编译原理实验四 实验报告

殷乔逸 151220152

程序功能

- [x] 代码中**已实现**了针对 C 语言代码的词法分析，语法分析和语义分析，能够正确地判别各项词法、语法或语义错误(**包括所有必做和选做要求**)。
- [x] 代码中**已实现**对正确通过词法、语法及语义分析的 C 语言代码进行中间代码的生成。(**完成所有必做和选做要求**)
- [x] 代码中改进或实现了实验三的以下假设或者功能：**(亮点)**
    1. [x] 改进假设 1，支持八进制和十六进制数(转为十进制数进行计算)。
    2. [x] 改进假设 4，支持变量作用域。
    3. [x] 改进假设 6，支持函数声明。
    4. [x] 改进要求 3.2，高维数组也允许作为参数。此外，结构体和数组可以互相嵌套。
    5. [x] 优化控制流语句，尽可能少生成 GOTO 和 LABEL 而使用 fall 特性。
    6. [x] 优化常数表达式，可计算的常数表达式会被直接计算好。
    7. [x] 优化临时变量，避免生成不必要的中间变量，减少不必要的赋值表达式。
    8. [ ] 语句块优化，减少冗余中间代码的生成。
- [x] 代码中**已实现**生成 MIPS 汇编代码所需的数据结构，并且根据中间代码（三地址代码）生成 MIPS 汇编代码。
    1. [x] 朴素寄存器分配法
    2. [ ] 局部寄存器分配法
    3. [ ] 图染色算法
    4. [ ] 活跃变量分析

数据结构

为了能够生成 MIPS 汇编代码，首先要针对**寄存器描述符**和**地址描述符**建立合理的数据结构。

- [x] 采用朴素寄存器分配法，则基本无需考虑寄存器分配，所有变量（临时变量、局部变量等）都会溢出到内存中。所以设计一个数据结构，针对每一个操作数，存储其类型、作为基地址的寄存器（一般是`$fp`，溢出到栈中）和偏移量。

```
typedef struct MIPS_Operand_{
    /* The type of the operand */
    MIPS_OP_TYPE kind;
    union {
        int value;
        char* label;
        struct {
            // MIPS_OP_ADDR use both
            MIPS_REG reg;   // MIPS_OP_REG use this only
            int offset;
        };
    };
} MIPS_OP;
```

除此之外，针对每一条 MIPS 汇编代码，设计数据结构专门用于存储汇编代码中的指令，操作数等。

```
    typedef struct MIPS_INSTR_ {
        // Type of the instructions
        MIPS_INSTR_TYPE kind;
        union {
            struct {
                MIPS_OP *op1, *op2, *op3;
            };
        MIPS_OP* operand;
        char* label;
        };
    struct MIPS_INSTR_ *prev, *next;
} MIPS_INSTR;
```

为了将所有变量溢出到栈中存储，必须在生成汇编代码的同时合理分配栈中内存位置，也就是说，任何一个变量存储到栈中都必须记录其在栈中的位置，由于$sp 栈指针在构造活动记录的时候可能由于声明数组或结构体等局部变量而变化，因此采用以$fp 帧指针作为基地址，所有变量都计算到$fp 的偏移量，最终的$sp 的位置也可以根据所有临时变量或局部变量被分配到的总空间来得到。

```
/* Data structure for local variables */
typedef struct Local_Var_{
    OP_TYPE kind;
    int num;
    /* Offset in stack frame related to $fp */
    int offset;
    struct Local_Var_* next;
} Local_Var;

/* Data structure for stack frame */
typedef struct Frame_Info_{
    Local_Var* variable;
```

```
        /* Variable offset is initialized by -8 */
        int var_offset;
        /* Arguments offset is initialized by 4 */
        int arg_offset;
    } Frame_Info;
```

- [ ] 局部寄存器分配法
- [ ] 局部寄存器分配法
- [ ] 图染色算法
- [ ] 活跃变量分析

实现思路

- 采用朴素寄存器分配法，则只需要考虑将变量溢出到内存的什么位置是合适的。显然在设计 `Local_Var` 结构体的时候就考虑到，任何一个栈内的位置（某一帧）存储的变量必须具备标识符属性的成员变量，譬如 `kind` 代表了变量类型，而 `num` 则表示变量序号，则两者共同唯一地标识一个变量。当需要使用该变量的时候，则在 `Frame_Info` 结构体中的 `varaible` 链表中查找，更新后存入原内存位置，即完成了变量的存取操作。

  除此之外，比较重要的是针对所有的 MIPS 汇编代码中的运算符，需要将操作数（符）先装载到寄存器中，不同的操作数类型有不同的装载方式，譬如针对 `*a=b` 这样的需要取存放 `a` 的内存位置为 `addr`，装载 `b` 到寄存器 `reg`，在写 `reg` 的内容进入 `addr` 代表的位置。

实验难点

- 对于朴素寄存器分配法，主要难点在于**分配变量溢出到栈后的内存位置**。

解决方案

- 解决朴素寄存器分配法中的难点需要找到合适的基地址寄存器，譬如`$fp`，由于在产生新的活动记录（栈帧）时`$fp` 的值保持不变，所以以它为基地址比较方便，并且容易寻址其他变量。基本的解决思路如下，具体见代码实现。

```
_____________________  _> High Address
|_____...___________|
|_____arg2__________|  _> $fp+4 eg. store the arg1 :$fp->0x44
|_____arg1__________|  _> $fp eg. store the old $fp :$fp->0x40
|_____old $fp_______|  _> $fp-4 eg. store the old $ra :$fp->0x36
|_____$ra___________|  _> $fp-8 eg. store the tempvar1 :$fp->0x32
|_____tempvar1______|  _> $fp-12 eg. store the tmp[2] :$fp->0x28
|_____tmp[2]________|  _> $fp-16 eg. store the tmp[1] :$fp->0x24
|_____tmp[1]________|  _> $fp-20 eg. store the tmp[0] :$fp->0x20
|_____tmp[0]________|  _> $fp-24 eg. store others :$fp->0x16
|_____...___________|  _> Low Address
```

编译及运行方法

首先压缩包解压后得到的文件夹 Lab4,Test 文件夹中存储了 5 个测试用例，Code 文件夹存储了所有代码文件。

- 如果想要直接得到 MIPS 汇编代码，在 shell 命令行执行./run.sh -a，则会直接编译并且对 Test 文件夹下的用例生成 MIPS 汇编代码，输出的结果以.asm 的文件格式按序存储在 Test_ASM 文件夹中。
- 如果直接编译运行，在 shell 命令行执行./run.sh，则会按默认模式(仅生成中间代码)直接编译并且对 Test 文件夹下的用例进行测试，输出的结果以.ir 的文件格式按序存储在 Test_IR 文件夹中。
- 如果想要删除中间文件、可执行文件和默认的结果存储文件夹，在 shell 命令行执行./run.sh --clean 即可。
- 如果想要采用不同的模式或者不同结果路径进行测试，在 shell 命令行执行./run.sh [-p] [-s] [-dag] [-ir] -[a] [result path]即可，默认的资源文件夹是 Test。
- 如果仅仅想要编译，则进入 Code 文件夹执行 make parser，则会重新编译并在上层目录 Lab4 得到可执行文件 parser。可以执行./parser 来查看可用的参数选项。
- 如果想要**对某一个测试用例生成中间代码**，则在 Lab4 目录下执行./parse -a source [result]，其中 source 代表测试用例的路径，result 代表结果文件路径，如果该参数为空则打印至屏幕。