编译原理

实验一 词法和语法分析

171860580 甘宇航

· 实验要求

通过 Lex 和 Bison 实现 C-- 语法的词法&语法分析器

· 实验内容说明

词法分析

对于词法分析，内容好像比较少，就是按照要求将正则表达式写出来，然后赋值给内置变量yylval，因为实验中要求需要构造（打印）一棵语法树，这里可能是词法分析（和语法分析）的比较重要的一个实验的细节，关于如何传递和保存值的问题，具体打印时需要打印节点的类型和内容，实验手册上给了一种将不同类型的数据分别保存的参考，但这样会带来**语法树节点类型复杂**和**打印程序判断条件过多**的情况，而针对这种情况，基于C语言的“良好”的底层特性（可强制类型转换，用什么指针，内存内的东西就解读为对应的类型），所以我设计了一个统一的节点，里面包含节点的类型、内容和行号，其中内容也用了一个char[]数组作为保存，只需要在获取的时候（比如打印，后续实验可能用到的计算），直接用 atoi()、atof() 这样的函数转换即可，其中特别注意的是，因为手册上没有提到节点内容长度的限制（特别是ID），于是节点内部仅用了两个指针，然后在创建节点时根据当前节点名称和内容的长度，（malloc长度+10）的空间进行存储，这种统一存储变量而非按照变量类型设计不同节点（大概）是本次实验的一个亮点吧；

除此之外，在进行实际测试时发现，有些语法未定义符号（例如“~”）在进行语法分析的时候，若直接丢弃，如果存在将未定义符号作为分号、括号使用的情况下，错误恢复会很难办，若不丢弃，则极端情况下，也会使错误恢复跳过一大段内容而导致可能直接跳过下一个错误，于是在进行了尝试后，直接定义了一个语法单元MYCHA（mysterious character）当作正常字符来处理，为了尽可能少的跳过符号，将MYCHA进行到Exp的规约，再配合相应的错误恢复机制，效果还不错，跳过的内容比较少后就能碰到错误恢复的产生式，这应该是词法分析的另一个亮点吧；

```
. { have_err = true; fprintf(stderr,"Error type A at Line %d: Mysterious character \"%s\"\n", yylineno, yytext);yylval.type_node = crea_node("MYSCHA",yytext); re

%%
```

```
    | ID LP error {prterror("Missing \")\" or content in \"()\" wrong?\n");}
    | Exp LB error ASSIGNOP {prterror("array index invalid\n");} ////////////////////////////////////////////
    //| Exp error SEMI {prterror("invalid expression the error is %s | %s | %s\n",$1,$2,$3);}
    | MYSCHA {prterror("");}          规约到Exp
;
```

```
    | WHILE LP error RP Stmt {prterror("");}
    //| Exp error {prterror("Missing \";\" or something before went wrong?\n");} ////////////////////////////////////////
    | Exp error SEMI {prterror("invalid expression\n");}
    //| error RC {prterror("");}          配套的错误恢复之一
```

语法分析

这个部分能说的（踩过的坑）比词法分析部分稍微多一些，主体就是将实验讲义上的产生式敲出来，但细节主要集中在错误恢复，其中一个比较重要的点就是上面的关于未定义符号的处理，还有一个就是错误恢复的一个tradeoff,一开始比较贪心，为了尽可能找出多的错误，跳过尽可能少的符号，在没有仔细思考的情况下就写了很多偏向底层产生式的错误恢复规则，而“冲突”的编译提示在命令行中比较灰暗和小，一开始没有注意到，于是就对很多莫名其妙的错误（特别是规约的行为）百思不得其解，头秃到深夜，后来终于发现了“移入规约冲突”以后，注释冲突的错误恢复；秉持的原则仍然是比较贪心的尽可能少的跳过符号，于是还是选择了偏向比较底层的错误恢复，仅在比较偏顶层的产生式中添了少量几个error应对特殊情况和保险，后来效果就还不错；

除此之外就是附加任务了，我领到的任务是注释的判断和删除，“//”比较好处理，直接将对应后面整行删除即可，讲义中也给了相应的实现，而“/*” 则利用了 Flex 的 input()，在遇到 “*/”之前将所有的符号都抛弃，其中有一个边界情况是注释没有“*/”而直接达到文件末尾，这种情况下若不处理则程序陷入死循环，原本是没有想到的，但后来有同学在课程群内提到了文件结束符，则增加了具体的处理，而特别的，因为已经写成了具体循环的input()的实现，正则 “<<EOF>>” 比较难以加进去，于是手动测试了在文件结束时的符号并做了特判，实现了功能（但很担心测试时因为环境不同导致该点测试失败）；

```
"/*" {
    char c;
LOOP:
    c = input();
    //printf("%d<---\n",EOF);
    if(c == 0) {have_err = true;printf("Error type A at Line %d: The annotation arrived the end of file(EOF)\n", yylineno);goto END;}
    while(c != '*' && c != 0) {c = input();}
    if(c == 0) {have_err = true;printf("Error type A at Line %d: The annotation arrived the end of file(EOF)\n", yylineno);goto END;}
    c = input();
    if(c == 0) {have_err = true;printf("Error type A at Line %d: The annotation arrived the end of file(EOF)\n", yylineno);goto END;}
    //printf("%c,%d , line-> %d\n",c,c, yylineno);
    if(c != '/' && c != -1) {goto LOOP;}
END: ;
}
```

边界条件判断

抽象语法树AST

因为语法分析是LALR方法，语法树需要自底向下构建，这个构建方法对于多叉树其实比较自然，本次实验中的多叉树采用了“左子女右兄弟”的构建方法而将整个多叉树转换为等价的二叉树，即对于每个节点的左子节点是他的第一个子节点，右子节点是与它同一层的下一个节点，这样转化后，讲义中要求的多叉树的先序遍历就对应了等价二叉树的前序遍历,打印起来也比较方便，于是对于每个节点都需要再加两个指针域（fst_child fst_bro）指向对应节点，考虑到因为是自底向上的，于是每个高层节点第一次产生时当且仅当由其对应的产生式的右部规约而成，于是在语法处理的动作中加入“创建产生式左部节点“$$ = crea_sy_node(name, val)”，将产生式右部节点插入到该节点的下一层上insert_child($$,$i)”，这里还有一个坑，就是产生式有可能产生 ε，所以在遇到这种情况时需要将$$置为NULL，否则在后期记录高层产生式的行号（产生式右部行号最小节点）时用到这个 ε 有可能导致段错误（又是一个头秃的晚上）；

```
void insert_child(node* cur_node, node* child) {  // we set that the first child i
    if(child != NULL) {//epsilon might be empty;
        if(cur_node->lineNum > child->lineNum) cur_node->lineNum = child->lineNum;
    }
    if(cur_node->fst_child == NULL) { cur_node->fst_child = child; return;}
    node* p = cur_node->fst_child;
    while(p->fst_bro != NULL)
        { p = p->fst_bro;}
    p->fst_bro = child;
```

记录行号

一些其他的

还有一些比较小的其他的点，例如在语法分析的语法动作实际进行语法树创建时，由于1个产生式左部最多可以由6个右部组成，则插入子节点需要写6遍而仅仅是参数不同，重复性很强，导致在写的时候很有可能因为重复造成潜在的BUG（因为大量的重复意味着大量的复制粘贴），于是就写了一个插入的宏减少了代码的长度，可读性也更好了：

```
#define IC1(father, C1) insert_child(father,C1);
#define IC2(father,C1,C2) IC1(father,C1); insert_child(father,C2);
#define IC3(father,C1,C2,C3) IC2(father,C1,C2) insert_child(father,C3);
#define IC4(father,C1,C2,C3,C4) IC3(father,C1,C2,C3) insert_child(father,C4);
#define IC5(father,C1,C2,C3,C4,C5) IC4(father,C1,C2,C3,C4) insert_child(father,C5);
#define IC6(father,C1,C2,C3,C4,C5,C6) IC5(father,C1,C2,C3,C4,C5) insert_child(father,C6);
```

IC代表 insert child

```
Stmt: Exp SEMI {$$ = crea_sy_node("Stmt"); IC2($$,$1,$2)}
    | CompSt {$$ = crea_sy_node("Stmt"); IC1($$,$1)}
    | RETURN Exp SEMI {$$ = crea_sy_node("Stmt"); IC3($$,$1,$2,$3)}
    | IF LP Exp RP Stmt %prec LOWER_THAN_ELSE {$$ = crea_sy_node("Stmt"); IC4($$,$1,$2,$3,$4)}
    | IF LP Exp RP Stmt ELSE Stmt {$$ = crea_sy_node("Stmt"); IC6($$,$1,$2,$3,$4,$5,$6)}
    | WHILE LP Exp RP Stmt {$$ = crea_sy_node("Stmt"); IC5($$,$1,$2,$3,$4,$5)}
```

递归打印语法树

```
/***print AST***/
void prt_cur_node(struct node* root, int space_num) {…
}
void prtAST(struct node*root, int space_num) {
    if(root == NULL) return;
    prt_cur_node(root,space_num);
    struct node* child_ptr = root->fst_child;
    while(child_ptr != NULL) {
      //printf("
      prtAST(child_ptr,space_num+1);
      child_ptr = child_ptr->fst_bro;
    }
}
```

还有就是由于判断错误时就不打印语法树，所以在全局设置了一个 flag “have_err”,当且仅当遇到词法、语法错误时设置为 true (由于 C 没有 true， #define true 1 增加可读性)，除此之外，设置了一个全局的根节点 “struct node* root”,并将Program的指针传给 root;
在错误打印方面则因为yyerror()定位错误比较准确，同时利用了 yyerror() 以及自己写了一个prterror(),prterror()是在有比较确切把握的情况下输出一些辅助信息，提出修改意见：

```
Error type A at Line 4: Mysterious character ":"
Error type B at line 4: Something wrong near this line
                        Reference supplementary comments : Something wrong in function parameter?

Error type B at line 7: Something wrong near this line
                        Reference supplementary comments : invalid expression

Error type B at line 12: Something wrong near this line
```

yyerror()输出定位信息

prterror()输出辅助说明

但prterror()由于是在错误恢复时打印的，所以有时候可能打印的信息不太准确，仅作为参考。