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

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.

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.

Листинг 1. Итерация по хэш-таблице в Perl

```
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.

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.

Листинг 2. Простое lex-описание

```
%{
#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.

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.

Листинг 3. Входные и выходные данные простого приложения lex

```
$ 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.

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.

Листинг 4. Регулярные выражения и специальные символы

```
%{
#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.

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.

Листинг 5. Возврат лексем

```
%{
#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.

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.