LE-compiler 实验报告

1 功能

- 逻辑表达式求值
- 计算因为短路跳过的**比较**次数
- 类似 C 语言语法的四则运算支持
- 一次测试多个逻辑表达式（但出现语法错误后不能够恢复）
- 命令行测试和测试文件测试

2 设计方案

为了满足如上所述的功能，我的实验需要在原来的要求上进一步支持：四则运算以及相关报错的词法设计、文件和命令行测试。为此，我设计了如下的词法和语法：

2.1 词法设计方案

词法记号	正则表达式	备注
RELOP	>|<|>=|<=|==|!=	关系运算符
PLUS	"+"	加号
MINUS	"-"	减号
STAR	"*"	乘号
DIV	"/"	除号（真除法）
AND	"&&"	与运算符
OR	"||"	或运算符
NOT	"!"	非运算符
LP	"("	左括号
RP	")"	右括号
INT	0|[1-9]+[0-9]*	整形
INT8	0[0-7]+	八进制整形
INT16	0[xX][0-9a-fA-F]+	十六进制整形
WINT8	0(digit|letter)+	错误的八进制整形 （词法错误提示用）

WINT16	0[xX](digit|letter)+	错误的十六进制整形 （词法错误提示用）
WINT	digit*|digit+(digit|[a-zA-Z])+	错误的整形（词法错误提示用）
FLOAT	digit+"."digit+ | digit*"."digit+[eE][+-]?digit+ | digit+"."digit*[eE][+-]?digit+	浮点数（支持科学计数法）

如上表所示，我设计了四则运算所用的所有符号和数值的词法，并且支持了整形、浮点数格式错误的词法提示。此外，在词法分析的时候，我就进行了一定的语义处理。我使用枚举量`NodeType`来对不同的 token 进行区分。如果 NodeType 为 Token_Int，则将 yytext 转化为 int 并赋值给对应词法记号的 intVal 属性；如果 NodeType 为 Token_Float，则将 yytext 转化为 float 并赋值给对应的词法记号的 floatVal 属性；如果 NodeType 为 Token_Other 类型，则将 yytext 直接赋值给 strVal 属性。

2.2 语法语义设计方案

2.2.1 数据结构设计

为了解决四则运算的语义问题以及计算因短路运算跳过的比较次数，我设计了如下的数据类型：

```
// node type declaration
typedef struct node {
    int lineNo;              // line no.
    size_t strLen;           // length of strVal
    NodeType type;           // node type
    char* name;              // node name
    unsigned int counted;    // counted before or not
    union                    // node value
    {
        int intVal;
        float floatVal;
        char* strVal;
    };
```

```
    struct node* child; // first child node of non-terminal node
    struct node* next;  // next brother node of non-terminal node
}Node;
```

其中，lineNo 是词法或语法错误位置，strLen 是 strVal 的长度（如果 strVal 不为空），type 为 NodeType 枚举类型，counted 表示是否计算过（用来计算跳过的比较次数），child 和 next 分别表示当前结点的孩子结点和兄弟结点。自然而然的，bison 栈中的数据类型即为 Node 指针类型 pNode。

2.2.2 语法语义设计

根据如上的数据结构我们可以知道，我为每个非终结符设计了 child、next、floatVal 三个综合属性，同时为了简便处理，我们将 TOKEN_INT 类型的词法记号的 intVal 直接赋值给 floatVal 属性；为每个非终结符设计了 lexeme 属性，其值为其语义本身（即 Token_Int 类型为整数，Token_Float 类型为浮点数，Token_Other 类型为字符串，存储在 strVal 属性中）。

由于支持四则运算后语法制导定义过长，限于篇幅，我将 SSD 放在了第 5页附录 SSD 部分。

2.3 难点与解决方案

经过以上的词法语法语义设计，我们能很简单的得到逻辑表达式的真值（即 Line.floatVal!=0）。我遇到的难点便成了如何计算因为短路运算跳过的次数。由于我没能设计出一个比较好的语义制导定义，又考虑到我采用了多叉树来表示整个输入，我最后选择使用深度优先遍历的方式来求解因为短路运算跳过的比较次数。算法可以表示成如后述伪代码。

简单的来说就是：每当当前节点为 Exp，其兄弟结点为 AND 或 OR，并且短路条件成立，则跳到其兄弟节点进行深度优先遍历，统计跳过的 RELOP，并将该结点标记为访问过。

3 构建和测试

我使用了 make 管理项目。因此，进入 src 源代码目录后，make 便可构建生成可执行程序 parser，make test 便可执行自动测试。

```
cd src
# 构建
make
# 自动化测试
make test
# 命令行测试
./parser
```

算法 1 计算因短路运算跳过的比较次数

输入： $Node*\ curNode$ 需要计算的结点指针

输出： $curNode$ 非空

```
1: function countSkip(Node* curNode)
2:     if curNode! = NULL then
3:         return
4:     end if
5:     if oper of Exp equals to "AND" and short circuit then
6:         dfs(curNode.next)
7:     end if
8:     if oper of Exp equals to "OR" and short circuit then
9:         dfs(curNode.next)
10:    end if
11:    countSkip(curNode.child)
12:    countSkip(curNode.next)
13: end function
14:
15: function dfs(Node* curNode)
16:    if curNode==NULL or curNode.counted==True then
17:        return
18:    end if
19:    if curNode.name=="RELOP"==0 then
20:        skipNum = skipNum + 1
21:        curNode.counted ← 1
22:    end if
23:    dfs(curNode.child)
24:    dfs(curNode.next)
25: end function
```

附录

A 语法制导定义

产生式	语义规则
Start → Input	Start.child = Input root=Start
Input → $Input_1$ Line	Input.child = $Input_1$ Input.next=Line **if** Line != NULL and Line.child != NULL and Line.next !=NULL **then** Input.floatVal = Line.floatVal printf("Output: %s, %d", Input->floatVal==0?"false":"true", skipNum) skipNum=0
Input → ϵ	Input=NULL
Line → NEWLINE	Line.child=NEWLINE
Line → Exp NEWLINE	Line.child=Exp Line.next=NEWLINE **if** Exp!=NULL **then** Line.floatVal=Exp.floatVal countSkip(Line)
Exp → Exp_1 PLUS Exp_2	Exp.child=Exp_1 Exp.next=PLUS Exp.next.next=Exp_2 Exp.floatVal = $Exp_1.floatVal + Exp_2.floatVal$
Exp → Exp_1 MINUS Exp_2	Exp.child=Exp_1 Exp.next=MINUS Exp.next.next=Exp_2 Exp.floatVal = $Exp_1.floatVal - Exp_2.floatVal$
Exp → Exp_1 STAR Exp_2	Exp.child=Exp_1 Exp.next=STAR Exp.next.next=Exp_2 Exp.floatVal = $Exp_1.floatVal * Exp_2.floatVal$

Exp → Exp_1 DIV Exp_2	Exp.child=Exp_1 Exp.next=DIV Exp.next.next=Exp_2 Exp.floatVal = $Exp_1.floatVal$ / $Exp_2.floatVal$
Exp → Exp_1 AND Exp_2	Exp.child=Exp_1 Exp.next=AND Exp.next.next=Exp_2 Exp.floatVal = $Exp_1.floatVal$ && $Exp_2.floatVal$
Exp → Exp_1 OR Exp_2	Exp.child=Exp_1 Exp.next=OR Exp.next.next=Exp_2 Exp.floatVal = $Exp_1.floatVal$ || $Exp_2.floatVal$
Exp → Exp_1 RELOP Exp_2	Exp.child=Exp_1 Exp.next=RELOP Exp.next.next=Exp_2 **if** RELOP.strVal == "==" **then** Exp.floatVal=Exp_1.floatVal-Exp_2.floatVal<1e-6 **elif** RELOP.strVal == "!=" **then** Exp.floatVal=Exp_1.floatVal-Exp_2.floatVal>=1e-6 **elif** RELOP.strVal == ">" **then** RELOP.floatVal=Exp_1.floatVal>Exp_2.floatVal **elif** RELOP.strVal == "<" **then** RELOP.floatVal=Exp_1.floatVal<Exp_2.floatVal **elif** RELOP.strVal == ">=" **then** RELOP.floatVal=Exp_1.floatVal>=Exp_2.floatVal **else** RELOP.floatVal=Exp_1.floatVal<=Exp_2.floatVal

Exp → MINUS Exp_1	Exp.child=MINUS Exp.next=Exp_1 Exp.floatVal=-Exp_1.floatVal
Exp → NOT Exp_1	Exp.child=MINUS Exp.next=Exp_1 Exp.floatVal=!Exp_1.floatVal
Exp → LP Exp_1 RP	Exp.child=LP Exp.next=Exp_1 Exp.next.next=RP Exp.floatVal=Exp_1.floatVal
Exp → INT	Exp.child=INT Exp.floatVal=INT.lexeme
Exp → FLOAT	Exp.child=FLOAT Exp.floatVal=FLOAT.lexeme