编译原理实验3：中间代码生成

实验报告

吴澄杰　　151220122　　wuchengjie@smail.nju.edu.cn　　2018年6月17日

一、实验进度

此次试验，在词法、语法分析和语义分析的基础上，利用在之前实现的抽象语法树结构，我实现了 C 语言代码的中间代码生成程序。该程序可以将语法语义正确的 C 语言代码翻译成符合实验要求规范的中间代码，并输出到文件中。

我完成了实验的必做要求：程序可以在简单整型变量、所有变量均不重名等条件下，完成包括条件语句、循环语句、函数调用等部分的中间代码分析和生成。此实验将抽象语法树转换为线型的中层次中间代码。我通过了所有必做测试样例。

二、编译与实验

在 main.c 中设置宏变量 LAB 值为 3，然后在工程目录下，输入 make 命令，即可完成编译，可生成程序 parser。工程目录下的 example3 文件夹中有实验 3 的 2 个必做样例的源代码和我的程序输出的结果。若要测试程序在样例上的运行，只需将样例文件名和输出中间代码结果的文件名作为参数给到 parser 运行即可。比如，./parser ./example3/example3 ./example3/example.ir 就可以得到程序在第一个测试样例上给出的中间代码，并写入文件 ./example3/example.ir 中。

三、具体功能描述

中间代码生成的过程和语义分析有相似之处，都是通过遍历抽象语法树，并且递归地将树型结构转换为线型结构。在具体实现上表现为，定义一系列 translate 函数，实现对抽象语法树的递归翻译。涉及到中间代码生成的定义在 translate.h 中，相关实现均在 translate.c 中。

1. 中间代码数据结构

中间代码以链表的形式进行存储。相比静态数组，用链表来表示的中间代码可以更加方便且高效地进行插入、合并、移动等操作。每条中间代码是一种三地址代码，共有 op1、op2、result 三个运算符 operand 位。每条中间代码用 kind 属性指明其是何种指令。

运算符有常数、变量、符号、标记（label）等类型。若是常数，则将常数的值存储在运算符的结构体中；若是其他类型，则将相应符号的指针存在运算符结构体中。

中间代码用 CB 结构体来管理。它有 begin 和 end 两个指针，分别指向当前代码块的第一条指令代码和最后一条指令代码。用 CB 结构体来管理代码块，可以非常方便地完成代码块之间的组合和合并。具体定义如下。

```
struct Operand {
    enum op_kind kind;
    union {
        int value;
        struct Symbol_t *sym;
    };
};
struct InterCode {
    enum ir_kind kind;
    struct Operand op1;
    struct Operand op2;
    struct Operand result;
    struct InterCode *next;
};
struct CodeBlock {
    struct InterCode *begin;
    struct InterCode *end;
};
typedef struct CodeBlock CB;
```

2. 使用翻译模式进行翻译

使用翻译模式进行翻译是整个中间代码生成的核心。此次试验中，我采用递归下降的方法执行翻译，从抽象语法树的根节点一步步向下，用多层次的 translate 函数执行翻译。其中，继承属性可以通过函数参数一步步向下传递。

```
CB translate_IR();
CB translate_function(struct Node *extdef);
CB translate_FunDec(struct Node *fundec);
CB translate_CompSt(struct Node *compst);
CB translate_DefList(struct Node *deflist);
CB translate_StmtList(struct Node *stmtlist);
CB translate_Stmt(struct Node *stmt);
CB translate_Exp(struct Node *exp, struct Symbol_t *place);
CB translate_Cond(struct Node *exp, struct Symbol_t *label_true,
                  struct Symbol_t *label_false);
CB translate_Args(struct Node *args);
```

按照实验讲义中提供的基本表达式、条件语句、循环语句、条件表达式、函数调用等语句的翻译模式，可以方便地实现这一部分的翻译函数。其中，要在合适和位置生成临时变量做存储，要恰当地生成 label 和无条件、条件转移语句完成控制流的转移。

上述是我定义的一系列 translate 函数。它们都以当前抽象语法树上的结点作为参数。其中 translate_Exp 和 translate_Cond 函数还分别将待赋值的变量指针 place 和待确定位置的标签 label_true 和 label_false 作为继承属性参数传入。

3. 独立的中间代码生成模块

如实验讲义中所说，中间代码生成可以包装成单独的模块，也可以放在语义分析的过程中同时实现。我选在了将其设置为单独的模块，这样可以具有更高的灵活性。中间代码生成和语义分析所使用到的结构信息还是有不同的，并且为中间代码生成部分和语义分析部分单独维护符号表这样的数据结构，会使得实现更简单、更直观。语义分析过程中，为

了找到所有的语义错，不可避免地会产生较多复杂逻辑细节。中间代码生成基于无语义错这一事实，因此其实现代码可以更加简洁高效。

四、实验总结

此次实验我实现了一个 C 语言代码的中间代码生成软件，可以依据抽象语法树分析结果生成相应的中间代码。翻译过程使用了递归向下的方法，对不同的产生式设定了相应的翻译规则，并且将继承属性通过函数参数向下传递。我使用了链表的方法存储并管理中间代码，以此来实现高效的合并排列。