编译原理 实验 1: 词法分析和语法分析

冯诗伟 161220039

1 实现的功能

1.1 词法分析

使用 Flex 工具，将 C - -源程序解析为一个个词法单元，作为接下来 Bison 语法分析的输入。对于不合法的输入，比如数字开头的 ID、未定义的符号 (#、@、...) 能够识别并且报错。

实现方式是使用正则表达式对输入进行识别。根据实验手册中的定义，每识别成功一个 token 就返回相应的类型。对于 INT、FLOAT、TYPE、ID 这四种 token 还要返回它的属性值 (如数值大小、变量名称)。

需要注意的是 ID 的识别要放在最后，因为语法中的关键词是不能作为变量的名称，只有变量名称不是已定义的关键词才能被成功识别。同时最后需要加一条通配符，以检测非法的输入。

1.2 语法分析

使用 Bison 工具，对 Flex 分析出来的词法单元进行语法分析。对于合法的输入，构建语法树并输出；对于不符合文法的输入，确定错误的类型的位置，并进行错误恢复以推进接下来的语法分析。、

实现方式是在 Bison 源程序中定义语法，并在识别出某条产生式后进行相应的操作，比如语法树的创建 (如何创建见 section2)。如果检测到非法的输入，即找不到匹配的文法，则启动相应的错误恢复例程 (如何检测并恢复见 section2)。

分析结束后，查看用来记录“是否有语法错误”的全局变量，如果没有错误则打印构建好的语法树，否则不打印。

2 (自认为的) 程序亮点

2.1 语法树的创建

主要思想是自底向上的构建语法树 (顺应语法分析的方向)，由小的子树合并成大的子树。关于语法树的创建主要使用下面四个函数:

```
1 MultiNode *createMultiTree(const char *name);
2
3 void insertNon(MultiNode *parent, MultiNode *child);
4 void insertTerm(MultiNode *parent, char *childName, int lineno);
5 void insertTermAttr(MultiNode *parent, char *childName,
6                     char* attribute, int lineno);
```

其中 MultiNode 是自定义的数据结构，代表一个节点，里面存储该节点的类型、属性值、子节点个数、子节点列表、该节点对应的输入的最小行号 (first_line) 等。

createMultiTree() 是每匹配成功一条产生式就会被调用一次，以创建产生式左边的单元对应的节点。接下来三个关于 insert 的函数分别代表插入非终结符、无属性的终结符和有属性的终结符。不同函数需要的参数不同，在此不再赘述。

在插入的过程中，除了更新新节点的子节点列表，还要更新新节点对应输入的最小行号。正好利用树的递归构建的过程，每插入子节点时，父节点的最小行号都更新为它和这个子节点最小行号的较小值，这样当语法树构建结束后，每个节点的最小行号也就得到了。

关于语法树的打印，只需要在开始符号被归约成功时调用。打印采用递归方式，每一层记录当前的 depth 以方便打印出前面的缩进。

2.2 错误检测和恢复

错误检测和恢复是本次实验最诱 (dan) 人 (teng) 的部分。难点之一是很难穷尽所有的情况，难点之二是错误定位的层次很难把握，若层次太低则掌握的信息不够，有些错误无法检测，若层次太高则会跳过很多输入，低层的错误会被直接忽略。对于如此的 paradox，我采取“结果驱动”的炼丹思想，以奢求拥有较准确的错误检测。

在错误检测阶段，可以把能想到的情况先列出来，然后根据其层次的情况选择一些合适的产生式的错误进行检测。其实检测的任务都是 Bison 完成的，我们自己只需要考虑如何错误恢复。

在错误恢复阶段，Bison 在检测到非法输入时会进入 error 状态，所以把 error 也当成是一种 token，然后相当于添加的新的规则，这样就可以对错误的情况进行归约，进而可以继续对后面的输入进行分析。比如对于 Stmt: RETURN Exp SEMI 这条产生式，可以新增一条 Stmt: RETURN error SEMI, 这样就可以检测中间缺少返回值的情况；也可以再新增一条 Stmt: RETURN Exp error，这样就可以检测缺少分号的情况。

在错误恢复例程的具体实现方面，我采用与操作系统系统调用类似的做法，给我能想到的每种错误一个编号，然后当 Bison 检测到错误之后就启动错误恢复例程，把当前错误的类型号和出错位置传进去，然后错误恢复例程根据错误类型号和出错位置把错误信息打印出来供用户参考。这样的好处是可以复用一些错误类型，同时方便管理相应的报错任务。

```
#define ERR_MISS_SEMI 1
#define ERR_MISS_COMMA_VARLIST 7
#define ERR_MISS_RP_FUNC 15
#define ERR_INDEX 12
#define ERR_EXPECT_EXP 11
...
void errorReport(int errno, int lineno){
    HAS_ERROR = 1;
    switch (errno){
        case ERR_MISS_SEMI:
            printf("Error Type B at Line %d: Missing \";\".\n", lineno);
            break;
            ...
    }
```

```
15 }
```

我目前定义了 15 种错误，主要分为三类，一种是 missing 一些东西，一种是出现违法操作 (如非法的索引)，还有一种是表达式缺少左/右值。

关于报错信息，我不仅提供了语法上缺少什么符号，还试图在报错信息中提供较为准确的”语义”信息，这里的语义仅指帮助用户更明确怎么修改程序。对于 missing 的情况，我考虑了不同场景进行了更细的处理，比如 return 语句缺少分号、stmt 语句缺少分号、参数列表缺少逗号、高层声明缺少分号；违法操作也考虑不同场景，比如变量声明出错、数组索引出错；关于表达式，对于“int a=;”的输入，会报错“Expected expression”。