语法分析程序的设计与实现 YACC

杨晨
学号 2021212171
北京邮电大学计算机学院

日期：2023 年 11 月 15 日

1 概述

1.1 实验内容

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

$$E \to E+T|E-T|T$$
$$T \to T*F|T/F|F$$
$$F \to (E)|num$$

要求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。

实验要求和实现方法要求如下：

- 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。

1.2 开发环境

- Windows10
- Visual Studio Code
- Win flex-bison

2 LEX 程序的功能模块划分

2.1 常见的标记模式和正则表达式

```
O    [0-7]
D    [0-9]
NZ   [1-9]
L    [a-zA-Z_]
A    [a-zA-Z_0-9]
H    [a-fA-F0-9]
HP   (0[xX])
E    ([Ee][+-]?{D}+)
P    ([Pp][+-]?{D}+)
```

```
FS    (f|F|l|L)
IS    (((u|U)(l|L|ll|LL)?)|((l|L|ll|LL)(u|U)?))
CP    (u|U|L)
SP    (u8|u|U|L)
ES    (\\(['"\?\\abfnrtv]|[0-7]{1,3}|x[a-fA-F0-9]+))
NE    [^"\\\n]
WS    [ \t\v\f]
end   [$\r]
```

这段代码用于定义一些常见的标记模式和正则表达式。下面是对每个定义的模式的详细解释：

- O: 匹配八进制数字的字符集，范围从 0 到 7。
- D: 匹配十进制数字的字符集，范围从 0 到 9。
- NZ: 匹配非零的十进制数字的字符集，范围从 1 到 9。
- L: 匹配字母、下划线的字符集，包括小写字母、大写字母和下划线。
- A: 匹配字母、数字、下划线的字符集，包括小写字母、大写字母、数字和下划线。
- H: 匹配十六进制数字的字符集，包括小写字母、大写字母和数字。
- HP: 匹配十六进制数的前缀，即以”0x” 或”0X” 开头。
- E: 匹配浮点数中的指数部分，包括可选的正负号，后面跟着十进制数字。
- P: 匹配浮点数中的十六进制指数部分，包括可选的正负号，后面跟着十六进制数字。
- FS: 匹配浮点数类型后缀，可以是”f”、”F”、”l” 或”L”。
- IS: 匹配整数类型后缀，可以是各种组合，如”u”、”U”、”l”、”L”、”ll”、”LL” 等。
- CP: 匹配字符类型后缀，可以是”u”、”U” 或”L”。
- SP: 匹配字符串类型前缀，可以是”u8”、”u”、”U” 或”L”。
- ES: 匹配转义序列，包括转义字符（如”\n”、”\t” 等）、八进制字符（如”\377”）和十六进制字符（如”\xFF”）。
- NE: 匹配非转义字符和换行符的字符集。
- WS: 匹配空白字符，包括空格、制表符、垂直制表符和换页符。
- END: 匹配’$’ 符号，代表输入结束

2.2 C 语言部分

```
%{

#include <stdio.h>
#include "parser.tab.h"

%}
```

这段代码是使用 Lex 工具编写的 C 语言词法分析器。让我们对其进行详细解释：

`#include <stdio.h>`这是 C 语言中的预处理指令，用于包含标准输入输出库的头文件`#include <stdio.h>`，以便在词法分析器中使用`printf`和其他相关函数。

`#include "parser.tab.h`是我们用 YACC 编译.h 文件后，产生的.h 文件，在后续的链接编译中有重要作用

2.3 翻译规则部分

2.3.1 继续解析

```
"//".* { return -1; }  /* single-line comment */
"#".+  { return -1; }  /* preprocessor directive */
{WS}   {return -1;}    /* white space */
```

在 Lex 中，规则可以返回一个整数值，用于指示词法分析器采取的下一个操作。通常情况下，返回-1 表示继续解析，而返回其他值可能表示要进行一些特定的操作或者终止解析过程。上述代码的作用如下：

- 当遇到以"//"开头的注释时要执行的操作。它匹配从"//"开始的任意字符序列（.*表示匹配任意字符零次或多次）
- 当遇到以"#"开头的预处理指令时要执行的操作。它匹配以"#"开头的任意字符序列（.+表示匹配任意字符至少一次）
- 当遇到空字符时，跳过。

2.3.2 整数常量

```
{HP}{H}+{IS}?              { return I_CONSTANT; }  /* constants */
{NZ}{D}*{IS}?              { return I_CONSTANT; }
"0"{O}*{IS}?               { return I_CONSTANT; }
```

这段 Lex 代码用于匹配并返回 C 语言中的整数常量（integer constants）。它包含了三个规则：

- 匹配十六进制整数常量的模式。它以 HP（0x 或 0X）作为起始，后面跟随一个或多个十六进制数字（H），并可选择性地包含一个后缀（IS）来指示整数的类型。
- 匹配十六进制整数常量的模式。它以 HP（0x 或 0X）作为起始，后面跟随一个或多个十六进制数字（H），并可选择性地包含一个后缀（IS）来指示整数的类型。
- 匹配八进制整数常量的模式。它以字符 0 作为起始，后面可以跟随零个或多个八进制数字（O），并可选择性地包含一个后缀（IS）来指示整数的类型。

2.3.3 浮点常量

```
{D}+{E}{FS}?                   { return F_CONSTANT; }
{D}*"."{D}+{E}?{FS}?           { return F_CONSTANT; }
{D}+"."{E}?{FS}?               { return F_CONSTANT; }
{HP}{H}+{P}{FS}?               { return F_CONSTANT; }
{HP}{H}*"."{H}+{P}{FS}?        { return F_CONSTANT; }
```

```
{HP}{H}+"."{P}{FS}?              { return F_CONSTANT; }
```

这段 Lex 代码用于匹配并返回 C 语言中的浮点数常量（floating-point constants）。它包含了六个规则，如下：

- 以一个或多个数字（D）开头，接着是一个指数部分，由字母 E 表示，后面跟着一个可选的符号和一个或多个数字。最后，可能会包含一个可选的浮点数后缀（FS）。
- 以零个或多个数字（D）开头，接着是一个小数点，然后是一个或多个数字。可选地，可以包含一个指数部分，由字母 E 表示，后面跟着一个可选的符号和一个或多个数字。最后，可能会包含一个可选的浮点数后缀（FS）
- 以一个或多个数字（D）开头，接着是一个小数点。可选地，可以包含一个指数部分，由字母 E 表示，后面跟着一个可选的符号和一个或多个数字。最后，可能会包含一个可选的浮点数后缀（FS）。
- 以十六进制前缀（HP）开头，后面跟着一个或多个十六进制数字（H），然后是一个指数部分，由字母 P 表示。最后，可能会包含一个可选的浮点数后缀（FS）。
- 以十六进制前缀（HP）开头，后面跟着零个或多个十六进制数字（H），然后是一个小数点，接着是一个或多个十六进制数字。最后，需要包含一个指数部分，由字母 P 表示。最后，可能会包含一个可选的浮点数后缀（FS）。
- 以十六进制前缀（HP）开头，后面跟着一个或多个十六进制数字（H），然后是一个小数点，接着是一个指数部分，由字母 P 表示。最后，可能会包含一个可选的浮点数后缀（FS）。

这些规则按照特定的模式匹配浮点数常量，并在匹配成功时返回 F_CONSTANT 标识符，以便在 Lex 文件的其他规则中对浮点数常量进行特殊处理或进一步分析。

2.3.4 标点符号

```
{end}                            { return END; }
"("                              { return LP; }
")"                              { return RP; }
"+"                              { return PLUS; }
"-"                              { return MINUS; }
"*"                              { return TIMES; }
"/"                              { return DIV; }
```

这段 Lex 代码片段用于匹配和识别我们需要的基本运算符号。

- 如果是”(”，则标记为 LP
- 如果是”)”，则标记为 RP
- 如果是”+”，则标记为 PLUS
- 如果是”-”，则标记为 MINUS
- 如果是”*”，则标记为 LP
- 如果是”/”，则标记为 LP

- 如果是”$”，则标记为 END

总之，这段代码定义了一系列模式，用于匹配和识别各种标点符号，根据不同的标点，返回不同的标记模式

2.3.5 错误

```
/* errors */
{D}+({A}|\.)+    { yyerror("illegal name"); return -1; }
{SP}?\"({NE}|{ES})*{WS}*  { yyerror("unclosed string"); return -1; }
.                { yyerror("unexpected character"); return -1; }
```

这部分定义了三个规则：

- D+(A|)+：匹配非法的名称。它匹配一个或多个数字（D+），后跟一个字母或点号（A|.)+）。当匹配到这种模式时，词法分析器将输出一个错误消息 “illegal name”。
- SP?(NE|ES)*WS*：匹配未关闭的字符串常量。它匹配一个可选的前缀（SP?），后跟一个双引号，然后匹配零个或多个非转义字符（NE）或转义字符（ES），再匹配零个或多个空格字符，由于缺少结束双引号”，词法分析器将输出一个错误信息”unclosed string”
- 匹配除换行符外的任意字符。当词法分析器遇到任何无法匹配其他规则的字符时，它会匹配这个规则。当匹配到这种模式时，词法分析器将返回一个错误消息”unexpected character”，表示遇到了意外的字符。

3 YACC 程序的功能模块划分

3.1 C 语言部分

```
%{
#include <stdio.h>
# define ACC 1
extern int yylex();
int yyerror(const char* s);
int success = 1;
%}
```

这部分代码是 Yacc 程序的声明和全局定义部分。以下是对每个部分的解释：

- %{ ... %}: 这是 Yacc 源文件的可选部分，它允许在其中插入 C 代码。在这个部分中，可以包含任意的 C 头文件、宏定义和全局变量的声明。
- #include <stdio.h>: 这是一个包含标准输入输出函数的头文件。它使得我们可以使用 printf 和其他标准输入输出函数。
- #define ACC 1: 这里定义了一个名为 ACC 的宏常量，它的值为 1。这个宏常量可能在后面的代码中用于表示解析成功的状态。

- `extern int yylex();`: 这是一个函数声明，用于声明一个名为 `yylex` 的外部函数。这个函数通常是由词法分析器生成的函数，用于获取输入并返回词法单元的类型。
- `int yyerror(const char* s);`: 这是一个函数声明，用于声明一个名为 `yyerror` 的函数。这个函数通常是由语法分析器调用的错误处理函数，用于处理语法错误并输出错误消息。
- `int success = 1;`: 这是一个全局变量的定义，名为 `success`，并初始化为 `1`。这个变量可能在程序的其他部分用于表示解析的成功与否的状态。

总之，这部分代码主要用于包含必要的头文件、声明外部函数和全局变量，以及为程序提供一些常量和错误处理函数。这些声明和定义为后续的 Yacc 规则和主程序提供了必要的基础。

3.2 语法规则

```
%token PLUS MINUS TIMES DIV LP RP I_CONSTANT F_CONSTANT END
%start S

%%
S : E END { printf("S -> E\n"); return ACC; }
  ;

E: E PLUS T { printf("E -> E + T\n"); }
 | E MINUS T { printf("E -> E - T\n"); }
 | T { printf("E -> T\n"); }
 ;

T: T TIMES F { printf("T -> T * F\n"); }
 | T DIV F { printf("T -> T / F\n"); }
 | F { printf("T -> F\n"); }
 ;

F: LP E RP { printf("F -> (E)\n"); }
 | I_CONSTANT { printf("F -> num (I_CONSTANT)\n"); }
 | F_CONSTANT { printf("F -> num (F_CONSTANT)\n"); }
 ;

%%
```

这部分代码是 Yacc 程序的语法规则部分。它定义了语法规则和对应的动作，用于构建语法树或执行相关操作。下面是对每个部分的解释：

- %token PLUS MINUS TIMES DIV LP RP I_CONSTANT F_CONSTANT END

  这一行指定了词法单元的符号名称，也可以称为终结符号。每个符号名称代表了一个特定

的词法单元类型。在这个例子中，定义了 PLUS、MINUS、TIMES、DIV、LP、RP、I_CONSTANT、F_CONSTANT 和 END 这些终结符号。

- %start S

  这一行指定了起始符号，也就是语法分析的起点。在这个例子中，起始符号为 S。

- S : E END `printf("S -> E\n"); return ACC;`

  这是一个语法规则，定义了 S 的产生式。它表示 S 可以推导出 E $。在推导成功时，执行动作 `printf("S -> E\n"); return ACC;` ，这里的动作是输出一条消息，并返回 ACC 状态。

- E: E PLUS T `printf("E -> E + T\n");`

  这是一个语法规则，定义了 E 的产生式。它表示 E 可以推导出 E + T。在推导成功时，执行动作 `printf("E -> E + T\n");` ，这里的动作是输出一条消息。

- | E MINUS T `printf("E -> E - T\n");`

  这是 E 的另一个产生式，表示 E 可以推导出 E - T。在推导成功时，执行动作 `printf("E -> E - T\n");` ，这里的动作是输出一条消息。

- | T `printf("E -> T\n");`

  这是 E 的第三个产生式，表示 E 可以推导出 T。在推导成功时，执行动作 `printf("E -> T\n");` ，这里的动作是输出一条消息。

- T: T TIMES F `printf("T -> T * F\n");`

  这是 T 的一个产生式，表示 T 可以推导出 T * F。在推导成功时，执行动作 `printf("T -> T * F\n");` ，这里的动作是输出一条消息。

- | T DIV F `printf("T -> T / F\n");`

  这是 T 的另一个产生式，表示 T 可以推导出 T / F。在推导成功时，执行动作`printf("T -> T / F\n");` ，这里的动作是输出一条消息。

- | F `printf("T -> F\n");`

  这是 T 的第三个产生式，表示 T 可以推导出 F。在推导成功时，执行动作 `printf("T -> F\n");` ，这里的动作是输出一条消息。

- F: LP E RP `printf("F -> (E)\n");`

  这是 F 的一个产生式，表示 F 可以推导出 (E)。在推导成功时，执行动作 `printf("F -> (E)\n");` ，这里的动作是输出一条消息。

- | I_CONSTANT `printf("F -> num (I_CONSTANT)\n");`

  这是 F 的另一个产生式，表示 F 可以推导出整数 num 。在推导成功时，执行动作 `printf("F -> num (I_CONSTANT)\n");` ，这里的动作是输出一条消息。

- | F_CONSTANT `printf("F -> num (F_CONSTANT)\n");`

  这是 F 的第三个产生式，表示 F 可以推导出浮点数 num。在推导成功时，执行动作 `printf("F -> num (F_CONSTANT)\n");` ，这里的动作是输出一条消息。

- %%: 这是规则部分的结束符号，用于标识语法规则的结束。

3.3 主函数与错误处理

```
int main(){
    yyparse();
    if (success == 1)
        printf("\033[32mParsing done.\033[0m\n");
    return 0;
}

int yyerror(const char *msg)
{
    extern int yylineno;
    printf("\033[31mParsing Failed\nLine Number: %d %s\033[0m\n",
       yylineno, msg);
    success = 0;
    return 0;
}
```

这部分代码包含了主函数 main 和错误处理函数 yyerror。

- int main(): 这是程序的主函数。它调用了 yyparse() 函数，该函数是由 Yacc 自动生成的用于执行语法分析的函数。yyparse() 函数将根据之前定义的语法规则对输入进行语法分析，并根据语法规则中的动作执行相应的操作。在这个例子中，yyparse() 函数完成语法分析后，通过检查 success 变量的值来判断解析是否成功。如果 success 的值为 1，表示解析成功，将输出一条成功的消息；否则，表示解析失败。
- int yyerror(const char *msg): 这是错误处理函数 yyerror 的实现。当语法分析过程中发生错误时，Yacc 会调用这个函数。函数的参数 msg 是错误消息的字符串。在这个例子中，yyerror 函数使用 printf 函数输出错误消息，并包含当前发生错误的行号yylineno。同时，它将 success 变量的值设置为 0，表示解析失败。

总之，这部分代码用于调用语法分析函数 yyparse()，并根据解析结果输出相应的消息。同时，它还实现了错误处理函数 yyerror，用于在发生语法错误时输出错误消息并记录解析失败的状态。

4 使用说明

4.1 Bison 编译程序

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

```
>  D:\win_flex_bison-latest\win_bison.exe -d parser.y
```

4.2 LEX 编译程序

利用 win_flex 编译写好的.l 文件，生成对应的.c 代码

```
> D:\win_flex_bison-latest\win_flex.exe --wincompat lexcial.l
```

4.3 C 语言混合编译程序

将生成的 2 个.c 进行链接编译，生成可执行文件 parser.exe

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

5 测试

5.1 测试 1+2

该测试集用于测试一个简单的算数表达式能否被正确识别

5.1.1 输出结果

```
> ./parser.exe
1+2$
F -> num (I_CONSTANT)
T -> F
E -> T
F -> num (I_CONSTANT)
T -> F
E -> E + T
S -> E
Parsing done.
```

5.1.2 输出结果分析

给定的语法分析程序的输出结果表明，输入字符串”1+2$” 符合给定的文法规则。根据给出的输出结果，可以得出以下简单的输出分析总结：

1. 输入表达式为 1+2$
2. 首先，识别到 1，将其作为 I_CONSTANT（整数常量）规约为 F。
3. 接着，识别到 2，同样将其作为 I_CONSTANT 规约为 F。
4. 将 F 规约为 T。
5. 再次将 F 规约为 T。
6. 将 T 规约为 E。
7. 最后，将 E + T 规约为 E。
8. 整个表达式成功地被语法分析器归约为起始符号 S。

因此，可以总结出以上输出的分析结果为：给定的输入表达式 1+2$ 符合定义的语法规则，并且成功地被规约为起始符号 S。

5.2 测试 1+2*(3-(4/0))

5.2.1 输出结果

```
1+2*(3-(4/0))$
F -> num (I_CONSTANT)
T -> F
E -> T
F -> num (I_CONSTANT)
T -> F
F -> num (I_CONSTANT)
T -> F
E -> T
F -> num (I_CONSTANT)
T -> F
F -> num (I_CONSTANT)
T -> T / F
E -> T
F -> (E)
T -> F
E -> E - T
F -> (E)
T -> T * F
E -> E + T
S -> E
Parsing done.
```

5.2.2 输出结果分析

通过一系列操作执行，程序成功地完成了对输入字符串”1+2*(3-(4/0))” 的语法分析。

根据给出的输出结果，可以得出以下简单的输出分析总结：

1. 首先，识别到 1，将其作为 I_CONSTANT（整数常量）规约为 F。
2. 接着，识别到 2，同样将其作为 I_CONSTANT 规约为 F。
3. 将 F 规约为 T。
4. 继续识别到 3，将其作为 I_CONSTANT 规约为 F。
5. 再次将 F 规约为 T。
6. 接下来，识别到 4，将其作为 I_CONSTANT 规约为 F。
7. 将 F 规约为 T。
8. 将 T / F 规约为 T。
9. 再次将 F 规约为 T。
10. 将 (E) 规约为 F。
11. 将 F 规约为 T。
12. 将 T * F 规约为 T。
13. 将 T 规约为 E。
14. 将 (E) 规约为 F。
15. 将 F 规约为 T。
16. 将 T - F 规约为 T。

17. 将 T 规约为 E。
18. 最后，将 E + T 规约为 E。
19. 整个表达式成功地被语法分析器归约为起始符号 S。

因此，可以总结出以上输出的分析结果为：给定的输入表达式 1+2*(3-(4/0))$ 符合定义的语法规则，并且成功地被规约为起始符号 S。

5.3 测试 1+2*/(3-4/0))

5.3.1 输出结果

```
1+2*/(3-4/0))
F -> num (I_CONSTANT)
T -> F
E -> T
F -> num (I_CONSTANT)
T -> F
Parsing Failed
Line Number: 1 syntax error
```

5.3.2 输出结果分析

根据给出的输出结果，可以得出以下简单的输出分析总结：

1. 首先，识别到 1，将其作为 I_CONSTANT（整数常量）规约为 F。
2. 接着，识别到 2，同样将其作为 I_CONSTANT 规约为 F。
3. 将 F 规约为 T。
4. 遇到 /，但是没有合适的终结符号可以规约，因此语法分析失败。
5. 输出结果指示语法分析失败，给定的输入表达式存在语法错误。
6. 在行号 1 处发现语法错误。

因此，可以总结出以上输出的分析结果为：给定的输入表达式 ‘1+2*/(3-4/0))‘ 存在语法错误，无法被正确规约。语法错误发生在行号 1 处。

6 实验总结

在本次实验中，我使用了 LEX 工具和 YACC 工具编写了一个语法分析程序，以实现对 C 语言中整型和浮点型常量的词法分析。

首先，我仔细阅读了 C11 的 ISO 标准，特别关注了标准中对常量的描述。这帮助我明确了 C 语言中整型和浮点型常量的定义和规则。

接着，我根据标准中给出的语法和我的课堂学习，设计了用于匹配整型和浮点型常量的正则表达式。这些正则表达式可以用于 LEX 工具生成词法分析器。

在设计词法分析程序时，我预先定义了一系列常用的正则表达式，以便后续的词法分析可以直接使用这些工具。这些正则表达式包含了整型和浮点型常量的模式，例如匹配整数、十六进制数、指数表示法等。此外，还考虑了 C 语言中常量的后缀，如 L、U、F 等。

接下来，我根据 C 语言的语法规则，使用 YACC 工具编写了相应的语法规约程序。这样，我可以定义 C 语言中的语法规则，包括算术表达式的结构等。

为了提高程序的鲁棒性，我还编写了简单的错误处理函数，用于处理词法分析和语法分析过程中可能出现的错误情况。这些错误处理函数可以帮助我在程序出现错误时给出相应的提示信息。

通过这次实验，我深入理解了语法分析的流程和相关知识点。通过使用 LEX 和 YACC 工具，我能够更方便地实现词法分析和语法分析，并且对 C 语言中整型和浮点型常量的词法规则有了更牢固的掌握。我还能通过自定义的错误处理函数，增强程序的健壮性，提供更好的错误提示和恢复机制。