
2 N [0-9]+
3 %%
4
5 {S}{N}\.{N}          TOC(Num,NUM)      /* numbers */
6 {S}{N}[eE]{S}{N}     TOC(Num,NUM)      /* floating point */
7 {S}{N}               TOC(Num,NUM)      /* exponential */
8 0x[0-9A-F]+          TOC(Int,NUM)       /* integer */
9 0b[01]+              TOC(Hex,NUM)       /* hex */
                      TOC(Bin,NUM)       /* binary */
```

При распознавании чисел используется подстановка regex-переменных {S} (знак числа) и {N} (цифры), заданных в области определений.

Символ

Все нераспознанные блоки текста, состоящие из латинских букв и цифр, распознаются как **символ 5.2**:

lpp.lpp

```
1 [a-zA-Z0-9_]+      TOC(Sym,SYM)      /* symbol */
```

Операторы

lpp.lpp

```
1 \ (                TOC(Op,LP)
2 \ )                TOC(Op,RP)
3 \ [                TOC(Op,LQ)
```


4	\]	TOC(Op,RQ)
5	\{	TOC(Op,LB)
6	\}	TOC(Op,RB)
7	\<	TOC(Op,LL)
8	\>	TOC(Op,GG)
9	\<\<	TOC(Op,LLLL)
10	\>\>	TOC(Op,GGGG)
11	\+\<=	TOC(Op,ADDEQ)
12	\=	TOC(Op,EQ)
13	\:	TOC(Op,COLON)
14	\@	TOC(Op,AT)
15	\.	TOC(Op,DOT)
16	\,	TOC(Op,COMMA)
17	\+	TOC(Op,ADD)
18	\-	TOC(Op,SUB)
19	*	TOC(Op,MUL)
20	\/	TOC(Op,DIV)
21	\^	TOC(Op,POW)

Глава 4

Парсер, утилита **bison**

Для разбора вложенных синтаксических конструкций, от арифметических выражений до исходных текстов программ, необходимо использование синтаксического парсера.

4.1 Парсер языка Ы

ypp.ypp

```
1 %{  
2 #include "hpp.hpp"  
3 %}  
4 %defines %union { sym*o; }  
5 %token <o> SYM NUM STR          /* symbol number string */  
6 %token <o> EQ ADD SUB MUL DIV POW /* = + - * / ^ */  
7 %token <o> ADDEQ                 /* += */
```

```

8%token <o> AT DOT COMMA COLON      /* @ . , : */
9%token <o> LP RP LQ RQ LB RB LL GG  /* ( ) [ ] { } < > */
10%token <o> LLLL GGGG               /* << >> */
11%type <o> ex list vector           /* expression [list] <vector> pair */
12%type <o> lambda                   /* {lambda} */
13
14%left EQ ADDEQ
15%left ADD SUB
16%left MUL DIV
17%left POW
18%left COLON
19%left LLLL GGGG
20%left AT
21%left LP RP
22%%
23 syntax: | syntax ex      { W($2); W("\n_____");
24                          W($2->eval()); W("\n====="); } ;
25 ex: SYM | NUM | STR
26     | LQ list RQ         { $$=$2; }
27     | LL vector GG       { $$=$2; }
28     | LB lambda RB       { $$=$2; }
29     | ex ADD ex          { $$=$2; $$->push($1); $$->push($3); }
30     | ex SUB ex          { $$=$2; $$->push($1); $$->push($3); }
31     | ex MUL ex          { $$=$2; $$->push($1); $$->push($3); }
32     | ex DIV ex          { $$=$2; $$->push($1); $$->push($3); }
33     | ex POW ex          { $$=$2; $$->push($1); $$->push($3); }
34     | ex EQ ex           { $$=$2; $$->push($1); $$->push($3); }
35     | ex ADDEQ ex        { $$=$2; $$->push($1); $$->push($3); }

```

```

36      | ex GGGG ex      { $$=$2; $$->push($1); $$->push($3); }
37      | ex COLON ex    { $$=new Pair($1,$3); }
38      | ex AT ex       { $$=$2; $$->push($1); $$->push($3); }
39      | ex LP list RP  { $$=new Op("@"); $$->push($1); $$->push($3); }
40 ;
41 list:
42     | list ex          { $$=$1; $$->push($2); }
43 ;
44 vector:
45     | vector ex        { $$=$1; $$->push($2); }
46 ;
47 lambda:
48     | lambda SYM COLON { $$=$1; $$->setpar($2); }
49     | lambda ex        { $$=$1; $$->push($2); }
50 ;
51 %%

```

Глава 5

Синтаксис и реализация языка Ы

5.1 Комментарий

script.bl

```
1 # comment
2 # \ outer list
3 # < integers , hex : binary > vector , pa:ir
4 # [ float numbers ] nested list
5 # lambdas
```

Комментарии вырезаются лексером:

lpp.lpp

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

5.2 AST: абстрактный символьный тип

Язык Ы построен на символьных вычислениях: интерпретации структур данных, состоящих из элементов **AST**¹. Идеология "программа есть данные" была взята из Lisp, и дополнена Python-подобным синтаксисом и динамической объектной системой а-ля SmallTalk.

Применение в основе иерархии объектов языка Ы **виртуального** базового класса позволяет работать с объектами классов-наследников, используя указатели `sym*` на базовый класс.

```
                                hpp.hpp
1 struct sym {                    // == abstract symbolic data type ==
2     std::string tag;             // type/class tag
3     std::string val;             // object value in string form
4     sym(std::string, std::string); // T:V constructor
5     // ----- nested objects
6     std::vector<sym*> nest;
7     void push(sym*);             // push to nest[] as stack
8     sym* pop();                  // pop from nest[] as stack
9     // ----- parameters
10    std::map<std::string, sym*> par;
11    void setpar(sym*);            // set parameter
12    // ----- object evaluation
13    virtual sym* eval();
14    // ----- operators
15    virtual sym* at(sym*);         // A @ B = A.at(B)
```

¹ [A]bstract [S]yntax [T]ree

```

16 virtual sym* eq(sym*);           // A = B
17 virtual sym* add(sym*);          // A + B
18 virtual sym* div(sym*);          // A/B
19 virtual sym* addeq(sym*);         // A += B
20 // ----- textual object dump
21 std::string dump(int depth=0);    // dump object in tree form
22 protected:
23     std::string pad(int n);        // pad dump with TABs
24     virtual std::string tagval();  // return "<tag:val>"
25 }

```

tag	тэг, тип/класс объекта
val	символьное значение объекта
sym(string T,string V)	создает символьный объект из строк для тэга и начального значения
nest[]	каждый AST-объект может содержать вложенные объекты
push(sym*)	втолкнуть объект в nest[] как в стек
sym* pop()	взять объект из стека
par[]	словарь именованных параметров
setpar(sym*)	добавить параметр
dump()	вывод дампа объекта, включая вложенные, в древовидной форме
pad(n)	выравнивание пробелами слева при работе dump()
tagval()	возвращает минимальное текстовое представление объекта в форме <tag:val>
eval()	метод символьного вычисления объекта, включая вложенные
virtual sym* op(sym*)	методы, реализующие операторы

AST-объект поддерживает хранение вложенных элементов, что позволяет легко реализовать хранение древовидных структур, характерных для программ символьных вычислений и трансляторов.

Функции push/pop используются при определении пользовательских анонимных и именованных функций ?? в стиле языка Forth.

hpp.hpp

```
1 #define TOC(C,X) { yyval.o = new C(yytext); return X; }
```

cpp.cpp

```
1 sym::sym(std::string T, std::string V) { tag=T; val=V; } // create AST object
2 void sym::push(sym*o) { nest.push_back(o); }
3 void sym::setpar(sym*o) { par[o->val]=o; }
4 std::string sym::pad(int n) { std::string S; for (int i=0; i<n; i++) S+="\t"; return S; }
5 std::string sym::tagval() { return "<" + tag + ":" + val + ">"; } // header <tag:val>
6 std::string sym::dump(int depth) { // dump as text tree
7 sym* sym::eval() { // eval/compute
8 sym* sym::at(sym*o) { assert(nest.size()==1); // default: move to nest[]ed
9 sym* sym::eq(sym*o) { env[val]=o; // env[A]=B
10 sym* sym::add(sym*o) { push(o); return this; } // A+B
11 sym* sym::div(sym*o) { push(o); return this; } // A/B
12 sym* sym::addeq(sym*o) { return this->eq(this->add(o)); }
13 sym* Op::eval() { sym::eval(); }
```

5.3 Скалярные типы

5.3.1 Символ <sym:symbol>

lpp.lpp


```
1 [a-zA-Z0-9_]+      TOC(Sym,SYM)      /* symbol */
```

hpp.hpp

```
1 struct Sym:sym { Sym(std::string); };
```

cpp.cpp

```
1 Sym::Sym(std::string V):sym("sym",V) {}
```

5.3.2 Строка <str:'строка'>

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

lpp.lpp

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

ypp.ypp

```
1 %token <o> SYM NUM STR      /* symbol number string */  
2 %type <o> ex list vector    /* expression [list] <vector> pair */  
3 ex: SYM | NUM | STR  
4     | lambda SYM COLON { $$=$1; $$->setpar($2); }
```

hpp.hpp

```
1 struct Str:sym { Str(std::string); std::string tagval(); };
```

Дамп строки выводится с кавычками:

cpp.cpp

```
1 Str::Str(std::string V):sym("str",V) {}
2 std::string Str::tagval() { return "<"+tag+":'"+val+"'>"; }
```

5.3.3 Числа

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

ypp.ypp

```
1 %token <o> SYM NUM STR /* symbol number string */
2 %type <o> ex list vector /* expression [list] <vector> pair */
3 ex: SYM | NUM | STR
4 | lambda SYM COLON { $$=$1; $$->setpar($2); }
```

5.3.4 Int: целое число <int:1234>

script.bl

```
1 -01 00 +02
```

lpp.lpp

```
1 {S}{N} TOC(Int,NUM) /* integer */
```

hpp.hpp

```
1 struct Int:sym { Int(std::string); std::string tagval(); long i;  
2     Int(long); };
```

cpp.cpp

```
1 Int::Int(std::string V):sym("int","") { i = atoi(V.c_str()); }  
2 Int::Int(long V):sym("int","") { i = V; }  
3 std::string Int::tagval() {
```

5.3.5 Hex: машинное шестнадцатеричное <hex:0x12AF>

script.bl

```
1 0x12AF
```

lpp.lpp

```
1 0x[0–9A–F]+ TOC(Hex,NUM) /* hex */
```

hpp.hpp

```
1 struct Hex:sym { Hex(std::string); };
```

cpp.cpp

```
1 Hex::Hex(std::string V):sym("hex",V) {}
```

5.3.6 Bin: машинное двоичное <bin:0b1101>

script.bl

```
1 0b1101
```

lpp.lpp

```
1 0b[01]+          TOC( Bin ,NUM)      /* binary */
```

hpp.hpp

```
1 struct Bin:sym { Bin(std::string); };
```

cpp.cpp

```
1 Bin::Bin(std::string V):sym("bin",V) {}
```

5.3.7 Num: число с плавающей точкой <num:1.23> <num:-3e+5>

script.bl

```
1 -01.20 -03e+04
```

lpp.lpp

```
1 {S}{N}\.{N}      TOC(Num,NUM)      /* floating point */
2 {S}{N}[eE]{S}{N} TOC(Num,NUM)      /* exponential */
```

hpp.hpp

```
1 struct Num:sym { Num(std::string); std::string tagval(); double f; };
```

cpp.cpp

```
1 Num::Num(std::string V):sym("num","") { f = atof(V.c_str()); }  
2 std::string Num::tagval() {
```

5.4 Композитные типы

5.4.1 List: список $\langle[:]\rangle$

5.4.2 Vector: вектор $\langle{:}\rangle$

5.4.3 Pair: пара $\langle x:y\rangle$

5.5 Функциональные типы

5.5.1 Op: оператор $\langle op:+\rangle$

5.5.2 Fn: встроенная функция $\langle fn:\sin\rangle$

5.5.3 Lambda: лямбда-функция $\langle _:_ \rangle$

Глава 6

ANTLR

Литература

Основы компиляторов

- [1] [Книга Дракона \(Dragon Book\)](#):
Альфред Ахо, Моника С. Лам, Рави Сети, Джефффри Ульман
Компиляторы: принципы, технологии и инструментарий
- [2] Habrahabr: [Компиляция. 1: лексер](#)
- [3] Habrahabr: [Компиляция. 2: грамматики](#)
- [4] Википедия [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^AT_EX: система верстки для научных публикаций

- [10] Википедия: [система верстки Л^AT_EX](#)

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

Л^AT_EXпо-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.