

编译原理：实验一

姓名：黄秉焜 学号：171240513

2020 年 4 月 17 日

第一部分 实验一

1 实现了哪些功能

完成了所有必做和选做要求.

其中错误恢复部分完成度不高，很多个错误或某些类型的错误，程序不一定能恢复，而是陷入 Panic.

2 如何实现相应功能

- 必做部分的词法分析，书写相应的正则表达式完成，若有符号无法被正则表达式匹配，则输出错误类型 A, mystery character.
报错都输入到 stderr 中.
- 选做部分的词法分析，各进制数由正则表达式匹配成功后，用 `strtol` 函数转化为 `int`。此外，写了额外的正则表达式来匹配错误的词法单元，如 0 开头的多位 0-9 数字串，如果没有被正确的十进制和八进制正则匹配，则认为是错误的八进制，报词法错误，错误信息为 `Illegal octal number`，错误的十六进制，浮点数，标识符都写了相应的正则表达式来匹配。
多行注释参考了 flex 手册，由状态完成，其中额外对 «EOF» 进行了检测，若在多行注释的匹配过程中读到了文件结束，则重置状态为 `INITIAL`，并报 `Incomplete multiline comment` 的词法错误.
- 语法分析部分，重点是 `YYSTYPE` 的设计，考虑到要打印解析树，暂且定义为一个结构体：

```
struct Node {  
    int token;  
    char* symbol;  
    int line;  
    int n_child;  
    struct Node** child;  
    union {  
        int ival;  
        float fval;  
        double dval;  
        char* ident;  
    };  
};
```

其中 token 记录该值的 token 值，在词法分析过程中赋值，非终结符的 token 值为 0，symbol 代表该变量的字面量，如 `FLOAT`，`W_FLOAT`(错误的浮点数) 等，主要用来在打引解析树时输出，line 记录该单元出现的位置，child 则记录其子树，最后的匿名结构体存储该终结符或非终结符的值，如 int 值，标识符的值等等。

关于空程序的判定，主要看非终结符 Program 的 ExtDefList 有无子树，若没有，说明是空串，讲 Program 的 yylval 中的 line 值修改为 yylineno，以指示 EOF 所在位置。

错误恢复部分主要是 `error SEMI` 和一些有括号匹配的地方。具体见 `syntax.y` 中产生式的书写。

使用了一个全局变量 `errors` 来记录文件中的词法错误和语法错误总数，若没有错误则会打引语法树。

对负号和减号的区分，在语法部分完成，使用 `%prec`，定义了 `NEG` 算符来定义负号的优先级。

调用了两个实验讲义未提及的 API，用以输出更详细的语法错误信息和定义 `YYSTYPE`:

```
%define parse.error verbose
#define api.value.type {struct Node *}
```

3 程序如何被编译

在 Code 目录下输入 `make parser` 即可，会生成可执行程序 `parser`，对于多个文件 `file1.cmm file2.cmm ...`，按照如下输入指令即可：

```
parser file1.cmm file2.cmm ...
```

☺ **Remark** 由于使用 docker container，程序文件修改时间可能有问题，因此 `make` 可能出错，如有提示 “Clock skew detected. Your build may be incomplete.” `make clean` 之后重新编译也许能解决问题。

4 其他

- 原来的 `YYSTYPE` 不是结构体指针，而是结构体，试图用 `&$$`，直接获得指向 `yylval` 的指针，但后来发现 `&$$` 的地址是固定，原来只是每个终结符或终结符的引用，打引出来的解析树有问题。
- 虽然有两个 git 分支，但已经合并过，应该都可以用于测试。
- 感谢同学们提供的测试文件

第二部分 实验二

5 实现了哪些功能

实现了所有必做和选做要求

其中内存回收做的不是很好，大量的 `malloc` 可能导致程序在处理超长测试样例时栈溢出

6 如何实现相应功能

实验二的全部代码在 `syntab.h` 和 `semantic.h` 中

整体上依照实验一生成的语法树来进行语义分析，由 `Analysis` 系列函数分析，从 `SemanticAnalysis` 开始分析语法树的根节点，而后按照不同的产生式逐步调用多个函数

对函数功能做了尽可能的拆解，故每个函数都不会太长，也能根据函数名和开头注释（部分）理解相应的含义，除了 `ExpAnalysis` 有点长

符号表的数据结构，综合运用了哈希表和链表，其中结构体的定义都存储在链表中，函数的声明和定义也都存储在链表中。全局变量和局部变量则存储在以链表为结点的哈希表中，按照实验指导，局部变量会存储在一个表栈链表中，离开 `CompSt` 后，会删除相应的变量

数据结构上基本采用了实验指导给出的样板，自定的函数表存储了函数的名字，初次声明或定义的行号，返回类型，参数类型，是否被定义

专门设计了一个表达式结构体，并修改了语法树结点，添加了 `eval` 和原来的 `ival`, `fval`, `ident` 组成匿名联合体来存储结点的值（如果有的话），其中表达式结构体涉及到具体的值（用 `double` 存储），表达式类型，是否是右值

数组类型的构建，使用了一个可变长的 `size` 数组，在追溯非终结符 `VarDec` 的过程中，不断记录数组的每一维的长度，并在超出上限时，使用 `realloc` 扩大指针所指空间

函数定义的时刻为解析 `CompSt` 时，若该 `CompSt` 所属的函数还没有被定义，则获取它的参数列表，将其加入局部变量，并修改函数属性为已被定义

7 程序如何被编译

在 `Code` 目录下直接 `make` 即可，生成可执行程序 `parser`，`GDBMakefile` 是为了调试程序使用，不用理睬

8 心得总结

实验总体较顺利，其中有几个点值得关注

- 尽可能地拆解了函数功能，不让一个函数过长，这样很容易定位错误，同时也容易想清楚逻辑上的问题

- 尽可能的采用了递归的想法设计函数，虽然没有非递归高效，但同上，更容易发现逻辑漏洞
- 使用了大量的 `assert`，在调试过程中让错误出现时间和程序 Bug 位置更加接近
- 熟练使用 `gdb`，看堆栈，打断点，十分高效地解决了很多段错误