1 Знакомство с генераторами лексических и синтаксических анализаторов lex и yacc

1 Введение

Задача создания эффективных анализаторов исходного кода возникла вместе с первым компилятором. С тех пор прошло много времени, и появилась мощная теория, позволившая автоматизировать рутинный процесс написания однообразных конечных автоматов. Программирование анализаторов исходного кода может быть полезно в учебных или исследовательских целях, но для анализа реальных языков — C++, Python или SQL — в настоящее время используют исключительно автоматические средства генерации лексических и синтаксических анализаторов. Далее речь пойдет только о них: lex и yacc.

2 Установка

Поскольку изначально lex и yacc были проприетарными программами, Сообщество разработало их свободные аналоги: flex и bison.

Установка программ происходит очень просто:

```
# apt-get install flex flex-doc bison bison-doc
```

Как только команда apt-get завершит свою работу, flex и bison готовы к использованию.

Если по какой-то причине ваша ОС не использует пакетный менеджер APT, обратитесь к документации пакетного менеджера вашей системы, чтобы найти и установить эти пакеты.

3 flex

flex означает fast lex, и он является генератором лексических анализаторов. Ни для кого не секрет, что программы на языках программирования состоят не из букв и цифр, но из более крупных единиц — лексем. Лексемы очень удобно задавать при помощи регулярных выражений. Например, идентификатор программы любого языка можно описать так:

```
[A–Za–z_][A–Za–z0–9_]*
```

Это выражение означает, что идентификатор должен состоять как минимум из одной буквы или знака подчеркивания, после которой может стоять сколько угодно букв, цифр или знаков подчеркивания.

flex предназначен для генерации программ—лексических анализаторов на языках C и C++. В самом простом случае такой анализатор, встретив очередную лексему, просто возвращает ее числовой код. В более сложных случаях анализатор совершает некоторое действие над лексемой, например, преобразует последовательность цифр в число.

Как правило, flex используется совместно с генератором синтаксических анализаторов bison.

4 bison

bison является генератором синтаксических анализаторов, полностью совместимым с yacc.

Синтаксический анализатор получает на вход поток лексем и проверяет их на соответствие некоторой контекстно-свободной грамматике. Грамматика в bison описывается в БНФ:

assignment : NAME ’=’ expr ’;’

По традиции нетерминальные символы грамматики записываются строчными буквами, терминальные — прописными или выделяются кавычками.

bison предназначен для генерации программ—синтаксических анализаторов на языках C, C++ и Java. В самом простом случае такой анализатор просто проверяет поток лексем на соответствие указанной грамматике. В более сложных случаях анализатор может выполнять некоторые действия, например, вычислять значения математических выражений или строить деревья.

Как правило, bison используется совместно с генератором лексических анализаторов flex.

5 Проектирование калькулятора

Теперь самое время перейти от абстрактного описания инструментов к реальному примеру. Реальным примером будет *несложный калькулятор*. От *простого калькулятора* его отличает наличие скобок в выражениях и именованные величины, то есть константы и переменные. Константность объекта является, как в C++, постоянным свойством. Это означает, что переменную нельзя превратить в константу, а константу — в переменную. Однако любую величину можно скопировать в любую переменную или создать новую константу со значением любого выражения, содержащего как константы, так и переменные. Это важно для понимания грамматики и поведения калькулятора.

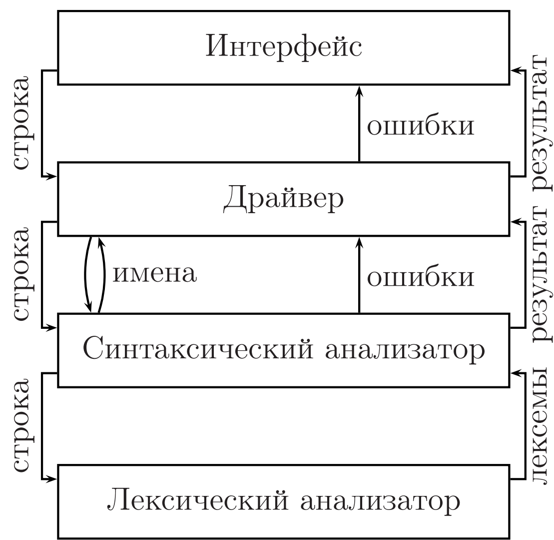
Грамматика калькулятора имеет следующий вид:

```
 1  program ::= ∅
 2     | CONST NAME = expr
 3     | NAME = expr
 4     | expr
 5
 6  expr ::= expr + expr
 7     | expr − expr
 8     | expr ∗ expr
 9     | expr / expr
10     | −expr
11     | (expr)
12     | term
13
14  term ::= NUMBER
15     | NAME
```

Итак, программой калькулятора может быть пустая строка. В этом случае значение выражения не определено и считается равным нулю.

Кроме того программа калькулятора может сохранять значения в переменных или константах. Константа создается при помощи ключевого слова const, переменная создается (или изменяется) просто фактом своего присутствия слева. В этом случае значением всего выражения является значение его левой части.

Интерес представляет необычное описание математических действий. Обычно предполагается, что операторы низкого приоритета описываются в одном правиле, операторы высокого — в другом, причем операндами операций низкого приоритета служат как раз нетерминалы с операторами высокого. Однако на практике писать такие грамматики нелегко, поэтому был найден способ решить проблему приоритета иным способом. Это будет описано в дальнейшем.



Сами выражения состоят термов, которыми являются числа и имена. Регулярные выражения этих лексем можно увидеть в разделе «Лексический анализатор».

Функционально калькулятор состоит из 4-х частей: интерфейса, драйвера, синтаксического и лексического анализаторов.

Основным языком программы является C++. Идеи навеяны главой 6 книги [1].

6 Драйвер

Драйвер является надстройкой над синтаксическим анализатором. Он предназначен для хранения результата, значений величин и сообщений об ошибках.

Можно обойтись и без драйвера, bison это позволяет сделать. Но это неудобно по ряду причин. Во-первых, синтаксический анализатор является классом, интерфейс (и внутренности) которого менять нельзя. Во-вторых, глобальное пространство имен засорится типами и глобальными переменными, использование которых неизбежно, что крайне нежелательно.

Теперь рассмотрим класс драйвера более детально.

```
8   #include "parser.hpp"
```

Восьмая строка подключает заголовок, генерируемый bison-ом. В нем находится класс Parser, который осуществляет синтаксический анализ.

```
10  #define YY_DECL \
11    yy::Parser::token_type \
12    yylex(yy::Parser::semantic_type* yylval, yy::Parser::location_type* yylloc)
13
14  YY_DECL;
```

В строках 10–14 помещено объявление функции лексического анализа. Эта функция объявлена в виде макроса YY_DECL, как это требуется соглашениями о вызовах между bison и flex.

Аргументами функции являются лексема и ее местоположение во входном потоке. Функция возвращаем тип лексемы.

```
16  class Driver
17  {
18    friend class yy::Parser;
```

Со строки 16 начинается собственно объявление драйвера. Класс yy::Parser, генерируемый bison-ом, объявлен дружественным. Это вызвано тем обстоятельством, что таблица символов по архитектурным соображениям является закрытой, а синтаксический анализатор должен иметь к ней полный доступ.

```
20  public:
21    class Error
22    {
23    private:
24      int from_;
```

```
25      int to_;
26      std::string message_;
27
28    public:
29      Error() {}
30      Error(const std::string& m) : from_(0), to_(0), message_(m) {}
31      Error(int f, int t, const std::string& m) : from_(f), to_(t), message_(m) {}
32
33      int from() const { return from_; }
34      int to() const { return to_; }
35      const std::string& message() const { return message_; }
36    };
```

Далее идет описание класса Error. Этот класс хранит текстовое описание ошибки и позиции в исходной строке, где эта ошибка встретилась.

В данном случае обработка ошибок заключается просто в прекращении дальнейшего анализа строки. Драйвер хранит последнюю ошибку. Альтернативой было бы использование исключений, никаких препятствий этому нет.

```
38  private:
39    struct Value
40    {
41      bool is_const;
42      double value;
43
44      static Value constant(double v) { Value r = { true, v }; return r; }
45      static Value variable(double v) { Value r = { false, v }; return r; }
46    };
47    typedef std::map<std::string, Value> Symbols;
48
49    Symbols symbols;
```

Строки 38–49 содержат описание таблицы символов. Каждый символ имеет (неизменяемый) признак константности и значение. Статические функции constant и variable предназначены для удобства написания исходного кода.

Сама таблица символов является словарем имен.

```
51    double result_;
52    bool hasError_;
53    Error error_;
```

Со строки 51 начинается определение переменных-свойств. Оба свойства предназначены только для чтения, устанавливать значения переменных могут только классы

Driver и yy::Parser.

```
24 int Driver::parse(const std::string& s)
25 {
26   hasError_ = false;
27   YY_BUFFER_STATE buff = yy_scan_string(s.c_str());
28   if(!buff)
29   {
30     setError("a problem occured while reading a string");
31     return -1;
32   }
33
34   yy::Parser parser(*this);
35
36   result_ = 0;
37   int res = parser.parse();
38   yy_delete_buffer(buff);
39   return res;
40 }
```

В строках 24–40 файла driver.cpp находится определение функции parse().

По умолчанию анализаторы читают символы из файла yyin. Но описанная функция предполагает считывать данные из строки. Для этого необходимо создать дескриптор при помощи функции yy_scan_string(). После завершения анализа строки дескриптор необходимо уничтожить функцией yy_delete_buffer().

Класс yy::Parser содержит в интерфейсе функцию parse(). Эта функция возвращает 0 в случае успеха и отличное от 0 значение при ошибке.

Исходный код драйвера находится в файлах driver.h и driver.cpp.

7 Синтаксический анализатор

```
%{
  Пролог
%}
Объявления bison-a
%%
Правила грамматики
%%
Эпилог
```

Описание синтаксического анализатора находится в файле parser.y. Этот файл состоит из трех разделов: пролога, правил грамматики и эпилога. Кратко рассмотрим назначение каждого раздела и работу генератора синтаксических анализаторов bison в целом.

Программа bison генерирует исходный файл на одном из языков: C, C++ или Java. Далее речь пойдет исключительно о языке C++, но генерация файлов на других языках выполняется аналогично.

В случае языка C++ bison генерирует файл, содержащий

класс в пространстве имен yy. Имя класса можно менять, в данном случае класс называется Parser.

Общий принцип работы bison-а таков: при помощи программы m4 bison создает исходные файлы на языке C++. При этом в начало определения класса и в заголовочный файл помещаются строки из пролога. Также пролог содержит настройки для bison-а: имя класса, аргументы функций, типы, лексемы и пр. Основная часть класса, то есть функция yy::Parser::parse(), генерируется на основе правил грамматики. В конец определения класса помещаются строки из эпилога.

Теперь подробно рассмотрим файл parser.y.

```
23 %require "2.4.1"
```

В самом начале файла помещена информация о минимальной версии bison-а. Дело в том, что bison постоянно развивается: добавляются новые опции, меняются форматы шаблонов. Параметр %require позволяет указать необходимую версию генератора анализаторов и тем самым избежать многочисленных сообщений о нераспознанных параметрах.

```
24 %language "C++"
25 %skeleton "lalr1.cc" /* -*- C++ -*- */
26 %defines
27 %define parser_class_name "Parser"
```

В первых строках пролога содержится указание целевого языка, шаблон (параметр %skeleton) и константы. Одной из констант является имя класса, которое здесь принимает значение «Parser».

```
38 %parse-param { Driver& driver }
```

Параметр parse-param содержит аргументы для функции yyparse(), если программа генерирует анализатор на языке C, или для конструктора класса Parser, как в данном случае.

```
40 %locations
```

Этот параметр включает трассировку. Каждая лексема снабжается информацией о местоположении в анализируемом файле. Трассировка замедляет анализ, но зато позволяет более детально информировать пользователя о ходе разбора и возникающих ошибках.

```
42 %union {
43   double value;
44   std::string* name;
45 }
```

Строки 41–44 содержит описание семантического типа yy::Parser::semantic_type, он же YYSTYPE. Этот тип представляет собой объединение. В данном случае объединение содержит два поля: для числовых значений и для имен. Имя определено как указатель на строку. Это не ограничение bison-а, это ограничение C++[1].

```
47 %code {
48 #include <memory>
49
50 #include "driver.h"
51
52 typedef std::auto_ptr<std::string> StringPtr;
53 }
```

Секция code содержит код, помещаемый перед определением класса. Синтаксическому анализатору необходимо знать структуру класса Driver, чтобы обращаться к таблице символов.

В строке 51 объявлен синоним типа auto_ptr. Выше упоминалось, что одним из членов объединения является указатель на строку. Функции, работающие с объединением, должны освобождать занятые ресурсы. Конечно, в примитивном случае можно вызвать оператор delete, и это сработает. Но как будет видно дальше, функции имеют несколько точек выхода. В таких случаях гораздо эффективнее использовать один из классов с автоматической сборкой мусора (или создать свой), чем вызывать оператор delete перед каждой точкой выхода[2].

```
55 %token CONST "const"
56 %token END 0 "end_of_file"
57 %token INVALID_CHARACTER "invalid_character"
58 %token NOT_A_NUMBER "not_a_number"
59
60 %token <name> NAME "identifier"
61 %token <value> NUMBER "number"
```

Строки 54–60 содержат самое интересное (после описания правил грамматики): описание лексем. Описание лексемы начинается со слова %token, после чего может идти тип лексемы в терминах YYSTYPE, ее символьное значение, необязательное числовое значение (лексема END по соглашению должна быть равна 0) и строку с удобным

[1] Дело в том, что любой класс имеет свою внутреннюю структуру, которая создается при вызове конструктора объекта. Запись в поле value разрушила бы структуру name, если бы name был объектом. Это как минимум вызвало бы утечку ресурсов, а в худшем случае нарушение прав доступа при вызове деструктора объекта. По этой причине объекты не могут быть членами объединений.

[2] Конечно, можно оставить одну точку выхода, однако пострадает понятность исходного кода. Кроме того, даже в этом случае нет гарантии освобождения ресурсов при внезапно возникшем исключении.

описанием лексемы. Последнее позволяет bison-у более понятно сообщать об ошибках.

Все символьные константы помещаются в структуре yy::Parser::token. Это важно помнить при написании лексического анализатора.

```
63 %type <value> expr term
```

Генерируемые bison-ом функции могут возвращать значения разных типов. В данном случае типы выражений и термов определяются как value, то есть число.

```
65 %destructor { delete $$; } "identifier"
```

Параметр %destructor описывает действия по уничтожению того или иного типа. Если возвращаемым значением функции является идентификатор, то его необходимо уничтожить вызовом оператора delete.

```
67 %error-verbose
```

Этот параметр включает подробные сообщения об ошибках.

На этом пролог заканчивается и начинается описание правил грамматики.

```
71 %start program;
72 program : /* nothing */ {}
73   | CONST NAME '=' expr { StringPtr p($2);
74     if(driver.symbols.find(*p) == driver.symbols.end())
75     {
76       driver.symbols.insert(Driver::Symbols::value_type(*p,
           Driver::Value::constant($4)));
77       driver.result_ = $4;
78     }
79     else
80     {
81       std::ostringstream msg;
82       msg << "cannot rebind '" << *p << "'";
83 
84       yy::location l;
85       l.begin = @1.begin;
86       l.end = @3.end;
87 
88       error(l, msg.str());
89       YYABORT;
90     }
91   }
```

Bison строит анализатор по схеме LALR(1). Синтаксический анализатор представляет собой конечный автомат. Автомату необходимо начальное значение. Строка 70 устанавливает это значение.

Правила грамматики, как видно из примера, описываются в виде БНФ, причем после правила следует собственно действие, соответствующее правилу. Поскольку грамматика допускает пустое правило, то строка 71 содержит просто фигурные скобки без какого-либо кода.

Зато альтернатива «CONST NAME '=' expr» содержит куда больше кода.

Здесь надо отметить, что функция принимает число аргументов, равное числу лексем в правой части правила. Каждый аргумент описывается в виде $\$n$:

\$1	\$2	\$3	\$4
CONST	NAME	'='	expr

Возвращаемое функцией значение описывается как $$. Правило program ничего не возвращает, поэтому значение $$ в нем не используется.

Что касается самой функции, то ее действия описываются просто. По условию константность величины не может быть изменена, поэтому функция осуществляет поиск имени $2 в таблице символов. Если это имя уже есть в таблице, то функций просто завершается с ошибкой (макрос YYABORT в строке 120). В противном случае в таблицу добавляется новая константа, а переменная result_ драйвера получает ее значение. Класс StringPtr позволяет не заботиться об освобождении ресурсов в *двух* точках выхода (а их тут именно *две*). Компилятор вызовет деструктор объекта p в необходимый момент.

Остальные альтернативы правила program мало отличаются от описанной, поэтому нет смысла подробно рассматривать их.

```
120 %left '+' '-';
121 %left '*' '/';
122 %right UMINUS;
123 expr : expr '+' expr { $$ = $1 + $3; }
124    | expr '-' expr { $$ = $1 - $3; }
125    | expr '*' expr { $$ = $1 * $3; }
126    | expr '/' expr { if($3)
127         $$ = $1 / $3;
128       else
129       {
130         error(@3, "division by zero");
131         YYABORT;
132       }
133    }
```

```
134     | '−' expr %prec UMINUS { $$ = −$2; }
135     | '(' expr ')' { $$ = $2; }
136     | term { $$ = $1; }
137   ;
```

Правило expr описывает арифметические действия: сложение, вычитание, умножение, деление, смену знака и скобки. Выполняемые действия тривиальны. Некоторый интерес представляет операция деления. В случае равенства нулю правого операнда ($3) происходит вызов функции-члена yy::Parser::error(), причем первым параметром этой функции является @3. Параметры вида @n доступны только при включенной опции %locations и содержат местоположение данной лексемы. В случае попытки деления на 0 синтаксический анализатор точно укажет недопустимое (т. е. нулевое) подвыражение.

Правило expr описывает все действия невзирая на их приоритет. Все дело в строках 119–121, в которых описаны приоритет и ассоциативность операций. Чем ниже по коду описана операция, тем выше ее приоритет. Аддитивные и мультипликативные операции левоассоциативны, тогда как унарный минус правоассоциативен.

Приоритеты специально введены для удобства описания математических выражений.

```
139   term : NUMBER { $$ = $1; }
140     | NAME { StringPtr p($1);
141       Driver::Symbols::const_iterator i = driver.symbols.find(*p);
142       if(i != driver.symbols.end())
143         $$ = i−>second.value;
144       else
145       {
146         std::ostringstream msg;
147         msg << "undefined symbol '" << *p << "'";
148         error(@1, msg.str());
149         YYABORT;
150       }
151     }
152     | INVALID_CHARACTER { YYABORT; }
153     | NOT_A_NUMBER {
154       error(@1, "not a number");
155       YYABORT;
156     }
157   ;
```

В правиле term наибольший интерес представляют две последние альтернативы: INVALID_CHARACTER и NOT_A_NUMBER. Они предназначены только для од-

ного: обработки ошибок лексического анализатора. Дело в том, что функция yylex() возвращает целое число—номер лексемы. Поэтому единственный способ передать «наверх» сообщение об ошибке — определить специальную лексему. Еще один способ заключается в возбуждении исключения.

```
161 void yy::Parser::error(const yy::Parser::location_type& l, const std::string& m)
162 {
163   driver.setError(l.begin.column, l.end.column, m);
164 }
```

В эпилоге определена функция-член error(). Эта функция обязательно должна быть определена программистом. В данном случае функция просто передает ошибку драйверу.

8 Лексический анализатор

Структура входного файла лексического анализатора также состоит из трех частей.

Определения %% Правила %% Код	Первая часть файла содержит разнообразные настройки flex-а, определения регулярных выражений и исходный код, помещаемый перед кодом анализатора. Вторая часть содержит сами правила и соответствующие им действия. Как правило, действия заключаются в возврате той или иной символьной константы. Последняя часть файла содержит код функций, помещаемых после кода лексического анализатора. Например, там может быть функция main(), если программа небольшая.

Теперь рассмотрим файл scanner.l, который содержит исходный код сканера, подробнее.

```
1 %option 8bit
2 %option noyywrap
3 %option warn
```

Файл начинается с настроек генератора flex. Здесь указано использование 8-битной кодировки (можно выбрать и 7-битную), отказ от функции yywrap (необходима для обработки нескольких входных файлов) и включены все предупреждения flex-а.

Далее в файле описано включение заголовочных файлов, как стандартных, так и принадлежащих калькулятору. Как и в parser.y, в scanner.l исходный код C++ обрамлен символами «%{» и «%}».

```
14 // By default yylex returns int, we use token_type.
15 // Unfortunately yyterminate by default returns 0, which is
16 // not of token_type.
```

```
17 #define yyterminate() return yy::Parser::token::END
```

Разбор лексем завершается вызовом функции yyterminate(), которая возвращает целый 0, что не соответствует типу yy::Parser::token, поэтому функция переопределяется таким образом.

```
21 %{
22 #define YY_USER_ACTION yylloc->step(); \
23   yylloc->columns(yyleng);
24 %}
```

flex позволяет добавлять в код действия пользователя после распознания очередной лексемы. В данном случае начало лексемы передвигается в конец предыдущей, после чего устанавливается новый конец лексемы. Это необходимо для понятного вывода информации об ошибках.

Кроме этого в раздел определений можно поместить определения регулярных выражений, но в данном случае в этом нет практического смысла.

Второй раздел содержит описание регулярных выражений, которыми описываются лексемы, и действия анализатора при распознании таких лексем.

```
28 [+\-*/=()] return yy::Parser::token_type(yytext[0]);
```

Считанная из потока лексема записывается в переменную yytext. В данном случае знаки операций и скобок напрямую превращаются в тип token_type. Можно было бы отдельно описать константы PLUS, MINUS и др., но в данном случае это усложнит код, не увеличив его понятности.

```
29 "const" return yy::Parser::token::CONST;
```

Слово «const» распознается очень простым регулярным выражением.

```
31 [[:alpha:]_][[:alnum:]_]* yylval->name = new std::string(yytext); return
       yy::Parser::token::NAME;
```

Как сообщалось выше, идентификатор начинается с буквы или знака подчеркивания, после чего может следовать любое число букв, цифр или знаков подчеркивания. flex понимает регулярные выражения в формате POSIX, что позволяет описывать алфавитные символы как [:alpha:], а не как [A-Za-z].

Встретив идентификатор, лексический анализатор копирует его в экземпляр класса std::string. Уничтожить экземпляр класса должен уже синтаксический анализатор.

```
33 [[:digit:]]+(\.[[:digit:]]*)?|\.[[:digit:]]+ {
34   errno = 0;
35   double d = std::strtod(yytext, 0);
36   if(errno == ERANGE)
37     return yy::Parser::token::NOT_A_NUMBER;
```

```
38     yylval->value = d;
39     return yy::Parser::token::NUMBER;
40 }
```

Число может быть записано несколькими способами для удобства ввода: как «15», как «3.14» и даже как «.5». Регулярное выражение отражает эти способы. Кроме собственно распознания лексемы функция преобразует строку в значение типа double. Если преобразование не удалось, синтаксический анализатор получает «лексему» NOT_A_NUMBER.

```
42 [ \t]+ {}
```

Пробельные символы просто игнорируется.

```
44 . { return yy::Parser::token::INVALID_CHARACTER; }
```

Последнее правило определяет действия анализатора на случай непредвиденных лексем. Здесь анализатор возвращает специальную лексему INVALID_CHARACTER, и уже задачей синтаксического анализатора является обработка ошибки.

9 Отладка

Разумеется, было бы наивно предполагать, что грамматику произвольной сложности можно сразу описать без ошибок. Для отладки flex и bison имеют специальные опции и флаги.

Для отладки flex имеет следующие флаги:

–d, %option debug генерирует отладочную версию анализатора. Как только flex встречает этот флаг и переменная yy_flex_debug не равна 0 (по умолчанию она не равна), лексический анализатор выводит в стандартный поток ошибок строку вида

–accepting rule at line 53 (“the matched text”)

Номер строки соответствует номеру строки с примененным правилом в исходном файле.

–s, ––nodefault, %option nodefault изменяет поведение лексического анализатора при считывании неизвестного символа. По умолчанию этот символ дублируется в стандартный вывод. При включенной опции лексический анализатор немедленно прерывает работу с сообщением об ошибке. Это полезно для поиска дыр в списке правил.

Для отладки синтаксического анализатора необходимо добавить директиву %debug а также вызывать функцию yy::Parser::set_debug_level() с ненулевым аргументом.

10 Интерфейс

Калькулятор может работать в пакетном или интерактивном режимах. Если в командной строке содержатся аргументы, каждый из них рассматривается как команда калькулятора. В противном случае калькулятор считывает данные из стандартного ввода до конца файла. Для упрощения работы с двумя источниками входных данных описан класс Input.

Исходный текст интерфейса содержится в файле calc.cpp.

11 Python Lex-Yacc

Для языка программирования Python есть специальный модуль ply (Python Lex-Yacc). Python Lex-Yacc основан на *идеях* lex и yacc, но описание грамматик хранится не в отдельных файлах, а непосредственно в .py. Перед запуском такой программы ply генерирует и компилирует дополнительные модули, основываясь на интерфейсе .py.

Python предоставляет информацию о любом модуле или классе прямо в ходе работы программы. ply извлекает из интерфейса модуля отвечающие соглашению имена и генерирует лексический и синтаксический анализаторы. Несмотря на разницу в подходах, исходные файлы имеют много общего.

11.1 Установка

Установка ply происходит очень просто:

```
# apt-get install python-ply python-ply-doc
```

Как только команда apt-get завершит свою работу, ply готов к использованию.

Если по какой-то причине ваша ОС не использует пакетный менеджер APT, обратитесь к документации пакетного менеджера вашей системы, чтобы найти и установить эти пакеты.

11.2 Проектирование

В версии калькулятора на Python-е нет драйвера по двум причинам. Во-первых, обработка ошибок сведена к возбуждению исключений, то есть можно отказаться от хранения последней ошибки. Во-вторых, тип функции parse() определяется динамически, то есть можно отказаться от переменной result_.

В отличие от примера на C++, где синтаксический анализатор является классом, а лексический — функцией, здесь все наоборот: лексический анализатор является классом, а синтаксический — функцией.

11.3 Синтаксический анализатор

Описание синтаксического анализатора находится в файле calc_parser.py.

```
26 import ply.yacc as yacc
```

Во-первых, надо подключить модуль ply.yacc.

```
29 from scanner import Scanner, Error
```

Потом из модуля scanner, описание которого будет приведено ниже, импортировать класс Scanner для лексического анализа и класс Error для возбуждения исключений.

```
32 _symbols = { "pi" : ("const", math.pi),
33   "e" : ("const", math.e) }
```

Таблица символов представляет собой словарь, как в предыдущем примере.

```
35 _scanner = Scanner()
36 tokens = Scanner.tokens
```

Далее необходимо создать объект—лексический анализатор и обозначить лексемы переменной tokens. Константы лексем описываются ниже в классе Scanner модуля scanner.

```
38 precedence = (("left", "PLUS", "MINUS"),
39   ("left", "MUL", "DIV"),
40   ("right", "UMINUS"))
```

Приоритет и ассоциативность операций по соглашению описываются переменной precedence.

```
43 def p_program_empty(p):
44   """program :"""
45   p[0] = 0.0
```

Имена продукций начинаются, опять же по соглашению, с префикса «p_». Грамматика продукции обозначается в виде *docstring*, далее идет исходный текст функции. В данном случае продукция пустая, а результат, как описано выше, равен 0.

```
47 def p_program_const_assignment(p):
48   """program : CONST NAME ASSIGN expr"""
49   v = _symbols.get(p[2], None)
50   if v:
51     raise Error, ("cannot rebind '%s'" % p[2], p.lexspan(1)[0], p.lexspan(3)[1])
52   _symbols[p[2]] = ("const", p[4])
53   p[0] = p[4]
```

Код функций анализатора похож на код исходных файлов bison-а. Есть разница в обозначениях: переменная $$ называется p[0], а $n — p[n]. Важно помнить, что отрицательные индексы в данном случае имеют совершенно другой смысл, поэтому их здесь нельзя использовать.

Переменные @n, в которых хранится положение лексемы в строке, здесь доступны через вызов функции p.lexspan(n).

```
114 def p_error(p):
115   raise yacc.SyntaxError, "syntax error at token '%s', position %d" % (p.value,
          p.lexpos)
```

Функцию p_error() вызывает непосредственно анализатор при возникновении ошибки.

```
117 yacc.yacc()
```

Функция yacc() обрабатывает модуль, создавая синтаксический анализатор. Кроме того, она выводит в файл parser.out описание сгенерированного конечного автомата в понятной человеку форме. Этот файл полезен при отладке.

```
119 def calculate(data):
120   return yacc.parse(data, lexer=_scanner.lexer, tracking=True)
```

Напоследок определим функцию calculate(), которая вызывает parse() с нужными параметрами: ссылкой на лексический анализатор и установленным флагом трассировки. Трассировка, как и в примере на C++, замедляет обработку входных данных, но позволяет получать информацию о местоположении лексем в исходном тексте.

11.4 Лексический анализатор

Описание лексического анализатора находится в файле calc_scanner.py.

```
3 import ply.lex as lex
```

Во-первых, надо импортировать модуль ply.lex.

```
 6 class Error(Exception):
 7   def __init__(self, m, begin, end):
 8     Exception.__init__(self, m)
 9     self.begin = begin
10     self.end = end
```

Производный класс Error хранит сообщение об ошибке и место ее возникновения. Класс используется для возбуждения исключений.

```
12 class Scanner:
```

Лексический анализатор реализован в виде класса Scanner. Этот класс должен соответствовать определенному интерфейсу.

```
13    # The list of reserved words. Optional
14    reserved = {
15      "const" : "CONST"
16    }
```

Переменная reserved необязательна и не является частью интерфейса, однако ключевые слова программы описаны именно в ней. Ниже будет надо объяснение этому факту.

```
18    # The list of tokens. Mandatory
19    tokens = [
20      "NAME", "NUMBER",
21      "PLUS", "MINUS", "MUL", "DIV",
22      "ASSIGN", "LP", "RP"
23    ] + reserved.values()
```

В отличие от C++ лексемы определяются не константами-числами, а константами-строками[3].

```
25    # Regular expressions for simple tokens
26    t_PLUS = r"\+"
27    t_MINUS = r"-"
28    t_MUL = r"\*"
29    t_DIV = r"/"
30    t_ASSIGN = r"="
31    t_LP = r"\("
32    t_RP = r"\)"
```

Простые лексемы описываются регулярным выражением. По соглашению имена лексем начинаются с префикса «t_».

```
34    # Complex tokens
35    def t_NAME(self, t):
36      r"[A-Za-z_]\w*"
37      t.type = self.reserved.get(t.value, "NAME")
38      return t
```

«Сложными» считаются лексемы, требующие особых действий при их появлении в исходном тексте. В переменной *docstring* функции описывается регулярное выражение, а сама лексема передается в функцию вторым аргументом (или единствен-

[3]Впрочем, в Python-е все строки являются константами по соображениями производительности.

ным, если функция не является методом). Функция t_NAME() устанавливает тип лексемы, записывая его в переменную t.type.

Лексический анализ в ply немного отличается от bison-овского. Если определить отдельное регулярное выражение для ключевого слова «const», то можно столкнуться с неприятной ситуацией, когда в исходном тексте появится (допустимый) идентификатор «constant». Анализатор разобьет его на две лексемы: «const» и «ant», что недопустимо.

Поскольку всякое ключевое слово является правильным именем, но не каждое правильное имя — ключевым словом, то при обнаружении идентификатора происходит поиск среди ключевых слов.

```
58    def __init__(self, **kwargs):
59      self.lexer = lex.lex(object=self, **kwargs)
```

Конструктор объекта Scanner создает лексический анализатор на основе самого себя. Аргумент **kwargs позволяет регулировать поведение объекта.

```
61    def input(self, data):
62      self.lexer.input(data)
```

И, наконец, каждый лексический анализатор должен иметь функцию input(), которая его запускает.

11.5 Интерфейс

Интерфейс калькулятора находится в файле calc.py.

12 Задание

Пользуясь вышеприведенными примерами и указанными ниже источниками, опробовать генераторы анализаторов.

Список литературы

[1] Б. Страуструп. Язык программирования C++, спец. изд./Пер. с англ. — М.; СПб.: «Издательство БИНОМ» — «Невский Диалект», 2001 г. — 1099 с., ил. 5

[2] Bison — GNU parser generator. http://www.gnu.org/software/bison/manual/

[3] Lexical Analysis With Flex. http://flex.sourceforge.net/manual/

[4] Лучшая обработка ошибок с помощью Flex и Bison. Советы по созданию более дружественных к пользователю компиляторов и интерпретаторов. http://www.ibm.com/developerworks/ru/library/l-flexbison/index.html

[5] PLY (Python Lex-Yacc). http://www.dabeaz.com/ply/ply.html