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>

28 декабря 2015 г.

Оглавление

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/	пакет компилятора и утилит C^{++} GNU GCC toolchain (g++, flex, bison, make)
git-scm	https://git-scm.com/	git-клиент
gvim	http://www.vim.org/download.php#pc	минималистичный редактор кода с самой простой подсветкой синтаксиса (на регулярках 3.3)
clang	http://llvm.org	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

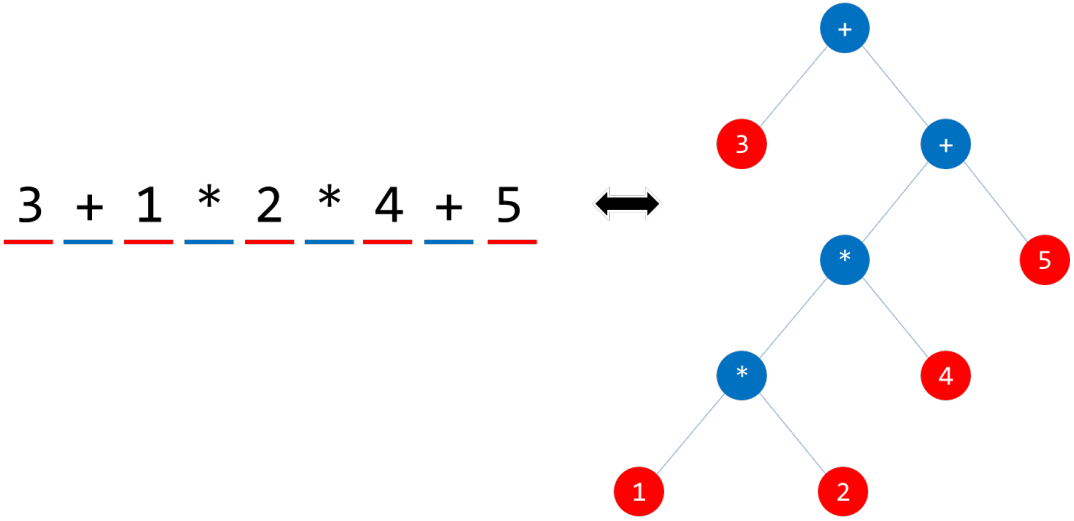
лексер 3 программный компонент, выполняющий выделение синтаксических элементов (токенов) из входного потока символов.

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.

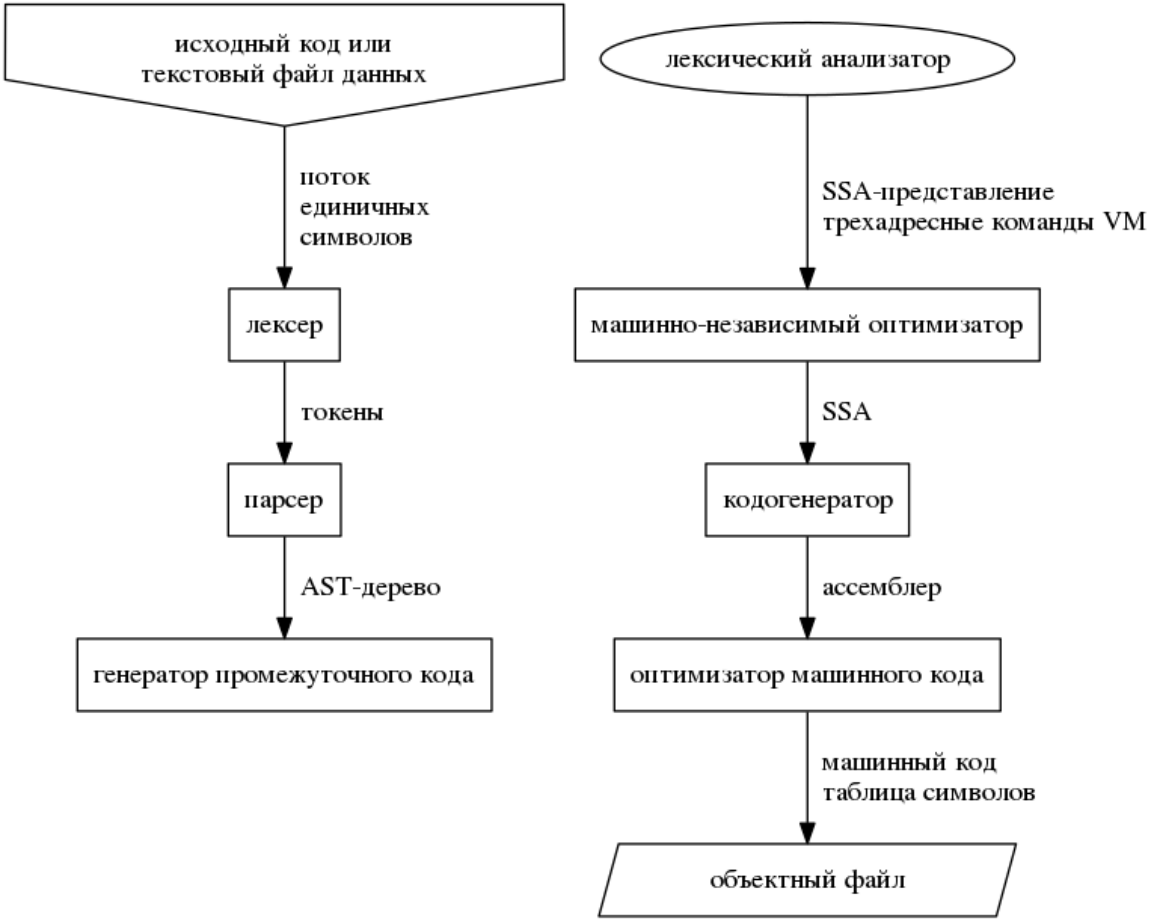
AST [A]bstract [S]yntax [T]ree, абстрактное синтаксическое дерево

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

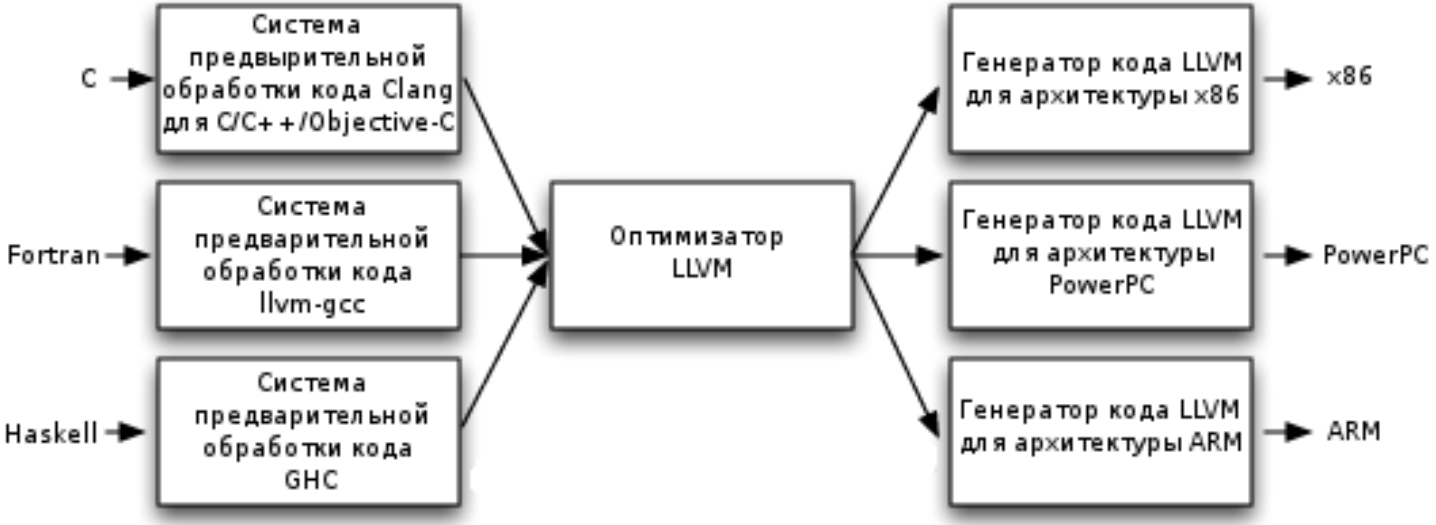


SSA ?? [S]ingle [S]tate [A]ssignment, однократное назначение: промежуточное представление, в котором каждой переменной значение присваивается лишь единожды. Переменные исходной программы разбиваются на версии, обычно с помощью добавления суффикса, таким образом, что каждое присваивание осуществляется 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 ??
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

```
# ![logo](logo.png) Mega script language

(c) Vasya Pupkin <pupkin@gmail.com>, all rights reserved

license: http://www.gnu.org/copyleft/lesser.html

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

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

Модуль заполняется автоматически по имени текущего каталога:

Makefile

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

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

Makefile

```
.PHONY: exec
exec: ./$(MODULE)$(EXE)
	./$(MODULE)$(EXE) < $(MODULE).bl > $(MODULE).blog && tail $(TAIL) $(MODULE).blog
```

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

Makefile

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

Сборка C^{++} части

Makefile

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

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

Генерация кода парсера

Makefile

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

Генерация кода лексера

Makefile

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

Компиляция файла ресурсов (win32)

Makefile

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

2.3 bat.bat

bat.bat

```
@start .
@gvim -c "colorscheme darkblue" -p lexman.scr lexman.log \
    ypp.ypp lpp.lpp hpp.hpp cpp.cpp \
    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/комипилятора, 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

```
%option main
%%
%%
```

```
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

```
#line 3 "lex.yy.c"
#define  YY_INT_ALIGNED short int
#define FLEX_SCANNER
#define YY_FLEX_MAJOR_VERSION 2
#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 пишите 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

```
%{
#include <iostream>
using namespace std;
string StringParseBuffer;
```

```
%}
%option main
%%
#.+\n        { cout<<yytext; }
[ \t\r\n]+   {}  /* spaces */
.            {}  /* undetected chars */
%%
```

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 regexp — 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 regexp `[0-9A-F]` задает одну цифру шестнадцатеричного числа; набор `[abcd]` можно заменить на диапазон `[a-d]`.

\ (обратная косая черта) используется для экранирования специальных симолов, для представления как текстовых симолов 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 max=Rmax
- число элементов в строке max=Xmax
- данные построчно

Fi.dat

```
20.0000
90.0000
1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  1.0000  ...
1.0060  1.0060  1.0060  1.0060  1.0059  1.0059  1.0058  1.0058  1.0057  1.0056  1.0054  1.0053  ...
1.0049  1.0049  1.0049  1.0048  1.0047  1.0046  1.0045  1.0043  1.0042  1.0040  1.0037  1.0034  ...
...
0.9936  0.9936  0.9935  0.9935  0.9934  0.9933  0.9932  0.9931  0.9929  0.9927  0.9925  0.9922  ...
0.9932  0.9932  0.9932  0.9932  0.9931  0.9931  0.9930  0.9930  0.9929  0.9928  0.9927  0.9926  ...
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

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

int item=0;
int R=0,Rlimit;
int X=0,Xlimit;

double Fi[Rmax][Xmax];

```

```
%}
%option noyywrap
S [\+\-]?
N [0-9]+
%%
{S}{N}(\.{N})?       {
    item++;
    if (item==1) { Rlimit=atoi(yytext); cout << "R limit:\t" << Rlimit <<"\n"; }
    if (item==2) { Xlimit=atoi(yytext); cout << "X limit:\t" << Xlimit <<"\n";
        X=R=0; assert(Rlimit<Rmax); assert(Xlimit<Xmax);
    cout << "\nFi[] phield data:"; }
    if (item>2) { Fi[R][X++] = atof(yytext); }
    }

[\r\n]+      { if (item>2) { X=0; R++; } }

<<EOF>> {
    for (int r=0;r<=Rlimit;r++) {
        cout << "\n\n" << r <<": ";
        for (int x=0;x<Xlimit;x++) {
            cout << Fi[r][x] << " ";
    }}
    yyterminate();
}

.    {}
%%
```

hpp.hpp

```
#ifndef _H_PIJ2D
#define _H_PIJ2D

#define TE "te.log"

#define Rmax 20+1
#define Xmax 90+1

//#include <mpi.h>

#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include <cfloat>
#include <cstdlib>
#include <cstdio>
#include <cassert>
using namespace std;

extern int doit();

extern int yylex();

extern double Fi[Rmax][Xmax];

#endif // _H_PIJ2D
```

cpp.cpp

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

```
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
```

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

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.

ypp.ypp

int	yyparse()	запуск парсера
union	yylval { int i; float f; std::string *s; sym*o; }	структура для одного синтаксическогоо 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 ypp.ypp.
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	4.2	базовый виртуальный класс для символьных типов данных
			скалярые типы данных:
sym	Sym	4.3.1	символ
sym	Str	4.3.2	строка
sym	Int	4.3.4	целое число
sym	Num	4.3.7	число с плавающей точкой
			функциональные типы данных:
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

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

Макрос `TOC` выполняет токенизацию символьного объекта 4.2, возвращая ссылку на объект и код токена в парсер:

hpp.hpp

```
1 #define TOC(C,X) { yylval.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                 yylval.o = new Str(StringParseBuffer); return STR;}
8 <stringstate>.  {StringParseBuffer+=yytext;}
```

В области определений задана строка — буфер для накомпления 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                                       /* numbers */
5 {S}{N}\.{N}           TOC(Num,NUM)    /* floating point */
6 {S}{N}[eE]{S}{N}      TOC(Num,NUM)    /* exponential */
7 {S}{N}                TOC(Int,NUM)    /* integer */
8 0x[0-9A-F]+           TOC(Hex,NUM)    /* hex */
9 0b[01]+               TOC(Bin,NUM)    /* binary */
```

При распозанавании чисел используется подставновка regexp-переменных {S} (знак числа) и {N} (цифры), заданных в области определений.

Символ

Все нераспознанные блоки текста, состоящие из литинских букв и цифр, распознаются как символ 4.2:

lpp.lpp

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

Глава 4

Синтаксис и реализация языка Ы

4.1 Комментарий

script.bl

```
# comment
# \ outer list
# < integers, hex : binary > vector, pair
# [ float numbers ] nested list
#{Y:Y} {X:Y: X+Y}   # lambdas
```

Комментарии вырезаются лексером:

lpp.lpp

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

4.2 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 Lispa, 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 constuctor
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     // ------------------------------------ textual object dump
13     std::string dump(int depth=0);        // dump object in tree form
14 protected:
15     std::string pad(int n);               // pad dump with TABs
```

[1] [A]bstract [S]yntax [T]ree

```
16      virtual std::string tagval();        // return "<tag:val>"
17  }
```

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 next[] как в стек
sym* pop()	взять объект из стека
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>

AST-объект поддерживает хранение вложенных элементов, что позволяет легко реализовать хранение древовидиных 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) { yylval.o = new C(yytext); return X; }
```

4.3 Скалярные типы

4.3.1 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)     {}
```

4.3.2 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 pair            /* expression [list] <vector> pa:ir */
3 ex: SYM | NUM | STR
4     | lambda SYM COLON  { $$=$1; $$->setpar($3); }
```

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+"'>";}
```

4.3.3 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 pair                  /* expression [list] <vector> pa:ir */
3 ex: SYM | NUM | STR
4     | lambda SYM COLON  { $$=$1; $$->setpar($3); }
```

4.3.4 Int: целое число <int:1234>

script.bl

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

lpp.lpp

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

hpp.hpp

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

cpp.cpp

```
Int::Int(std::string V):sym("int","")   { i = atoi(V.c_str()); }
std::string Int::tagval() {
    std::ostringstream os; os<<"<"<<tag<<":"<< i <<">"; return os.str(); }
```

4.3.5 Hex: машинное шестнадцатеричное <hex:0x12AF>

script.bl

```
0x12AF
```

lpp.lpp

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

hpp.hpp

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

cpp.cpp

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

4.3.6 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)     {}
```

4.3.7 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() {
3     std::ostringstream os; os<<"<"<<tag<<":"<< f <<">"; return os.str(); }
```

4.4 Композитные типы

4.4.1 List: список <[:]>

4.4.2 Vector: вектор <:>

4.4.3 Pair: пара <x:y>

4.5 Функциональные типы

4.5.1 Op: оператор <op:+>

4.5.2 Fn: встроенная функция <fn:sin>

4.5.3 Lambda: лямбда-функция <^:^>

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] Ыкипедия 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

LaTeX: система верстки для научных публикаций

[10] Википедия: система верстки LaTeX

[11] Котельников И. А., Чеботаев П. З.
LaTeX 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 электроном 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.