北京郵電大學

实验报告：语法分析程序的设计与实现

——YACC 实现

学院：计算机学院（国家示范性软件学院）

专业：计算机科学与技术

班级：2022211305

学号：2022211683

姓名：张晨阳

2024 年 11 月 18 号

目录

1.实验概述

1.1. 实验题目及要求

利用 YACC 自动生成语法分析程序，实现对算术表达式的语法分析。要求所分析算数表达式由如下的文法产生。

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow (E) \mid num$$

在对输入的算术表达式进行分析的过程中，依次输出所采用的产生式。

1.2. 实现方法要求

根据给定文法，编写 YACC 说明文件，调用 LEX 生成的词法分析程序。

1.3. 实验环境

- Windows 11
- Visual Studio Code
- WinFlexBison

2.程序设计说明

2.1. LEX 程序功能模块

2.1.1.功能概述

Lex 程序的主要作用是实现词法分析器，将输入的字符串拆分成符合语法分析器（YACC）要求的词法单元（tokens）。

在本程序中，Lex 文件的功能是识别算术表达式中的基本符号（如数字和操作符），并为语法分析器返回对应的标记（token）。

2.1.2.模块设计与实现

1. 文件头部声明

```
%option yylineno
%option noyywrap
%{
#include "parser.tab.h"
%}
```

- yylineno：启用行号跟踪。
- noyywrap：关闭默认的 yywrap 函数调用，避免未定义的函数错误。
- #include "parser.tab.h"：引入 YACC 自动生成的头文件，确保返回的 token 与 YACC 端匹配。

2. 词法规则

```
[0-9]+        { yylval = atoi(yytext); return NUM; }
"+"           { return '+'; }
"-"           { return '-'; }
"*"           { return '*'; }
"/"           { return '/'; }
"("           { return '('; }
")"           { return ')'; }
"$"           { return 0; }
[ \t\n]       ;
.             { printf("Unknown character: %s\n", yytext); }
```

- **数字匹配**：正则表达式 [0-9]+ 匹配一个或多个连续的数字，将其转换为整数后存储在 yylval 中，并返回 NUM 标记。
- **操作符和括号**：对 +, -, *, /, (,) 等符号直接返回对应字符作为 token。
- **输入结束符**：字符 $ 作为输入结束标记，返回 0。
- **忽略空白**：忽略空格、制表符和换行符。
- **错误处理**：对未知字符打印错误信息 Unknown character: ...

3. 功能实现流程

- 从输入流中逐字符读取。
- 根据定义的正则表达式，匹配输入中的数字、操作符或括号，并生成对应的 token。
- 如果无法匹配规则，输出未知字符错误信息。

2.2. YACC 程序功能模块

2.2.1.功能概述

YACC 程序的主要功能是实现语法分析器，对 Lex 提供的词法单元进行语法分析，并根据给定的文法规则验证输入的合法性。

在本程序中，YACC 文件实现了一个算术表达式解析器，使用如下文法规则：

$$E \rightarrow E + T \mid E - T \mid T$$

$$T \rightarrow T * F \mid T / F \mid F$$

$$F \rightarrow (E) \mid num$$

在解析过程中，程序会输出每次应用的产生式。如果解析成功，则输出 Accept!，否则输出 Reject!。

2.2.2.模块设计与实现

1. 文件头部声明

```
%{
#include <stdio.h>
#include <stdlib.h>
int yylex();
void yyerror(const char *s);
int step = 0; // 用于计数产生式的顺序
%}
```

- 引入标准输入输出和标准库。
- 声明 yylex() 函数，调用词法分析器。
- 定义 yyerror() 函数，用于语法错误处理。
- 声明 step 变量，用于计数每次应用的产生式顺序。

2. 终结符与非终结符

```
%token NUM
%start E
```

- 定义 NUM 作为终结符，代表数字。
- 定义 E 为起始符号。

3. 文法规则

```
E : E '+' T    { printf("(%d) E → E + T\n", ++step); }
  | E '-' T    { printf("(%d) E → E - T\n", ++step); }
  | T          { printf("(%d) E → T\n", ++step); }
  ;

T : T '*' F    { printf("(%d) T → T * F\n", ++step); }
  | T '/' F    { printf("(%d) T → T / F\n", ++step); }
  | F          { printf("(%d) T → F\n", ++step); }
  ;

F : '(' E ')'  { printf("(%d) F → (E)\n", ++step); }
  | NUM        { printf("(%d) F → num\n", ++step); }
  ;
```

3.1 表达式规则：

- E → E + T、E → E - T：匹配加法或减法操作的表达式。
- E → T：匹配基础表达式。

3.2 项规则：

- T → T * F、T → T / F：匹配乘法或除法操作的项。
- T → F：匹配基础项。

3.3 因子规则：

- F → (E)：匹配括号中的表达式。
- F → num：匹配数字。

4. 错误处理与主程序

```
int main() {
    printf("Enter an expression ending with $:\n");
    if (yyparse() == 0) {
        printf("\033[32mAccept!\033[0m\n");  // 绿色输出
    } else {
        printf("\033[31mReject!\033[0m\n");  // 红色输出
    }
    return 0;
}

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

- yyerror 函数：处理语法错误并打印错误信息。
- main 函数：
  - 调用 yyparse() 执行语法分析。
  - 如果成功，输出绿色 Accept!；否则，输出红色 Reject!。

3.测试设计与分析

3.1. 测试方法

1. 利用 win_bison 编译写好的.y 文件，生成对应的.h 和.c 代码

```
> ./win_bison -d parser.y
```

2. 利用 win_flex 编译写好的.l 文件，生成对应的.c 代码

```
> ./win_flex lexer.l
```

3. 将生成的 2 个.c 文件进行链接编译，生成可执行文件 parser.exe

```
> gcc parser.tab.c lex.yy.c -o parser
```

4. 执行 parser.exe 进行测试

```
> ./parser.exe
```

3.2. 测试 1+2$

测试目的:

验证程序基本功能是否正常工作，确保程序能够正确解析简单的加减法表达式。

输入：

```
1+2$
```

输出：

```
(base) PS E:\WILLIAMZHANG\Compilation\Labs\YACC\src> ./parser
Enter an expression ending with $:
1+2$
(1) F → num
(2) T → F
(3) E → T
(4) F → num
(5) T → F
(6) E → E + T
Accept!
```

测试结果分析：

- 首先将 1 识别为 num，匹配 F → num，输出 (1)。
- 将 F 归约为 T → F，输出 (2)。
- 将 T 进一步归约为 E → T，输出 (3)。
- 接着，将 2 识别为 num，匹配 F → num，输出 (4)。
- 再次归约 F → T，输出 (5)。
- 最后应用加法规则，将表达式归约为 E → E + T，输出 (6)。

3.3. 测试 3+5*(2-8)/4$

测试目的:

验证程序是否能够正确解析复杂表达式，支持嵌套括号、多级运算符优先级（乘除优先于加减），并生成完整的产生式。

输入：

```
3 + 5 * (2 - 8) / 4 $
```

输出：

```
(base) PS E:\WILLIAMZHANG\Compilation\Labs\YACC\src> ./parser
Enter an expression ending with $:
3 + 5 * (2 - 8) / 4 $
(1) F → num
(2) T → F
(3) E → T
(4) F → num
(5) T → F
(6) F → num
(7) T → F
(8) E → T
(9) F → num
(10) T → F
(11) E → E - T
(12) F → (E)
(13) T → T * F
(14) F → num
(15) T → T / F
(16) E → E + T
Accept!
```

测试结果分析：

- 将 3 识别为 num，匹配 F → num，输出 (1)。
- 将 F 归约为 T → F，输出 (2)。
- 将 T 进一步归约为 E → T，输出 (3)。
- 将 5 识别为 num，匹配 F → num，输出 (4)。
- 将 F 归约为 T → F，输出 (5)。
- 将 2 识别为 num，匹配 F → num，输出 (6)。
- 将 F 归约为 T → F，输出 (7)。
- 将 T 进一步归约为 E → T，输出 (8)。
- 将 8 识别为 num，匹配 F → num，输出 (9)。

- 将 F 归约为 T → F，输出 (10)。
- 应用减法规则，将表达式归约为 E → E - T，输出 (11)。
- 将括号内的表达式归约为 F → (E)，输出 (12)。
- 应用乘法规则，将 5 * (2 - 8) 归约为 T → T * F，输出 (13)。
- 将 4 识别为 num，匹配 F → num，输出 (14)。
- 应用除法规则，将 T → T / F，输出 (15)。
- 最终应用加法规则，将整体表达式归约为 E → E + T，输出 (16)。

3.4. 测试 3+*5$

测试目的:

验证程序的错误处理功能，确保能够识别并提示语法错误，同时正确输出 Reject!。

输入：

```
3 + * 5 $
```

输出：

```
(base) PS E:\WILLIAMZHANG\Compilation\Labs\YACC\src> ./parser
Enter an expression ending with $:
3 + * 5 $
(1) F → num
(2) T → F
(3) E → T
Error: syntax error
Reject!
```

测试结果分析：

- 将 3 识别为 num，匹配 F → num，输出 (1)。
- 将 F 归约为 T → F，输出 (2)。
- 将 T 归约为 E → T，输出 (3)。
- 当程序遇到 + * 时，检测到语法错误：
  - 根据文法规则，+ 后应接一个合法的 T，但此处直接出现了 *，违反文法定义。
  - 程序调用 yyerror 输出错误提示 Error: syntax error。
- 程序停止进一步解析，并输出 Reject!，表示输入表达式不合法。

4.心得总结

在本次项目中，通过设计和实现基于 LEX 和 YACC 的语法分析器，我们深入学习了编译原理中词法分析和语法分析的基本概念及其具体实现方法。

在实现过程中，我特别关注了程序的清晰性和可维护性，例如通过合理设计文法规则和适当的打印逻辑，使解析过程一目了然。此外，给输出结果添加颜色区分（绿色的 Accept! 和红色的 Reject!）不仅让程序更易用，也为测试和调试带来了便利。

然而，开发过程中也遇到了不少挑战。例如，刚开始时由于 Flex 默认会调用 yywrap 函数，导致链接错误。经过查阅资料，我尝试了多种方法，包括禁用 yywrap 和定义一个空实现，最终成功解决了问题。在复杂表达式解析上，调试时输出的产生式顺序有时与预期不符，需要对文法的优先级和递归规则进行细致推敲。

通过这次项目，我不仅巩固了基础知识，也对编译器设计中的实际问题有了更多的认识和解决能力。尽管程序已具备一定的功能，但我也在思考如何进一步扩展和优化。例如，可以支持更复杂的表达式，例如浮点数或变量赋值；同时加入错误恢复机制，使程序在检测到错误后依然能够继续分析剩余部分。

本次实践让我深刻感受到理论与实践结合的重要性。未来，我希望能够在此基础上进一步学习和拓展编译器的其他部分，例如语义分析或代码生成，为更复杂的编译任务打下坚实的基础。

这次项目让我获益匪浅，也增强了我对编译技术的兴趣和信心。