编译原理实验报告

实验四：目标代码生成

高亦远 151160014

一、 实验目的

实验四的任务是在词法分析、语法分析、语义分析和中间代码生成程序的基础上，将C--源代码翻译为 MIPS32 指令序列(可以包含伪指令)，并在 SPIM Simulator 上运行。

二、 实验要求

将实验三中得到的中间代码经过与具体体系结构相关的指令选择、寄存器选择以及栈管理之后,转换为 MIPS32 汇编代码。输入是一个包含 C--源代码的文本文件,你的程序需要能够接收一个输入文件名和一个输出文件名作为参数。例如,假设你的程序名为 cc、输入文件名为 test1.cmm、输出文件名为 out.s,程序和输入文件都位于当前目录下,那么在 Linux 命令行下运行./cc test.cmm out.s 即可将输出结果写入当前目录下名为 out.s 的文件中。输出文件应为相应的相应的 MIPS32 汇编代码。

3.1 程序结构及功能

实验四的程序的源文件主要由以下文件组成：
词法分析文件 lexical.l；
语法分析文件 syntax.y；
语法树相关代码 tree.c 及其头文件；
语义分析相关代码 semantic.c 及其头文件；
中间代码生成相关代码 intercode.c 及其头文件；
目标代码生成相关代码 objectcode.c 及其头文件
程序入口 main.c。

实验四在实验三的基础上，新增了目标代码生成文件 objectcode.c 及其对应头文件 objectcode.h。在 objectcode.c 中定义和实现了生成 MIPS32 代码的相关操作。在 main 函数中也有相关变动。

使用 makefile 可以快速进行编译。经过编译后生成中间文件 lex.yy.c、syntax.tab.c、syntax.tab.h 及执行程序 complier。

测试时在根目录下使用 make 命令进行编译，使用./complier test1.cmm out1.s 来对测试文件进行测试，并将结果输出到文件中。

本程序实现了将 C—源代码翻译为 MIPS32 代码的功能，使之能在 SPIM Simulator 上运行。当源码通过了语法和语义检查后，会输出对应的目标址码。

3.2 功能实现

3.2.1 数据结构

在 objectcode.h 中定义了变量描述符表 VarDesciptor、寄存器描述符表 RegDesciptor 和栈描述符表 StkDesciptor 三个数据结构,其中标量描述符表用链表形式表示,寄存器描述符表和栈描述符表用数组形式描述。

变量描述符表 VarDesciptor 的定义如下：

```
typedef struct Var_t
{
```

```
    int reg_no;          //寄存器编号
    Operand op;          // 操作符类型
    struct Var_t *next;
} VarDesciptor;
```

其中， reg_no 是该变量所在寄存器的编号，0 ~ 31;op 是操作符的类型，如 PLUS_IR、ARG_IR，对应了中间代码的 14 种类型。

寄存器描述符表 RegDesciptor 定义如下：

```
typedef struct RegDesciptor
{
    char name[6];       //寄存器名
    int old;            //存储的时间
    struct Var_t *var;
} RegDesciptor;
```

其中： name 存储了寄存器的名称，如$zero,$at,$v0 等;old 记录了该变量在寄存器中的存储时间，当寄存器不够用时，最旧的变量将从寄存器中被释放。

3.2.2 代码生成

首先在 writeAllObject() 函数中对寄存器描述符的初始化,打印出程序头部信息、 read() 函数和 write() 函数的 MIPS 代码。例如此时会生成一句“_prompt: .asciiz \"Enter an integer:\”，实现 read()函数的提示文字“Enter an integer:”。

然后将实验三的每条中间代码 IRList 逐条转为目标代码。函数 irToObject()定义了一条中间代码到目标代码的转换规则,首先判断每条中间代码的类别,程序中实现了实验三中的 14 种类型:

LABEL_IR、FUNCTION_IR、ASSIGN_IR、PLUS_IR、MINUS_IR、STAR_IR、DIV_IR、GOTO_IR、IF_GOTO_IR、RETURN_IR、ARG_IR、CALL_IR、READ_IR、WRITE_IR

3.2.3 寄存器分配

本程序对寄存器的分配使用了一种简单的局部寄存器分配算法，通过 getReg()函数实现。在寄存器描述符中通过 old 字段记录该变量在寄存器中的存储时间，每个新变量存入寄存器中时，其 old 值将比前一个大 1;当寄存器不够用时，最旧的那个变量（old 值最小的）将被释放，以在寄存器中存储新的变量。

3.2.4 栈和函数

StkDescipter 用来进行栈管理。进行函数调用前先将返回地址$ra 寄存器的内容压入到栈中,然后根据参数个数将参数$a0-$a3 压入栈中，然后将所有变量保存到 StkDescipter 中，并且将变量压入栈中,将对应的寄存器清零,然后将变量链表 VarList 清零；函数调用结束后将之前保存的变量恢复到 VarList 中，并且恢复之前的寄存器，将 StkDescipter 清零；然后恢复参数寄存器$a0-$a3，恢复返回地址寄存器$ra，最后将返回值从$v0 中取出。

在函数体中，先将所有的变量列表和寄存器列表清零，然后将参数从$a0-$a3 中取出来，在函数返回时，只需要将返回值移动到$v0 中，并且跳转到$ra 指定的地址。

在函数 storeAllVar()中，将所有变量寄存器中的内容压入到栈中，并且将变量保存到 StkDescipter 中，然后将变量链表 VarList 清零。

在函数 loadAllVar()中，先将变量链表清零，然后将 StkDescipter 中内容回复到 VarList 中，并且将栈中的内容恢复到寄存器中。