Compiler Lab1 Report

151220019 崔臻 151220023 段建辉 邮箱:1026680660@qq.com

完成进度

完成了实验一必做全部内容和选做要求 1.1;

实现功能

1、 对任意浮点数、id、保留字、符号、八进制十进制十六进制整形等的匹配。

通过正则匹配实现。对整型数字的识别通过识别所有的数字以及字母然后进行处理。这样的好处是让词法错误尽量留在词法区域，不让词法的错误再次传递到语法分析中进行处理。

2、 对八进制、十进制、十六进制的词法错误的发现。

在识别出来应该是八进制或者十六进制或者十进制的数字中进行字符串的判别，如果识别出来的字符不是该进制所有的数字，进行错误的输出。然后返回-1。

3、 语法树的构建与输出

如果发现了语法错误，让布尔变量 hasError 置 true，然后通过判断这个布尔变量决定是输出语法树还是错误。

4、 语法错误的发现

通过利用 bison 保留字 error 在语法树构建过程之中发现 error 然后在特定位置进行语法错误的输出。可以输出存在的所有的错误。

在这个过程之中可能会出现想法和语法树构建方面的不一致导致的移入规约冲突。这样的冲突虽然在目前的分析中不会出现问题，但是可能在以后的复杂语法分析中出现问题，因此必须解决。并且可能 error 错误种类千奇百怪，并不能枚举出来所有的错误，并且错误判别一旦多起来，各种语法分析的冲突也会随之出现，那么只能在合适的地方添加错误的发现与输出，但是也要考虑退栈是否能够到达我们所期望的位置。

编译运行

通过 makefile 进行编译，在终端执行 make run，生成 parse 可执行文件，然后运行 ./parse <cmmfile> 进行程序的执行。

实验以及亮点内容简介

1. 词法分析

针对不同的词法单元执行不同功能，如下图：正则表达式匹配到相应的词素返回相应的词法单元；

```
\n|\r\n              {yylineno++; yycolumn = 1;}
"+"                 {yylval.type_node=NewNode("PLUS");return PLUS;}
"-"                 {yylval.type_node=NewNode("MINUS");return MINUS;}
"*"                 {yylval.type_node=NewNode("STAR");return STAR;}
"/"                 {yylval.type_node=NewNode("DIV");return DIV;}
";"                 {yylval.type_node=NewNode("SEMI");return SEMI;}
","                 {yylval.type_node=NewNode("COMMA");return COMMA;}
```

进行八进制以及十六进制的判断以及错误输出。因为十六进制可能出现字母，并且和八进制十进制都是整型变量，因此需要放在一起处理。我们在 lex 文件中写了一个函数用来单一处理整型变量，在识别过程中并不是识别 0[0-7]+或者 0[Xx][0-9A-Za-z]+，而是识别出 0 或者 0x 之后后面的字母均进行识别。变成了：

letter　[A-Za-z]

digit　[1-9]

int　(0|[1-9]({digit}|{letter})*|0({digit}|{letter})+|0[Xx]({digit}|{letter})+)

这样可以把连续数字均识别出来并不会对后续的其他词法分析产生影响。然后通过判断是否合法决定生成一个整型变量，输出错误返回-1.

如果生成-1 还继续使用该变量肯定是不正确的，因此如果产生错误并且能够让词法继续分析下去，将这里的整形变量在出错情况下生成一个 ID 的 node，把错误的串看作是一个 ID，这样可以让整型变量的判断不影响后续分析并且能够很好地处理该处的变量。

2.语法和语法分析树的构建

定义结构体来实现多叉树结构

```
typedef struct Node
{
    int type; //0 terminal 1 noterminal 2 int 3 float 4 Id
    int linenum;
    int intval;
    float floatval;
    char idval[50];
    char strval[50];
    int childnum;
    struct Node * child[100];
}Node;
```

在 buildtree.c 中实现如下函数，

```
Node *NewNodeSyn(char *,int);
Node *MergeNode1(Node*,Node*);  //1 child
Node *MergeNode2(Node*,Node*,Node*); //2 child
Node *MergeNode3(Node*,Node*,Node*,Node*); //3 child
Node *MergeNode4(Node*, Node*,Node*,Node*,Node*); //4 child
Node *MergeNode5(Node*, Node*,Node*,Node*,Node*,Node*); //5 child
Node *MergeNode7(Node*, Node*,Node*,Node*,Node*,Node*,Node* ,Node*);//6 child
```

然后在每次条文法规约时，讲产生式左边作为父节点，产生式右边为该父节点的子节点。如图：

```
ExtDefList : ExtDef ExtDefList {Node *p=NewNodeSyn("ExtDefList",@$.first_line); $$=MergeNode2(p,$1,$2);}
    |{$$=NULL;}
```

如果产生式右边为空，则为子节点为 NULL，不作为父节点的孩子。

输出语法书函数:DisplayTree，按照语法树的输出要求对多叉树进行前序遍历。

```
void DisplayTree(Node *head,int n)
{
    int i=0;
    PrintTab(n);
    switch (head->type)
    {
        case 0: printf("%s\n",head->strval); break;
        case 1: printf("%s  (%d)\n",head->strval,head->linenum);break;
        case 2: printf("%s: %d\n",head->strval,head->intval);break;
        case 3: printf("%s: %f\n",head->strval,head->floatval);break;
        case 4: printf("%s: %s\n",head->strval,head->idval);break;
        default : printf("should not reach!\n");
    }
    for(;i<head->childnum;i++)
    {
        DisplayTree(head->child[i],n+1);
    }
}
```

3.错误恢复

在这个过程之中，使用宏定义定义了普遍出现的一些错误：

```
#define ERR_COMMA    Missing"\",\""
#define ERR_LP       "Missing \"(\""
#define ERR_RP       "Missing \")\""
#define ERR_LB       "Missing \"[\""
#define ERR_RB       "Missing \"]\""
#define ERR_RC       "Missing \"}\""
#define ERR_LC       "Missing \"{\""
#define ERR_SEMI "Missing \";\""
```

这样的定义让识别输出更方便，然后进行错误识别的过程之中，由于一开始直接添加了能够出现的各种错误，移入规约冲突很多，因此只能通过在一些地方进行错误删除。

最主要的亮点是尽量将词法分析不干扰语法分析，语法分析中尽量使用简单的错误分析或者是错误判断更广的错误分析，类似“error COMMA”或者“error SEMI”的句子，如果使用类似“IF LP Exp error Stmt ELSE Stmt”这样的错误分析语法就显得过于精细并且可能发生严重的移入规约冲突。