Lab1-词法分析与语法分析

程序功能说明

词法分析

词法分析部分使用GNU Flex工具将输入文本识别为词法单元，包括各种符号，保留字，整数值（八进制、十进制、十六进制），实数值，标识符，注释。同时实现了对错误词法的检测，包含整数错误，实数错误，标识符错误，当检测到错误词法时输出相应报错信息。

实现方式：使用正则表达式匹配词法，对于每一次词法分析的调用，Flex将选取当前输入下可匹配长度最长的规则进行处理，正常的词法会创建树节点作为属性值，并返回对应token（在bison源码部分定义），此外：

- 如果词法单元是整数值或实数值，根据不同进制交给不同函数处理成int或float，作为树节点的属性值；
- 如果词法单元是标识符，直接将字符串类型的匹配文本作为树节点的属性值；
- 如果是错误词法，立刻输出报错信息，置全局变量Error为1，并返回正常token（尽可能不干扰语法分析）；

语法分析

语法分析部分使用GNU Bison工具，通过CFG描述语法规则，将一份输入转化为语法树结构，并最终输出语法树及其各个结点的信息。此外，还能识别出特定的语法错误并报错。

实现方式：bison依据规则对词法单元输入进行移入、规约操作，最终规约为一个根节点`Program`，对于中间出错的部分进行错误恢复。对于每一个没有错误的语法单元规则，创建语法树节点，并进行父子结点连接。当分析结束时检查全局变量Error，如果Error为0，可以打印语法树结构。

编译使用说明

源文件：`lexical.l`，`syntax.y`，`main.c`，`tree.h`

编译测试文件：`Makefile`，`runtest`，`test/*.cmm`

`Makefile` 中定义了若干目标，要编译本实验项目，可以输入：

```
make
```

编译会产生中间文件`lex.yy.c`，`syntax.tab.h`，`syntax.tab.c`，生成可执行程序`parser`.

`parser` 接收一个文本文件作为输入，进行分析和输出，例如：

```
./parser test/tmp0.cmm
```

`test` 目录下包含了10个测试样例，要对`test` 目录下所有测试样例进行测试，可以执行：

```
./runtest
# ./runtest [outputfile]
```

如不携带参数，则输出到屏幕；否则将输出到指定文件。

细节说明

bison错误恢复

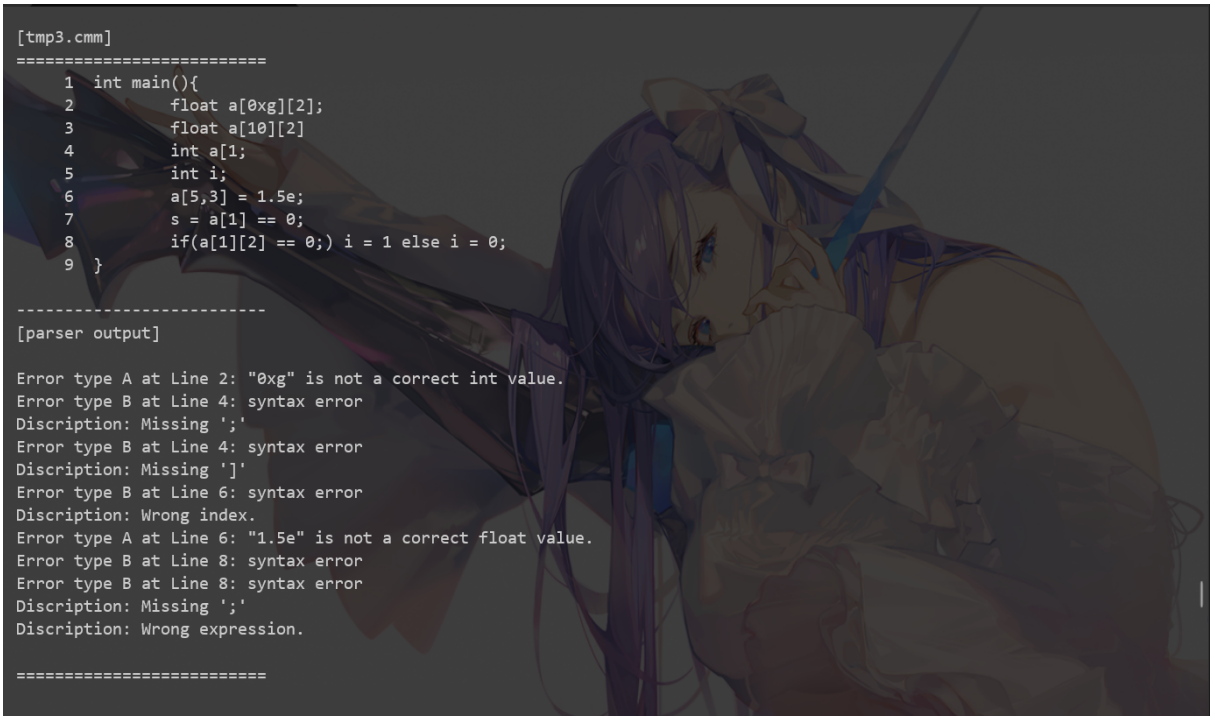
bison遇到error（即下推自动机无法继续运行）时的默认动作是结束分析（如果没有定义接收error的语法规则），这使得只能识别出至多一个语法错误，因此需要指定特殊语法规则以进行错误恢复。

本项目实现了针对若干特定类型的错误恢复，部分相关代码：

```
| VarDec LB error RB    { printf("Discription: Wrong index.\n"); yyerrok;}
| VarDec LB INT error   { printf("Discription: Missing ']'\n"); }

| ID LP error RP        { printf("Discription: Wrong args.\n"); yyerrok; }

| IF LP Exp RP error    { printf("Discription: Wrong if Stmt.\n");yyerrok;}
| IF LP error RP Stmt   { printf("Discription: Wrong expression.\n"); yyerrok;}
```



树结构的设计与打印

本项目实现的树结构以单向链表存储子结点，`firstChild` 指针指向第一个孩子，`next` 指针指向兄弟结点。在连接父子结点时，检查父亲的`firstChild`，为空则置该子结点为`firstChild`，否则从`firstChild`开始，找到最后一个子结点，用`next`连接。

打印部分，为了打印出足够美观易读的树结构，`printTree` 函数使用了UTF-8制表符，并且维护一个`flag`数组来标识当前深度是否需要打印竖线，部分代码如下。

```
flag[depth] = 0;
    for (int i = 0; i < depth; i++) {
        if (i == depth - 1) {
            // 检查当前结点是否是其兄弟中的最后一个可打印结点
            TreeNode* checker = now;
            int check = 0;
            while (checker -> next)
            {
                checker = checker -> next;
                check |= checker -> nodeType != NULL_SYN;
```

```
            }
            if (check) {
                printf("├─");
            } else {
                printf("└─");
                flag[i] = 1;
            }
        } else {
            if (!flag[i]) printf("│ ");
            else printf("  ");
        }
    }
```

编译与调试

`Makefile` 功能丰富，可以进行bison调试，并且无需手动`make clean`。

开启bison调试需要修改`main.c`，并添加bison编译选项`-t`，在`Makefile`中使用了sed命令进行匹配替换，完成了对main.c的自动化修改。

执行`make debug`，可以生成调试器`debug`，用法与`parser`一致，会打印出bison栈的记录。