编译原理实验报告四

151220135 许丽军 xulj.cs@gmail.com

零、实验进度描述

完成了所有的必做内容

一、实验内容

从不存在词法错误、语法错误和语义错误的C--源代码,先生成中间代码，最终生成MIPS32指令序列。至此，独立编写完成了一个可以实际运行的C--语言编译器。

二、小组成员

151220135 许丽军

三、实现细节

指令选择

将逐条处理中间代码，翻译为对应的MIPS32指令（逐条是指中间代码与汇编代码是一对一或多对一关系，没有多条中间代码翻译成一条汇编代码的情况。这样也可能会导致了目标代码不够高效）

```
void translate_ir(InterCode* ic)
```

对于函数调用和函数定义，因为前后中间的翻译结果可能是相互关联的，故这几条中间代码采用单独的翻译函数
相互关联是指：

1. ARG语句的数量决定了函数的形参个数，当参数个数大于4的时候需要在栈中保存参数，当函数调用结束后，需要恢复原来的栈顶
2. PARAM语句的数量决定了函数的实参个数，当参数大于4的时候需要从栈中取出参数保存的寄存器中

```
InterCodes *translate_func_param(InterCodes *p)
InterCodes *translate_arg_call(InterCodes *p)
```

寄存器管理

用到的寄存器包括：

1. $zero
   当使用到常数零的时候，用$zero寄存器
2. $t0-$t9 $s0-$s7
   这些寄存器用来加载中间变量
   对于这18个寄存器的分配采用的分配算法是：**局部寄存器分配算法**，即基于基本块进行分配。算法细节同实验指导手册，这里不再赘述。
3. $a0-a3
   用于传递函数调用时的前4个参数

4. $sp $fp
   $sp指向一个函数活动记录的栈顶位置，$fp指向一个函数活动记录的栈底位置
   这里采用了类似Inter x86 ISA的处理方式，$sp = $esp $fp = $ebp
   在某次函数执行过程中，$sp可能是不断变化的，但是$fp是不变的，所以**变量的在活动记录上的地址用其与$fp的偏移地址**表示（一个函数对应这唯一的fp值，且函数内的变量作用域在函数内部，故这种表示是可行的）
   函数开头保存原来的$fp，将$sp的值赋给$fp；函数返回值将$fp的值赋给$sp，并恢复$fp原来的值

栈管理

函数调用时的栈如图所示：

这里采取了把中间代码CALL当作一个基本块，所以CALL之前的所有修改过的变量都溢出到了内存上，所以callee中不必再保存$s0-$s7，即**相当于所有的寄存器都由caller保存**

四、实验亮点

- 对于含有结构体和高维数组的源代码（不作为函数参数和返回值），也可以正确编译
- 整体的代码模块性强，除了主函数所在文件main.c意外，每一个源文件只对应其中一个模块

```
/*****  LAB1 *****/
yyparse();
if(ERROR)
    return -1;
printAST();
/*****  LAB2 *****/
semanticCheck();

/*****  LAB3 *****/
if(SERROR)
    return -1;
InterCodeGen();
//print_ICROOT();

/*****  LAB4 *****/
print_ASMs();
```

- 利用makefile的静态模式规则和多目标简化编译过程

```
SRCS := $(shell find $(SRC) -name "*.c")
OBJS := $(BisonObj_O) $(patsubst $(SRC)%.c,$(OBJ)%.o,$(SRCS))

$(target): $(OBJS)
    $(CC) -o $@ $^ $(LDFLAGS)

$(OBJ)%.o:$(SRC)%.c
    @mkdir -p $(@D)
    $(CC) -o $@ $< $(CFLAGS)
```

五、编译和运行

- 编译并生成可执行目标文件：make
- 运行可执行目标文件： ./mycc input_file output_file
- 清除中间生成文件：make clean-temp
- 清除所有生成文件: make clean