编译原理实验报告

221240073 李恒济*

2025 年 5 月 29 日

1 完成进度

完成了所有的必做任务，主要包括目标代码生成 (oc.c) 的实现.

2 编译方式

使用了课程网站上指定的 Makefile，进入 Code/文件夹后键入 **make** 即可在该文件夹下生成 parser.

3 实现细节

3.1 目标代码生成

生成目标代码可分解成指令选择、寄存器分配、栈管理三个子任务.

3.1.1 指令选择

遍历实验三生成的中间代码链表，逐条生成目标代码. 需要注意的是中间代码中的立即数，我们需要检查操作数是否为立即数，若是，我们直接将之加载进入寄存器而不需要经过内存的计算. 此外，对于四则运算，若两操作数均为立即数，我们可以预先计算结果.

3.1.2 寄存器分配

由于近期精力有限，暂时选定使用朴素寄存器分配算法，不区分变量和局部变量，仅使用 $\$t0, \$t1, \$t2$ 三个寄存器存储（临时）变量用于计算，后续或将使用局部寄存器分配算法加以优化.

*Email:221240073@smail.nju.edu.cn

3.2 栈管理

为了方便栈管理，我使用了 $\$fp$ 寄存器来进行计算. 对于中间代码中的每个函数，使用**头插法**维护一个栈条目链表，链表中的每一个节点都存储了对应的操作数信息和相对于栈底的偏移量，栈条目的定义如下：

```
struct Stack_Item_{
    Operand op;     //操作数
    int offset;     //相对于栈底偏移
    struct Stack_Item_ *next; //下一个栈条目
};
```

栈管理的重中之重是处理函数调用，为了方便实现，本次实验未按照 MIPS32 的约定使用寄存器传参数，而是完全将参数压栈，并在调用之前，将 $\$sp, \$fp, \$ra$ 的值也压栈，并在调用结束后将对应寄存器恢复.

```
...
fprintf(fp," addi $sp, $sp, -12\n"); //栈顶指针下移
fprintf(fp," sw $ra, 0($sp)\n"); //将$ra寄存器的值存储到栈中
fprintf(fp," sw $fp, 4($sp)\n"); //将$fp寄存器的值存储到栈中
fprintf(fp," sw $sp, 8($sp)\n");//将$sp寄存器的值保存到栈中

fprintf(fp," jal %s\n",p->code.u.two.right->u.name);
fprintf(fp," lw $sp, 8($fp)\n"); //从栈中加载$sp寄存器的值
fprintf(fp," lw $fp, 4($sp)\n"); //从栈中加载$fp寄存器的值
fprintf(fp," lw $ra, 0($sp)\n"); //从栈中加载$ra寄存器的值
fprintf(fp," addi $sp, $sp, 12\n"); //栈顶指针上移
...
```

4 不足 & 可改进的地方

- 实验三生成的中间代码存在大量重复、无用的中间代码，本次实验又是根据实验三生成的中间代码链表逐一生成目标代码并未进行优化，加之朴素寄存器分配，生成的目标代码性能有**很大**的优化空间.