编译原理实验

第四次实验报告

孙楷文 学号：111100027 （5班） email: kaiwensun@smail.nju.edu.cn

1. 代码文件

实验源代码文件包含：

g--	lexical.l	mips.c	semantic.h	syntaxtree.h
iropt.c	main.c	mips.h	syntax.y	translate.c
iropt.h	Makefile	semantic.c	syntaxtree.c	translate.h

其中用红色标出的文件为 Lab4的主要内容。

编译和使用方法：

用 Makefile 生成名为“g--”的可执行程序，运行./g-- <c--源代码文件名> <MIPS 汇编文件名>，如：

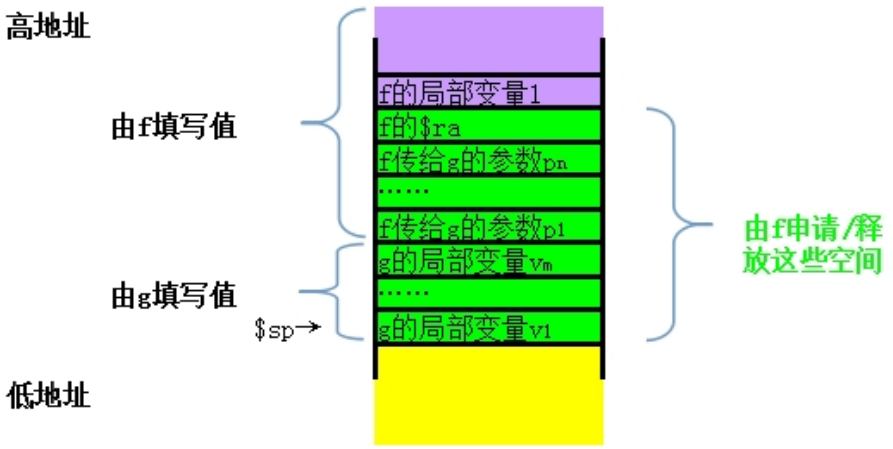
```
./g-- test.cmm test.s
```

另外，用 Makefile 生成的其他部分文件也在提交的代码中。

2. 活动记录

我翻译的 MIPS 代码在传递参数时，全部使用栈进行传参，没有使用寄存器$a0~$a3（write 函数除外）。返回值使用$v0返回。简便起见，所有的变量和参数在内存栈中都有其对应的位置。

以函数 f 调用子函数 g 时的活动记录为例，我生成的 MIPS 代码会像下图这样构造活动记录：



为了构建活动记录中的访问链，让函数知道某个局部变量/参数存在哪里，需要知道局部变量/参数相对于$sp 指向的地址的偏移量。这个信息的存储由下面《3. 数据结构》介绍，这个信息的计算方式由《4. 填写 struct FuncLocalvarList 链表——生成 MIPS 代码前的准备工作》介绍。

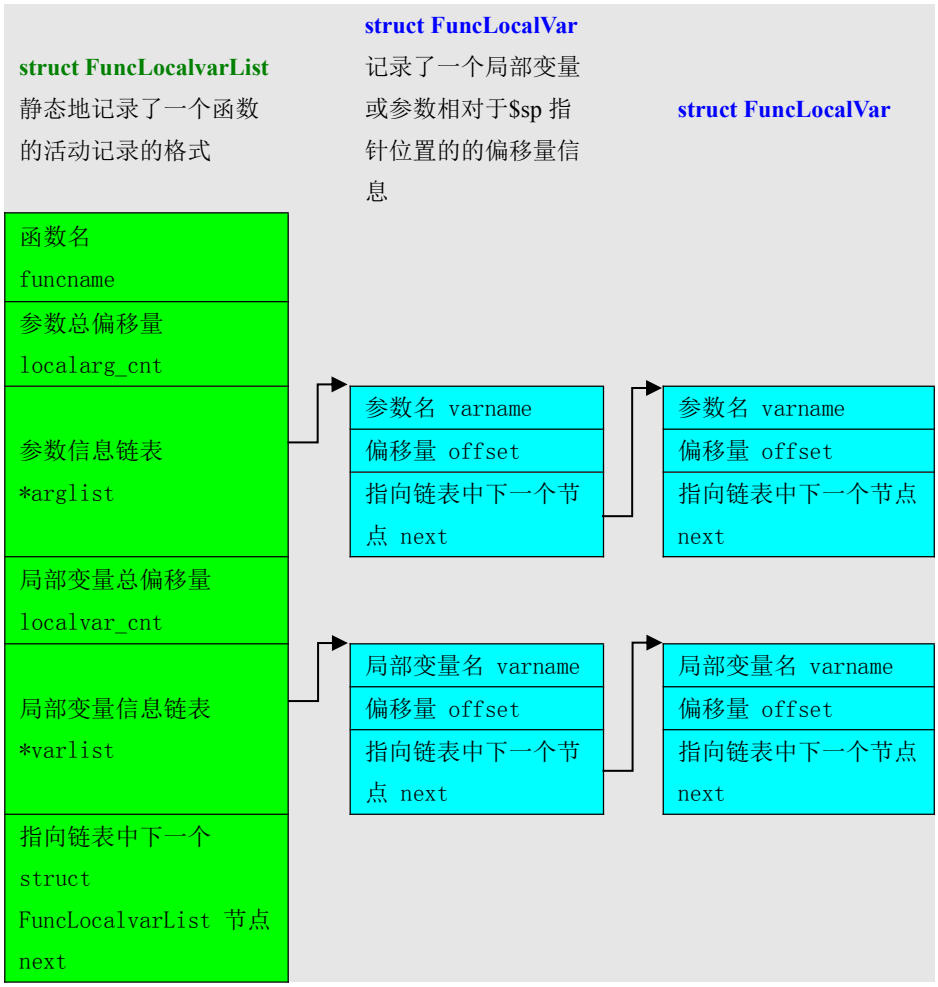
3. 数据结构

对于内存栈中的一个函数的活动记录，需要知道函数的访问链（包括局部变量和参数）。一个函数的这些信息由结构 **struct FuncLocalvarList** 存储，以$sp 指针为参考位置，记录了所有局部变量、参数的偏移量。具体的某个局部变量/参数的偏移量，被串在节点类型为 **struct FuncLocalVar** 的链表里。

另有 MIPS 代码结构体 struct MIPSCode，用字符串和链表存储 MIPS 目标代码。

```
struct FuncLocalVar
{
        //一个局部变量/形参
        char varname[256];              //局部变量名
        int offset;                     //局部变量所占内存对sp的偏移量（byte）
        struct FuncLocalVar* next;      //函数中下一个局部变量
};
struct FuncLocalvarList
{
        //一个函数静态"活动记录"模板的链表节点
        char funcname[256];             //函数名
        int localarg_cnt;               //函数形参相对于$sp的最大偏移量（单位byte）
        struct FuncLocalVar* arglist;   //函数形参链表
        int localvar_cnt;               //函数内局部变量相对于$sp的总偏移量（单位byte）
        struct FuncLocalVar* varlist;   //函数内局部变量名链表
        struct FuncLocalvarList* next;  //下一个函数活动记录
};
```

下图是 **struct FuncLocalvarList** 和 **struct FuncLocalVar** 结构体的图形化表示：



4. 填写 struct FuncLocalvarList 链表——生成 MIPS 代码前的准备工作

在翻译中间代码为 MIPS 代码之前，先由函数 void prepareFuncLocalVarList()，根据全部的中间代码，填写 C--函数的活动记录（访问链）的格式的模板 **struct FuncLocalvarList**。该函数从头到尾顺序扫描中间代码，每当遇到一个"FUNCTION xxx :"格式的中间代码时，就新建立一个 **struct FuncLocalvarList** 变量加入链表，并在遇到下一个"FUNCTION xxx :"中间代码之前一直填写这个变量。当被扫描到的中间代码中有未曾记录过的局部变量/参数时，将把该局部变量/参数的信息加到 arglist 或 varlist 链表中，并更新 localarg_cnt 或 localvar_cnt（遇到"DEC xxx #"时要格外

留意变量所占空间大小）。具体的对单条中间代码的分析，由 prepareFuncLocalVarList()调用 countLVStack()完成。countLVStack()计算新的偏移量，判断该变量名是局部变量还是参数，并调用 addOpdToList()具体完成把一个变量名和对应的偏移量加入到 **struct FuncLocalVar** 类型的链表中。

5. 生成 MIPS 代码——void genMips()

首先，genMips()调用上述 prepareFuncLocalVarList()函数，计算活动记录中各变量的偏移量。

其次，genMips()调用 genMipsHead()函数，生成.text、.global main、write 函数、read 函数等所有 C--程序公用的部分。

最后，genMips()遍历所有中间代码，对每一条中间代码调用 mipsCode()以生成对应的 MIPS 代码。下面介绍 genMips()的工作。

6. 生成一条中间代码所对应的 MIPS 代码——mipsCode()

我寄存器的选择方法采用了最简单、最低效的方法——即时读写内存，有大量寄存器被闲置。

对于一般的三地址代码，会将参与计算的两个值 lw 到$t1 和$t2 中，计算结果放在$t0 中。如果涉及到指针运算，会用$t3 寄存器辅助，把指针所指向的地址 lw 入$t3，然后再把 0($t3)从内存 lw 到相应的$t1 或$t2，或者把$t0 用 sw 存到内存 0($t3)里。

既然涉及到 lw 和 sw 操作，那么，如何在内存栈里找到变量对应的地址？答：对于每个 C--函数都已经由 prepareFuncLocalVarList()构造好了 **struct FuncLocalVar** 访问链。问题又来了，对于当前处理的这一条中间代码，我怎么知道这一条中间代码属于哪个 C--函数？答：在 mipsCode()中有一个静态变量`static struct FuncLocalvarList* funclist = NULL;`，每当 mipsCode()要分析的中间代码是“FUNCTION xxx :”时，mipsCode()就会在 **FuncLocalvarList** 链表中搜索名为 xxx 的 **FuncLocalvarList** 节点，并让 funclist 指针指向它。在遇到下一个“FUNCTION xxx :”格式的中间代码之前，所有的中间代码的访问链均由当前的 funclist 寻找而得。

对于“ARG a”格式的中间代码，同样可以通过访问链来知道 a 的源地址。但这是第几个参数呢？要把参数压栈到哪里？这是如何计算 offset 的问题。这个 offset 同样由一个静态变量 static int parameter_offset 记录，当遇到一个 FUNCTION 或 CALL 中间代码时，parameter_offset 将被赋为-4.（为$ra 预留空间，因而不是赋为 0）。当遇到一个 ARG 中间代码时，把实参存放在 parameter_offset($sp)的内存位置，然后 parameter_offset 将被赋为 parameter_offset-4。

对于“r := CALL f”格式的中间代码，由于此前已经由对 ARG 中间代码分析完成了传参的动作，所以现在只需要：①把$ra 存至-4($sp)；②$sp = $sp-4；③jal f；④恢复$sp；⑤恢复$ra；⑥用 sw 把返回值$v0 存至 r 在内存中的位置。

其他指令都比较简单，不赘述。

7. 几个特殊的地方

7.1. 对于 C--的 main()函数，要在遇到中间代码“FUNCTION main :”时由 main()为自己准备局部变量的栈空间，$sp = $sp - <main 函数的局部变量所占据的空间大小>。当 main 函数结束并返回时，经在命令行 spim 中试验，无需恢复$sp。

7.2. 支持非标准的中间代码

在 Lab3 中，我向助教老师说过那个 python 写的虚拟机无法支持非标准的 “*a := *b [+-*/] *c”格式的中间代码。然而，由于我自己的程序在生成 MIPS 代码时使用的寄存器分配算法比较懒，只使用$t0、$t1、$t2 参与三地址代码的直接计算，**由专门的$t3 辅助指针操作**（见上一小节的第二自然段），所以我的目标代码生成器是支持这种非标准的中间代码的。

7.3. 如何支持全局变量？（我的程序没有实现）

全局变量的存储空间可以在进入 main 时申请到栈里。全局变量的访问链不需要在活动记录中体现，这是因为我的寄存器分配算法中有大量永远闲置的寄存器，所以可以随便找一个永远闲置的寄存器过来用于记录第一个全局变量的存储地址。其他全局变量的地址在此地址上加上相对偏移量即可。