Проектирование компиляторов. Лабораторная работа №2. Создание синтаксического анализатора.

МАИ, 24.03.2023

1 Введение

Для упрощения и автоматизации создания синтаксических анализаторов (парсеров) был разработан специальный класс программ, называемых «генераторы синтаксических анализаторов». Программа `bison` является классическим примером генератора синтаксических анализаторов. На основе грамматики, заданной в ФБН (Форма Бэкуса-Наура) она создаёт модуль на языке C, выполняющий разбор входящего потока токенов. Функция считывания входного потока может быть как написана вручную, так и основываться на лексическом анализаторе, созданном программой `flex`.

2 Задание

На языке C11 реализовать синтаксический анализатор, допускающий корректные выражения для калькулятора с функцией запоминания значений. Анализатор следует создать при помощи программы `bison`. В качестве лексического анализатора необходимо использовать сканер, созданный в рамках лабораторной работы №1. Если подаваемая на вход парсеру программа корректна, то он возвращает 0. При наличии ошибки парсер выводит сообщение с описанием ошибки. Код возврата в этом случае будет ненулевой.

2.1 Описание входного языка

Предложенный язык калькулятора представляет из себя арифметичекие выражения с операциями и данными, определёнными в лр. №1. Каждое выражение из которых записано на новой строке. Выражение может либо состоять из вычислений, в которых участвуют литералы, переменные и вызовы функций, либо представлять из себя запись, присваивающую значение переменной.

2.2 Описание входных данных

На вход программе подаётся название файла с распознаваемым текстом. Если файл не подан, программа считывает данные со стандартного ввода. Подача нескольких файлов в качестве аргумента вызывает заверение программы с ошибкой некорректного использования.

Входной файл может содержать произвольные данные. Примеры входных данных:

Пример 1

```
(T && T) || !F
f = F
T || f
is_true(f)
```

Пример 2

```
(3 + 4) / 7
i = 1
pow(i, 3)
```

Пример 3

```
(0x1 | 0x2) & 0xF)
b = 0x7
bswap(b)
```

2.3 Описание выходных данных

Пример 1

```
$ echo $?
0
```

Пример 2

```
$ echo $?
0
```

Пример 3

```
In line 1: syntax error

$ echo $?
1
```

3 Критерии выполнения

Когда автор посчитает что его работа готова, он отправляет преподавателю архив, содержащий исходный код программы, Makefile и набор тестов, на котором синтаксический анализатор отлаживался. Письмо должно содержать фамилию и имя автора, тема должна содержать слова «МАИ, Лабораторная работа 2». Архив упаковывается одним из следующих форматов: `.bz2`, `.tgz`, `.xz`. Письма, не выполняющие указанные требования, не рассматриваются.

4 Сроки

При сдаче работы начиная с 08.04.2023 максимально возможная оценка за лабораторную работу снижается на один балл в неделю.

5 Литература

- *А. В. Ахо, М. С. Лам, Р. Сети, Д. Д. Ульман* Компиляторы: принципы, технологии и инструменты.
- CS 143 — Compilers, Stanford University

6 Автор

Задание предоставлено ассистентом кафедры вычислительной математики и информатики факультета прикладной математики и физики Московского авиационного института Маркиным Алексеем Леонидовичем в 2023 году.

7 Справочное приложение

Полную справку по работе генератора синтаксических анализаторов bison можно получить введя команду info bison в своей операционной системе. Ниже представлена краткая справка по работе с bison, содержащая материалы, которые могут понадобиться для выполнения лабораторной работы.

7.1 Структура программы на bison

Программа на bison состоит из 4 основных секций:

```
%{
 Пролог
%}
Объявления
%%
Правила грамматики
%%
Эпилог
```

Пролог содержит код на языке C, который будет использоваться при обработке синтаксических правил.

Секция объявлений содержит объявления терминалов и нетерминалов грамматики, которые будут использованы при разборе входного потока.

В секции правил грамматики содержится непосредственно описание языка в формате БНФ.

Секция эпилога содержит определения процедур на языке C, которые будут использованы в программе.

7.1.1 Секция объявлений

Терминальные символы грамматики задаются при помощи директивы %token token_name. Например чтобы задать токен целочисленного типа необходимо написать:

```
%token TOK_INT
```

Нетерминальные символы задаются директивой %type token_name. Например для того чтобы задать нетерминальный символ выражения, необходимо написать:

```
%type exp
```

Секция объявлений также служит для задания ассоциативности приоритетов токенов:

```
%left '+'
%left '*'
%precedence TOK_UMIN
```

В данном случае чем выше находится объявление, тем ниже приоритет соответствующего токена. Директива `%left` служит для задания ассоциативности токена, а директива `%precedence` служит только для задания приоритета.

7.1.2 Секция правил грамматики

Правила грамматики задаются в БНФ (Форма Бэкуса-Наура) и выглядят следующим образом:

```
нетерминал: правило_1 {действия}
          | правило_2 {действия}
          ...
;
```

С левой стороны находятся нетерминалы, с правой — правила, по которым они раскрываются. В правилах могут находиться терминалы, нетерминалы или пустые символы:

```
input:
  %empty
| input line
;
```

Помимо непосредственно терминалов и нетерминалов, в правилах могут присутствовать символьные литералы:

```
line:
  '\n'           {  }
| exp '\n'  { printf ("Got exp\n"); }
| error '\n'      { yyerrok; }
;
```

Видно что данная грамматика допускает символ переноса строки, более того он является символом завершения выражения. В фигурных скобках можно указывать различные действия, выполняемые при распознавании данного правила. В рамках данной л.р. это можно использовать для отладки.

Возможность дописать правила для нетерминала `exp` предоставляется читателю.

7.2 Входной поток

Программа, сгенерированная `bison` получает следующую лексему при помощи функции `int yylex (void);`. Возвращаемым значением является код токена. Данная функция может быть реализована программистом как вручную, так и с применением генератора лексических анализаторов `flex`.

Запуск анализа производится вызовом процедуры `int yyparse(void)`.

7.3 Взаимодействие с flex

Для того чтобы синтаксический анализатор, сгенерированный при помощи `bison` мог взаимодействовать с лексическим анализатором, сгенерированным `flex`, необходимо чтобы последний мог возвращать коды считанных токенов. Для этого в секции определений программы на `flex` необходимо подключить заголовочный файл, сгенерированный `bison`. Он будет содержать определения токенов. После этого при распознавании лексемы можно будет использовать конструкции вида:

```
return TOK_INT;
```

Таким образом, правило распознавания целого числа принимает следующий вид:

```
{DIGIT} { return TOK_DIGIT; }
```

Если токен является одним символом, то его можно передавать в качестве возвращаемого значения:

```
[+*\n] { return yytext[0]; }
```

7.4 Сборка программы на bison

В модуле синтаксического анализа определяются классы токенов, и для того чтобы модуль лексического анализа мог корректно их возвращать, необходимо подключить заголовочный файл с описанием этих классов. Сделать это можно в секции кода, файл будет называться `grammar.h`.

Аналогичным образом, необходимо сообщить модулю синтаксического анализа сигнатуры функций считывания токенов, располагающихся в модуле лексического анализа. Для этого в секции кода необходимо подключить заголовочный файл `lexer.h`, который указан в опциях создания модля.

Последним действием будет являться создание функции `yyerror` в файле разбора грамматики. Сделать это необходимо, т.к. генератор не создаёт её самостоятельно. Для этого в секции кода создадим пустую функцию:

```
void yyerror(char const * msg)
{
}
```

Для получения исполняемого файла необходимо сначала транслировать файл из формата `bison` в файл на языке C:

```
$ bison grammar.y --defines=grammar.h -o grammar.c
```

В данной команде в качестве входного файла подаётся `grammar.y`. Опция `-o` задаёт выходной файл. Опцией `--defines=` задаётся название генерируемого заголовочного файла. Он нужен в случае если определения грамматики будут использоваться внешним модулем, например программой, написанной на `flex`.

После трансляции синтаксического анализатора в файл на языке C, можно его скомпилировать и скомпоновать с модулем лексического анализа:

```
$ gcc grammar.c lexer.c -o parser
```