《编译技术》

课程设计文档

学号：15061111

姓名：刘斯盾

2018年 1月 13日

## 需求说明

### 文法说明

* 原始文法

＜加法运算符＞ ::= +｜-

＜乘法运算符＞ ::= \*｜/

＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

＜数字＞ ::= ０｜＜非零数字＞

＜非零数字＞ ::= １｜．．．｜９

＜字符＞ ::= '＜加法运算符＞'｜'＜乘法运算符＞'｜'＜字母＞'｜'＜数字＞'

＜字符串＞ ::= "｛十进制编码为32,33,35-126的ASCII字符｝"

＜程序＞ ::= ［＜常量说明＞］［＜变量说明＞］{＜有返回值函数定义＞|＜无返回值函数定义＞}＜主函数＞

＜常量说明＞ ::= const＜常量定义＞;{ const＜常量定义＞;}

＜常量定义＞ ::= int＜标识符＞＝＜整数＞{,＜标识符＞＝＜整数＞}

| char＜标识符＞＝＜字符＞{,＜标识符＞＝＜字符＞}

＜无符号整数＞ ::= ＜非零数字＞｛＜数字＞｝

＜整数＞ ::= ［＋｜－］＜无符号整数＞｜０

＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝

＜声明头部＞ ::= int＜标识符＞ |char＜标识符＞

＜变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;}

＜变量定义＞ ::= ＜类型标识符＞(＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’){,(＜标识符＞|＜标识符＞‘[’＜无符号整数＞‘]’ )}

＜常量＞ ::= ＜整数＞|＜字符＞

＜类型标识符＞ ::= int | char

＜有返回值函数定义＞ ::= ＜声明头部＞‘(’＜参数＞‘)’ ‘{’＜复合语句＞‘}’|＜声明头部＞‘{’＜复合语句＞‘}’ //第一种选择为有参数的情况，第二种选择为无参数的情况

＜无返回值函数定义＞ ::= void＜标识符＞(’＜参数＞‘)’‘{’＜复合语句＞‘}’| void＜标识符＞{’＜复合语句＞‘}’//第一种选择为有参数的情况，第二种选择为无参数的情况

＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

＜参数＞ ::= ＜参数表＞

＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}

＜主函数＞ ::= void main‘(’‘)’‘{’＜复合语句＞‘}’

＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞}

＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

＜因子＞ ::= ＜标识符＞｜＜标识符＞‘[’＜表达式＞‘]’|‘(’＜表达式＞‘)’｜＜整数＞|＜字符＞｜＜有返回值函数调用语句＞

＜语句＞ ::= ＜条件语句＞｜＜循环语句＞| ‘{’＜语句列＞‘}’｜＜有返回值函数调用语句＞;

|＜无返回值函数调用语句＞;｜＜赋值语句＞;｜＜读语句＞;｜＜写语句＞;｜＜空＞;|＜情况语句＞｜＜返回语句＞;

＜赋值语句＞ ::= ＜标识符＞＝＜表达式＞|＜标识符＞‘[’＜表达式＞‘]’=＜表达式＞

＜条件语句＞::= if ‘(’＜条件＞‘)’＜语句＞else＜语句＞

＜条件＞ ::= ＜表达式＞＜关系运算符＞＜表达式＞｜＜表达式＞ //表达式为0条件为假，否则为真

＜循环语句＞ ::= while ‘(’＜条件＞‘)’＜语句＞

＜情况语句＞ ::= switch ‘(’＜表达式＞‘)’ ‘{’＜情况表＞[＜缺省＞] ‘}’

＜情况表＞ ::= ＜情况子语句＞{＜情况子语句＞}

＜情况子语句＞ ::= case＜常量＞：＜语句＞

＜缺省＞ ::= default : ＜语句＞

＜有返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’|<标识符> //第一种选择为有参数的情况，第二种选择为无参数的情况

＜无返回值函数调用语句＞ ::= ＜标识符＞‘(’＜值参数表＞‘)’|<标识符> //第一种选择为有参数的情况，第二种选择为无参数的情况

＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}

＜语句列＞ ::= ｛＜语句＞｝

＜读语句＞ ::= scanf ‘(’＜标识符＞{,＜标识符＞}‘)’

＜写语句＞ ::= printf ‘(’ ＜字符串＞,＜表达式＞ ‘)’| printf ‘(’＜字符串＞ ‘)’| printf ‘(’＜表达式＞‘)’

＜返回语句＞ ::= return[‘(’＜表达式＞‘)’]

* 对文法的改写

设计时对文法的改写主要在全局变量声明和函数定义这一部分。由于全局变量声明和函数定义的头部都是int/char加上标识符，不能通过向前看一个符号的方式进行区分，不满足LL(1)分析法。所以我增加新的语法成分<varOrFunc>用来识别int/char以及之后的第一个标识符，区分其为变量或函数（数组变量将其长度一并处理）。之后的全局函数定义或全局变量声明用于处理剩余部分。

### 目标代码说明

生成的目标代码为MIPS指令和扩展指令，包括add, sub, addi, subi, mul, div, lw, sw等。

### 优化方案

实现DAG优化，减少不必要的运算和重复运算，删去无用临时变量。

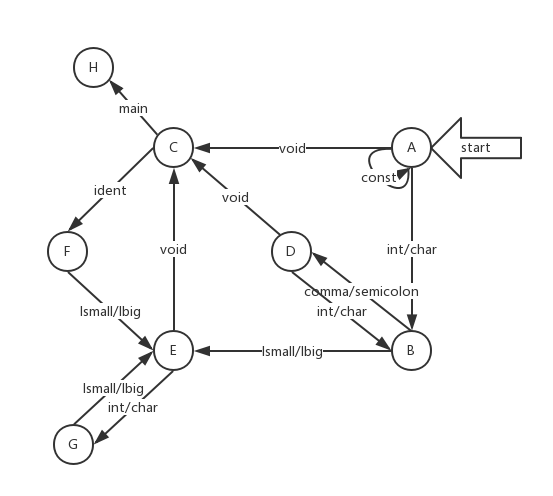
实现着色算法优化，将活跃变量放入寄存器，充分利用寄存器，减少对内存的访问。

常数优化，右侧操作数均为常数的运算在编译时计算。

## 详细设计

### 程序结构

如下是语义分析与中间代码生成的程序结构。通过自动机实现分析程序的上层结构



|  |  |  |  |
| --- | --- | --- | --- |
|  | 前置sym集合 | 描述 | 后置sym集合 |
| A | const | 处理一个常量声明语句，一直到分号。 | const, void, int, char |
| B | int, char | 判断这个声明是函数声明还是全局变量声明。 | comma, semicolon, lsmall, lbig |
| C | void | 带void标识符的声明一定是函数声明，C用来区分是主函数还是其他函数 | main, ident |
| D | comma, semicolon | 处理变量声明剩余部分，如果是分号就吞下分号直接往后走 | void, int, char |
| E | lsmall, lbig | 处理函数主体部分，如果是小括号还需要处理参数 | void, int, char |
| F | ident | void只能是声明函数，读入不是main，后面只能是小括号或大括号 | lsmall, lbig |
| G | int, char | 出现在函数之后的int或char只能是其他的子函数，G吞下ident，让sym指向小括号或大括号 | lsmall, lbig |
| H | main | 主函数的主体部分，注意确保包括空括号 | NULL |
|  |  |  |  |

### 函数功能

grammaticalAnalysis.cpp:

1. NFA\_program()：顶层自动机的实现，驱动整个语法分析程序。

delaration.cpp：

1. constDeclare(lev, tab)：识别到const关键字后进行常量定义，分析在遇到行末分号时停止。lev表示其为全局定义还是函数内定义。如果在函数内定义则tab指针非空，指向当前函数在符号表中位置，用作重名检查。
2. global\_varOrFunc()：在识别到int或char关键字时进入，处理int/char关键字和其后的第一个标识符。返回函数或变量在符号表中位置指针。
3. global\_varDeclare(type)：当varOrFunc识别为变量时处理剩余定义。type表示类型。
4. local\_varDeclare(tab)：处理函数内变量定义。tab指向当前函数在符号表位置。
5. global\_funcDeclare(tab)：当varOrFunc识别为函数定义时处理剩余函数定义。tab指向该函数在符号表中位置。该函数处理至函数声明末尾的右大括号。

statement.cpp：

1. compoundStatement(tab)：由global\_funcDeclare函数调用。处理函数内的复合语句。tab为当前函数在符号表中位置。
2. statementArray(tab)：处理语句列。tab为当前函数在符号表中位置。
3. statement(tab)：处理语句。根据读到的第一个符号确定语句类型，并调用相应处理函数。tab为当前函数在符号表中位置。
4. ifStatement(tab)：处理if语句。tab为当前函数在符号表中位置。
5. whileStatement(tab)：处理while语句。tab为当前函数在符号表中位置。
6. switchStatement(tab)：处理switch语句。tab为当前函数在符号表中位置。
7. scanfStatement(tab)：处理scanf语句。tab为当前函数在符号表中位置。
8. printfStatement(tab)：处理printf语句。tab为当前函数在符号表中位置。
9. assignStatement(tab)：处理赋值语句。tab为当前函数在符号表中位置。
10. callStatement(tab)：处理函数调用语句。tab为当前函数在符号表中位置。
11. returnStatement(tab)：处理return语句。tab为当前函数在符号表中位置。
12. expression(&type, &res)：处理表达式。type返回表达式类型，res返回表示该表达式值的变量名。
13. expression(&type, &res, &exp\_bool, &exp\_value)：处理表达式。前两个参数同上。exp\_bool表示该表达式能否在编译阶段获得值。exp\_value在exp\_bool为真的情况下值为该表达式的值。这两个返回值用于检查数组越界静态检查。
14. term(&type, &res, &exp\_bool, &exp\_value)：处理项。参数含义同上。
15. factor(&type, &res, &exp\_bool, &exp\_value)：处理因子。参数含义同上。

symbolTable.cpp:

1. enter(ident, kind, type, length, value, lev)：向符号表登记标识符。ident为标识符名，kind为标识符种类，分为func, var, cons, para，type为标识符类型，分为t\_int, t\_char, t\_void, t\_tmp。length表示数组长度或函数参数个数。value为值。lev表示全局或是局部。在enter中存在如下逻辑：

全局变量存入global\_tab，局部变量存入local\_tab，每在global\_tab中存入一个func类型标识符，就在local\_tab创建新的空间用于保存下一函数块的标识符。

函数返回插入的符号表元素指针，用于后续补充信息（如函数参数个数）。

1. lookup(ident, local\_flag, \*element)：在符号表中查找标识符，防止重名。ident为标识符名。local\_flag标志查找范围。element在找到时填入相关信息。函数返回0,1标志是否找到。
2. show\_tables()：打印符号表。

mcode.cpp:

1. emit：插入中间代码。
2. newLabel()：生成新的唯一的跳转标签并返回。
3. setLabelVal(label, val)：设置label到val行。
4. newTmpVar()：生成新的唯一的临时变量，并将其插入到符号表中。
5. embeddingLabel()：将label表嵌入中间代码。如果有多个label指向该行则将其替换删除。返回label，code结构体向量。
6. printMcode(path)：将中间代码输出到文件。

translate.cpp:

1. getGlobalOffset(ident, &offset, &constkind)：在全局符号表中查找标识符。返回offset。如果constkind为真，则offset为该常量值，否则为偏移量。
2. getLocalOffset(ident, &offset, &constkind)：在局部符号表中查找标识符。参数意义同上。
3. loadVal(reg, ident)：将变量的值放入reg表示的寄存器中。
4. storeVal(reg, ident)：将reg表示的寄存器中的值放入ident表示的内存中。
5. translate(embedcode)：遍历中间代码，根据不同代码生成不同的目标代码。

dag\_optimzed.cpp:

1. enter\_mcode()：将中间代码分块，提取变量名。
2. enter\_var()：根据变量在基本块中的分布构造保留变量池。
3. enter\_code()：填入优化后中间代码。
4. enter\_label()：填入label。
5. calculate(dag\_link)：获得子结点的变量名。它包含这样的逻辑：

如果该结点是叶子结点，返回叶子结点中的变量名（可确保唯一性）。

如果该结点只有一个子结点，则子结点必为叶子结点，返回叶子结点中变量名（可确保唯一性）。

如果该结点中存在保留变量，则返回最先找到的保留变量。

否则，递归调用calculate函数获得该结点的左右子结点变量名，根据结点中记录的操作符打印中间代码。随机选择该结点中的一个变量升级为保留变量并计算其值。返回该变量。

1. DAG(begin, end)：处理一个基本块。构造dag图—>获得结点导出队列—>队列反向导出中间代码（使用calculate函数获得子结点变量名；如果该结点没有保留变量就不做处理）。begin和end表示该基本块的开始和结束位置。
2. dag\_optimize()：dag图优化的驱动函数。切块，填变量，对于每一个长度大于1的基本块使用DAG函数做处理。
3. printMcode\_optimized(path)：将优化后的中间代码打印到指定路径。

coloring.cpp:

1. enter\_use(blk, var)：向blk块的use集合中加入var，在var为局部非数组变量或参数且不出现在def集合中时。同时向in集合加入该变量。
2. enter\_def(blk, var)：向blk块的def结合中加入var，在var为局部非数组变量或参数且不出现在use集合中时。
3. update\_out(blk) & update\_in(blk)：更新blk块的out集合和in集合。更新完in集合后检查in集合有无扩充。返回布尔值。
4. update\_active()：更新所有块的active集合。active集为use集和out集的并集。
5. build\_blk\_list(begin, end)：为函数划分基本块。
6. link\_blk\_list()：根据跳转语句连接基本块。
7. build\_conflict\_map()：建立冲突图，并根据冲突图导出函数内的活跃变量寄存器分配方案。
8. distribute\_reg\_to\_blks()：分配寄存器给每一个基本块。对于基本块active集里的变量，如果在活跃变量寄存器分配方案中为其分配了寄存器，则在该块内为其分配该寄存器。将剩余的寄存器分配给(def-out)集合中的变量。
9. coloring(begin, end)：处理一个函数内的活跃变量分析和寄存器分配。并在最后调用coloring\_trans.cpp中的to\_tcode(func\_name)翻译成目标代码。
10. coloring\_translate(mcode)：根据函数声明头部将其分为若干块，分别调用coloring处理。

coloring\_trans.cpp:

1. ett(string)：添加text段目标代码。
2. etd(string)：添加data段字符串。
3. load(reg, v)：返回变量v的寄存器。如果v不在寄存器内，则将v从内存中导入reg表示的寄存器，并返回该寄存器。
4. loadArr(reg, v)：将v表示的数组的起始偏移位置放入reg表示的寄存器中。
5. setblksz()：计算该函数所占内存的大小。
6. to\_tcode(func\_name)：根据寄存器分配方案生成目标代码。

### 函数依赖关系

NFA\_program

constDeclare

global\_varOrFunc

global\_varDeclare

global\_funcDeclare

componentStatement

constDeclare

local\_varDeclare

statementArray

statement

ifStatement

expression

term

factor

expression

statement

whileStatement

expression

statement

switchStatement

expression

statement

assignStatement

expression

callStatement

expression

returnStatement

expression

scanfStatement

expression

printfStatement

expression

statementArray

translate

getGlobalOffset

getLocalOffset

setBlock

loadVal

storeVal

dag\_optimize

enter\_mcode

enter\_var

DAG

calculate

calculate

coloring\_translate

coloring

build\_blk\_list

link\_blk\_list

build\_conflict\_map

distribute\_reg\_to\_blks

to\_tcode

setblksz

load

loadArr

### 符号表管理方案

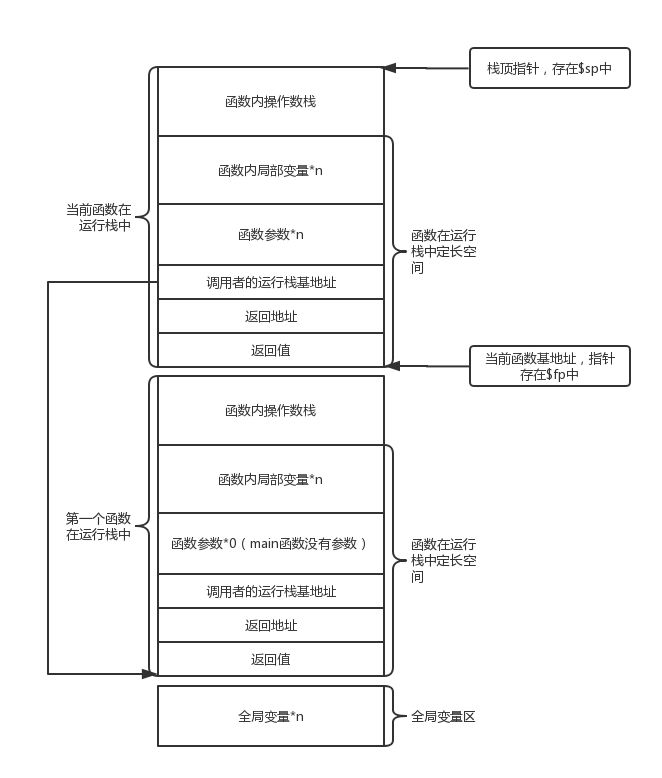
符号表分为global\_tab和local\_tab两个部分。Global\_tab是一维数组，存储全局变量，全局常量和函数。Local\_tab是二维数组，每个函数分配一个local\_tab中的一个数组。数组中存储局部常量和局部变量。

两个指针display和offset：display是指向当前函数块对应的local\_tab下标。Global\_tab每加入一个函数，display增加1。Offset指向变量和参数相对运行栈中当前函数运行块基地址的偏移，初值为0。每存入一个参数或变量，offset增加1，如果变量是数组，offset增加数组的长度。Global\_tab每加入一个函数，offset就置为FUNC\_OFFSET（该常数值为3，作为返回值，返回地址和上一个帧指针的预留空间）。

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | ident | kind | type | length | value |
| 函数 | 函数名 | func | 返回值类型{t\_void, t\_int, t\_char} | 参数个数 | 该函数对应的局部符号表(local\_tab)的下标 |
| 参数 | 参数名 | para | 参数类型{t\_int, t\_char} | 0(传参不能传数组) | 参数在运行栈中相对当前帧指针的偏移offset |
| 变量 | 变量名 | var | 变量类型{t\_int, t\_char, t\_tmp} | 0(如果不是数组) 或 n (如果是数组，n为数组长度) | 变量在运行栈中相对当前帧指针的偏移offset |
| 常量 | 常量名 | cons | 常量类型{t\_int, t\_char} | 0(常量不能是数组) | 常量的值，char类型则是其ascii码 |

### 存储管理方案

运行栈如下图所示。返回值，返回地址，调用者的运行栈基地址为每个函数运行栈中定长的部分。参数区和局部变量区对确定函数的运行栈来说是定长的。操作数栈随着函数内部指令的执行而变化。在指令执行过程中遇到新的函数调用则创建新的运行栈。



在coloring\_translate函数中建立运行栈时新开辟一片空间，用于在函数调用时保存现场。

### 四元式设计

|  |  |
| --- | --- |
| 函数入口 | int/char/void function\_name() |
| 简单赋值 | a = b |
| 基本运算 | c = a + b  c = a – b  c = a \* b  c = a / b |
| 数组操作 | a = b[x]  a[x] = b |
| 函数传参 | PUSH a 0 // 把a放入第0个参数位  PUSH a 1 // 把a放入第一个参数位 |
| 函数调用 | CAL function\_name |
| 函数返回值 | RETV a |
| 函数返回 | RETURN |
| 获得返回值 | a = RET |
| 分支 | GEZ a #label  GZ a #label  NEZ a #label  EZ a #label  LZ a #label  LEZ a #label |
| 跳转 | JMP #label |
| 输出 | PRINTFN a // 输出数字  PRINTFC a // 输出字符  PRINTFS str // 输出字符串 |
| 输入 | SCANFN a // 输入数字  SCANFC a // 输入字符 |
| 跳转标签 | #Label\_x\_ // x为唯一标识序号 |
| 临时变量 | #tx // x为唯一标识序号  #t\_x |

### 目标代码生成方案

1. 数据结构

中间代码保存格式：vector<pair<string, string>>，其中first为label域，当没有label时该域为空。second为中间代码，格式如上。

1. 关键算法

以功能实现为目的的目标代码生成思路很简单。将有关操作数放入内存，运算后再存入左值对应的内存中，对寄存器的使用很少。将取值和存值两个过程抽象成loadVal和storeVal两个函数。loadVal函数传入reg和ident两个字符串。

1. 检查ident是否为a[x]格式。如果是该格式则将a的偏移取出放入寄存器，再次使用loadVal将x的值放入内存。打印目标代码将x值乘以4加上a的偏移量。以该值为地址取出内存中的值放入reg寄存器中。返回
2. 检查是否为数字。如果是数字则使用li指令直接放到reg寄存器中。返回
3. 检查是否为RET。如果是RET则从返回值域获取返回值到reg寄存器中。返回
4. 顺序检索局部符号表和全局符号表。如果是常量，使用li指令把值放到reg寄存器中。返回
5. 否则使用lw指令从内存中取值放到reg寄存器中。返回

storeVal流程相对简单一些。参数同样为reg和ident两个字符串。

1. 检查ident是否为a[x]格式。实现方案类似以上。
2. 顺序检索局部符号表和全局符号表。可以保证为变量。使用sw将reg中的值放到偏移所指内存中。返回

### 优化方案

1. DAG图消除公共子表达式和无用中间变量。优化目标是连续的加、减、乘、除、赋值和取数组操作。其余部分不做优化。

* 相关结构体和枚举变量

1. 枚举变量 optype { \_null, \_add, \_sub, \_mul, \_div, \_arr\_get }。
2. 结构体 quadruple { string label; string code; string value; string left; string right; optype op; }
3. DAG图结点dag\_node {

bool inQ; // 是否在队列中，用于导出中间代码

bool leaf; // 是否为叶子结点

string leaf\_var\_name; // 叶子结点的变量名，非叶子结点该项为空

optype op; // 操作符，简单赋值和叶子结点op为\_null

dag\_node \*left;

dag\_node\* right;

vector<dag\_node\*> parent; // 左右子结点和父结点数组

} \*dag\_link; // 指向dag\_node类型结构体的指针

* 算法简要说明

1. 分割基本块。如果是非加、减、乘、除、赋值、取数组中间代码，则当前行为入口语句且下一行为入口语句，即该行单独成块。如果该行存在label，则该行为入口语句。
2. 构造保留变量池。根据划分的基本块提取变量。记录每个变量出现的基本块。如果某个变量是用户定义的变量，或是出现在不同的基本块，将其放入保留变量池。
3. 对每一个行数超过1 的基本块，进行DAG图优化。维护结点数组node\_list和变量结点对应表node\_map，建立dag图。对于每一行中间代码：
   1. 寻找第一个操作数对应的结点。如果没有找到则创建结点。存入node\_list中，并在node\_map中记录对应关系。
   2. 如果是直接赋值语句，检查第一个操作数对应的结点。如果是非叶子结点，将左值对应到同一个结点。否则，遍历其父结点寻找表示直接赋值的结点。如果找到则左值与这一找到的结点共用同一结点。否则为其创建一个新表示直接赋值的结点。
   3. 如果是两操作数语句，检查第二个操作数对应的结点。方法同a)。寻找两操作数共同的且操作符相同的父结点，如果没有找到则创建结点。将左值存入该结点中。

构造完成后可以保证，叶子结点至多只有一个变量；数组，常数和常量只出现在叶子结点中；表示直接赋值的结点图中必包含叶子结点。

1. 将node\_map对应关系翻转，由结点对应一系列变量名。如果该结点中没有对应的变量，则增加新的临时变量，填入该结点中。如果没有临时变量的结点为叶子结点，则添加中间代码为新添加的临时变量赋值为该叶子结点原始的变量名leaf\_var\_name，并将该临时变量加入到保留变量池。
2. 启发式算法得到结点导出队列。
3. 反向循环结点队列。循环该结点中的变量，如果不存在保留变量池中的变量，则跳过该结点。否则，使用calculate得到该结点的左结点变量名，如果存在右结点，则同样使用calculate得到右结点。打印中间代码：结点中任取的变量赋值为左结点变量名（op右结点变量名），并将该结点剩余保留变量池中变量更新为该变量的值。一下简述calculate算法实现：
   1. 如果该结点为叶子结点，返回该结点对应的变量名（可保证只有一个）。
   2. 如果该结点只有一个子结点，返回该子结点的变量名（同样可保证只有一个）
   3. 如果该结点中存在保留变量池中的变量，则返回该变量。
   4. 否则递归调用calculate得到左结点名和右结点名。任取该结点中的一个变量赋值为这两个变量名的操作结果，并将其存入保留变量池，返回该变量。
4. 基于活跃变量分析的着色算法优化，包含常数优化。简述寄存器分配方案的实现。

* 数据结构
  1. 结构体 blk\_node {

set<string> use; // 使用优先于定义

set<string> def; // 定义优先于使用

set<string> in; // in = use U (out – def)

set<string> out; // out = U in

set<string> active; // active = in U out = use U out

vector<string> code; // 该基本块的代码

string laebl; // 该基本块的跳转标签，没有则为空

string jump\_label; // 该块跳转到的目标标签，没有则空

string branch\_label; // 类似以上

bool ret\_blk; // 该块是否return块

set<blk\_node\*> precursor; // 该块的后继基本块

set<blk\_node\*> successor; // 该块的前驱基本块

map<string, int> register\_distribute; // 该块的寄存器分配方案

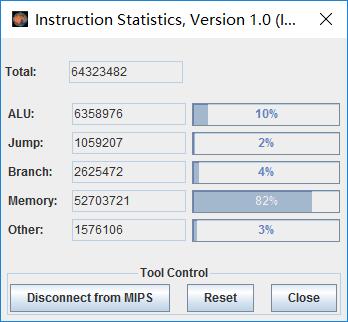
} \*blk\_link // 指向blk\_node结构体的指针

* 算法简述（寄存器分配方案的获得）

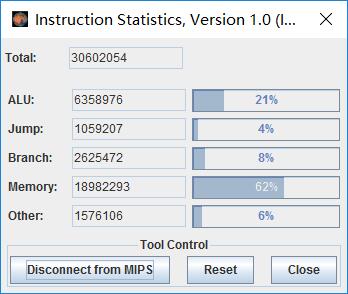
1. 建立局部变量池维护局部非数组变量。建立常量池维护局部和全局常量。
2. 划分基本块。该行有label，则该行为入口语句。该行为分支或跳转或返回语句，则下一行为入口语句。同时记录use集和def集。如果是常量则将其替换为数字。
3. 删去所有空的基本块。根据每个基本块的类型进行连接。
4. 循环计算out集和in集，直到所有基本块的in集不再扩大。
5. 计算active集。。
6. 根据每一基本块的active集构建冲突图矩阵。使用图着色算法得到全局寄存器分配方案。
7. 根据全局寄存器分配方案为每一个基本块分配寄存器。对于每一个基本块的active集中变量，如果全局寄存器分配方案为其分配了寄存器，则为其分配该寄存器，否则不分配寄存器。对于基本块的out – def集合中变量，为其分配剩余的寄存器。

* 在中间代码CAL的翻译过程中，需要保存现场。对只需对out集中的变量做sw和lw操作。

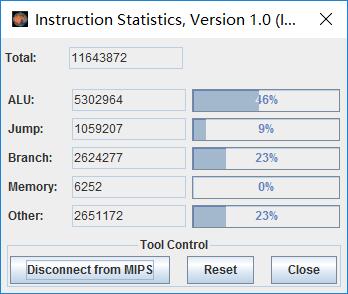
1. 优化效果



未优化



DAG图优化



DAG图与图着色共同优化

### 出错处理

该编译器出错处理采用跳读的方案。大致逻辑：当读到的符号不是预期的符号时，给定一个目标符号集合，当前符号不在该集合中时，不断向后跳读。循环跳读结束后可将当前符号类型限制在有限的集合中。使用条件判断对不同类型的符号做不同类型的错误处理，处理的共同目的是将其与之后的编译同步。编译器使用skip(set<SYMBOL> symset)函数进行跳读。传入的symset是目标符号集合。

错误信息共有39条。具体如下

"illegal string format", // 0

"no match symbol for current char", // 1

"can't declare variable after function declaration", // 2

"program start with const or var or func declaration", // 3

"missing `,` or `;` if it's var declaration, missing `(` or `{` if it's func declaration", // 4

"should be function name after `void`", // 5

"declaration start with `void`, `char`, `int`", // 6

"missing `(` or `{`", // 7

"should be `const`", // 8

"should be identity", // 9

"should be `=`", // 10

"should be int number and larger than 0", // 11

"ident repeated or invalid", // 12

"should be char", // 13

"should be `int` or `char`", // 14

"should be `;`", // 15

"should be `]`", // 16

"should be `)`", // 17

"should be `{`", // 18

"can't be left value", // 19

"identity not defined", // 20

"should be `}`", // 21

"should be `(`", // 22

"missing `else`", // 23

"missing `case`", // 24

"should be `:`", // 25

"parameter not matched", // 26

"return statement not exist", // 27

"should't be array", // 28

"should be array", // 29

"return type not matched", // 30

"should be int", // 31

"should be `[`", // 32

"can't call void function in expression", // 33

"switch type not matched", // 34

"main is not callable", // 35

"index out of range", // 36

"reach end of file", // 37

"file not exist" // 38

其中错误编号大于等于37的错误应当立即停机，其余错误保存下来，分析结束后打印。

## 操作说明

### 运行环境

codeblocks 16.01工程项目，编译器支持C++11标准。

### 操作步骤

编译得到compiler\_c0.exe，命令行输入：

compiler\_c0.exe filepath

或命令行输入：

compiler\_c0.exe

在程序提示输入文件路径时输入filepath。

如果程序存在语法语义错误，打印相关错误信息并结束程序。

如果程序正确，则输出如下文件：

mcode.c0 // 原始的中间代码

tcode.asm // 简单翻译策略得到的目标代码

mcode\_dag\_optimized.c0 // DAG图优化后的中间代码

tcode\_dag\_optimized.asm // 从DAG图优化后中间代码基于简单翻译策略得到目标代码

tcode\_coloring\_optimized.asm // 从DAG图优化后中键代码基于图着色寄存器优化翻译策略得到的目标代码

## 测试报告

### 测试程序及测试结果

|  |  |  |  |
| --- | --- | --- | --- |
| 测试程序 | 描述 | 核心覆盖范围 | 测试结果 |
| 测试1（正确） | 输入整数x，计算并输出第x个斐波那契数 | 函数递归调用，if-else语句，表达式内函数调用，独立的语句函数调用，输入输出。 | 正确 |
| 测试2（正确） | 输入整数x，随后输入x个整数。计算这x个数的和，平均数，最大值和最小值 | while循环语句，空语句，四则运算，数组存值和取值操作。 | 正确 |
| 测试3（正确） | 输入整数x，随后输入x个整数。使用快速排序对这x个数进行排序 | 函数递归调用，全局数组变量的取值和存值，不区分大小写。 | 正确 |
| 测试4（正确） | 输入两个整数x和y，计算这两个数的最大公约数 | 函数递归调用，函数调用内嵌函数调用，多数值输入，字符串和值联合输出 | 正确 |
| 测试5（正确） | 输入三个整数表示年，月，日，计算这一天是一年里的第几天。 | switch-case语句，if-else语句，联合输入和输出语句。 | 正确 |
| 测试6（错误） | 在全局和函数内变量声明之后加上常量声明。以及常量声明时没有为其赋值，变量声明时为其赋值。 | 声明顺序错误，声明时能否为其赋值错误。 | 正确 |
| 测试7（错误） | 测试词法分析程序，对非法字符和数字的检查。常量声明中将数字赋给字符和将字符赋给整数的错误。 | 常量赋值错误，词法中对非法字符和数字的检查。 | 正确 |
| 测试8（错误） | 测试无参函数的定义与调用时加上空括号的情况。表达式中调用无返回值函数的情况。 | 函数调用检查，函数声明检查，有参与无参的区分 | 正确 |
| 测试9（错误） | 针对数组的测试。检查静态数组越界的情况，包括向上越界和向下越界。检查范围包括整数和常量。数组操作时丢失右括号情况。 | 静态数组越界检查。数组完整性检查。 | 正确 |
| 测试10（错误） | 有返回值函数丢失return语句。函数内定义与函数同名变量或常量。无返回值函数返回值。对常量赋值。 | 返回值与函数声明的对应关系。对常量的赋值。函数内变量与当前函数同名。 | 正确 |

### 测试结果分析

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | t1 | t2 | t3 | t4 | t5 | t6 | t7 | t8 | t9 | t10 |
| 功能实现 | | | | | | | | | | |
| 函数递归 | √ |  | √ | √ |  |  |  |  |  |  |
| if-else | √ | √ | √ | √ | √ |  |  |  |  |  |
| while |  | √ | √ |  |  |  |  |  |  |  |
| switch-case |  |  | √ |  | √ |  |  |  |  |  |
| 空语句 |  | √ | √ |  |  |  |  |  |  |  |
| 强化四则运算 |  | √ | √ |  | √ |  |  |  |  |  |
| 数组访问 |  | √ | √ |  |  |  |  |  |  |  |
| 函数调用内嵌函数调用 |  |  |  | √ |  |  |  |  |  |  |
| 表达式函数调用 | √ | √ | √ | √ |  |  |  |  |  |  |
| 独立语句函数调用 |  |  | √ |  |  |  |  |  |  |  |
| 输出语句的值输出 | √ | √ | √ | √ | √ |  |  |  |  |  |
| 输出语句的字符串输出 |  | √ | √ | √ | √ |  |  |  |  |  |
| 整型输入语句 | √ | √ | √ | √ | √ |  |  |  |  |  |
| 字符型输入语句 |  |  |  | √ |  |  |  |  |  |  |
| 常量声明 |  |  | √ |  |  |  |  |  |  |  |
| 变量声明 | √ | √ | √ | √ | √ |  |  |  |  |  |
| 不区分大小写 |  |  | √ |  |  |  |  |  |  |  |
| 错误处理 | | | | | | | | | | |
| 声明顺序检查 |  |  |  |  |  | √ |  |  |  |  |
| 重复定义 |  |  |  |  |  |  |  |  |  | √ |
| 常量中字符型与整型赋值检查 |  |  |  |  |  |  | √ |  |  |  |
| 静态数组越界检查 |  |  |  |  |  |  |  |  | √ |  |
| 返回值与函数类型匹配 |  |  |  |  |  |  |  | √ |  |  |
| 调用时参数不匹配 |  |  |  |  |  |  |  | √ |  |  |
| 无参函数声明和调用加括号 |  |  |  |  |  |  |  | √ |  |  |
| 词法分析非法字符和数字 |  |  |  |  |  |  | √ |  |  |  |
| if不包含else的检查 |  |  |  |  |  |  |  |  | √ |  |
| if和while条件判断为空 |  |  |  |  |  |  |  |  | √ |  |
| 左值为常量或数字 |  |  |  |  |  |  |  |  |  | √ |

## 总结感想

编译技术课程设计是一门对综合程序设计能力很高的课程，一方面要求较高的设计编码水平，需要用工程实现的角度去思考程序的结构，而不是数据结构等课程中仅从算法流程的角度去思考，对C++编码能力和工具的运用能力也有很高的要求；另一方面要求对编译课上的内容融会贯通并且灵活运用，理论课中的内容不能直接用在自己的编译器中，一部分内容需要更多的补充和修改，还有一部分内容略显冗余，这些都需要在设计实现时根据实际情况做决定。

实现编译器之前，应当想到这是一个十分庞大的工程，必须保证程序结构设计的完整与正确才可以动手实现，避免在编码时发现之前考虑不周而重头来过。同时每一步的正确性必须得到保证，才能在整合时不出现问题。在完整工程运行时出现问题是最要命的。另一点很重要的就是程序结构清晰，关键代码的注释有助于对程序的理解。后期最后悔的一件事情就是没有做命名空间，在实现类似功能时出现重名函数或变量的问题导致后期不能以最直接的命名方式去命名一个变量或函数，降低程序的可读性。另一部分就是使用字符串保存中间代码，在输出时方便很多，但是在后续的翻译中不得不采用资源耗费极大的正则匹配的方式提取相关名字。

优化部分的实现需要对编译和数据流动有比较深的理解。教材上的算法较为抽象，都是在最理想的情况下，在实现过程中却并非理想情况，需要考虑更多的问题。这一部分需要更多的思考与扩充设计。

最终编译器完整功能的实现和优化的实现给了我很强烈的成就感，我也能感到自己对工程项目的理解和认识。这也是我生平第一次个人完成将近三千行的代码，是锻炼自己的极好的平台。

最后对编译课程有一个小小的建议，希望课程团队能总结历届问的比较多的文法实现的问题做成文法解释文档，帮助同学们更好的理解文法语法语义，而不需要频繁的看论坛来保证自己的理解和官方解释一致。