第二周实验课 Lex + Yacc入门

Flex（Lex）和Bison（Yacc）是一套代码生成器，可生成C代码。该C代码的输入是目标语言的源代码，会按你写的规则来解析源代码并构建出抽象语法树（Abstract Syntax Tree，AST）。

词法分析的文件是 `lexer.lex`，语法分析的文件是 `parser.yacc`。虽然编译的步骤是先进行词法分析，但编写解析规则时却不是先写 `lexer.lex`。

这两个文件都分为三个部分，两个部分之间用两个百分号 `%%` 隔开：

```
声明部分
%%
规则部分
%%
辅助函数
```

本次实验的目标是实现一个简单的计算器，以整数加减法举例，我们使用以下文法：

```
exp -> term | exp + exp | exp - term
term -> number
number -> [0-9]+
```

Step 1 确定终结符、非终结符 、token

根据文法，可以确定终结符与非终结符。对于上述文法，整数和运算符是终结符，`exp`、`term`、`number`是非终结符。

`lexer.lex`以终结符为输入，向`parser.yacc`返回`token`，可以把`token`理解为"词"。`parser.yacc`则以`token`为输入，根据每个非终结符的规则进行解析，最后构建出 AST。

Step 2 yacc 声明

`token`在`parser.yacc`的声明部分定义，因此应该先写yacc的声明部分，再写lex，并且`lexer.lex`的开头也要导入yacc编译出的`y.tab.h`库。

给 token 和非终结符的属性值定义类别，可以是数字类或抽象类（自定义的类），单个字符可以也不定义而直接作为 char 返回。

将token的名字和对应属性值的类别名按以下的格式写入 `parser.y` 的声明部分：

```
%union {
    double floatval;
    int val;
}

// 终结符的类
%token <val> NUMBER
%token ADD SUB
%token EOL

// 非终结符的类
%type <val> term exp
```

Step 3 编写 lexer 规则

在 `lexer.lex` 的规则部分添加 lexer 规则。

每条规则的形式：`正则表达式 { c 代码 }`

这部分的 c 代码一般做两件事：

- 对 `yylval` 进行赋值。`yylval` 的类型是上面写的union，用来记录这个 token 的属性值。
- `return token_name;` 返回token的名字。

你可以在用户自定义函数部分定义新的函数，并在规则定义处使用。

```
%{
# include "test.tab.h"
void yyerror(char*);
%}

%%
"+" { return ADD; }
"-" { return SUB; }
[0-9]+  { yylval.val = atoi(yytext); return NUMBER; }

[ \t]   { /* ignore white space */ }
.   { yyerror("Unknown character!\n"); }
\n      { return EOL; }
%%

//用户自定义函数
```

需要注意：

- lex 规则有优先级：优先匹配最长的，再匹配靠前的规则。

Step 4 编写 yacc 规则

在 `parser.y` 的规则部分添加 yacc 规则。

yacc 规则的形式是：`非终结符名字 : 导出式右部 { 带变量 $$ 和 $1 的 c 代码} | 更多的导出式右部与处理`

其中：

- `$$` 表示返回值，没有显式的 `return` 语句。返回值类型需与非终结符记号属性的类型一致。
- `$1` 表示导出式的第几部分，从1开始编号。

`parser.yacc` 代码续：

```
%%
calclist:
    | calclist exp EOL { printf("= %d\n> ", $2);}
    | calclist EOL { printf("> "); }
    ;

exp: term
    | exp ADD exp { $$ = $1 + $3; }
    | exp SUB term { $$ = $1 - $3; }
    ;
```

```
term: NUMBER
    ;
%%
```

Step 5 yacc库中其他接口

编译`parser.yacc`时，yacc会根据我们写的规则生成 C 语言的 parser 代码，其中可能涉及几个常用的 yacc 库接口。为了保证编译成功，我们需要在 yacc 的声明部分声明以下外部接口：

```
%{
#  include <stdio.h>
void yyerror(char *s);
int yylex();
%}
```

再在结尾的辅助函数部分增加以下代码：

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

Step 6 编译parser和lexer

```
yacc -d parser.yacc  # 用yacc生成parser代码
bison -d parser.yacc
yacc --help          # 查看yacc使用帮助
lex lexer.lex        # 用lex生成lexer代码，需要确保lexer中已经添加了parser库
flex lexer.lex
```

Step 7 使用这些生成的代码

在主程序中调用`yyparse()`。`yyparse()`从`stdin`中读取源代码，然后执行 lexer 和 parser 的工作。

```
int main()
{
    printf("> ");
    yyparse();
}
```

使用Make

Makefile如下定义：

```
test: test.l test.y
    bison -d test.y
    flex test.l
    cc -o $@ test.tab.c lex.yy.c -lfl

clean:
    rm -f test \
    lex.yy.c test.tab.c test.tab.h
```

在命令行中执行

```
make
```

```
make clean
```

Step 8 lex和yacc的debug

lex的debug可以在结尾增加这样一个通配的正则表达式，将未被识别的字符打印出来：

```
. {
    printf("Illegal input \"%c\"\n", yytext[0]);
}
```

yacc的debug可以使用自带的debug模式：

1. 在yacc开头的定义部分增加`extern int yydebug=1`
2. 运行`yacc a.yacc -d -v --debug`，这一步相当于生成了一个debug版本的C代码，所以要回到正常模式，使用`yacc a.yacc`等语句正常编译，或删除yydebug标识重新编译即可。
3. 正常make