HOWTO: как написать интерпретатор* (программирование в свободном синтаксисе)

Дмитрий Понятóв <dponyatov@gmail.com>

21 февраля 2016 г.

Оглавление

Введение*

Программа, выполняющая преобразование *исходного текста* программы:

транслятор — в другой язык программирования

компилятор — в машинный код, сохраняемый в *объектные файлы*

интерпретатор — вариант **компилятора**, выполняющий **генерацию программы в ОЗУ** в виде машинного кода, или *байт-кода*[1]. Практически все современные интерпретаторы используют технику *JIT-компиляции* — генерация машинного кода в ОЗУ из исходного текста[2] или *промежуточного представления* программы в виде байт-кода.

Интерпретатором например является Java[3]: байт-код из .class и .jar файлов преобразуется в машинный код после запуска программы (в рантайме). Одновременно Java можно называть и компилятором, так как он компилирует программы в байт-код, эквивалентный машинному коду виртуальной Java-машины: стадии компиляции и выполнения разделены.

скрипт-движок — интерпретатор, встраиваемый в другие программы, и используемый для чтения файлов конфигурации и написания макросов. В комплект поставки обычно включается хост-среда, позволяющая запускать интерпретатор скрипт-языка в пакетном режиме[4], в составе веб-сервера, или интерактивной среды[5]

* в этом HOWTO описана реализация скриптового языка с алголоподобным синтаксисом.

---

[1] машинный код виртуальной машины, выполняемый программно, т.е. **интерпретируемый**

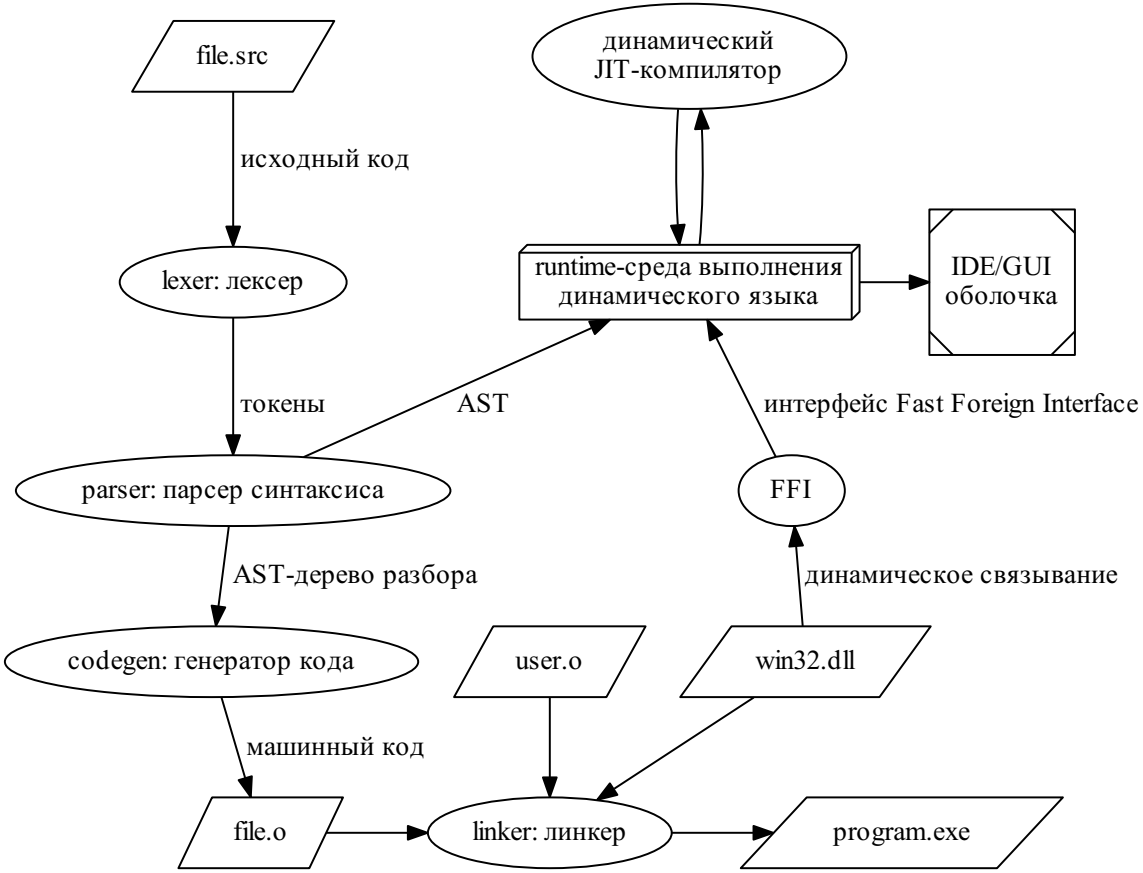
[2] точнее AST-дерева, полученного *разбором* исходника

[3] и Android без кода NDK

[4] как обычную консольную программу

[5] IDE разработчика, или полноценной графической пользовательской среды, как языки SmallTalk-группы

Структура компилятора



FSP: программирование в свободном синтаксисе

FSP[6] — метод *программирования в свободном синсаксисе*: центральную роль в вашей программе занимает *интерпретатор входного языка. Свободный синтаксис* входного языка вы можете произвольно менять, описывая на нем логику программы, файлы конфигурации, GUI и пользовательский командный интерфейс (CLI).

Скелет лексической программы

Создадим Скелет программы использующей лексический разбор → лексической программы → lexical skeleton → skelex:

```
mkdir script
cd script
touch src.src log.log ypp.ypp lpp.lpp hpp.hpp cpp.cpp Makefile bat.bat .gitignore
echo gvim -p src.src log.log ypp...cpp.cpp Makefile bat.bat .gitignore >> bat.bat
bat
```

Отредактируйте файлы[7]:

bat.bat : gvim win32 helper

```
1 @start .
2 @gvim -p src.src log.log ypp.ypp lpp.lpp hpp.hpp cpp.cpp Makefile bat.bat .gitignore
```

Если вы пользуетесь Git, создайте на GitHub репозиторий, и подключите его к проекту[8]:

[6] [F]ree [S]yntax [P]rogramming

[7] если вы пользуетесь Vimom под Windows

[8] у вас уже должны быть настроены ключи SSH-доступа для вашего аккаунта на GitHub

```
git init
git remote add gh git@github.com:yourname/script.git
git commit -a -m "+skelex"
git push -u gh master
```

.gitignore

```
1 *~
2 *.swp
3 exe.exe
4 log.log
5 ypp.tab.?pp
6 lex.yy.c
```

src.src	исходный код
log.log	лог транслятора
ypp.ypp	лексер /flex/
lpp.lpp	парсер /bison/
hpp.hpp	хедеры
cpp.cpp	C++ ядро интерпретатора
Makefile	
bat.bat	gvim win32 helper
.gitignore	

Makefile

```
1 .PHONY: log.log
2 log.log: ./exe.exe src.src
3 	./exe.exe < src.src > $@ && tail $(TAIL) $@
4 C = cpp.cpp ypp.tab.cpp lex.yy.c
```

```
5 H = hpp.hpp ypp.tab.hpp
6 CXXFLAGS = -std=gnu++11 -DMODULE=\"$(notdir $(CURDIR))\"
7 ./exe.exe: $(C) $(H)
8 ____$(CXX) $(CXXFLAGS) -o $@ $(C)
9 ypp.tab.cpp: ypp.ypp
10 ____bison $<
11 lex.yy.c: lpp.lpp
12 ____flex $<
```

Содержимое остальных файлов, описанное далее, легко скопипастить с

`https://github.com/ponyatov/script.git`.

В качестве примера рассмотрим описание скелета FSP-программы на нашем собственном языке:

skeleton.src

```
1
2 skelex = [
3     files = [ src.src log.log
4         core = [ ypp.ypp lpp.lpp hpp.hpp cpp.cpp ]
5     ]
6     Makefile bat.bat
7     .gitignore = '*~\n*.swp\nexe.exe\nlog.log\nypp.tab.?pp\nlex.yy.c'
8 ]
9
10 bat.bat = [
11     '@start .'
12     '@gvim -p '+files/'\s'
13 ]
```

Синтаксис языка bI

```
                                    # строчный комментарий

                                    # скалярные типы:
Sym.bol_01                          # символ
'строка
\tможет быть\смногострочной'        # \t табуляция \n конец строки \s пробел

                                    # композитные типы:
[ flat list ]                       # плоский список
lisp,like,cons,list                 # список на базе cons-ячеек в стиле Lisp

                                    # выражения в инфиксном синтаксисе
A+B-C*D/E^F                         # арифметические операторы
sin@X+cos@Y/exp@Z                   # применение встроенных функций
Z = sqrt@(X^2+Y^2)                  # биндинг выражений (назначение, уравнение)

pp = {P: P+'.pp'}                   # лямбда-функции, создание именованной функции
x.x = {X: pp@X+'.'+pp@X}
files = x.x@y + x.x|y,l,h,c         # одиночное и map- применение функции

class:object                        # объектно-ориентированное программирование
class:lexeme                        # одиночное наследование без модификации
object,lexeme:comment               # множественное наследование
    % regexp = '#[^\n]*'            # поле данных
    % lex = {lexrule@regexp,'{}'}   # метод (вызов функции-генератора правила лексера)
comment@apply <comment:bracks>      # инстанциация объекта через @ и <:>
```

Глава 1

Синтаксический анализатор

1.1 Лексер /flex/

Лексер — компонент, выполняющий разделение исходного текста программы на элементы: *токены*.

Лексер умеет работать только последовательно, и не способен разпознавать такие элементы как вложенные скобки. Задача лексера — объединить определенные смежные символы в один объект-токен, и присвоить ему метку типа (число, строка, оператор,..). Также лексер может запоминать в токене его положение в исходнике: имя файла, номер строки и столбца.

Лексер можно написать полностью самому, реализовав интерфейс:

hpp.hpp интерфейс лексера

```
1 extern int yylex();
2 extern int yylineno;
3 extern char* yytext;
4 #define TOC(C,X) { yylval.o = new C(yytext); return X; }
```

Макрос `TOC(C,X)` описывает внутреннюю логику функции `yylex()`: набирать символы из входного потока в буфер, создать объект-токен, вернув указатель через `yylval.o`(bject), и вернуть целочисленный код токена.

В большинстве случаев[1] удобнее воспользоваться типовой программой-генератором лексеров **flex**.

Структура файла правил лексера **lpp.lpp**:

```
%{
заголовок (C++ код)
%}
%option noyywrap
%option yylineno
%x state1 state2..
%%
правила
%%
необязательный C++ код
```

Если в файл лексера добавить опцию `%option main`, полученный код лексера можно скомпилировать как самостоятельную программу, и использовать ее как инструмент интеллектуального поиска/замены текста, как пример см. файл `doc/src2lst.lex`

lpp.lpp заголовок лексера

```
1 %{
2 #include "hpp.hpp"
```

[1] когда набор правил лескера не меняется в процессе работы программы

```
3 string LexString;
4 %}
```

Правила лексера задаются парами: `<регулярное выражение> <C++ код>`

Если набор символов совпадает с правилом, заданным регулярным выражением, срабатывает указанный сишный код:

lpp.lpp комментарий

```
1 #[^\n]*          {}
```

игнорировать любой текст от символа # до конца строки

Самый сложный набор правил — разбор строки, т.к. он требует отдельный набор правил. Переключение набора правил лексера выполняется через **состояния**. Стартовое (и единственное) состояние лексера `INITIAL`. Дополнительные состояния добавляются через директиву `%x`. Добавим дополнительное состояние `lexstring`, и укажем специальный набор правил для разбора строки:

lpp.lpp разбор строки

```
1  string LexString;
2  %}
3  %x lexstring
4  %%
5  '␣␣␣␣␣␣␣␣␣␣␣␣␣␣{BEGIN(lexstring);␣LexString="";}
6  <lexstring>'     {BEGIN(INITIAL); yylval.o = new Str(LexString); return STR; }
7  <lexstring>\\s   {LexString+='␣';}
8  <lexstring>\\t   {LexString+='\t';}
9  <lexstring>\\n   {LexString+='\n';}
10 <lexstring>\n    {LexString+=yytext[0];}
11 <lexstring>.     {LexString+=yytext[0];}
```

Макрос `BEGIN()` переключает состояния, закрывающее правило `<lexstring>'` переключает лексер в начальное состояние, и возвращает токен-строку.

Основным объектом нашего языка является *символ*:

lpp.lpp лексемы-символы

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

Как символы определятся все *лексемы*, состоящие из букв, цифр, знаков подчеркивания и точки[2].

lpp.lpp скобки

```
1 \(                TOC(Op,LB)
2 \)                TOC(Op,RB)
3 \[                TOC(Op,LQ)
4 \]                TOC(Op,RQ)
5 \{                TOC(Op,LC)
6 \}                TOC(Op,RC)
```

lpp.lpp инфиксные операторы

```
1 \=                TOC(Op,EQ)
2 \@                TOC(Op,AT)
3 \+                TOC(Op,ADD)
4 \/                TOC(Op,DIV)
```

И наконец, будем игнорировать все пробельные символы:

lpp.lpp пробелы

```
1 [ \t\r\n]+        {}
```

[2] нужна для распознавания имен файлов как символов

Вызв генератора **flex** приведет к появлению файла `lex.yy.c` с кодом лексера:

Makefile: генерация кода лексера

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

1.2 Парсер /bison/

Парсер

Makefile: генерация кода парсера

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

Предметный указатель