申优文档

15061103 李海南

详细的设计在设计文档中描述很多，本文主要进行要点难点分析和一些需要特别注意的部分，词法分析较为简单，此处不再进行说明。

1. **语法分析**

在外层程序中因为变量（数组）声明与函数声明的first集合相交，int和char可能是变量（数组）声明，也可能是函数声明，所以需要预读一个单词来进行判断，读到左括号则是函数，读到做方括号即为数组，读到逗号或分号即为变量，变量或数组之后若为逗号，之后也只能是变量或数组，不可以是函数，另外值得注意的是，变量声明与函数声明有明确的先后顺序，所以判断若是函数，则应该跳出循环，不能继续进行变量声明，因为主函数必须在程序的最后，所以在读到main函数时跳出，同时要对main的前一个单词进行检查，若不是void应该进行报错。而在函数内部int和char只能是变量或数组声明。

对于因子的错误，因为因子出现在多种情况下，对于因子错误后的跳过代码的处理便是一大难题，这里我使用了facerrflag这个全局变量，每次调用表达式前，将facerrflag置为0，对于因子的错误只进行报错并将facerrflag设为1，不进行跳过代码，跳过代码的过程在具体语句中进行。

1. **语义分析**

符号表结构如下：

struct Tab

{

char name[100];

int paranum;

enum objecttyp obj;

enum types type;

int ref;

int normal;

int adr;

int link;

};

struct Tab tab[tmax];

函数表结构如下：

struct funtab

{

char name[100];

int paranum;

int offset;

int rt;

struct Ref\_Tab ref\_tab[100];

int ref;

int size;

enum types type;

}ftab[bmax];

入表时需要对变量进行检查，值得注意的是不同层的变量可以同名这就需要对符号表进行特殊的设计，我是通过link实现的这一功能，符号表的下表为0的一项不会固定的存放某个变量，变量入表时传入isfun参数，对于一般的变量，link为新变量下标减1，对于函数变量，link为上一个函数的ref，即上一个函数在符号表中的位置，对于第一个参数变量，link为0.使用变量时，在符号表中查找变量的操作即为先从表头开始按link向前查找，若没有找到，则从当前函数名在符号表中的位置按link向前查找。

语义分析要检查数据的类型，不可以对常量赋值，要检查调用函数时的参数个数，这里我通过一个次数等于循环次数的循环来获取参数，若参数不足则提前跳出循环，报错，若参数过多，则后续要检查的标识符不正确，报错并跳到可以识别的符号（右括号）。

读语句的标识符必须是变量类型。

若是语句的第一个标识符不能识别，则跳到下一个语句，但不会结束语句列，只有读到右大括号时语句列才会结束。

数组下标越界检查。

对于return语句的处理，通过检查函数是否有返回值进行处理，对于无返回值的函数，若return语句有括号，则报错。若有返回值函数无括号，则报错，且通过设置flag检查有返回值函数是否有return语句。

1. **中间代码生成**

这里需要注意的就是label的位置，有很多地方是先生成新label，然后写入跳转指令，最后才写入此label的位置。

另外一大难点就是写入PUSH的顺序，对于函数的实参是多个有返回值函数这种情况，如果没读一个表达式马上就生成PUSH指令，就会导致PUSH和call之间混入其他函数的参数或是计算表达式，只是后续生成目标代码的过程中出现已经传入的参数被其他函数的参数覆盖的情况，导致程序出错。

1. **运行栈设计**

使用t8寄存器保存全局变量的栈顶，每次调用函数，先进行保存现场，用t9保存当前层栈底指针，用sp寄存器保存当前层栈顶指针，函数结束时恢复现场。

1. **目标代码生成**

struct Addr

{

char name[100];

int addr;

int type;

int isglo;

}addrlist[2000];

int ap=0;

通过此表保存变量及其相对于基地址的偏移量，类型，是否全局变量，若是全局变量，则基地址为t8，若不是全局变量，则基地址为sp。

对于PUSH语句，即值参的传递，通过将每个函数的参数顺序固定并都从函数顶端开始存放，将值参直接放到使用它的函数中的确定位置。

1. **代码优化**

优化一：DAG图：

struct Nodetab

{

char name[100];

int index;

}nodetab[100];

struct Nodelist

{

char op[100];

int is\_leaf;

int in\_order;

int pi;

int left\_index;

int right\_index;

int parent\_index[100];

}\*nodelist[100] = { NULL };

notetab为结点表，nodelist为结点地址表，notetab中的index存放的就是该结点在nodelist数组中的下标，结点结构中的左右孩子和双亲都是nodelist数组中的下标。叶子结点中如果是变量且后续会改变这个变量的值，则将该变量变为$变量，同时生成一条$变量=变量的语句。算法即为书中的生成DAG图，以及导出的启发式算法，不过我改进了一下，若结点中有程序中的变量，导出时将程序中的变量全部导出，临时变量删掉，若只有临时变量，则选择第一个保留，其余删掉。另外导出时还要遍历叶子结点，若叶子结点中出现多个变量，则输出这多个变量互相相等。

此处需要注意一个细节，我的临时变量的内存分配是通过遍历中间代码，寻找带有临时变量特征（以$开头）的变量进行分配，代码优化虽然会使原本的临时变量减少，但也会生成一些新的临时变量（叶子结点是变量且后续会改变），为保证程序的正确运行，要使新的临时变量同样以$开头。

优化二：引用计数法分配寄存器

在ftab中加入此函数的所有变量，并统计变量的个数然后进行排序，这里需要注意的是并不统计参数的引用次数，因为PUSH会将实参放到内存中，所以参数必须从内存中获得，而不能从寄存器中获得，之后对变量按引用次数进行排序。

int refloc(int j,char name[]) {

int i;

for (i = 0; i < ftab[j].rt && i<8; i++) {

if (strcmp(ftab[j].ref\_tab[i].name, name)==0) {

return i;

}

}

return -1;

}

目标代码生成时会使用此函数进行查找变量，此函数限定只能返回0到7之间的数字，若在表中0到7的下标出没有找到，则该变量没有分配寄存器（我只是用了s0到s7这8个寄存器）。