Создание анализаторов текста при помощи yacc и lex

Мартин Браун, внештатный автор, консультант

Описание: В этой статье на примере создания простого калькулятора показано, как создать анализатор при помощи инструментов lex/flex и yacc/bison, а затем более подробно рассмотрено, как применить эти принципы к синтаксическому разбору текста. Синтаксический разбор текста - анализ и извлечение ключевых частей текста - важная часть многих приложений. В UNIX® многие элементы операционной системы зависят от синтаксического анализа текста: оболочка, которая используется для взаимодействия с системой, распространенные утилиты и команды типа awk или Perl, вплоть до компилятора Си, используемого для разработки приложений. Анализаторы собственной разработки можно использовать в UNIX-программах (и не только UNIX) для создания простых анализаторов конфигурации или даже для создания своего собственного языка программирования.

Дата: 03.12.2008
Уровень сложности: средний

Грамматический анализ с использованием yacc

Грамматические правила используются для распознавания последовательности лексем и выполнения подходящих действий. В основном грамматические правила используются в комбинации с lex; оба эти инструмента - yacc и lec - и составляют анализатор.

Простые грамматические правила в yacc

Основная часть описания грамматики в yacc - описание последовательности ожидаемых лексем. Например, распознать выражение A + B, где A и B - это числа, можно при помощи следующего грамматического правила:

```
NUMBER PLUSTOKEN NUMBER { printf("%f\n",($1+$3)); }
```

NUMBER - это идентифицирующая лексема для числа, а PLUSTOKEN - идентифицирующая лексема для знака "плюс".

Код в фигурных скобках определяет действия, которые должны быть выполнены при распознавании искомой последовательности. В этом примере выполняется код на Си, который складывает два обнаруженных числа. Для описания первой и третьей лексемы в грамматическом правиле используются условные обозначения $1 и $3 соответственно.

Чтобы грамматическое правило было идентифицировано, необходимо присвоить ему имя, как показано в листинге 6.

Листинг 6. Присвоение имени грамматическому правилу

```
addexpr: NUMBER PLUSTOKEN NUMBER
        {
            printf("%f\n",($1+$3));
        }
        ;
```

Имя грамматического правила - addexpr, описание грамматического правила заканчивается точкой с запятой.

Грамматическое правило может содержать несколько идентифицируемых последовательностей (и связанных с ними действий), у которых есть что-то общее. Например, операции сложения и вычитания в калькуляторе идентичны, поэтому их можно сгруппировать вместе. Отдельные правила в группе отделяются друг от друга вертикальной чертой (|) (см. листинг 7).

Листинг 7. Сгруппированные правила отделяются друг от друга символом "|"

```
addexpr: NUMBER PLUSTOKEN NUMBER
        {
            printf("%f\n",($1+$3));
        }
        | NUMBER MINUSTOKEN NUMBER
        {
            printf("%f\n",($1-$3));
        }
        ;
```

Теперь, после изложения основных правил описания грамматики, следует объяснить, как комбинировать несколько правил и рассмотреть их приоритеты.

Приоритеты грамматических правил

В большинстве языков, например литературном, математическом, языке программирования, есть такое понятие как приоритет (значимость) фразы. Выражения с высоким приоритетом более важны по сравнению с выражениями, у которых более низкий приоритет.

Например, в большинстве программных языков, в математических выражениях умножение имеет более высокий приоритет, чем сложение и умножение. Например, выражение: 4+5*6 эквивалентно: 4+30.

В конечном счете, данное выражение равняется 34. Причина этого в том, что операция умножения выполняется первой (5 на 6 равно 30), и, далее, полученное выражение используется в последней операции - результат умножения складывается с 4 и получается 34.

В yacc приоритеты задаются путем определения множественных групп грамматических правил и связывания их; порядок расположения отдельных правил в группе помогает определить приоритет. В листинге 8 представлен код, который описывает поведение, упомянутое выше.

Листинг 8. Связывание грамматических правил и установка приоритетов

```
add_expr: mul_expr
      | add_expr PLUS mul_expr  { $$ = $1 + $3; }
```

```
        | add_expr MINUS mul_expr { $$ = $1 - $3; }
        ;
mul_expr: primary
        | mul_expr MUL primary { $$ = $1 * $3; }
        | mul_expr DIV primary { $$ = $1 / $3; }
        ;
primary: NUMBER { $$ = $1; }
        ;
```

Все выражения, которые просматривает yacc, обрабатываются слева направо. Поэтому в сложном выражении (типа рассмотренного примера 4+5*6) yacc в первую очередь будет искать соответствие наиболее приоритетному правилу, описывающему выражение.

Сопоставления выполняются в порядке расположения правил - сверху вниз. Таким образом, сначала выполняется сопоставление грамматическим правилам в add_expr. Однако, поскольку приоритет умножения выше, в правилах указывается, что yacc должен прежде всего проработать mul_expr. Данное указание принуждает yacc искать в первую очередь умножение (как определено в mul_expr). Если ничего найти не удалось, yacc переходит к следующему правилу в блоке add_expr и так далее до тех пор, пока выражение не будет распознано. В случае, если распознать выражение не удастся, будет выведена ошибка.

При оценке выражения 4+5*6 набором правил, определенных выше, yacc сначала пытается использовать add_expr, затем его перенаправляют в mul_expr (первое правило, с которым надо сопоставлять), откуда в свою очередь yacc перенаправляется на самое главное правило primary. Правило primary присваивает значения числам в выражении. $$ определяет значение, возвращаемое частью выражения.

Как только все числа были распознаны, yacc (при помощи описания в строке | mul_expr MUL primary { $$ = $1 * $3; }) определяет операцию умножения. Правило: | mul_expr MUL primary { $$ = $1 * $3; } сохраняет возвращаемое значение подвыражения 5*6. yacc генерирует код, который выполняет вычисление (при этом извлекая значения по предыдущему главному правилу), и заменяет исходную часть входного выражения (с умножением) на результаты вычисления. Это означает, что теперь yacc будет искать и сопоставлять со своими правилами следующее выражение: 4+30.

Это выражение соответствует правилу | add_expr PLUS mul_expr { $$ = $1 + $3; }, которое в конечном счете и выдаст результат 34.

Упорядочивание лексем по приоритету в грамматическом описании

Грамматические описания устанавливают правила, согласно которым проводится синтаксический разбор входных лексем (их формата и размещения) в некоторый формат данных, который можно использовать в дальнейшем. Описанные выше наборы правил указывают yacc, как распознать входной текст и выполнить соответствующий фрагмент кода Си. Инструмент yacc генерирует код Си, который необходим для анализа информации; yacc сам по себе не производит анализ.

Программный код в фигурных скобках является исходным кодом на псевдо-Си. Yacc управляет преобразованием из программного кода на псевдо-Си в исходный код на Си, основываясь на правилах и остальной части кода, которая фактически используется для анализа входных данных.

Кроме наборов правил, которые определяют метод анализа входных данных, имеются дополнительные функции и опции, которые помогают определить остальной исходный код на Си. Формат файла описания yacc (такой же как у lex/flex) показан в листинг 9.

Листинг 9. Формат yacc-файла

```
%{
/* требуемые глобальные и заголовочные определения */
%}

/* описания (дополнительные определения токенов) */
```

```
%%
/* определение наборов правил для анализа
%%

/* дополнительный исходный код C */
```

Определения глобальных переменных и заголовочных файлов (глобальные/заголовочные) такие же как и в файлах lex/flex. Эти определения нужны для того чтобы можно было использовать какие-либо особые функции при анализе входных данных и обработке выходных.

Описания лексем относятся не только к ожидаемым лексемам, но и к приоритетам, использующимся при анализе. В этих описаниях, путем установления принудительного порядка исследования лексем, определяется порядок того, как yacc будет обрабатывать правила. Также в этом описании указывается, как yacc будет работать с выражениями при сопоставлении их с правилами.

Например, для знака плюс (+) был установлен порядок старшинства слева направо, т.е. любое выражение, расположенное слева от этого знака, будет сопоставлено первым. Это означает, что выражение 4+5+6 сначала будет оценено как 4+5, а затем как 9+6.

Однако для некоторых знаков может понадобиться правый (справа налево) порядок старшинства. Например, для анализа логического НЕТ (логическое отрицание), (например, ! (expr)) нам нужно, чтобы expr оценивалось прежде, чем его значение будет инверсировано символом !.

Устанавливая порядок старшинства лексем, кроме управления приоритетами отношений между правилами, можно определить порядок анализа входных данных.

Для управления приоритетами есть три типа маркеров: %token (без приоритетов), %left (приоритет отдается входным данным слева от указанной лексемы) и %right (приоритет отдается входным данным справа от указанной лексемы). Например, в листинге 10 описано несколько лексем согласно требуемому порядку старшинства.

Листинг 10. Описание лексем согласно их требуемому старшинству

```
%token EQUALS POWER
%left PLUS MINUS
%left MULTIPLY DIVIDE
%right NOT
```

Следует заметить, что лексемы расположены в порядке увеличения приоритетов (сверху вниз), и лексемы на одинаковых уровнях имеют одинаковые приоритеты. В примере (см. листинг 10) MULTIPLY и DIVIDE имеют одинаковые приоритеты, которые выше, чем у PLUS и MINUS.

Любые лексемы, не описанные как в листинге 10, но присутствующие в каких-либо правилах, послужат причиной ошибки.

Пользовательская инициализация и ошибки

Две наиболее важные функции, которые нужно определить, это функция main(), в которой будет находиться инициализация, и функция yyerror(), которая выводит ошибку при отсутствии правила выполнения анализа. Сам анализ осуществляется функцией yyparse(), которую генерирует yacc. Типичные описания пользовательских функций приведены в Listing 11.

Листинг 11. Функции анализатора, определяемые пользователем

```
#include <stdio.h>
#include <ctype.h>
char *progname;
double yylval;

main( argc, argv )
char *argv[];
{
  progname = argv[0];
  yyparse();
}

yyerror( s )
char *s;
{
    fprintf( stderr ,"%s: %s\n" , progname , s );
}
```

Функция yyerror() принимает строку, которая в случае ошибки выводится вместе с названием программы.

Компилирование грамматического описания в файл на Си

yacc и GNU-инструмент bison принимают на вход файл грамматического описания и создают необходимый исходный код Си, из которого можно скомпилировать анализатор. Этот скомпилированный анализатор будет принимать соответствующие входные данные.

Файлы грамматического описания обычно имеют разрешение .y.

По умолчанию при вызове yacc он создает файл с именем yy.tab.c. Если yacc используется в комбинации с lex, то, возможно, полезно будет сгенерировать дополнительный заголовочный C-файл, который будет содержать макроопределение лексем, идентифицированных в обеих системах. Для создания заголовочного файла, в дополнение к исходному коду на C, нужно использовать опцию -d:

```
$ yacc -d calcparser.y
```

Кроме того, bison выполняет некоторые основные операции. По умолчанию он создает файлы, имя которых начинается с префикса имени исходного файла грамматического описания. Таким образом, в упомянутом выше примере bison создаст файлы calcparser.tab.c и calcparser.tab.h.

Это различие важно не только потому, что необходимо знать, какой файл компилировать, но и потому, что необходимо импортировать корректный заголовочный файл в lex.l-описание, чтобы тот мог прочитать корректные определения лексем.

Ниже перечислены основные шаги при компилировании приложений из исходного кода:

1. Выполнить yacc для описания синтаксического анализатора.
2. Выполнить lex для лексикографического описания.
3. Скомпилировать исходный код, сгенерированный yacc.
4. Скомпилировать исходный код, сгенерированный lex.
5. Скомпилировать любые другие необходимые модули.
6. Скомпоновать файлы от lex, yacc, и другие файлы с исходным кодом в исполняемый файл.

Для выполнения этих задач я предпочитаю использовать Makefile способом, показанным в листинге 12.

Листинг 12. Простой Makefile для lex/yacc

```
YFLAGS        = -d

PROGRAM       = calc

OBJS          = calcparse.tab.o lex.yy.o fmath.o const.o

SRCS          = calcparse.tab.c lex.yy.c fmath.c const.c

CC            = gcc

all:          $(PROGRAM)

.c.o:          $(SRCS)
              $(CC) -c $*.c -o $@ -O

calcparse.tab.c:        calcparse.y
              bison $(YFLAGS) calcparse.y

lex.yy.c:       lex.l
              flex lex.l

calc:          $(OBJS)
              $(CC) $(OBJS)  -o $@ -lfl -lm

clean:;         rm -f $(OBJS) core *~ \#* *.o $(PROGRAM) \
              y.* lex.yy.* calcparse.tab.*
```

Обмен данными с lex

Основным методом обмена данными между lex и yacc являются определения лексем. Эти определения генерируются при создании исходного C-кода из файла грамматического описания yacc и при создании заголовочного файла, который содержит макросы Си для каждой лексемы.

Однако если обмена лексемами недостаточно, нужно определить значение, которое будет хранить передаваемую информацию.

Сделать это можно в два шага. Первый шаг: согласно необходимым требованиям следует определить тип YYSTYPE. Для обмена данными между двумя системами YYSTYPE используется по умолчанию. Для простого калькулятора тип значения будет целочисленным (int; между прочим, этот тип используется по умолчанию), или float, или double. Если необходимо осуществлять обмен текстом, типом YYSTYPE будет символьный указатель или массив. Если необходимо обмениваться и числами и текстом, типом YYSTYPE будет union (этот тип может хранить оба типа значений).

Далее следует объявить переменную yylval (и, при необходимости, любую другую переменную, которую будут одновременно использовать lex и yacc) в описаниях yacc и lex; тип этой переменной - YYSTYPE.

Например, в заголовочном блоке обоих файлов (lex и yacc) определяется тип YYSTYPE (см. листинг 13).

Листинг 13. Определение типа

```
%{
...
#define YYSTYPE double
...
%}
```

В lex-описании также определяется внешняя переменная yylval этого типа:

```
extern YYSTYPE yylval;
```

И, наконец, эта переменная определяется в блоке пользовательской инициализации yacc-файла:

```
YYSTYPE yylval;
```

Теперь можно обмениваться данными между исходными кодами на C, сгенерированными обеими системами.

3 из **8** | предыдущая | следующая