# 编译原理实验二：语义分析与中间代码生成

郭松 2015301500205

January 1, 2018

我基本完成了要求的全部内容。另外，我的代码还支持以下超出假设的内容，由于我是先写中间代码生成，然后逐步加上语义分析的功能，所以有些语义分析的功能会受到中间代码生成的制约（这是因为我读 Project2 的要求的时候读错了，当我注意到这一点的时候，Project2 已经快写完了）：

1. 支持多维数组，支持结构体，支持结构体套结构体，支持结构体数组，支持结构体套数组套结构体，结构体赋值，但是结构体和数组之间的赋值操作没有经过测试，我相信它在大多数情况下是正确的。
2. 局部变量可以和全局变量重名，特别的，局部变量显然会掩盖全局变量的作用域。
3. 对于有些错误，为了让翻译工作继续完成，我并没有直接退出，这可能会使得代码在后续报出一些其它的错误，这一问题主要出现在和 Exp 相关的语句上，当出现错误的时候，Exp 会返回一个类型为 int 的临时变量。
4. 为了支持继承属性，使用了 2 pass 而不是 1 pass，这有一定的效率问题，但是代码相对而言优雅很多。

## 1 编译方法和项目结构

### 1.1 编译方法

与和项目一起的，有一个 Makefile 文件，进入含有 Makefile 文件的目录，直接调用 make 即可，生成的可执行文件叫”ejq_cc”

代码在 macOS 10.12.3, GCC 7.2 和 Ubuntu 17.10, GCC 7.2 下编译和测试通过，理论上所有支持 c99 标准的编译器都能通过编译

### 1.2 项目结构

project1.h 是实验一的部分代码抽取、整理出来的内容
project2.h 是实验二的中间代码生成所需要的结构体定义
project2.2.h 是用到的一些符号常量的定义
symbol.c 是符号表的定义
trie.c 是可持久化字典树的定义
translate.c 是中间代码生成的核心代码
second.y 仍然是项目的主文件

## 2 翻译

我觉得我的翻译部分没有什么好讲的，如果别的同学做完了的话，我的东西和他们的应该是大同小异的。另外，我有自信别人要是抄我的代码的话，绝对弄不清楚某些细节的实现（比如下面的可持久化线段树）。

对于符号表的实现，方法和书上的没有太大区别，特别的，为了实现在符号表中快速地查找代码，我使用了名为“可持久化 trie”或者叫“可持久化字典树”的数据结构来实现这一功能。在提交的源代码中，可持久化字典树的代码已经写在了”trie.c” 这一文件当中。

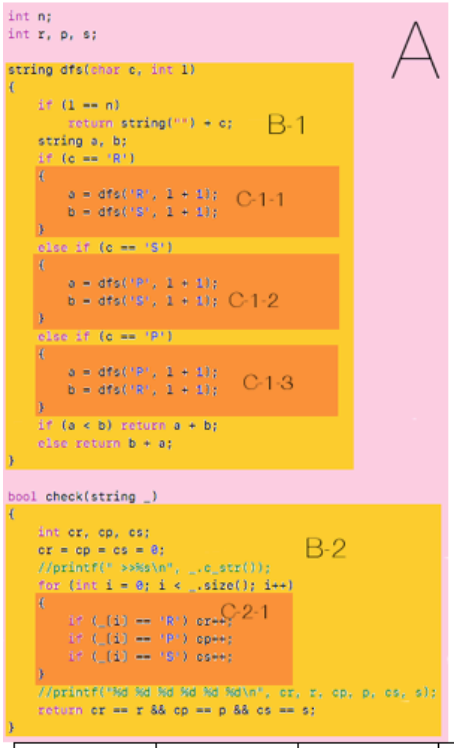
# 3 可持久化字典树

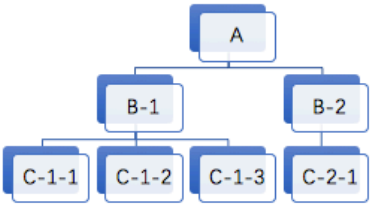
一般的编译器中，每个作用域对应一棵新的字典树，当我们需要查询变量的作用域的时候，会从当前的字典树开始查找，如没有查询到，则到父作用域里面查找，以此类推，直到找到根作用域。这种实现需要维护多棵字典树，同时在查询的时候也会有些复杂，如需要遍历很多棵字典树，其缺点是在查询的时候效率比较低，其优点是往字典树里面添加节点的效率很高。

我的想法是，通过一个统一的方式，把所有作用域里面的变量名统一存储到同一棵字典树里面。这要求我们在字典树的基础上进行改动，以支持下面几种操作：

1. 删除一个作用域里面的所有变量名（相当于在一个作用域结束之后，删除一颗字典树）
2. 查询一个作用域，及其父作用域里面是否有一个变量名

我们考虑把一个作用域画出来，如下图，下面这个作用域可以抽象成右边的树，对右边的树按我们进行翻译的顺序计算出 DFS 序，得：




| A | B1 | C11 | C12 | C13 | B2 | C21 |
|---|---|---|---|---|---|---|

我们发现，同一层级的节点互不相交，子节点被父节点完全包含，这一形式为我们删除一个特定作用域下的所有单词创造了方便——我们删除一个节点，完全不会影响到其它的节点。

接下来，我引入一个被称作版本的概念，事实上，整个“可持久化”字典树，与现有的版本管理非常相似。

当我们进入一个新的作用域的时候，我们记下当前的版本号，并创建当前版本的“快照”（即记下当前版本的树根——在后面，每一个新版本，都有一颗新的树根），当我们要插入一个单词的时候，我们创建一个新版本，保留与将插入单词无关的节点不动，并将这个单词经过的路径上的所有节点复制并改写——这也是“可持久化”一词的由来，每一个版本都利用了上一个版本中，没有改动的节点，这一技术似乎也被叫做“写时复制”。当我们要从某一个作用域中退出来的时候，只需要简单地将当前的版本号回退至之前创建的快照，如要节约内存的使用，只需删除在这个快照之后创建的所有节点——正如前文所说，它们不会影响到与这个作用域无关的作用域。同时，这样也完成了我们前面所要求的两个基本操作。

最后是对这一数据结构效率的分析。相比于建立多棵字典树，这一数据结构在查询的时候只需要一次字典树查询，即可找到我们需要的单词。但相应的，需要耗费一些额外的内存空间，同时在插入过程中，也需要更多的时间来进行节点复制的操作。但是，在一般的程序当中，变量被声明或者定义的次数要远少于变量被调用的次数，基于“加速热点过程”的取舍，这这样的改变从理论上是可行的（因为没有程序用来计算具体的效率差别）。

# 4 代码段分析

## 4.1 插入与创建新版本

```
int __E_trie_insert(char *s, int item_id) {
    size_t rt = __E_trie_new_version();
    for (int i = 0; s[i]; i++) {
        if (E_trie_nodes[rt].nxt[s[i]]) {
            E_trie_nodes[rt].nxt[s[i]] = __E_trie_fork_node(E_trie_nodes[rt].nxt[s[i]]);
        } else {
            E_trie_nodes[rt].nxt[s[i]] = __E_trie_new_node();
        }
        rt = E_trie_nodes[rt].nxt[s[i]];
    }
    E_trie_nodes[rt].flag = item_id;
    __E_trie_finalize_new_version();
    return 0;
}
```

将普通的 Trie 改写成可持久化 Trie 非常简单，只需要在插入的过程中加入“写时复制”的“复制”部分即可。为了加快效率，我仿照了 C++ 中 vector 中的实现，同时用数组下标模拟指针，相对于直接使用指针，在一定程度上速度会稍快一些，更重要的是这样可以有方便的版本回退（直接丢掉最后若干个元素，而不用逐一 delete）。

## 4.2 查询

```
int E_trie_find(char *s) {
    if (!E_trie_current_version) return 0;
    size_t rt = E_trie_roots[E_trie_get_current_version()];
    for (int i = 0; s[i]; i++) {
        if (E_trie_nodes[rt].nxt[s[i]])
            rt = E_trie_nodes[rt].nxt[s[i]];
        else
            return 0;
    }
    return E_trie_nodes[rt].flag;
}
```

查询操作和一般的 trie 查询操作完全一样。但却能够查询到父作用域和更上层的作用域的节点。同时也因为每次都新建了一些节点，对于与父作用域里面重名的变量，也能轻松对付。

# 5 总结及致谢

完成这一项目前后用时一个月，净代码量 1600 行，算是一个有一定难度的实验，在完成这一实验的过程当中，前后翻了很多次书，也修改过很多次语义分析器和中间代码生成的具体实现操作。总而言之有很大的收获。

如果我没有阅读过一些代码，那么我可能不可能完成实验的第二部分。其中比较有代表性的项目如下：

1. PostgreSQL 数据库管理系统的 SQL 解析部分，pgsql 的代码是用 C 写成的，这使得一些地方不得不使用一些危险的行为来完成，我借鉴了这些危险的行为，主要体现在 project2.h 这一头文件中，对 Project1 里的部分内容的重写，以在 Project2 当中使用可读性更高的代码。
2. 因为我平时使用 C++ 比较多，C 语言缺少一些好用的特性（比如函数重载）为了让自己写起来更舒服，我从 StackOverflow 上面搜索了一些代码段，并将其应用到自己的代码当中，当然，这些代码与这个项目的内容没有关联。
3. 可持久化数据结构和可持久化字典树并不是由我提出的，学术界提出了这些思想，并由算法竞赛的一些前辈们将它们具体化。