


编译原理课程实验——实验3报告

 **实验内容：**中间代码生成。在词法分析、语法分析和语义分析程序的基础上，将C--源代码翻译为中间代码。要求将中间代码输出成线性结构，并输出到文件。

- **功能架构：**借助于语法制导翻译（SDT），为每个主要的语法单元 `X` 都设计相应的翻译函数 `translate_X`，对语法树的遍历过程也就是这些函数之间互相调用的过程。每种特定的语法结构都对应了固定模式的翻译“模板”，本次所使用的模板主要来自实验指南，故相关结构不再赘述。下面主要介绍一些基础设施以及指南中没有提到的翻译模板：
- **中间代码的组织形式：**中间代码的传参和返回值均用字符串组织，这样所的好处是所有函数的接口统一，代码之间的拼接就是字符串拼接。考虑到 C 语言对字符串的支持不佳，尤其是字符串拼接，原生的 `strcat` 函数不能对空间进行分配。故本次实验重新写了一个字符串拼接函数：

```
1 char *strcatm(char *s1, char *mid, char *s2) {
2     if (s1 == NULL || strlen(s1) == 0) return s2;
3     if (s2 == NULL || strlen(s2) == 0) return s1;
4     char *ret = malloc(strlen(s1) + strlen(s2) + strlen(mid) + 4);
5     sprintf(ret, "%s%s%s", s1, mid, s2);
6     return ret;
7 }
```

该函数会造成内存泄漏，但本程序只是编译程序，不是常驻程序，故少量的泄露换取清晰的代码架构是值得的。同时，该函数还为空语句的拼接提供便利，避免出现不规整的空行的出现。

- **`translate_X` 函数的实现：**除了模板之外，对语法树的遍历基本就是匹配产生式+分段翻译并拼接的模式。以 `translate_CompSt` 为例：

```
1 char *translate_CompSt(Node *tree_node) {
2     tree_node = tree_node->child;
3     Node *defList = find_brother(tree_node, _DefList);
4     char *s1 = defList == NULL ? "" : translate_DefList(defList);
5     Node *stmtList = find_brother(tree_node, _StmtList);
6     char *s2 = stmtList == NULL ? "" : translate_StmtList(stmtList);
7     return strcatm(s1, "\n", s2);
8 }
```

由于在 `strcatm` 中对空串支持，故许多产生式匹配不需要完全匹配，只需要在不存在时置为空串即可。考虑到类似 `IF LP Exp RP Stmt [ELSE Stmt]` 这样的结构，这种设计带来许多方便。

- **翻译模式（指南未提及）**：指南中没有给出的翻译模式中有些比较简单，在这里一笔带过：

- `translate_CompSt`：将其拆分为 `translate_DefList` 和 `translate_StmtList` 并拼接。
- `translate_DefList`：通过树的遍历进行递归调用，拆分为若干 `translate_Dec`。
- `translate_StmtList`：通过树的遍历进行递归调用，拆分为若干 `translate_Stmt`。

- **HIGH LIGHT**：对于部分实现还是有可圈可点的地方，尤其是指南中未给出的部分翻译模式：

- `translate_Dec`：该函数需要考虑两件事：为数组分配空间（即 `DEC` 语句）以及为一些变量初始化值。
 - 对于分配空间，实现了 `get_mem` 函数。该函数会通过查表获得变量大小，并返回需要的 `DEC` 语句（`BASE` 类型返回空语句）。同时该函数还进行错误检测——传入结构体类型直接报错。值得一提的是对于数组类型，`get_mem` 函数会将变量名对应的临时变量置为对应的地址。这样做可以实现数组逻辑的统一。否则函数传参需要取一次地址而且还要作出区分。
 - 对于变量初始化，直接调用 `translate_Exp` 函数赋值给临时变量，再将临时变量赋值给变量名即可。
- 函数调用参数列表：不同于指南中提到的类似数组的实现，为了方便进行递归调用，这里使用了链表的数据结构：

```
1 typedef struct ArgsList {
2     char argName[64];
3     struct ArgsList *next;
4 } ArgsList;
```

- 高维数组：对于高维数组，其有关数据操作的部分全部在 `Exp` 节点中，至于其他部分的声明则由 `get_mem` 函数实现。对于数据操作，这里实现了 `get_arrLocation` 函数，接受 `Exp -> Exp LB Exp RB` 节点，并将得到的地址赋值给临时变量。随后的的解引用由调用方添加。

```
1 char *get_arrLocation(Node *tree_node, char *place, Type *eleType) {
2     tree_node = tree_node->child;
3     if (tree_node->type == _ID) {
4         *eleType = read_sem_node(tree_node->val.id)->type;
5         return strcatm(place, " := ", tree_node->val.id);
6     }
7     // ...
```

```
8     char *s1 = get_arrLocation(tree_node, t1, eleType);
9     // ...
10    char *s2 = translate_Exp(tree_node->brother->brother, t2);
11    *eleType = (*eleType)->data.array.elem;
12    int eleSz = size_of_arr(*eleType);
13    // ...
14 }
```

这里是部分代码。其中的 `size_of_arr` 函数接受 `Type` 并返回字节大小。值得一提的是 `eleType` 这个参数，其在函数递归调用的最底层完成赋值，并在回调过程中一层层解开。这是由于越靠后的函数调用对应的数组元素的类型越大。

- **代码编译：**使用OJ默认的makefile即可编译，这里同时提供本项目的makefile内容：

```
1 all: compile
2 compile:
3     bison -d -v syntax.y
4     lex lexical.l
5     gcc translate.c semantic.c syntax.tab.c main.c syntax.c -lfl -o scanner
```