编译原理实验 1 实验报告

鄢振宇（基础班）
171240518

2020 年 3 月 20 日

I 代码框架

```
/Lab
    /Code #存放代码的文件夹
        Makefile #用于编译的文件
        common.h #常用的宏/函数的声明/定义
        handlers.h #包含若干接受node*的函数的声明
        helper.c #一些接受node*作为参数的、具有分析作用的函数
                          #（目前只有一个，以后语义分析再增加）
        lexical.l #词法处理工具flex的源代码
        main.c #包含main函数和若干
        printer.c #一些接受node*作为参数的、用于打印的函数
        syntax.y #语法处理工具bison的源代码
    README #框架包含的一个自述文件
    /Test #诸多测试文件，.cmm后缀表示C—源代码
          #.out后缀表示预期的输出（用于本地测试用）
        ...
        syn*.cmm #测试语法错误可读性为主的C—源代码
    parser #预先编译好的二进制文件
    report.pdf #报告
```

II 我的程序应该如何被编译

使用给定的 Makefile

III 我的程序实现了哪些功能

我的程序按照要求实现了对词法错误和语法错误的检测和报告，并且尝试实现了可读性略高的语法错误提示（参见 Lab/Test/syn*.cmm 和对应的.out）

词法错误

为了方便语法阶段的处理，对于可被判定为词法错误或语法错误（或语义错误）的内容，我一般尽力把它归类到词法错误，诸如 09, 0x3G, 1.05e, 5. 等等不合法的整型/浮点数，都被我归类为词法错误

我采用的判据是“这样的输入流是否可能是某个合法程序的一部分”，如果一段输入（比如上面提到的那些）不可能在任何合法程序中出现，则可以考虑将其作为词法错误。

比如说.e-5，可能很多人会以为它是非法浮点数，然后在词法阶段把它处理掉。但是，假设有个结构体变量 x, 包含名为 e 的 INT 成员变量。那么 x.e-5 就是有意义的程序的片段，而词法分析器会先读取走 x，余下.e-5。如果.e-5 被当做一个非法浮点数处理，后面的分析便谬之千里。

对于非法整型/浮点数，由于相同长度匹配最前规则的特性，正确的整型/浮点数规则只要放在最前，必然能被正确识别，而我们在考虑错误的浮点数的时候，就可以直接写成 {digit}*.{digit}* 而不必特别地写成 ({digit}+.)|(.{digit}+)

语法错误

为了能让语法错误的报错更便于阅读，我采取的方式是，将 yyerror 声明为空函数，将报错放在各个产生式对应的语义动作中。虽然此处理方法较为暴力，但是至少是一个可行的方法。

我的程序能够报告简单的 Missing ']', Missing ';' 等语法错误

我还结合了另一种处理方法，即在 syntax.y 中加入常见错误（如缺少右括号）对应的产生式。如 Exp→LP Exp RP 是一个合法的产生式，对应的，我们可以加入 Stmt→LP Exp SEMI，然后在语义动作中进行报错"Missing ')'"

IV 自认为的亮点

跨行注释的处理

C–中的/*-*/注释是词法分析的一个难点，尤其是需要判断其是否 unterminated。在上网查询之后，我查到三种解决方案。

一种是利用语义动作，先识别"/*"，剩下的操作由语义动作完成。但是这样写，代码较为繁杂，也容易出错。

或者利用正则表达式。由于 flex 的 * 是贪婪的，所以表达式较为复杂

```
\/\*([^*]|(\*+[^*/]))*\*\/
```

我最后采取的方法是利用了 flex 的状态（state）读到"/*" 就进入特殊的状态，在碰到"*/" 之前，都会舍弃输入的所有字符，如果碰到 EOF 则报错

语法树节点的生成

对于实验 1 而言，不同产生式生成语法节点的规则其实非常接近，就是记录下当前节点的名字、所有的后继的指针

我实现了 Node 函数（见 common.h），它通过变长参数的特性，能够非常容易地生成对应的语法节点。

我又考虑到 syntax.y 的语法特性，对于由两个符号构成的节点，可以写成

```
$$ = Node(name, @1, 2, $1, $2)
```

但是，这样写仍然有些许繁琐，所以我阅读了 syntax.y 编译成的.c 文件，了解了 @1, $1 等对应的 C 语言代码。这样一来，由两个符号构成的节点，可以写成

```
$$ = Node2(name)
```

函数指针实现"OOP"

考虑到在以后的实验的词法分析，我们可能需要根据节点的类型（名字）的不同，采取某些不一样的操作。为了防止使用大量的 if-else 来判断。于是，我在 node 结构体中，加入了一个名为 func 的成员，它的类型是接受一个 node*，返回空的函数指针。在词法/语法分析的阶段，每个词法单元生成 node 的时候，func 都会被赋上合适的值或 NULL。由于实验一只要求打印整型/浮点数/类型/标识符的具体信息，所以目前我仅为这四种词法单元设计了函数，并且这个函数目前只有打印的功能。（函数定义见 printer.c）

在 main 函数的 preorder 函数中，就可以利用这个特性，来打印需要打印额外信息的词法单元：

```
if(cur -> func) {
    cur -> func(cur);
} else {
    puts(cur -> name);
}
```