Создание анализаторов текста при помощи yacc и lex

COLLABORATORS			
	TITLE : Создание анализаторов текста при помощи yacc и lex		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Мартин Браун	21 декабря 2012 г.	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Содержание

1 Прежде чем мы начнем **1**
1.1 Об этом учебном курсе . . . 1
1.2 Предварительные требования . . . 1

2 Лексический анализ и lex **1**
2.1 Что такое лексический анализ? . . . 2
2.2 Инструмент lex . . . 2
2.3 Создание исходного кода на языке Си . . . 3
2.4 Идентификация элементов . . . 4
2.5 Сложные лексемы . . . 4
2.6 Извлечение переменных данных . . . 5

3 Грамматический анализ с использованием yacc **5**
3.1 Простые грамматические правила в yacc . . . 5
3.2 Приоритеты грамматических правил . . . 6
3.3 Упорядочивание лексем по приоритету в грамматическом описании . . . 7
3.4 Пользовательская инициализация и ошибки . . . 8
3.5 Компилирование грамматического описания в файл на Си . . . 9
3.6 Обмен данными с lex . . . 10

4 Создание калькулятора **10**
4.1 Основы калькулятора . . . 10
4.2 Грамматический файл для калькулятора . . . 12
4.3 Настройка формата вывода . . . 15
4.4 Работа с регистрами . . . 15

5 Дальнейшее изучение lex и yacc **16**
5.1 Изучаем дальше калькулятор . . . 16
5.2 Обработка сложного текста . . . 17
5.3 Компиляторы языка программирования . . . 18

6 Заключение **19**

7 Ресурсы **19**

8 Об авторе **20**

Аннотация

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.

1 Прежде чем мы начнем

UNIX®-программистам часто требуется гибкий, но стандартизируемый формат для анализа текста и других структур данных. Используя инструменты lex и yacc, можно создать механизм синтаксического разбора текста (далее - анализатор), который будет анализировать текст согласно заданным правилам. Для достижения различных целей этот анализатор можно интегрировать в приложения, начиная с анализа задаваемой конфигурации и заканчивая созданием собственного языка программирования. К концу этого учебного курса читатель узнает, как выполнять распознавание лексем, как написать yacc-правила и как использовать механизм правил для создания и описания различных анализаторов и приложений.

1.1 Об этом учебном курсе

Для поиска и распознавания текста в UNIX существует много различных способов. Можно использовать grep, awk, Perl, и другие решения. Но иногда может возникнуть необходимость распознавать и извлекать данные в структурированном, но не строгом формате. В этом случае могут оказаться полезными UNIX-инструменты lex и yacc. Вышеупомянутые инструменты, awk, Perl, сама оболочка и множество других языков программирования используют lex и yacc для анализаторов, которые будут проводить синтаксический разбор текста и конвертировать его в информацию или структуру данных, которые нам нужны.

Lex - это инструмент для лексического анализа, который может использоваться для выделения из исходного текста определенных строк заранее заданным способом. Yacc - это инструмент для грамматического разбора; он читает текст и может использоваться для конвертирования последовательности слов в структурированный формат для дальнейшей обработки.

Вначале в этом учебном курсе объясняется использование lex и yacc для создания калькулятора. Далее, используя этот калькулятор в качестве примера, подробно рассмотрена информация, создаваемая и выводимая системами lex и yacc, и показано, как использовать эту информацию для анализа других типов информации.

1.2 Предварительные требования

Чтобы работать с примерами в этом учебном курсе, понадобятся следующие утилиты:

Lex Эта утилита является стандартным компонентом большинства ОС UNIX. Инструмент GNU flex обеспечивает такую же функциональность.

Yacc Эта утилита является стандартным компонентом большинства ОС UNIX. Инструмент GNU bison обеспечивает такую же функциональность.

Компилятор языка C Подойдет любой стандартный компилятор языка C, включая Gnu CC.

Make Этот инструмент требуется для использования обычного Makefile (чтобы упростить сборку приложения).

GNU-инструменты могут быть загружены с Web-сайта GNU или с его местного зеркала.

2 Лексический анализ и lex

Первое, что надо сделать при написании текстового анализатора - это реализовать возможность определения типа читаемых данных. Сделать это можно множеством различных способов, простейшим из которых является применение lex-инструмента, который конвертирует входную информацию в последовательность лексем.

2.1 Что такое лексический анализ?

Задумывались ли вы, когда писали программу на каком-либо языке или вводили команду в командную строку, каким образом конвертируются введенные символы в набор инструкций?

Процесс одновременно и прост, и сложен. Сложен потому, что существует бесконечный массив возможных комбинаций и последовательностей информации, которую можно ввести. Например, чтобы выполнить итерацию по хэш-таблице в языке Perl, можно использовать программный код из листинга 1.

```
foreach $key (keys %hash)
{
...
}
```

Каждый из элементов этого кода имеет разное значение, и это при том что команда очень проста. Для выражения в листинге 1 есть определенные синтаксические правила, так же как и в человеческих языках. Поэтому, если разбить на части введенный программный код и структурировать эту информацию, то выполнить синтаксический разбор содержимого будет достаточно легко.

Распознавание информации анализатором происходит в два этапа. На первом этапе надо только определить, что было напечатано или передано приложению. Необходимо уметь идентифицировать ключевые слова, фразы, последовательности символов из входного потока так, чтобы можно было определить, что с ними нужно делать. Второй этап состоит в осмысливании структуры поставляемой информации - входные данные нужно проверить и обработать. Использование круглых скобок в большинстве языков программирования является прекрасным примером грамматического анализа. Очевидно, что следующий программный код неверен:

```
{ function)( {
```

Фигурные скобки не согласованы, круглые скобки расположены в неправильном порядке. Чтобы анализатор смог осмыслить и распознать это выражение, он должен знать правильные последовательности и что делать, когда он распознает соответствующую последовательность.

Лексический анализ начинается с процесса идентификации входных данных и может быть реализован при помощи инструмента lex.

2.2 Инструмент lex

Инструмент lex (или GNU-инструмент flex) использует конфигурационный файл для создания исходного кода на C, из которого можно затем создать либо отдельное приложение, либо встроить этот исходный код в другое приложение. Конфигурационный файл определяет набор символов, ожидающихся в файле, который будет анализироваться, и какие действия надо выполнить, когда файл будет проанализирован. Формат файла является открытым; надо только распознать лексемы во входном файле и определить, что нужно сделать при их обнаружении (два этих элемента отделяются друг от друга запятыми или символом табуляции). Например:

```
sequence do-something
```

Листинг 2 показывает очень простое описание, которое принимает слова и выводит строку, основываясь на обнаруженном слове.

```
%{
#include <stdio.h>
%}

%%
```

```
begin  printf("Started\n");
hello  printf("Hello yourself!\n");
thanks printf("Your welcome\n");
end    printf("Stopped\n");
%%
```

Первый блок, ограниченный %{...%}, определяет текст, который будет вставлен в создаваемый исходный код на C. В этом случае, поскольку в примерах далее будет использован метод printf(), включен заголовок stdio.h.

Второй блок, ограниченный последовательностями %%, содержит описания идентифицируемых строк и действие, выполняемое при их обнаружении. В этом примере для простого слова печатается соответствующее сообщение.

2.3 Создание исходного кода на языке Си

Чтобы создать исходный код на Си, который будет анализировать некоторый входной текст, выполните для файла команду lex (или flex), как показано в листинге 1. Lex/flex-файлы имеют расширение "l", поэтому упомянутый выше файл может называться exampleA.l. Для создания исходного кода на Си необходимо выполнить команду:

```
$ flex exampleA.l
```

Вне зависимости от использованного инструмента выходной файл будет называться lex.yy.c. Изучение содержимого этого файла не для слабонервных: процесс, реализующий анализатор, достаточно сложен и базируется на сложной системе синтаксического анализа (использующей таблицу), которая сопоставляет входной текст с описаниями, заданными в lex. Из-за этого сопоставления анализ очень расходует память, особенно для больших и сложных файлов.

К преимуществам flex перед lex относятся дополнительные опции, разработанные для увеличения производительности (использования памяти или скорости работы), опции отладки и улучшенное управление сканером поведения (например, для игнорирования некоторых случаев). При помощи опции -l (для flex) можно создать исходный код на C, который близок к коду, генерируемому утилитой lex.

Теперь, когда получен исходный код на C, можно скомпилировать его в приложение для проверки процесса:

```
$ gcc -o exampleA lex.yy.c -lfl
```

Библиотека flex (подключается опцией -lfl, или опцией -ll для lex) содержит простой метод main(), который выполняет код, анализирующий текст. Когда созданное приложение запустится, оно начнет ожидать ввода. Листинг 3 показывает входные (и выходные) данные приложения.

```
$ exampleA
begin
Started

hello
Hello yourself!

end
Stopped

thanks
Your welcome

hello thanks
Hello yourself!
 Your welcome
```

```
hello Robert
Hello yourself!
 Robert
```

На простую входную строку ("begin") приложение реагирует выбранной командой (в этом случае выводит "Started"). Для многословных строк, в которых слова нужно распознать, приложение выполняет обе команды, отделяя друг от друга пробелом результаты их работы. Нераспознанные последовательности (включая те, которые содержат пробелы) в неизменном виде возвращаются назад

Данный пример показывает основные операции системы (использовались только стандартные слова). Для идентификации других комбинаций, как то символы и последовательности символов (элементы), доступен широкий диапазон различных решений.

2.4 Идентификация элементов

Идентифицируемые элементы не обязательно должны быть фиксированными строками (как в приведенном выше примере). Идентифицирующий механизм поддерживает регулярные выражения и специальные символы (например, знаки пунктуации), как показано в листинге 4.

```
%{
#include <stdio.h>
%}

%%
[a-z]    printf("Lowercase word\n");
[A-Z]    printf("Uppercase word\n");
[a-ZA-Z] printf("Word\n");
[0-9]    printf("Integer\n");
[0-9.]   printf("Float\n");
";"      printf("Semicolon\n");
"("      printf("Open parentheses\n");
")"      printf("Close parentheses\n");
%%
```

Примеры в листинге 4 должны быть понятными. Те же самые принципы могут использоваться для любых регулярных выражений или специальных символов, которые необходимо учитывать при синтаксическом анализе.

2.5 Сложные лексемы

В примерах, рассмотренных ранее, создавался код на Си, который рассматривал все слова по отдельности. Применение подобного подхода допустимо, однако для синтаксического разбора текста и других элементов, содержащих словосочетания, фразы, предложения, строящиеся по определенному правилу, рассмотренный выше метод анализа плохо подходит.

Для анализа предложения при таком подходе потребуется грамматический анализатор - программа, которая смогла бы распознать последовательность лексем. Но грамматический анализатор должен знать, какие лексемы следует ожидать. Для вывода распознанной лексемы в описание lex надо внести некоторые изменения: реакция на обнаружение лексемы изменяется с вывода какой-то сторонней строки на вывод этой же лексемы. Например, внесем изменения в исходный пример, чтобы тот выглядел как код из листинга 5.

```
%{
#include <stdio.h>
#include <y.tab.h>
%}
```

```
%%
begin  return BEGIN;
hello  return HELLO;
thanks return THANKS;
end    return END;
%%
```

Теперь вместо вывода строки при обнаружении лексемы "hello" будет возвращаться имя лексемы. Это имя будет использоваться в yacc для создания грамматического анализатора.

В этом примере имена лексем явно не определены. Они задаются в файле y.tab.h, который автоматически создается утилитой yacc при анализе грамматического yacc-файла.

2.6 Извлечение переменных данных

Если необходимо извлечь значение (например, нужно считать число или строку), тогда необходимо определить, как входные данные будут переконвертированы в данные требуемого типа. Это не очень нужно при работе в рамках lex, но крайне важно при создании полнофункционального анализатора с помощью инструмента yacc.

Для обмена данными обычно используются две переменные: переменная yytext хранит неформатированные данные, прочитанные lex при синтаксическом анализе, тогда как yylval используется для обмена действительными значениями между двумя системами. Более подробно методика работы этой интеграции будет рассмотрена далее в учебном курсе; для идентификации информации следует использовать lex-описание, подобное следующему:

```
[0-9]   yylval=atoi(yytext); return NUMBER;
```

В упомянутой выше строке кода переменной yylval присваивается значение, которое получено путем конвертирования фрагмента текстовой строки (удовлетворяющего условиям регулярного выражения) в целое число стандартной функцией atoi(). Следует заметить, что кроме конвертирования значения возвращается его тип. Это нужно для того чтобы yacc знал, какой тип значения использовался и мог использовать лексему в своих описаниях.

Давайте рассмотрим, как yacc описывает свои грамматические структуры.

3 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.

3.1 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.

```
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).

```
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.

3.2 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.

```
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

3.3 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.

```
%{
/* требуемыеглобальныеизаголовочныеопределения     */
%}

/* описаниядополнительные ( определениятокенов ) */

%%
/* определениенаборовправилдляанализа
%%

/* дополнительныйисходныйкодC    */
```

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.

```
%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.

3.4 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.

```
#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.

3.5 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 листинге12.

```
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.*
```

3.6 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).

```
%{
...
#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.

4 Создание калькулятора

Используя методики, показанные в предыдущем разделе, можно создать калькулятор, который работает с символьными выражениями.

4.1 Основы калькулятора

Давайте создадим программу, способную обрабатывать выражения (см. листинг 14)

```
4+5
(4+5)*6
2^3/6
sin(1)+cos(PI)
```

Определив дополнительные лексемы и грамматические правила для пользовательских функций и равенств (даже не входящих в язык Си и библиотеку Math), можно расширить функциональность калькулятора.

Чтобы реализовать это расширение, нужно для начала описать лексемы, которые будут идентифицированы инструментом lex. Полное lex-описание показано в листинг 15.

```
%{
#define YYSTYPE double
#include "calcparse.tab.h"
#include <math.h>
extern double yylval;
%}
D       [0-9.]
%%
[ \t]   { ; }
log      return LOG;
pi      return PIVAL;
sin     return SIN;
cos     return COS;
tan     return TAN;
and     return AND;
not     return NOT;
xor     return XOR;
or      return OR;
reg     return REGA;
ans     return ANS;
fix     return FIX;
sci     return SCI;
eng     return ENG;
const   return CONST;
bintodec return BINTODEC;
dectobin return DECTOBIN;
{D}+    { sscanf( yytext, "%lf", &yylval ); return NUMBER ; }
[a-zA-Z_]+ return IDENT;
"["     return OPENREG;
"]"     return CLOSEREG;
"<<"    return LEFTSHIFT;
">>"    return RIGHTSHIFT;
"++"    return INC;
"--"    return DEC;
"+"     return PLUS;
"-"     return MINUS;
"~"     return UNARYMINUS;
"/"     return DIV;
"*"     return MUL;
"^"     return POW;
"!"     return FACT;
"("     return OPENBRACKET;
")"     return CLOSEBRACKET;
"%"     return MOD;
"^^"     return XOR;
"!!"     return NOT;
"="     return ASSIGN;
"&&"    return LAND;
"||"    return OR;
"|"     return IOR;
"&"     return AND;
"~~"     return COMPLEMENT;
"\n"    return EOLN;
```

В этом описании присутствует большое количество лексем, причем смысл многих из них ясен из их же названий. В эти лексемы входят основные математические операции, некоторые функции (sin, cos и тому подобные) и логические операторы. Следует отметить два момента. Первый - для значений используется тип double, для преобразования строк чисел и десятичной точки в подходящее double-значение используется функция sscanf().

4.2 Грамматический файл для калькулятора

Основываясь на лексемах, описанных в предыдущем разделе, вводятся грамматические правила, согласно которым будет производиться синтаксический анализ текста. Полный исходный код грамматического анализатора показан в листинге 16. Давайте поподробнее взглянем на некоторые моменты и на то, как работает система.

```
%{
#include <alloca.h>
#include <math.h>
#include <stdlib.h>
#include <stddef.h>
#include <ctype.h>
#define YYSTYPE double
double calcfact();
double reg[99];
double ans;
char format[20];
%}

%token NUMBER SPACE MOD RIGHTSHIFT LEFTSHIFT SEMICOLON SIN EOLN PIVAL
%token PLUS MINUS DIV MUL POW OPENBRACKET CLOSEBRACKET UNARYMINUS
%token COS TAN ASIN ACOS ATAN FACT INC DEC LAND OR COMPLEMENT
%token NOT XOR ASSIGN IOR AND OPENREG CLOSEREG REGA ANS FIX SCI ENG
%token CONST
%left PLUS MINUS
%left MUL DIV
%left UNARYMINUS
%left LAND OR XOR NOT AND IOR
%left LOG
%%
list:   /* пусто */
        | list EOLN
        | list expr EOLN
          { printf( format , (double) $2 ); ans=$2; }
        ;
expr:   conditional_expr
        ;
conditional_expr: logical_or_expr
        ;
logical_or_expr: logical_and_expr
        | logical_or_expr OR logical_and_expr
          { $$ = (int) $1 || (int) $3; }
        ;
logical_and_expr: inclusive_or_expr
        | logical_and_expr LAND inclusive_or_expr
          { $$ = (int) $1 && (int) $3; }
        ;
inclusive_or_expr: exclusive_or_expr
        | inclusive_or_expr IOR exclusive_or_expr
          { $$ = (int) $1 | (int) $3; }
        ;
exclusive_or_expr: and_expr
        | exclusive_or_expr XOR and_expr
```

```
          { $$ = (int) $1 ^ (int) $3; }
        ;
and_expr: shift_expr
        | and_expr AND shift_expr
          { $$ = (int) $1 & (int) $3; }
        ;
shift_expr: pow_expr
        | shift_expr LEFTSHIFT pow_expr
          { $$ = (int) $1 << (int) $3; }
        | shift_expr RIGHTSHIFT pow_expr
          { $$ = (int) $1 >> (int) $3; }
        ;
pow_expr: add_expr
        | pow_expr POW add_expr { $$ = pow($1,$3); }
        ;
add_expr: mul_expr
        | add_expr PLUS mul_expr  { $$ = $1 + $3; }
        | add_expr MINUS mul_expr { $$ = $1 - $3; }
        ;
mul_expr: unary_expr
        | mul_expr MUL unary_expr { $$ = $1 * $3; }
        | mul_expr DIV unary_expr { $$ = $1 / $3; }
        | mul_expr MOD unary_expr { $$ = fmod($1,$3); }
        ;
unary_expr: assign_expr
        | MINUS primary %prec UNARYMINUS { $$ = -$2; }
        | INC unary_expr { $$ = $2+1; }
        | DEC unary_expr { $$ = $2-1; }
        | NOT unary_expr { $$ = !$2; }
        | LOG unary_expr { $$ = log($2); }
        ;
assign_expr: postfix_expr
        | REGA OPENREG expr CLOSEREG ASSIGN postfix_expr
          { reg[(int)$3]=$6; $$=$6; }
        | REGA OPENREG expr CLOSEREG
          { $$=reg[(int)$3]; }
        | REGA
          { int i;
            for(i=0;i<100;i++)
              if (reg[i]!=0)
                printf("%02d = %.2f\n",i,reg[i]);
            $$=0;
          }
        ;
postfix_expr: primary
        | postfix_expr INC { $$ = $1+1; }
        | postfix_expr DEC { $$ = $1-1; }
        | postfix_expr FACT
          { $$ = calcfact((unsigned long int)$1); }
        ;
 primary: NUMBER { $$ = $1; }
        | PIVAL { $$ = M_PI; }
        | OPENBRACKET expr CLOSEBRACKET { $$ = $2; }
        | ANS { $$ = ans; }
        | CONST OPENBRACKET expr CLOSEBRACKET { $$ = constval($3); }
        | set_format
        ;
set_format: function_call
        | FIX OPENBRACKET expr CLOSEBRACKET
             { sprintf(format,"%%.%df\n",(int)$3); $$=0; }
        | FIX { sprintf(format,"%%f\n"); $$=0; }
        | SCI OPENBRACKET expr CLOSEBRACKET
```

```
               { sprintf(format,"%%.%dg\n",(int)$3); $$=0; }
        | SCI { sprintf(format,"%%g\n"); $$=0; }
        | ENG OPENBRACKET expr CLOSEBRACKET
               { sprintf(format,"%%.%de\n",(int)$3); $$=0; }
        | ENG { sprintf(format,"%%e\n"); $$=0; }
        ;
function_call: SIN OPENBRACKET expr CLOSEBRACKET
                { $$ = (cos($3)*tan($3)); }
        | COS OPENBRACKET expr CLOSEBRACKET
               { $$ = cos($3); }
        | TAN OPENBRACKET expr CLOSEBRACKET
               { $$ = tan($3); }
        | ASIN OPENBRACKET expr CLOSEBRACKET
               { $$ = asin($3); }
        | ACOS OPENBRACKET expr CLOSEBRACKET
               { $$ = acos($3); }
        | ATAN OPENBRACKET expr CLOSEBRACKET
               { $$ = atan($3); }
        ;
%%

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

main( argc, argv )
char *argv[];
{
  progname = argv[0];
  strcpy(format,"%g\n");
  yyparse();
}

yyerror( s )
char *s;
{
  warning( s , ( char * )0 );
  yyparse();
}

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

В этом приложении присутствуют объявления трех глобальных структур данных:

- Массив reg используется в качестве регистра основной памяти, в который можно помещать значения и результаты вычислений.
- Переменная ans содержит значение последнего вычисления.
- Переменная format предназначена для хранения формата вывода, который используется при выводе результатов на экран.

Результаты вычисления печатаются только тогда, когда во входных данных содержится символ конца строки (end-of-line), наличие которого определяется при помощи лексемы EOLN. Благода-

ря этому длинное выражение может быть введено в одну строку. Содержимое длинного выражения будет проанализировано прежде, чем выведутся какие-либо результаты. Формат, в котором выводятся данные, хранится в глобальной переменной format, а выводимый результат хранится в ans.

В остальной части анализатора содержатся методы для распознавания фрагментов выражения и вычисления их значения, которые выполняются до тех пор, пока все выражение не будет обработано.

4.3 Настройка формата вывода

Формат вывода является всего лишь строкой форматирования для printf(). Изменить его можно вызовом функции fix. С помощью этой функции можно изменить точность и формат вывода. Например, чтобы выводить только три знака после десятичной запятой, можно использовать fix(3); чтобы вернуть формат вывода в значение по умолчанию, следует использовать fix().

Результат изменения формата вывода показан в листинге 17.

```
$ calc
1/3
0.333333
fix(3)
0.000
1/3
0.333
```

Поскольку строка формата является глобальной и результаты распечатываются по ходу выполнения каждого правила, строку формата можно использовать для контроля выходных данных.

4.4 Работа с регистрами

Регистр по существу является массивом значений с плавающей точкой, к которому можно обратиться при помощи числовой ссылки. Работа с регистрами выполняется во фрагменте кода, приведенном в листинг 18.

```
assign_expr: postfix_expr
        | REGA OPENREG expr CLOSEREG ASSIGN postfix_expr
          { reg[(int)$3]=$6; $$=$6; }
        | REGA OPENREG expr CLOSEREG
          { $$=reg[(int)$3]; }
        | REGA
          { int i;
            for(i=0;i<100;i++)
              if (reg[i]!=0)
                printf("%02d = %.2f\n",i,reg[i]);
            $$=0;
          }
        ;
```

Первое правило позволяет выполнять присваивание - анализатор разрешает использовать выражение для обращения к регистру. Но значение этого выражения должно соответствовать postfix_expr. Таким образом, это правило ограничивает значение, которое можно присвоить регистру (см. листинг 19).

```
reg[0]=26
26
reg[0]
26
```

```
reg[1]=(4+5)*sin(1)
7.57324
reg[1]
9
reg[4+5]=29
29
reg[9]
29
```

Заметьте, что выражение (4+5)*sin(1) возвращает правильный результат, но регистру присваивается только первая часть выражения. Это происходит потому, что правила позволяют выполнять присвоение только тех выражений, которые соответствуют правилу postfix_assign. Данное правило фактически является определяющим при работе с выражениями, содержащими круглые скобки. Поскольку вся строка не может быть присвоена регистру, программа анализирует первый фрагмент (4+5) и присваивает его значение регистру. Так как при присвоении регистру возвращается присвоенное значение, далее выполняются остальные вычисления и выводится корректный результат.

Существует простой прием для обхода этой проблемы - необходимо, чтобы выражение, значение которого должно быть присвоено регистру, было целиком заключено в круглые скобки:

```
reg[1]=((4+5)*sin(1))
```

Данная операция над выражением не потребует никаких изменений правил (поскольку анализатор остается тем же), но может потребоваться изменение документации.

Данный подход имеет еще одно преимущество - можно выполнять вложенные присвоения, поскольку при присвоении значения регистру возвращается само это значение. Поэтому следующее выражение будет корректно обработано (см. листинг 20).

```
reg[1]=(45+reg[2]=(3+4))
52
reg[1]
52
reg[2]
7
```

5 Дальнейшее изучение lex и yacc

При рассмотрении принципа работы калькулятора были затронуты основные моменты работы с yacc и lex, но возможности этих инструментов гораздо шире.

5.1 Изучаем дальше калькулятор

Правила, которые обрабатывают входные данные и выполняют на основе результатов этой обработки какие-то действия, могут устанавливаться и контролироваться по отдельности. Поэтому можно изменить способы извлечения и обработки информации (независимо один от другого).

Когда я впервые "познакомился" с lex и yacc, моей целью было создать калькулятор с обратной польской записью (Reverse Polish Notation (RPN)). В RPN сначала вводятся числа, а затем оператор желаемого действия. Например, для сложения двух чисел следует использовать: 4 5 +.

Для некоторых пользователей такая запись более понятна, чем стандартная, особенно если они узнали о подобной форме записи при изучении сложения в школе:

```
4
5 +
----
9
----
```

В результатах преобразования примечательно то, что порядок расстановки приоритетов (который определялся в правилах анализатора, yacc-описание) приводит к RPN-калькулятору, который обрабатывает пример, приведенный выше.

Это настолько эффективно, что если подать выходные данные инструмента equtorpn в RPN-калькулятор, то получится такой же результат:

```
$ equtorpn|rpn
4+5*6
34
```

Для загрузки полных примеров и программных кодов приложений rpn, equtorpn и rpntoequ и дальнейшего изучения принципов их работы посетите Web-сайт MCslp Coalface (см. раздел Ресурсы).

5.2 Обработка сложного текста

Ранее в статье мы рассмотрели, как создать анализатор, преобразующий математическое выражение в такой формат, чтобы выражение можно было посчитать. Была показана важность последовательности размещения лексем, правил анализа и приоритетов. Те же самые правила можно применить к системам обработки текстовых данных. Но, если не вводить такие же строгие правила как в примере с калькулятором, разбор текста может оказаться весьма сложным.

Листинг 21 содержит lex-файл, который создает простой анализатор для чтения/записи данных, а листинг 22 содержит yacc-правила для самого анализа выходных данных

```
%{
#include <stdio.h>
#include "text.tab.h"
%}

%%
set       return SET;
state     return STATE;
mode      return MODE;
get       return GET;
[a-zA-Z]+  { yylval=strdup(yytext); return STRING; }
%%
```

```
%{
#include <stdlib.h>
#define YYSTYPE char *
char mode[20];
char state[20];
%}

%token SET STATE MODE GET STRING EOLN
%%
list:   /* ничего */
        | list getorset
        ;
getorset: getvalue
         | setvalue
```

```
         ;
setvalue:
        SET STATE STRING  { strcpy(state,$3); printf("State set\n"); }
        | SET MODE STRING { strcpy(mode,$3); printf("Mode set\n"); }
        ;
getvalue:
        GET STATE  { printf("State: %s\n",state); }
        | GET MODE { printf("Mode: %s\n",mode); }
        ;

%%

#include <stdio.h>
#include <ctype.ha>
char *progname;

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

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

В отличие от калькулятора, в котором входной текст преобразовывался в число, входную текстовую строку можно использовать без преобразования (см. листинг 23).

```
$ textparser
get state
 State:

set state bob
  State set

get state
 State: bob
```

Более сложные текстовые анализаторы построены на таких же базовых принципах и могут использоваться для любых целей: анализаторы конфигурации, создание своего собственного языка скриптов или альтернативного компилятора.

5.3 Компиляторы языка программирования

Компиляторы языков программирования (например, компилятор Си) принимают входные данные и конвертируют их в код ассемблера, который выполняется на компьютерной платформе. Вот почему (в первом случае) компиляторы являются платформенно-зависимыми, и вот почему сравнительно легко написать компилятор, который генерирует машинный код для альтернативной платформы (кросс-платформенность). Компилятор только лишь генерирует код для альтернативной платформы. Например, чтобы сложить два числа вместе - сгенерировать соответствующий код ассемблера для Intel x86, можно изменить правило, рассмотренное ранее:

```
add_expr: mul_expr
        | add_expr PLUS mul_expr
          { printf("MOV ax,%d\nADD ax,%d\n",$1,$3); }
```

Далее, для выполнения полного преобразования языка программирования в код ассемблера можно добавить в систему другие структуры типа циклов, функций, имен переменных.

6 Заключение

Используя lex и yacc, можно генерировать программный код, составляющий анализатор. Инструмент lex реализует возможности распознавания текстовых фрагментов и элементов, после чего возвращает назад идентифицированные лексемы. Инструмент yacc предоставляет методику обработки структур этих лексем. Эта методика основывается на ряде правил, согласно которым определяется формат входных данных и определяются действия, которые будут выполнены с распознанными последовательностями лексем. Оба инструмента используют конфигурационный файл, при обработке которого создается исходный код анализатора на Си.

В этом учебном курсе было рассмотрено, как использовать lex и yacc для создания простого анализатора для калькулятора. В этой статье была объяснена важность распознавания лексем, наборов правил и показано, как при помощи взаимодействия правил реализовать законченное решение для синтаксического анализа данных (применительно к калькулятору). Также было рассмотрено, как основные принципы создания анализаторов текста могут использоваться для других задач: создание текстового анализатора и создание своего собственного языка программирования.

7 Ресурсы

Научиться :

- Write text parsers with yacc and lex: оригинал руководства (EN).
- MCslp Coalface: сайт с дополнительными примерами и информацией, в частности, связанной с RPN и утилитами для конвертирования в RPN. (EN)
- Podcasts: аудиозаписи презентаций экспертов IBM.(EN)
- Новичок в AIX и UNIX?: страница AIX и UNIX для новичко
- AIXpert blog: этот блог является лучшим источником информации по UNIX.(EN)
- Jumpstart your Yacc… and Lex too! (EN) (developerWorks, ноябрь 2000): эта статья предоставляет руководство по правилам и доступным опциям для обоих инструментов.(EN)
- Lex & Yacc Page: сайт содержит ссылки на различные ресурсы. (EN)
- GNU: последние версии lex и yacc на сайте GNU.(EN)
- Using SDL, Part 4: lex and yacc of the Pirates Ho! (EN) (developerWorks, май 2000): статья о том, как использовать lex/yacc для анализатора текста конфигурационного файла. (EN)
- AIX и UNIX: в данном разделе сайта developerWorks размещена различная информация по всем аспектам системного администрирования AIX, которая поможет лучше изучить UNIX.
- Команда IBM developerWorks проводит по всему миру сотни бесплатных технических консультаций.(EN)
- Safari bookstore: сайт магазина книг по ИТ.(EN)

8 Об авторе

Мартин Браун (Martin Brown) пишет статьи уже более семи лет. Он является автором многочисленных книг и статей по различным темам. Его квалификация охватывает множество платформ и языков разработки - Perl, Python, Java™, JavaScript, Basic, Pascal, Modula-2, C, C++, Rebol, Gawk, Shellscript, Windows®, Solaris, Linux, BeOS, Mac OS X и т.д., а также Web-программирование, системное управление и интеграция. Мартин является внутренним экспертом (SME) компании Microsoft® и регулярно пишет для ServerWatch.com, LinuxToday.com и IBM developerWorks. Он также принимает участие в блогах Computerworld, The Apple Blog и на других сайтах. Связаться с ним можно через его Web-сайт.