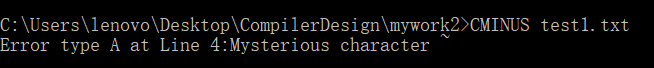
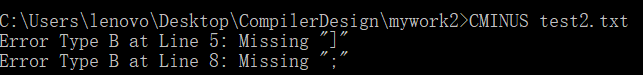
1. **实现的功能**
2. 发现**A类错误**：有未定义的字符。

test1.txt中放的是样例1的代码



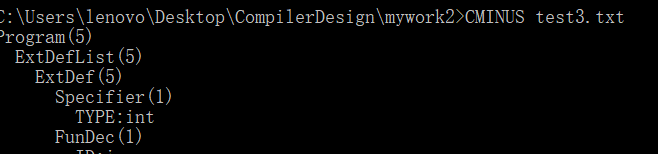
1. 发现**B类错误**：缺少字符。

test2.txt中放的是样例2的代码



1. 当没有语法错误的时候，打印**语法树**

Test3.txt中放的是样例3的代码，**为节省篇幅，只截图了一部分**



1. 可以将文件名作为命令的参数，这样可以简化输入——将代码写到文件中，而不用在命令行输入
2. **如何编译与使用**
3. **编译环境** 系统版本：windows10 版本号：1903

gcc version 6.3.0 (MinGW.org GCC-6.3.0-1)

flex version 2.5.4

bison (GNU Bison) 2.4.1

1. **编译项目**

打开命令行，定位到项目的工作目录，

输入 make CMINUS.fb

即可编译并生成 CMINUS.exe可执行文件

1. **使用项目**

有两种使用方式，第一种： CMINUS [文件名]. 如 CMINUS test1.txt

第二种，输入CMINUS ，回车。然后将待分析的代码输入到命令行中，再Ctrl+Z，结束输入。

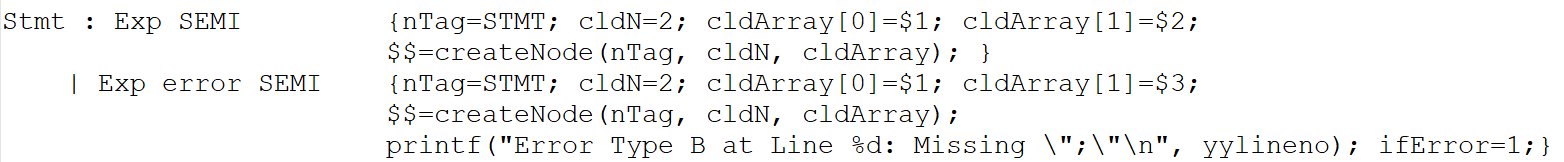
1. **实现思路**
2. 功能1：发现并定位A类错误, 未定义字符

这个可以直接在词法分析器中完成，按照flex的匹配优先规则——最长匹配，最先匹配。我在规则段末尾定义了一条规则：

. {ifError=1; printf("Error type A at Line %d:Mysterious character %c\n", yylineno, yytext[0]); return 1;}

这样可以保证，当发现了之前未定义的符号时，会输出错误

1. 功能2： 发现B类错误：缺少字符

这是一种语法错误，所以我利用bison的恐慌规则，分析器在一个含有错误恢复规则的状态出错时，首先从分析栈中弹出语法符号直到能 将 error 移进，在栈顶形成错误恢复规则中 error 前的句型为止。一个实现例子如下：

其中最后一句是打印错误，其它语句是为了构造语法树。

1. 功能3 ：当没有语法错误的时候，打印语法树

构造语法树主要分成了三个部分：叶子节点的构建、非叶子节点的构建、语法树的打印输出。关于语法树的节点定义、函数声明放在了tree.h中，函数定义放在了tree.c中。

1. 节点数据结构定义如下：

struct Node

{

int tag; // 节点类型

char\* value; // 节点值，主要针对id，type

struct Node \*childs[10]; // 子节点

int nchilds; // 子节点个数

int lineno; // 行号

};

然后我在.y文件里面将终结符和非终结符的语义值类型设置为Node\*。

当词法分析器分析到每个词法单元时，调用createLeaf() 函数生成相应的叶子节点。对于type、id等类型，除了设置tag以外，还需要设置节点值value。例子如下

"int" {yylval.val = createLeaf(TYPE, yytext); return TYPE;}

然后就是在语法分析的时候生成非叶子节点，首先在.y文件中设置全局变量 int nTag; struct Node \*cldArray[10]; int cldN; 分别用来记录每次规约对应非终结符的节点类型，子节点，子节点个数。然后创建相应的节点，并将语义值设置为指向该节点的指针。如下代码：

VarList : ParamDec COMMA VarList {nTag=VARLIST;

cldN=3; cldArray[0]=$1; cldArray[1]=$2; cldArray[2]=$3;

$$=createNode(nTag, cldN, cldArray); }

产生式右边有3个部分，也就是说VarList 的子节点个数是3，然后将 cldArray[0]， cldArray[1]， cldArray[2] 依此设置为指向子节点的指针。

最后，在规约非终结符Program时判断是否有语法错误，如果没有，就调用函数 treePrint ()打印该语法树，该函数直接使用递归的先序遍历方法遍历该语法树，再利用树的深度控制行首的缩进，就可以打印出语法树。

1. 功能4 ：将文件名作为命令的参数

首先判断参数个数，如果大于1，则说明有第二个参数作为文件名，打开文件，使用flex提供的yyrestart(f)使词法分析器读取标准输入输出文件f。

1. **总结与收获**

在本次实验中还是遇到了不少困难，其中最大的一个困难是如何打印出语法树，并且在非叶子节点后面打印出非终结符在展开时的行号。我的第一个方案是在产生式中间添加语义动作和副作用，边分析边打印语法树。首先我直接在尾部添加副作用，发现这样打印出的节点的顺序是不对的。然后我考虑在首部添加副作用，然后发现这样会产生太多的语法冲突，以至于语法分析器不能正常移进规约。再然后我想到bison采用的是LALR(1) 分析，可以向前看一个符号，然后我将一些产生式的语义动作从首部移到中间，重要是为了消除移进规约冲突，我这样做了过后确实消除了冲突，但又遇到了一开始的问题——打印的顺序不对，不是按照语法树的先序遍历顺序打印的，而且我发现行号的打印也没有预计的顺利。

于是我考虑使用第二个方案——构造语法树。我开始一直不想采用这个方案，一是觉得太麻烦，而是这样也不能按要求打印出行号。我还是只能再规约的时候把行号赋给节点，这样打印的结果和老师给的测试用例的结果稍微有一点点区别。但最后想的是退而求其次，还是使用语法树吧。我觉得语法树这条路算是走对了，虽然没有解决行号的问题，但感觉这样的语法分析思路才和理论课上讲到的语法分析器一致——语法分析器的输出是语法树，实现起来并没有我想象的困难，而且收获比较大。

在做这次大作业的过程中，我还学习了很多其它的知识，为了有个积累，我按内容分类写了几篇博客。链接如下：

Flex： <https://blog.csdn.net/qq_41956860/article/details/103010505>

Bison： 待审核

Makefile：<https://blog.csdn.net/qq_41956860/article/details/103017062>

Gcc： <https://blog.csdn.net/qq_41956860/article/details/103012404>

1. **待改进的点**

主要是有两个点，一个是行号，例子的要求的行号应该是非终结符在被展开时对应的行号，但bison是自底向上的，所以我打印的语法树里面非终结符对应的行号是被规约时候的行号，我现在也不清楚怎么弄成例子里面的效果。

第二个是符号表，由于时间的原因我没有弄符号表，只是把相关的数据结构定义了一下，如果有时间的话还是可以弄一下。