《编译技术》课程 设计文档

学号：16721145

姓名：史鼎元

2018年 11月 12日

## 一．需求说明

### 1．文法说明

获得的文法如下：

1、＜加法运算符＞ ::= +｜-

2、＜乘法运算符＞ ::= \*｜/

3、＜关系运算符＞ ::= <｜<=｜>｜>=｜!=｜==

4、＜字母＞ ::= ＿｜a｜．．．｜z｜A｜．．．｜Z

5、＜数字＞ ::= ０｜＜非零数字＞

6、＜非零数字＞ ::= １｜．．．｜９

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

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

9、＜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＞

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

11、＜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＞}

12、＜无符号整数＞ ::= ＜非零数字＞｛＜数字＞｝｜０

13、＜整数＞ ::= ［＋｜－］＜无符号整数＞

14、＜标识符＞ ::= ＜字母＞｛＜字母＞｜＜数字＞｝

15、＜声明头部＞ ::= int＜标识符＞|char＜标识符＞

16、＜变量说明＞ ::= ＜变量定义＞;{＜变量定义＞;}

17、＜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＞']') } //＜无符号整数＞表示数组元素的个数，其值需大于0

18、＜类型标识符＞ ::= int | char

19、＜有返回值函数定义＞ ::= ＜声明头部＞'('＜参数表＞')' '{'＜复合语句＞'}'

20、＜无返回值函数定义＞ ::= void＜标识符＞'('＜参数表＞')''{'＜复合语句＞'}'

21、＜复合语句＞ ::= ［＜常量说明＞］［＜变量说明＞］＜语句列＞

22、＜参数表＞ ::= ＜类型标识符＞＜标识符＞{,＜类型标识符＞＜标识符＞}| ＜空＞

23、＜主函数＞ ::= void main'('')' '{'＜复合语句＞'}'

24、＜表达式＞ ::= ［＋｜－］＜项＞{＜加法运算符＞＜项＞} //[+|-]只作用于第一个<项>

25、＜项＞ ::= ＜因子＞{＜乘法运算符＞＜因子＞}

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

27、＜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＞;｜＜返回语句＞;

28、＜赋值语句＞ ::= ＜标识符＞＝＜表达式＞|＜标识符＞'['＜表达式＞']'=＜表达式＞

29、＜条件语句＞ ::= if '('＜条件＞')'＜语句＞［else＜语句＞］

30、＜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

31、＜循环语句＞ ::= do＜语句＞while '('＜条件＞')' |for'('＜标识符＞＝＜表达式＞;＜条件＞;＜标识符＞＝＜标识符＞(+|-)＜步长＞')'＜语句＞

32、＜步长＞::= ＜无符号整数＞

33、＜有返回值函数调用语句＞ ::= ＜标识符＞'('＜值参数表＞')'

34、＜无返回值函数调用语句＞ ::= ＜标识符＞'('＜值参数表＞')'

35、＜值参数表＞ ::= ＜表达式＞{,＜表达式＞}｜＜空＞

36、＜语句列＞ ::=｛＜语句＞｝

37、＜读语句＞ ::= scanf '('＜标识符＞{,＜标识符＞}')'

38、＜写语句＞::= printf'('＜字符串＞,＜表达式＞')'|printf '('＜字符串＞')'|printf '('＜表达式＞')'

39、＜返回语句＞ ::= return['('＜表达式＞')']

有关说明如下：

（1）char类型的变量或常量，用字符的ASCII码对应的整数参加运算。

（2）标识符区分大小写字母。

（3）写语句中，字符串原样输出，单个字符类型的变量或常量输出字符，其他表达式按整型输出。

（4）数组的下标从0开始

上述文法虽然存在FIRST集相交的问题，不是LL(1)文法，但是在程序设计上进行预读和处理，可以避免出现问题，所以无需对文法进行改写。

### 2．目标代码说明

编译程序生成的目标代码为MIPS汇编指令。主要使用的指令（包含伪指令）如下表。

|  |  |
| --- | --- |
| 格式 | 含义 |
| add $t0 $t1 $t2 | $t0 = $t1 + $t2 |
| sub $t0 $t1 $t2 | $t0 = $t1 - $t2 |
| addi $t0 $t1 1 | $t0 = $t1 + 1 |
| mul $t0 $t1 $t2 | $t0 = $t1 \* $t2 |
| div $t0 $t2 $t3 | $t0 = $t1 / $t2 |
| li $t0 100 | $t0 = 100 |
| move $t0 $t1 | $t0 = $t1 |
| lw $t0 4($sp) | $t0 = $sp[4] |
| sw $t0 4($sp) | $sp[4] = $t0 |
| j $label | jump to $label |
| jal $func | jump and link to $label |
| jr $ra | jump register $ra |
| beq $t0 $t1 $label | branch to $label if $t0==$t1 |
| bne $t0 $t1 $label | branch to $label if $t0!=$t1 |
| bge $t0 $t1 $label | branch to $label if $t0>=$t1 |
| ble $t0 $t1 $label | branch to $label if $t0<=$t1 |
| bgt $t0 $t1 $label | branch to $label if $t0>$t1 |
| blt $t0 $t1 $label | branch to $label if $t0<$t1 |

### 3. 优化方案\*

为了提高生成代码的执行效率，采取了寄存器分配、常量传播、复制传播、保存现场优化以及窥孔优化。

3.1 寄存器分配方案

寄存器分配需要根据变量的性质进行。根据程序的运行特点，我把变量分成了三类：第一类是代码意义上的全局变量，也就是用户在代码头部声明的全局变量。第二类是基本块内的局部变量，这类变量有两种，一种是中间变量，它们一定是不跨越基本块的，还有一些用户声明的局部变量可能在使用过程中没有跨越基本块，因此一样可以看作是局部的。第三类是跨越基本块全局变量，这类变量虽然是局部的，但是它跨越了基本块。

对于上面的三类，分别采用下面的策略进行分配。

对于代码全局变量：理论上为了保证全局变量的线程安全性，是不应该为其分配任何寄存器的，但是我们只会执行一个线程，所以如果指定寄存器并且在整个程序的执行过程中始终不改变，就不会出现问题，而且还可以减少大量的访存指令。为此，我会给两个全局变量（如果存在）分配两个全局寄存器。

对于基本块内的局部变量：基本块的划分保证了在一个基本块内，程序是顺序执行的，这说明在编译一个基本块的时候就已经可以确定它在执行时的顺序，所以可以采用动态分配方法。动态分配的特点是一个寄存器可能会对应多个变量，所以可能出现寄存器别名问题，所以如果执行顺序未知的话，很可能出现问题。因此这种分配方式只能在一个基本块内使用，因为块内的编译顺序和执行顺序是一致的。寄存器如果动态分配，是可以借用内存管理中的页面分配的，可以使用最优的方法，但是代码会比较复杂，所以使用LRU（最近最久未使用的策略进行分配）的分配策略能够比较好的实现。

对于跨越基本块的全局变量：因为跨越了基本块，执行的顺序可能在编译过程中是不能够个确定的，这样必须要把寄存器和变量绑定，在这个函数的执行过程中不能变更。但是这些变量肯定是用户声明的局部变量，所以函数结束的时候可以解除绑定。

3.2 常量传播与复制传播

常量传播是一个渗透在中间代码生成和优化中间代码之间的过程，因为语法分析程序的设计特点，分析因子的时候如果是常数会返回常数的值，所以在生成中间代码时，如果有两个操作数都是常数，会把运算指令转化为赋值语句。

接下来对中间代码做常量传播时，如果遇到赋值语句，是把一个中间变量赋值为常量，则可以直接进行替换，因为中间变量在我的文法中是不会跨块的，所以不用担心对它的使用问题。同样，如果遇到中间变量被赋值成一个局部变量，在这个局部变量的值被改变之前，同样可以进行替换，如果发现当基本块的末尾依然没有出现对局部变量的改变，就可以把这个赋值语句删除。

3.3 保存现场优化

函数调用时需要保存现场，最简单的策略是把31个（0号除外）寄存器全部保存，但是这样每次函数调用会出现大量的访存指令，严重降低效率，所以采用的办法是个性化保存现场。因为寄存器的分配情况在编译的时候是知道的（即使是t寄存器采用动态分配策略，由于基本块中的编译顺序和执行顺序一致，所以依然是知道寄存器的分配情况的），因此可以只把使用的寄存器进行保存。这样寄存器的保存就仅仅局限在了s寄存器和t寄存器了，a寄存器仅仅是函数参数传递的媒介并不需要保存。

函数调用并不是一个基本块的划分依据，所以对于t寄存器仍然需要保存，如果能够对中间变量的生存期有着精准的认识，就可以及时的清理掉中间变量。这样一方面缓解t寄存器的压力，一方面也能在保存现场的时候少保存寄存器。

为此主要运用了中间变量的两处特征。

第一，如果赋值语句的左侧是一个用户定义的变量，那么这一定是一个语句的结束，中间变量可以全部去除。

第二，如果遇到一条函数调用指令，那么所有传参数使用的寄存器都可以删除。

s寄存器使用量是比较大的，因为通常一个函数都会有5个左右的跨块变量，他们都要占据s寄存器，为此，保存现场的时候还可以进行一些特判减少保存现场开销。

第一，如果被调用的函数是一个叶子函数，由于它不会再调用其他函数，那么可以很清楚的知道未来会被使用的s寄存器有哪些，这样只需要保存那些父过程和被调用的子过程都是用的寄存器就可以了。

第二，这其实是第一种情况的特殊情况，如果这个函数是一个叶子函数而且不跨块，那么可以直接不保存s寄存器，这其实是一种类似内联展开的做法。

3.5 窥孔优化

对中间代码的窥孔优化主要针对一个翻译过程中出现的问题，当把一个表达式的值赋给一个变量的时候，只要表达式不是一个单独的常数或变量，就出现类似下面的代码：

ADD #1 a, b

ASSIGN c #1

这样的代码只要有比较复杂的赋值语句就一定会出现，所以对其进行窥孔优化效果是十分明显的，考虑将其收缩成一条：

ADD c a, b

这样需要注意的问题是，数据一致性的问题。为保证数据一致性，如果值被改变的变量是一个没有分配寄存器的变量，那么在生成目标代码的时候会生成一条回写内存的指令。原来，这样对未分配寄存器的变量改值的指令只有四条：LELEM（取数组元素）、INV（读数字）、INC（读字符）、ASSIGN（赋值），但是现在的经过窥孔优化之后，就不止有这四条了，运算类的四条指令（ADD、SUB、MULT、DIV）同样需要这样对未分配寄存器的变量进行处理。

另外还做了一些让代码更加简洁的优化，为了避免函数体没有显式写返回语句的问题，我在函数编译结束的时候不管这个函数有没有返回值都会加一条返回指令，这条指令可能是多余的，现在也可以删掉，另外有些时候可能在同一个地方设置了多个标签，让代码变得繁琐，这些多重标签同样可以压缩成一个。如果GOTO（无条件跳转）到的标签恰好就在这条指令下面，那么这条GOTO语句同样可以删除。

## 二．详细设计

### 1．程序结构

整个程序可以分为五个部分，分别是词法分析、语法分析、语义分析、中间代码生成与优化、目标代码生成与优化。伴随着两个过程，错误处理和符号表管理。各个模块之间的相互作用配合如下图所示。

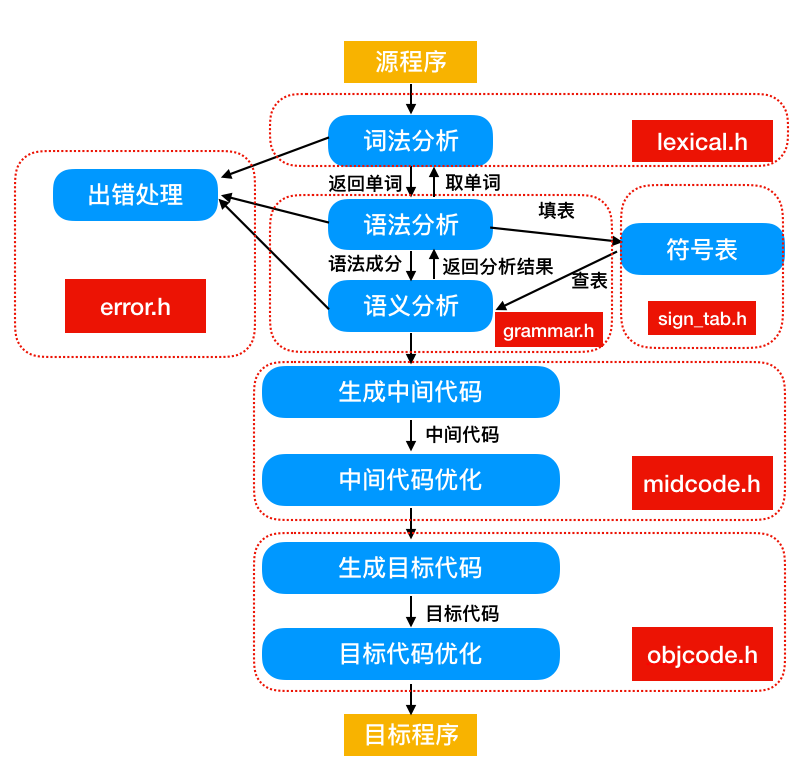


图 1 编译程序结构

### 2．类/方法/函数功能

2.1 函数功能

词法分析

|  |  |
| --- | --- |
| 函数名、参数、返回类型 | 功能 |
| void readFile() | 从文件读取源程序到数组 |
| bool isalu(char c) | 辅助函数，确定是否是字母或下划线 |
| bool isalnum(char c) | 辅助函数，确定是否是字母下划线或数字 |
| char nextch() | 辅助函数，读下一个字符，更新行列 |
| void nextsym\_inner(type\_set &type, int &val, string &name) | 内核函数，读取单词，返回类型、值、名称 |
| void nextsym(type\_set &type, int &val, string &name) | 核心函数，调用内核函数，保证返回一个正确的单词。也是外界调用此模块的唯一函数 |

错误处理

|  |  |
| --- | --- |
| 函数名、参数、返回类型 | 功能 |
| void errmg() | 输出报错信息。 |
| void skip() | 跳读函数。 |

符号表管理

|  |  |
| --- | --- |
| 函数名、参数、返回类型 | 功能 |
| int insert\_tab(bool islocal, string name, int feature, int address, int size) | 根据是否是全局变量，填写有关的符号表并建立索引，返回在符号表中的位置。 |
| int search\_tab(string name) | 查找符号表，返回位置 |

语法语义分析

|  |  |
| --- | --- |
| 函数名、参数、返回类型 | 功能 |
| void program() | 程序分析的开始。 |
| void constDef() | 分析常量的定义，并填写符号表。 |
| void varDecl() | 分析变量声明，并填写符号表。 |
| void funcDef() | 分析函数定义，并填写符号表，生成有关中间代码。进行语义分析。 |
| void paraList() | 分析参数表，生成有关代码。进行语义分析。 |
| void valpara(string funcname) | 分析值参数表，需要根据函数参数类型判定合法性。 |
| void integer() | 分析整数。 |
| void statements() | 分析复合语句，生成有关代码。进行语义分析。 |
| void stmtList() | 分析语句列，生成有关代码，进行语义分析。 |
| void becomestmt() | 分析赋值表达式。 |
| void ifstmt() | 分析if语句。 |
| void condition() | 分析条件语句。 |
| void dowhilestmt() | 分析do-while语句 |
| void forstmt() | 分析for语句。 |
| void expr() | 分析表达式。 |
| void term() | 分析项。 |
| void factor() | 分析因子。 |
| void print() | 分析printf语句。 |
| void scan() | 分析scanf语句。 |
| void ret(int type) | 分析返回语句。需要根据函数类型判断是否合法。 |

中间代码生成与优化

|  |  |
| --- | --- |
| 函数名、参数、返回类型 | 功能 |
| void gen(int op, int res, int num1, int num2) | 生成四元式并插入四元式表。 |
| void midcodeoptimize() | 中间代码优化函数总入口。 |
| void dagOpt() | DAG图优化。 |
| void constOpt() | 常量优化。 |
| void kkOpt() | 窥孔优化 |
| void globalOpt() | 全局公共子表达式优化。 |
| void flush() | 把中间代码输出到文件。 |

目标代码生成与优化

|  |  |
| --- | --- |
| 函数名、参数、返回类型 | 功能 |
| void genMips() | 实现代码映射。 |
| void mipsOptimize | 优化调用总入口。 |
| void regOpt() | 寄存器分配优化。 |
| void instrOpt() | 指令选择优化。 |

活跃变量分析与寄存器分配

|  |  |
| --- | --- |
| 函数名、参数、返回类型 | 功能 |
| void cal\_blocks() | 划分基本块。 |
| void gen\_DAG() | 计算流图。 |
| void cal\_def\_use() | 计算def和use集合。 |
| void cal\_in\_out() | 计算in和out集合。 |
| void cal\_reg() | 计算寄存器分配策略。 |

2.2 关键算法

由于文法存在FIRST集相交的问题，而且不好改写，所以有些地方需要进行预读，以程序分析为例。可以根据const判断进入常量分析的函数。但是后面遇到int、char都不能确定是变量还是函数，所以需要再往后读两个，也就是标识符的下一个符号，这个符号如果是分号，那么可以直接填表。如果是‘（’进入函数分析程序，同时把是否允许变量标记（canVar）置为false，因为已经出现函数了，所以接下来就只能是函数。如果是逗号，向后读一个，进入词法分析程序。

这里的巧妙之处在于，我们进入变量分析程序的时当前单词一定是标识符，这样的话不论是LL(1)（在函数内部，遇到int/char可以立即断定是变量声明调用词法分析）还是LL(2)（全局状态下需要往后读才能确定），都能够使用，就解决了可能存在的不一致问题，实现了代码复用。

空类型函数和main函数的分析类似。这里比变量的分析还要简单，因为这两个分析函数只会被调用一次，（不像常量变量分析在全局和函数内部都有），所以更加简单。

### 3．调用依赖关系

各个函数相互调用关系错综复杂，主程序的调用顺序如下图：



图 2 主函数调用顺序

program部分为语法分析和中间代码生成，有关函数的的相互调用如下：

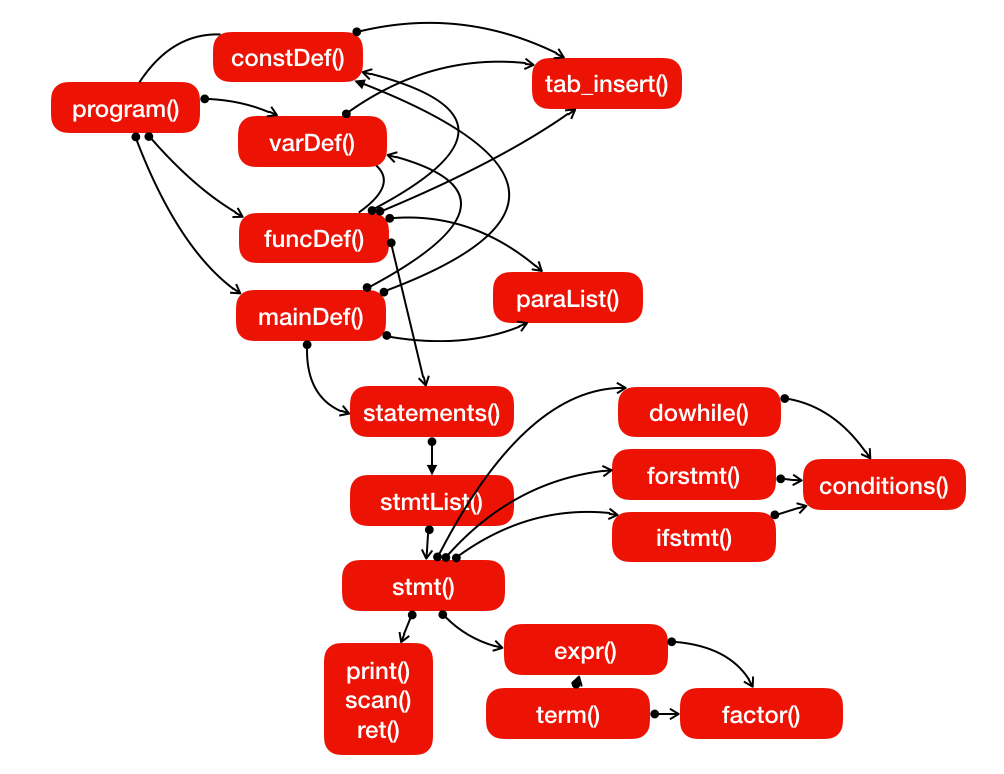


图 3 语法分析函数调用关系

mcOpt为中间代码优化，有关函数相互调用如下：

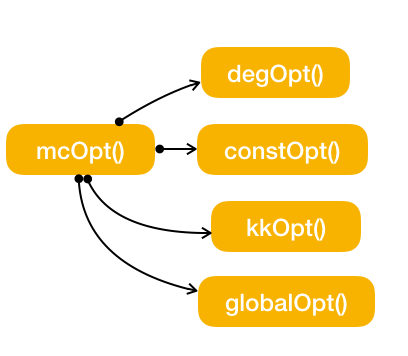


图 4 中间代码优化函数调用关系

mipsOpt为目标代码优化，有关函数相互调用如下：

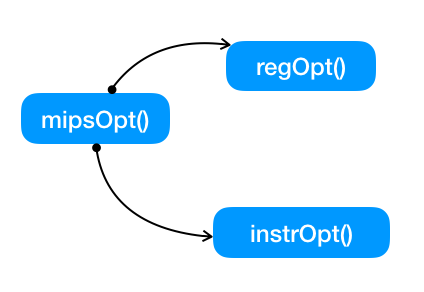


图 5 目标代码优化调用关系

### 4．符号表管理方案

4.1符号表结构

由于我们的语法规则限制了函数内部不能定义函数，这样可以极大地简化符号表的设计。符号表设计采用栈式符号表。由一张全局表（记录全局变量、常量、已经编译完成的函数名及其参数、当前正在编译的函数名和参数、局部常量、局部变量和字符串）和一张辅助表（记录字符串）实现。

同时为了加速查找的过程，采用C++的map建立一个全局符号的名字到表项的索引来加速查找。所以还要为全局表表建立一个索引。因为map是按照字典序存储的，所以查找和插入的过程都是O(logn)的时间复杂度，而且因为是STL，编程难度也很小。

全局表索引建立了标识符名称到标识符在全局表中位置的映射。在全局索引表中可以查找的标识符为全局变量、全局常量、函数名称。函数参数和局部变量虽然在符号表中，但是并不会建立索引，查找参数需要利用全局索引找到函数名的位置，由于栈式符号表的特点，函数参数会紧跟在函数名所在表项的后面，因此线性查找即可。

局部表索引建立了标识符名称到标识符在局部表位置的映射。局部索引表中可以查找局部变量、常量、字符串。

全局表各个域的名字和含义为：名字（name）、种类（kind）、类型（type）、特征值（feature）大小（size）和地址（addr），是否局部（islocal）。

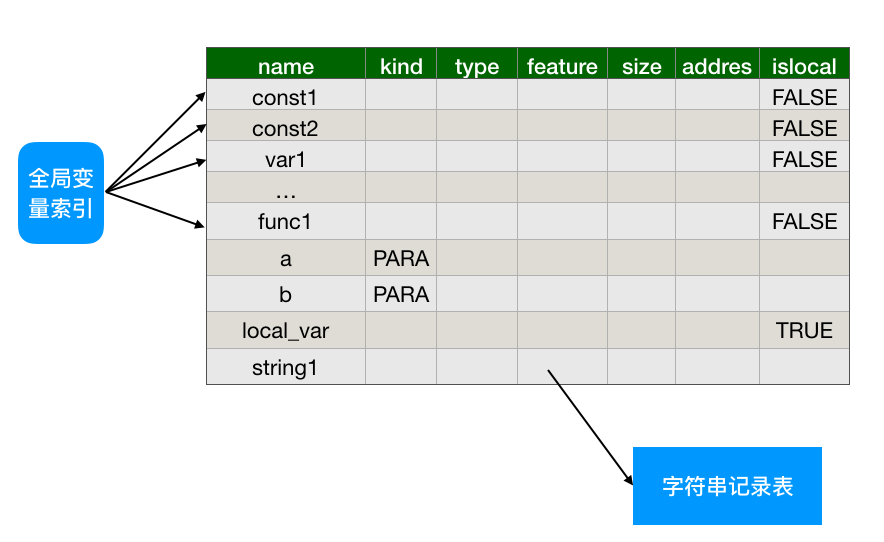


图 6 符号表结构示意

其中：

名字（name）为标识符的名字；其中字符串需要编译程序给一个名字，使用string1、string2编号。

种类（kind）为常量（const）、简单变量（var）、函数（function）、数组（array）、参数（parameter）和字符串（string）。

类型（type）为整型（int）和字符型（char）以及空类型（void，仅适用于函数）。

大小（size）为占据空间的大小，字符占据空间大小为了对齐方便分配4字节空间，字符串占据的空间并不是字符串实际的空间，只是存储其首地址的空间。

地址（addr），相对地址。

特征值（feature）的的意义根据标识符的种类和类型而定，具体意义见下表。

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| kind | const | | var | func | array | string |
| type | int | char | int/char | int/char/void | int/array |  |
| feature | 值 | ascii码 | 无意义 | 参数个数 | 长度 | 字符串首地址 |

注：字符串只需要记录首地址即可，因为在编译过程中会把存储的字符串末尾自动追加\0。

4.2 符号表的程序实现

首先是数据结构。

全局符号表。每个表项的类型为一个结构体定义，使用该结构体对应的结构体数组实现。局部符号表索引和全局符号表索引。索引采用map实现。

字符串表使用字符数组实现，每个串的结尾追加‘\0’。

全局符号表待写入位置：gstp（global sign table pointer）

字符串表待写入位置：strtabp。（string table pointer）

之后是操作对应的函数。

填写函数tab\_insert：负责符号表的填写。

查找函数tab\_saerch：负责符号表的查找。

具体过程见下一小节。

4.3 符号表的填写、查找与释放

符号表填写过程为：遇到一个声明，把变量插入符号表，之后再判断变量是全局还是局部，如果是全局变量建立索引。

符号表查找过程为：先利用函数名找到对应的表项函数起始位置，开始向下查找到待写入位置，如果找到就能够找到就说明是一个局部符号或者是参数。如果没有找到，则可能是去全局变量，把当前函数名在全局符号表中查找，利用map建立的索引，可以实现时间复杂度为O(nlog)的查找。然后找到对应位置，说明是一个全局变量。否则，如果没有找到，说明是一个未声明的变量，输出报错信息。

其实除了上面两个过程，教材上还提到了符号表的释放，这里我们是不需要也不能够释放符号表的。首先我们不能够释放符号表，因为现在的符号表里有很多常量的信息，在四元式中都只是一个标识符，我们需要在生成目标代码的时候利用它查找标识符对应量的值。另一方面也是没有必要释放的，因为我们的函数不会存在嵌套定义，所以变量只会出现在全局或者这个函数本身，而我们的查找方式，从当前函数所在的位置向下查找局部变量以及利用索引查找全局变量，根本不会进入其他函数的局部变量空间，所以很安全，不需要释放。

### 5. 存储分配方案

5.1 静态存储方案

根据此文法，可以知道，全局变量和全局常量以及字符串可以在编译阶段确定其地址。

针对上面的这类数据，应该分配在内存的最低地址区，生成编译代码的时候直接定位到对应的地址。

其中，全局常量其实是不需要分配存储空间的，因为常量可以直接被替换到指令中，值一点将在优化四元式的时候实现。针对字符串，使用mars中的伪指令.asciiz来实现存储分配，如果把MARS设置称为数据从0地址开始，那么字符串是从0开始的最低的地址区分配；针对全局变量，应该在字符串的上面放置。数据从0开始分配则代码会从0x3000开始分配。这样的话只要字符串和全局变量占据的空间不超过12KB，程序就可以正常运行。

从0x3000向上是编译出来的代码，值得注意的是，PC的初始值是0x3000，这里应该是函数的入口，但是我们编译的顺序总是最后编译main函数，所以0x3000位置的代码应该是一条无条件转移指令，转移到main标签，然后再执行。

5.2 动态存储方案

静态存储是从低地址到高地址分配的，而动态存储分配则相反。整个内存从高地址到低地址为函数分配运行栈，地址最低的函数为当前运行的函数。因为文法并不支持在函数运行时动态申请空间，所以并不涉及堆的使用，而且每个函数的占据空间的大小在编译阶段就可以完全确定。因此每个函数的运行栈大小是确定的，应该有如下关系：

其中，local\_variable是函数的局部变量（包括参数）占据的空间，contex是保护现场需要的内存空间。

在调用函数之前，先分配空间（$sp减去运行栈的大小），然后传递参数（从$sp向上存储），进入函数以后，先保存现场，上面是局部数据区。

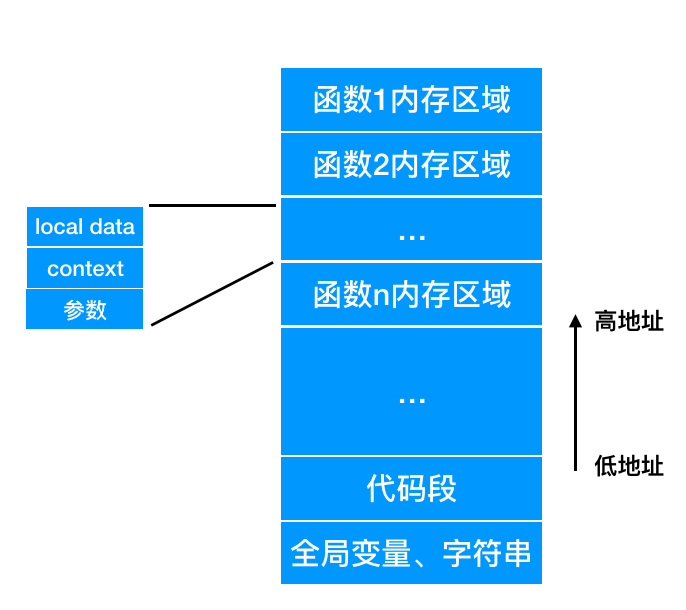


图 7 内存分配

### 6. 解释执行程序

此编译程序生成MIPS汇编指令使用MARS执行。无需解释执行程序。

### 7. 四元式设计

四元式及中间代码，目的是便于优化和生成汇编指令，基于这些考虑，设计出如下规则的四元式表示。

格式：结果（RES） 操作符（OP） 操作数1（NUM1） 操作数2（NUM2）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| OP | RES | NUM1 | NUM2 | 含义 |
| VAR | int/char | 变量名 |  | 声明变量语句 |
| CONST | int/char | 变量名 |  | 声明常量语句 |
| PARA | int.char | 变量名 |  | 参数声明语句 |
| LABEL | 函数名 |  |  | 声明一个int/char型函数 |
| PUSH | 参数名 |  |  | 传递参数 |
| CALL | 函数名 |  |  | 调用某个函数 |
| ASSIGN | 变量名 | 源 |  | 赋值，源为RET为函数返回值 |
| RET | 标识符 |  |  | 函数返回语句 |
| ADD | 和 | 加数 | 加数 | 加法语句 |
| SUB | 差 | 被减数 | 减数 | 减法语句 |
| MULT | 积 | 因子 | 因子 | 乘法语句 |
| DIV | 商 | 被除数 | 除数 | 除法语句 |
| EQ | 比较结果 | 比较数1 | 比较数2 | 进行比较得到结果0/1 |
| NE | 比较结果 | 比较数1 | 比较数2 | 进行比较得到结果0/1 |
| GT | 比较结果 | 比较数1 | 比较数2 | 进行比较得到结果0/1 |
| LT | 比较结果 | 比较数1 | 比较数2 | 进行比较得到结果0/1 |
| GE | 比较结果 | 比较数1 | 比较数2 | 进行比较得到结果0/1 |
| LE | 比较结果 | 比较数1 | 比较数2 | 进行比较得到结果0/1 |
| GOTO | 标签 |  |  | 无条件跳转至标签 |
| LELEM | 取的值 | 数组名 | 索引名 | 取数组元素 |
| SELEM | 数组名 | 索引名 | 存数值 | 存数组元素 |
| BNZ | 标签名 | 条件 |  | 不满足条件跳转 |
| BEZ | 标签名 | 条件 |  | 满足条件跳转 |
| INC | 变量名 |  |  | 从标准输入读取 |
| INV | 变量名 |  |  |  |
| OUTS | 字符串名 |  |  | 输出字符串 |
| OUTV | 标识符 |  |  | 输出值 |
| OUTC |  |  |  |  |
| EXIT |  |  |  | 请求OS杀死进程 |

中间代码中的各类操作数有可能是标识符、数字和函数返回值。其中，标识符可以是全局变量、局部变量和四元式生成的临时变量；函数返回值应该使用$v0寄存器存储，在四元式中使用#RET表示（区分可能的标识符）。

对示例中给出的修正，对于关系运算符，根据文法，它只在条件表达式中有意义，也就是其结果不会做为一个值参与表达式运算，所以可以直接对应成为一条有条件跳转指令，有助于后面的目标代码生成。

对于原有的示例，给出的扩充为，SCON、RCON与函数保存现场和恢复现场有关，IN、OUTS、OUTV与标准输入输出有关。EXIT，和程序终止有关。

为了便于使用DAG优化，把空缺的操作数用零补全。此外，为了防止出现不正确的优化结果，对于数组运算，应该改成中缀表示，如x = a[i]表示为x=a[]i，a[i] = x表示为a=i[]=x。

从高级语言（C0）到中间代码（四元式）的转化过程使用语法制导翻译来实现，设计方法采用自顶向下的方法。

中间代码的存储结构使用C++语言中的vector来实现，vector的每一个元素为一条四元式，每一条是一个结构体，有四个域，分别是op（枚举类型）、num1（字符串类型）、num2（字符串类型）、res（字符串类型）。

### 8. 目标代码生成方案

8.1 四元式到MIPS的转换

|  |  |
| --- | --- |
| OP | 转换方法（暂不考虑变量和寄存器的转化问题） |
| PUSH | 传参数，把参数加入到参数队列中。 |
| CALL | 从参数队列生成lw/move指令，放到对应的内存中或a寄存器中 |
| ASSIGN | 生成move或li指令 |
| RET | 生成对v0寄存器的赋值指令和jr $ra指令 |
| ADD | 如果NUM1和NUM2都是数字，编译器计算出结果（设为x），加载立即数指令。如果有一个是数字（设为NUM1），使用addi RES, NUM2, NUM1。如果都是变量使用 add RES，NUM1，NUM2，减法类似。乘除法可以根据是否为2的整数幂转化为移位运算。 |
| SUB |
| MULT |
| DIV |
| EQ | 比较运算一定出现在条件中，所以结合下面的BNE或BEZ生成对应的beq、bne、blt、bgt、ble、bge指令 |
| NE |
| GT |
| LT |
| GE |
| LE |
| BNZ |
| BNZ |
| LABEL | lebel： |
| GOTO | j label |
| SELEM | 生成地址运算指令，然后sw |
| LELEM | 生成地址运算指令，然后lw |
| INC/INV | 根据类型判断向v0写入5或12，然后存储到相应的变量中 |
| OUTS | 生成la指令加载字符串首地址，向v0写入4，系统调用 |
| OUTV/OUTC | 根据类型判断向v0写入1或11，然后把a0写入要输出的量 |
| EXIT | li $v0 10, syscall |

8.2 关键算法

代码映射是很直观的，代数运算、逻辑运算、跳转等都基本上是直接映射的，比较复杂的主要是CALL、PUSH、OUTS、OUTV、OUTC。

CALL和PUSH为函数调用和传参数指令，同时需要注意保存现场保存和恢复现场，主要应该保存需要保存的通用寄存器的值。

OUTS、OUTV和OUTC是针对三种不同的printf格式来制定的。printf(字符串)对应一句OUTS，目标代码中为la、li和syscall。printf(表达式)对应一句OUTV，目标代码中为la、li、syscall。printf(字符串，表达式)对应两者的组合。

8.3 四元式的数据结构

四元式采用结构体数组，每一个数组元素是一个四元式，它由4个整数域构成，分别代表了操作符码，操作数在符号表中的位置，如果是临时变量，采用负数，数字绝对值为临时变量的编码。

### 9. 优化方案

9.1 基本块与流图的建立

教材上的基本块划分语句为：

整个语句序列的第一条语句、能由跳转语句到达的第一条语句，紧跟在跳转语句后面的第一条语句。

考虑到中间代码的形式，能由跳转语句到达的第一条语句这一条应该被修改为标签，所以能划分基本块的开始语句有：

第一条语句、

LABEL、

RET、BEZ、BNZ、GOTO的后一条语句。

在划分好基本块之后建立流图。每个基本块的前驱只需要根据中间代码顺序即可确定。接下来说明基本块的后继。首先，将前面所述的基本块开始语句分类，可以分为：

条件/无条件跳转、函数的调用和返回。

对于第一类，可以根据其跳转的标签建立后继的关系，对于第二类，函数的调用和返回语句没有后继，因为返回以后空间会释放，不会和其他的代码产生关系。最后一句EXIT语句的后继为出口块。

通过上面的逻辑，可以为中间代码建立流图。

为其建立数据结构需要考虑两个问题，第一，在流图中一个基本块作为一个结点存在，第二，每个基本块和他所对应的中间代码的关系。

全局优化需要大量的数据结构作为基础，主要有数据流、定义-使用链（网）以及由此推导出的冲突图。

数据流的数据结构对于每个基本块而言是六个集合，即IN、OUT、GEN、KILL、DEF、USE。它们之间的关系由下面的公式决定。

根据这些公式不难设计出求解每个基本块进入和离开时的活跃变量和到达定义。

到达定义分析算法：

**Input**：kill and gen set of each block B

**Output**：in and out set of each block B

**Algorithm：**

**initial** Bexit and out[Bi]

**while** there still a set change:

in[B] = **union**(out[P])

out[B] = **union**(gen[B], (in[B]-kill[B]))

活跃变量分析算法：

**Input**: def and use block of each block B

**Output**:in and out block of each B

**Algorithm**:

**initial** Bexit and in[Bi]

**while** there still a set change:

out[B] = **union**(in[N])

in[B] = **union**(use[B], (out[B]-def[B]))

这两个算法的输入比较容易得到，只要简单的扫描和判断即可，唯一值得一提的是扫描的范围。如果变量是局部变量，只要在这个函数内的范围内扫描即可，不需要进行全局扫描。这样可以减少不必要的搜索，加快优化速度。

根据活跃变量分析的结果，可以比较容易得到定义-使用链（网），只要扫米in、out集合构建一张图即可，不再赘述。

9.2 寄存器分配[[1]](#footnote-1)

9.2.1 寄存器分配策略综述

寄存器分配主要的分配对象为s寄存器和t寄存器。其中s寄存器分配给全局变量，这里的全局变量指的是跨越基本块的变量，而t寄存器分配给局部变量，这里的局部变量是指不跨越基本块的变量，包括用户声明的未跨越基本块的变量和中间变量（根据我所抽取到的文法的特点，中间变量一定不会跨块）。

而真正的全局变量，也就是在源代码中函数定义之前所声明的变量，并不分配寄存器，因为按照教材中的有关描述，为了保证线程安全，全局变量不应该分配寄存器，所以每次使用都要访存，虽然这样可能带来效率低下的问题，但是这也是合理的，因为现代程序设计理念中就要求尽可能少用全局变量，违反这些的规范的程序员写出的代码势必不会有很好的效率。

基于上面的结论，我采用的寄存器分配方式为动态与静态相结合的方法。所谓动态分配，是指寄存器所对应的变量会随着程序的执行而改变，而静态分配则是指在一个函数运行的过程中，变量和寄存器的对应关系是固定不变的。

我采用的动态分配方式用于t寄存器的分配，静态分配方式用于s寄存器的分配。

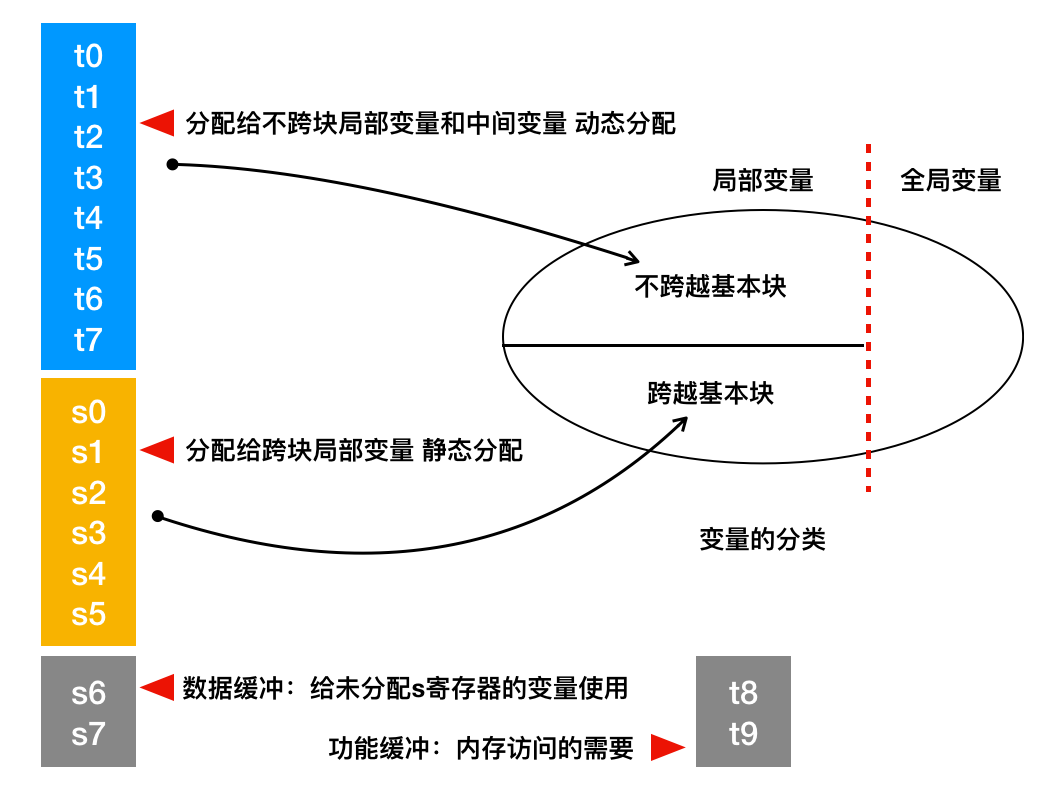


图 8 寄存器分配策略

9.2.2 t寄存器的动态分配

教材中对于t寄存器的分配介绍并不充分，只是提到了寄存器池的概念，但是并没有给出当寄存器用满时的取舍问题，其核心在于应该尽可能舍弃最不经常使用的寄存器，这就启发我联想到操作系统中的页面分配算法，他们所解决的问题有着一定的相似性。最优算法是可以实现的但是过于复杂，所以我采用了最近最久未使用算法（LRU）进行t寄存器的分配。

我把t0～t7寄存器分配给局部变量，t8～t9留作特殊功能使用（比如向内存中写入常数必须借助寄存器）。这8个寄存器构成一个栈，每次使用都取从栈底取寄存器，每次访问寄存器都把它放到栈的最顶部，这样栈底一定是最近最久未使用的变量，寄存器不够用的时候也会优先淘汰。

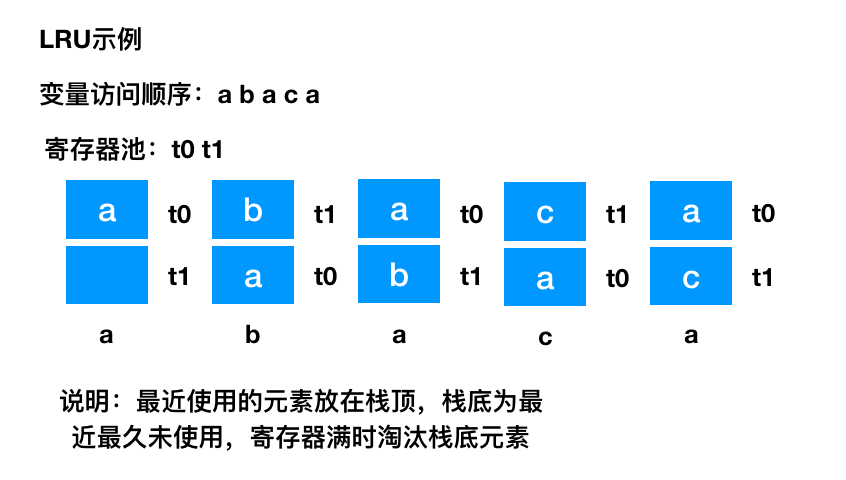


图 9 LRU示例

这种分配方式有很大的好处，和顺序静态分配相比，好处是很明显的，顺序分配并不能保证最频繁使用的变量获得寄存器，而循环则会导致劣势变得非常明显。

这样的分配方式会存在问题，考虑下面的情况。

程序运行到下面的中间代码时，因为是一个块的开始，所有的t寄存器都可用，假设在栈中是顺序排列的，那么会按照t7~t0的顺序分配。结合代码解释。

LABEL：

a = expr1 //a分配t7寄存器 因为是第一次使用所以不需要从内存中读取

b = expr1 //b分配t6寄存器 因为是第一次使用所以不需要从内存中读取

c = expr1 //c分配t5寄存器 因为是第一次使用所以不需要从内存中读取

d = expr1 //d分配t4寄存器 因为是第一次使用所以不需要从内存中读取

e = expr1 //e分配t3寄存器 因为是第一次使用所以不需要从内存中读取

f = expr1 //f分配t2寄存器 因为是第一次使用所以不需要从内存中读取

g = expr1 //g分配t1寄存器 因为是第一次使用所以不需要从内存中读取

h = expr1 //h分配t0寄存器 因为是第一次使用所以不需要从内存中读取

i = expr1 //i分配t7寄存器 使用前需要先把t7寄存器之前的值回写到a的内存空间

GOTO LABEL

这个基本块编译结束了，但是问题在于，变量a和i分配了同一个寄存器，而这段代码时循环执行的所以重新回到a = expr1这句话的时候，并没有任何把t7会写到i的内存之后把a加在到寄存器的操作，这会导致变量和寄存器之间的对应关系出现错误。这是寄存器别名问题。出现这个问题的原因是，编译的过程和代码的执行过程并不一致，所以编译时并不能发现可能存在的隐含问题。但是基本块已经保证了代码的执行顺在块内是一致的，为什么依然不能发现呢？原因在于从流图的角度看，这些变量会出现在这个块的in集合中，也就是说，并不能保证在进入这个块的时候此变量的状态的和第一次进入（这也是编译过程知道的状态）时是一样的。因此，严格来说，这类用户定义的变量时跨块变量。

针对上面的问题，我采用的解决方法是构建流图，进行分析。对于出现在in和out集合中的变量，都视为全局变量，为其分配s寄存器，采用冲突图着色（面临的问题会在后面详述）算法进行寄存器分配。in集合与out集合的定义与算法和教材上是一致的，唯一的区别在于in out def use四类集合都会去除掉用户定义的全局变量，因为它们不参与寄存器分配，写在集合中没有意义。

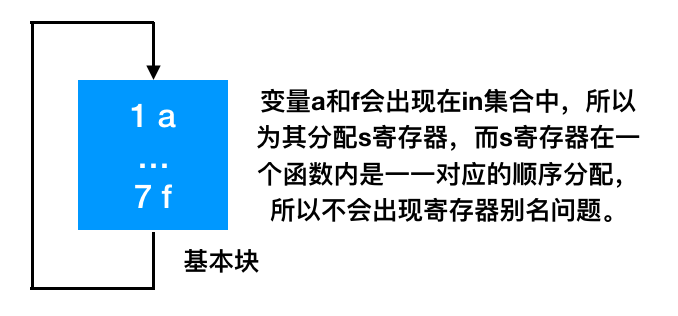


图 10 寄存器别名问题解决

至此，我的寄存器分配策略为：

不跨块的变量（用户定义的不跨块变量和中间变量）分配t寄存器

跨越基本块的变量分配s寄存器

跨越基本块的充分必要条件为：该变量出现在某个基本块的in集合中。

这样，我们就避免了寄存器别名问题，并且也能根据程序访问变量的特点来有效的分配临时寄存器了。

9.2.3 s寄存器的静态分配策略

为了让s寄存器能够有效的分配，决定采用冲突图着色的算法。

首先是冲突图的构建。按照教材的说法，两个变量冲突当且仅当它们在同一个基本块的入口处活跃。但是按照这样的说法，也就是可以把一个函数的每个基本块in集合的变量两两冲突，看上去就可以了。

但是考虑下面的情况。

x = a

y = b

显然x会出现在def集合里，所以不会出现在这个基本块的in集合中。但是可能出现在其他的基本块里，所以依然有可能分配s寄存器。而a和b出现在use集合里，所以一定会出现在这个基本块的in集合中分配s寄存器，如果仅仅要求in集合中的变量冲突，a和b是不会和x、y冲突的。这样的话，b可能和x分配了同一个s寄存器，这样就会出现错误。

这里之所以出问题，是因为变量可能出现在多个基本块，但并不总是出现在in集合里，所以，应该将in集合和def集合的元素也冲突，def和def集合的元素也要冲突。这样是合理的，因为如果这些变量出现在了后面的块里，它们还会被继续使用，分配同一个寄存器显然是不合理的。

这样，看上去我们的冲突图变的很强，好像不能起到高效分配寄存器的效果，但是因为不跨块的变量是不分配s寄存器的，所以其实最后我们用来着色的冲突图只是一个子图。

根据前面所述的冲突图建立方式，使用图着色算法，进行寄存器的分配，充分利用寄存器减少访问内存次数。首先是全局（相对基本块而言）寄存器算法如下：

**Input**：冲突图，全局寄存器数量K

**Output**：分配结果

**Algorithm**：

**while** 图中的点超过1个：

**while** 冲突图中存在度小于K的结点：

移走这个点

在图中选取一个适当的结点

标为不分配全局寄存器结点，移走。

对最后一个点任意分配一个寄存器

按照移走的顺序的逆序添加结点和边，保持不同色。

之后是临时寄存器分配算法，采用寄存器池的方法，这是一个基本算法，基本思路就是有就分配，没有就把即将生成的代码中不会使用的寄存器写会内存然后释放。

9.3 保存现场优化

9.3.1 保存现场策略综述

函数调用时需要保存现场，最简单的策略是把31个（0号除外）寄存器全部保存，但是这样每次函数调用会出现大量的访存指令，严重降低效率，所以采用的办法是个性化保存现场。因为寄存器的分配情况在编译的时候是知道的（即使是t寄存器采用动态分配策略，由于基本块中的编译顺序和执行顺序一致，所以依然是知道寄存器的分配情况的），因此可以只把使用的寄存器进行保存。这样寄存器的保存就仅仅局限在了s寄存器和t寄存器了，a寄存器仅仅是函数参数传递的媒介并不需要保存。

9.3.2 t寄存器保存现场

函数调用并不是一个基本块的划分依据，所以对于t寄存器仍然需要保存，如果能够对中间变量的生存期有着精准的认识，就可以及时的清理掉中间变量。这样一方面缓解t寄存器的压力，一方面也能在保存现场的时候少保存寄存器。

为此主要运用了中间变量的两处特征。

第一，如果赋值语句的左侧是一个用户定义的变量，那么这一定是一个语句的结束，中间变量可以全部去除。

第二，如果遇到一条函数调用指令，那么所有传参数使用的寄存器都可以删除。

如果这样的处理方法，增加了一个隐含的限制条件，也就是在做公共子表达式、复制传播的时候，不能跨越赋值语句和函数调用，否则可能会导致中间变量的生存期超过了预想的长度，不保存现场就会出现错误。

9.3.3 s寄存器保存现场

s寄存器使用量是比较大的，因为通常一个函数都会有5个左右的跨块变量，他们都要占据s寄存器，为此，保存现场的时候还可以进行一些特判减少保存现场开销。

第一，如果被调用的函数是一个叶子函数，由于它不会再调用其他函数，那么可以很清楚的知道未来会被使用的s寄存器有哪些，这样只需要保存那些父过程和被调用的子过程都是用的寄存器就可以了。

第二，这其实是第一种情况的特殊情况，如果这个函数是一个叶子函数而且不跨块，那么可以直接不保存s寄存器，这其实是一种类似内联展开的做法。

总之，上面的做法都极大降低了保存和恢复现场产生的访存指令。经过这样的优化以后，我的t寄存器几乎不需要保存，而s寄存器保存的数量也会降低。

9.4 a寄存器的使用

9.4.1 常规参数传递

如果直接把a固定在参数上，那么会造成的问题是增加了保存现场的负担。因为函数再调用其他就可能出现覆盖。所以我采用的办法是仅仅把a寄存器看作是一种媒介。如果这个参数分配了s寄存器，那么就把参数放到a上，进入到这个函数的时候，会第一时间从a上转移到应该对应的s寄存器上。如果这个参数没有分配s寄存器，那么就把它写到内存中，使用时在load出来。

9.4.2 针对特别函数的优化

上面的这种方式使得我对于那些参数没有分配s寄存器的函数对a的使用不充分。而且对于叶子函数，使用a寄存器之后其实没有必要转移到s寄存器上。因此针对叶子函数，我会直接指定它的前三个参数和a1、a2、a3对应，注意不能把a0分配参数，因为某些函数调用会用到a0，可能造成意料之外的错误。这样，就能把寄存器使用的更加充分了。

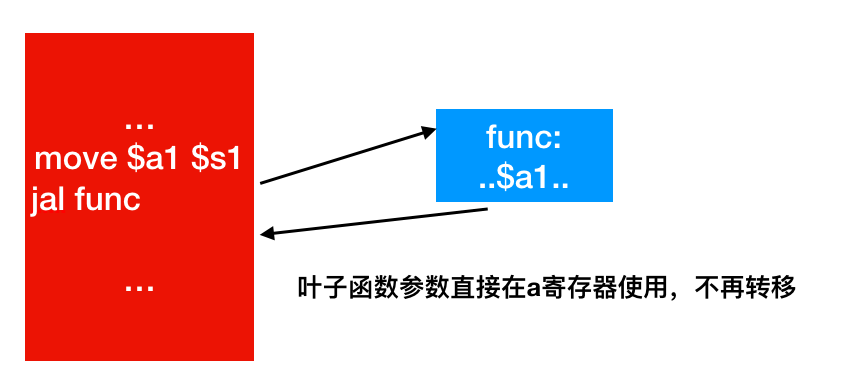


图 11 寄存器使用

上面的这些保存现场和参数传递的优化目的都是降低在发生函数调用时的访存开销，这些优化的效果也是立竿见影的，按照发布的测试样例来看，我的访存数量从一开始的一千五百万量级下降到一百五十万量级，减少了90%。

9.5 常量传播与复制传播

由于中间代码四元式的格式限制，最多只能有两个操作数，如果这两个操作数不都是常数，就会导致此项优化无法进行，所以真正做起来只能是表达式中出现连续的常量才能优化，者极大地限制了此项优化的效果。因此可以在生成中间代码之前进行预处理，对生成的表达式先做一些化简和移项一类的操作，提高常量合并优化的效率。

常量传播是一个很有效的优化手段，我也进行了传播，常量的传播不仅仅局限于替换，更重要的是它可能会让很多代数运算的两个操作数都变成常数，提前进行计算，进一步传播，这样会极大缩小运算量。

复制传播主要针对形如y=x的语句，在后面尽量使用x替代y，这需要满足以下条件：

所有到达考察点的路径上x已经求值，且最近一次到达没有被重新定义。如果最后全部都能够替换那么可以删除y=x这条语句。当然这样的优化效果极为有限，而且很麻烦，所以酌情考虑。

复制传播则有可能减少赋值语句，不过要注意，前面已经对中间变量的生存期有了严格的控制，所以如果赋值的来源是一个中间变量那么传播的范围也是有限的。

9.6 公共子表达式优化

9.6.1 教材方法

构建删除了公共子表达式的DAG图算法如下：

**Input**：中间代码序列

**Output**：删除局部公共子表达式的DAG图

**Algorithm**：

**Initial** table T

**for** each instruction in codes：

**if** z = x op y：

**search** x in T

**if** not found：

create a node i and **add** the entry into T

**search** y in T

**if** not found：

**create** a node j and **add** the entry into T

search op in T

**if** not found：

**create** a node k and **connect** i，k and j，k

**search** z in T

**if** found：

**update** z with k

**else**：

**add** z，k in T

这个算法和教材上一样的，此外，教材还指出此算法在面临数组时可能不正确，但是我们为数组存取构建了专门的四元式表示，所以没有问题。

根据前面生成的DAG图，可以生成公共子表达式的消除后的中间代码，采用启发式算法。算法如下：

**Input**：DAG

**Output**：middle code

**Algorithm**：

**Initial** a queue Q

**while** there still exists middle node not in queue：

**add** one of these middle node n whose parent node has in queue into Q

**while** n.left is one of those nodes:

**add** into Q

**output** the Q from end to begin

**generate** new middle code based on this order

此算法是一个启发式算法，而且基于X86架构，对于我们生成的MIPS，还需要根据实际情况进行调整。

9.6.2 公共子表达式的实现

公共子表达式的消除公认的方法是DAG图，但是这里面会有很多复杂的问题，而且最关键的是这种优化带来的好处其实并不太大，所以我采用了暴力扫描的方法进行解决。

其基本思想是，对于一条运算或者取内存元素的指令，我会检查后面是否有操作数和它一样并且没有被改变的指令，如果有，那么将其变成一条赋值语句，之后在使用复制传播进行优化。这样也会有不错的效果，实现起来也还比较简单。

9.7 窥孔优化

这是一种比较简单的优化，旨在消除一些在逻辑上没有意义的代码，主要针对一下几种情况。

第一种，无条件跳转到下一条指令，比如形如j label lable这样的指令是无意义的。

第二种，运算无意义指令，比如add a a 0 mult a 1这样的指令，没有对变量值做出改动，没有意义。

第三种，存取和赋值类指令，比如把一个数组元素紧接着赋值给自己这样的指令也是徒劳的。

第四种，有很多时候同一个位置就出现多个标签，这样地方进行合并，连续的两LABEL语句合并成一条，并把所有的分支跳转到这个位置的指令进行对应的修改，如果发现LABEL和GOTO可以相邻那么可以删除GOTO。

9.8 目标代码优化

目标代码中指令选择优化有以下几类。

第一类：关于运算类的优化。如果遇到常数的运算指令，那么可以把数字提前算好使用加载立即数指令实现。

第二类：关于访存类的优化，如果遇到连续的sw指令，并且访问的地址是一样的，那么只有最后一条指令是有效的，前面的可以删除。

第三类：关于系统调用的优化。系统调用时很费时间的，所以应该尽量减少系统调用次数。考虑到输出的美观，我约定了printf语句的输出最后会有换行符。这样的话如果输出了一个字符串，还要再为换行符专门进行系统调用并不合算，所以可以在编译时对于只输出字符串的printf语句里的字符串最后追加换行符，减少系统调用。 如果出现了两个连续的同类输出，那么v0寄存器的值是不需要再次赋值的。

### 10. 出错处理

10.1 错误处理流程

整个工程有一个专门的错误处理模块（error.cpp和error.h），这个模块中，主要的函数有两个，分别是报错信息的输出函数（errmsg）和错误局部化处理的函数（skip）。当词法或文法分析发现了错误时，会调用errormsg，errmsg接收当前的位置和报错信息输出。之后skip函数会跳过部分字符或单词，实现错误的局部化。同时为保持程序可读性，不对错误进行编号，程序中直接使用字符串常量进行表示和传参。

10.2 错误类型

词法错误

|  |  |
| --- | --- |
| 错误类型 | 输出信息 |
| 不能识别的字符 | Unrecognizable character |
| 期待一个单引号 | Expect a single quote mark |
| 期待一个‘=’符号 | Expect a ‘=’ sign |
| 有前导零 | Leading zero |
| 数字大小超出限制 | Number too big |
| 致命错误 字符串超出字符串表的限制 | Fatal too many string |
| 标识符长度超出限制 | Identifier too long |

语法错误

|  |  |
| --- | --- |
| 错误类型 | 输出信息 |
| 期待无符号整数 | Expect a unsigned integer |
| 期待字符 | Expect a character |
| 期待加法运算符 | Unrecognizable character |
| 期待标识符 | Expect a identifier |
| 期待数字 | Expect a number |
| 期待分号 | Expect a semicolon |
| 期待‘（’ | Expect a left parent |
| 期待‘）’ | Expect a right parent |
| 期待‘[’ | Expect a left bracket |
| 期待‘]’ | Expect a right bracket |
| 期待‘{’ | Expect a left brace |
| 期待‘}’ | Expect a right brace |
| 期待类型标识符 | Expect a ‘int’/‘char’ |
| 期待while | Expect a ‘while’ |

语义错误

|  |  |
| --- | --- |
| 错误类型 | 输出信息 |
| 除以0错误 | Divize zero |
| 标识符重复定义 | Unrecognizable character |
| 标识符未定义 | Undeclared identifier |
| 参数个数不一致 | Parameters number not match |
| 参数类型不一致 | Parameters type not match |
| 表达式类型不一致 | Expression type not match |
| 空类型参与运算 | void type in expression |
| 未指明返回值 | Function does not have a return value |
| 空类型函数有返回值 | Void function has a return value |
| 返回值类型不一致 | Return value not match |

## 三．操作说明

### 1．运行环境

实现代码：C++（C++11标准）

编写与编译环境：Visual Studio 2017 Community

运行环境：Windows7 Professional

### 2．操作步骤

运行compiler.exe程序，弹出控制台提示输入待编译的文件路径。

之后，输入路径后回车，如果有错误，会在控制台输出出错的位置和错误信息。如果没有错误，则会把mips汇编代码输出到当前路径（compiler.exe的路径）下的result.txt文件中。优化前中间代码在midcode.txt中，优化后中间代码在new\_midcode.txt中。

## 四．测试报告

### 1．测试程序及测试结果

1.1 正确测试1

意图：测试递归，使用了全排列和斐波那契数进行测试。

代码：

﻿char chs[4];

int fib(int n){

if(n==1)

return(1);

else if(n==2)

return(1);

return(fib(n-1)+fib(n-2));

}

void permutation(int m, int n){

int i;

char t;

char r;

if (m < n-1) {

permutation(m+1, n);

for (i = m+1;i < n;i = i+1) {

t = chs[m];

chs[m] = chs[i];

chs[i] = t;

permutation(m+1, n);

t = chs[m];

chs[m] = chs[i];

chs[i] = t;

}

}

else{

i = 0;

do{

r = chs[i];

printf(r);

i = i + 1;

}while(i<4)

printf(" ");

}

}

void main() {

int i;

printf("please input a num between 1~20");

scanf(i);

printf(fib(i));

chs[0] = 'A';

chs[1] = 'B';

chs[2] = 'C';

chs[3] = 'D';

printf("this is a permutation of ABCD: ");

permutation(0,4);

return;

}

结果：

please input a num between 1~202

1this is a permutation of ABCD: ABCD ABDC ACBD ACDB ADCB ADBC BACD BADC BCAD BCDA BDCA BDAC CBAD CBDA CABD CADB CDAB CDBA DBCA DBAC DCBA DCAB DACB DABC

1.2 正确测试2

意图：测试内存访问的正确性，使用矩阵相乘进行测试。

代码：

void main() {

int a[9];

int b[15], c[15];

int i, j, k;

b[0\*5+0] = 1;

b[0\*5+1] = 7;

b[0\*5+2] = 4;

b[0\*5+3] = 5;

b[0\*5+4] = 1;

b[1\*5+0] = 3;

b[1\*5+1] = 2;

b[1\*5+2] = 1;

b[1\*5+3] = 2;

b[1\*5+4] = 1;

b[2\*5+0] = 1;

b[2\*5+1] = 5;

b[2\*5+2] = 0;

b[2\*5+3] = 2;

b[2\*5+4] = 1;

c[0\*3+0] = 19;

c[0\*3+1] = 7;

c[0\*3+2] = 2;

c[1\*3+0] = 8;

c[1\*3+1] = 14;

c[1\*3+2] = 4;

c[2\*3+0] = 6;

c[2\*3+1] = 5;

c[2\*3+2] = 3;

c[3\*3+0] = 0;

c[3\*3+1] = 3;

c[3\*3+2] = 6;

c[4\*3+0] = 7;

c[4\*3+1] = 13;

c[4\*3+2] = 2;

for(i=0;i<3;i=i+1){

for(j=0;j<3;j=j+1){

a[i\*3+j] = 0;

for(k=0;k<5;k=k+1)

a[i\*3+j] = a[i\*3+j] + b[i\*5+k]\*c[k\*3+j];

printf(a[i\*3+j]);

printf(" ");

}

printf("|");

}

}

结果：106 153 74|86 73 31|66 96 36|

1.3 正确测试3

意图：测试寄存器分配的正确性，在一个循环和一个基本块内加入超过寄存器数量的变量进行访问。

代码：

void oneBlock(int a){

int t0, t1, t2, t3, t4, t5, t6, t7, t8, t9;

t0 = 12+5+8\*3-2+a;

t1 = t0-a\*a;

t2 = t1+7;

t3 = t0\*t1\*t2;

t4 = 32\*t3;

t5 = t4/3;

t6 = 5+3;

t7 = 12+4;

t8 = t7+2;

t9 = t0+t1;

printf(" ", t0);

printf(" ", t1);

printf(" ", t2);

printf(" ", t3);

printf(" ", t4);

printf(" ", t5);

printf(" ", t6);

printf(" ", t7);

printf(" ", t8);

printf(" ", t9);

}

void main(int argc, char\* argv[]){

int i;

i = 0;

int v0, v1, v2, v3, v4, v5, v6, v7, v8;

do{

v0 = i+1;

v1 = i\*5;

v2 = i/3;

v3 = i-6;

v4 = i\*i;

v5 = i+i\*i;

v6 = 6+v5;

v7 = v1+v0;

v8 = v1+v0;

printf("|", v0);

printf("|", v1);

printf("|", v2);

printf("|", v3);

printf("|", v4);

printf("|", v5);

printf("|", v6);

printf("|", v7);

printf("|", v8);

i= i +1;

}while(i<10);

oneBlock(v8);

}

结果：

|1|0|0|-6|0|0|6|1|1|2|5|0|-5|1|2|8|7|7|3|10|0|-4|4|6|12|13|13|4|15|1|-3|9|12|18|19|19|5|20|1|-2|16|20|26|25|25|6|25|1|-1|25|30|36|31|31|7|30|2|0|36|42|48|37|37|8|35|2|1|49|56|62|43|43|9|40|2|2|64|72|78|49|49|10|45|3|3|81|90|96|55|55 94 -2931 -2924 805602936 9490176 3163392 8 16 18 -2837

1.4 正确测试4

意图：测试a寄存器的使用，通过函数有大量参数来测试。

代码：

﻿int lots\_para\_oneBlock(int p1, int p2, int p3, int p4, int p5, int p6, int p7, int p8, int p9, int p10){

printf(p1);

printf(" ", p5);

printf(" ", p7);

printf(" ", p10);

printf(" ", p1+p2+p3+p4+p5+p6+p7+p8+p9+p10);

printf(" |");

return (p1+p2+p3+p4+p5+p6+p7+p8+p9+p10);

}

int lots\_para\_blocks(int p1, int p2, int p3, int p4, int p5, int p6, int p7, int p8, int p9, int p10){

int i;

i = 0;

do{

printf(" ", p1);

printf(" ", p5);

printf(" ", p7);

printf(" ", p10);

printf("", p1+p2+p3+p4+p5+p6+p7+p8+p9+p10);

printf(" |");

i = i+1;

}while(i<3);

return (p1+p2+p3+p4+p5+p6+p7+p8+p9+p10);

}

void main(){

printf(lots\_para\_oneBlock(1, 2, 4, 7, 16, 21, 25, 31, 37, 43)+lots\_para\_blocks(1, 2, 4, 7, 16, 21, 25, 31, 37, 43));

}

结果：

1 16 25 43 187 |1 16 25 43 187 |1 16 25 43 187 |1 16 25 43 187 |374

1.5 正确测试5

意图：测试复杂表达式和超级深度的函数嵌套调用。

代码：

﻿int foo2(){

return (2);

}

int foo3(int a){

return (a+3);

}

int call1(int a){

return (a+1);

}

int call2(int a){

return (a+2);

}

int call3(int a){

return (a+3);

}

int call4(int a){

return (a+4);

}

int call5(int a){

return (a+5);

}

int super\_expr(int a, int b){

return (12+(222222222+4+('a'))/10086\*100/(foo2())+(a+b)+(a+b)+(a+b)+foo3(a+b)+(a+b)+call5(call4(call3(call2(call1(5))))));

}

void main() {

printf(super\_expr(12, 30));

}

结果：

1101845

1.6 错误测试1

意图：测试变量声明错误、期待分号、数组长度错误、非法字符、重名错误、区分大小写不报错、数字开头非法的变量名。

代码：

void main(){

const int g\_c\_int = 1, char g\_c\_char = 'c';

const int leading\_zero = 001;

const char no\_qmark = a;

const g\_c\_chare = 'e';

const char g\_c\_char\_notsurpport = '?'

const char c;

int g\_v\_int\_b = 1;

char g\_v\_char\_arr[10], int g\_v\_int\_d;

int illegal\_len[-12];

int illegal?, \_1, \_\_\_\_\_\_, 0inhg;;

int o;

char o, O;

}

结果：报错声明错误、期待分号、数组长度错误、非法字符、重名错误、区分大小写不报错、数字开头非法的变量名。

1.7 错误测试2

意图：测试缺少括号、缺少类型、缺少返回值、语句位置错误。

代码：

﻿char noparalist{

return('a');

}

int lesstype(int a, b){

return(a);

}

int lessreturn(int a){

a = a\*2;

}

void main(){

printf("in");

int displaced;

return;

}

结果：报出缺少括号、缺少类型、缺少返回值、语句位置错误。

1.8 错误测试3

意图：测试数组位置错误、未定义标识符、缺少分号

代码：

void main(){

const int len = 10;

int arr[10];

int i;

int user\_input;

int res\_of\_expr;

for(i = 0; i < len; i = i + 1){

arr[i] = i;

}

arr[150] = 10;

arr[+1] = 10;

undef = 1;

printf("input the number (1~10)");

scanf(user\_input);

if(user\_input>=len)

printf("too big")

else{

i = 0;

do{

printf(" ", arr[i]);

i = i + 1;

}while(i<=user\_input);

}

}

结果：报出数组位置错误、未定义标识符、缺少分号

1.9 错误测试4

意图：测试输入输出的问题，不能直接读到数组，数组赋值缺少下标，函数调用缺少括号，缺少双引号和不支持的字符。

代码：

﻿ ﻿void test\_in(){

printf("in");

}

void main(){

int arr[10];

arr = 1;

test\_in;

scanf(arr[0], arr[1])

printf('cant use quote');

printf(need a quote);

printf("c\*12");

}

结果：报出不能直接读到数组，数组赋值缺少下标，函数调用缺少括号，缺少双引号和不支持的字符的错误。

1.10 错误测试5

意图：测试循环中的错误。缺少对i的赋值，步长不正确，缺少条件

代码：

﻿void main(){

int i;

i = 0;

for(;i<5;i=i+1){

printf(i);

}

for(i=0;i<5;i=i+'c'){

printf(i);

}

for(i=0;;i=i+1){

printf(i);

}

}

结果：报出缺少对i的赋值，步长不正确，缺少条件的错误。

### 2．测试结果分析

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | test1 | test2 | test3 | test4 | test5 | test6 | test7 | test8 | test9 | test10 |
| 递归 | √ |  |  |  |  |  |  |  |  |  |
| 访存 |  | √ |  |  |  |  |  |  |  |  |
| 表达式 |  |  | √ |  |  |  |  |  |  |  |
| 传参 |  |  |  | √ |  |  |  |  |  |  |
| 局部多变量 |  |  |  |  | √ |  |  |  |  |  |
| 声明错误 |  |  |  |  |  | √ |  |  |  |  |
| 缺失分号 |  |  |  |  |  | √ |  |  |  |  |
| 数组长度定义错误 |  |  |  |  |  | √ |  |  |  |  |
| 非法字符 |  |  |  |  |  | √ |  |  |  |  |
| 重名错 |  |  |  |  |  | √ |  |  |  |  |
| 标示符错误 |  |  |  |  |  | √ |  |  |  |  |
| 括号缺失 |  |  |  |  |  |  | √ |  |  |  |
| 类型缺失 |  |  |  |  |  |  | √ |  |  |  |
| 返回值缺失 |  |  |  |  |  |  | √ |  |  |  |
| 语句位置错 |  |  |  |  |  |  | √ |  |  |  |
| 数组索引错误 |  |  |  |  |  |  |  | √ |  |  |
| 未定义标识符 |  |  |  |  |  |  |  | √ |  |  |
| 不能直接读数组 |  |  |  |  |  |  |  | √ | √ |  |
| 缺失下标 |  |  |  |  |  |  |  | √ | √ |  |
| 缺失赋值 |  |  |  |  |  |  |  |  |  | √ |
| 步长不正确 |  |  |  |  |  |  |  |  |  | √ |
| 缺失条件 |  |  |  |  |  |  |  |  |  | √ |

## 五．总结感想

完成对C0文法的编译器后，还是很有成就感的，他的工程量很大，而且是一个确实有一定应用价值的东西，设计的过程中结合了理论与实践，收获很多。

我在这个过程中是具有探索和冒险精神的。在和同学的讨论过程中，我们发现寄存器池中寄存器的分配和内存管理中的页面分配很类似，于是将其中兼具了实现难度和性能的最近最久未使用算法（LRU）移植过来。这样做是很冒险的，因为并没有什么主流的用法这样做，所以正确性是存在隐患的。果然，在后来我发现了两个很致命的问题。

第一个问题是寄存器别名问题，就是循环结构会导致代码执行的顺序和编译顺序不一致，所以可能循环回来再次执行的时候，寄存器和变量之间的对应关系发生改变造成错误。经过和同学的讨论，我们提出了使用活跃变量分析解决方案，计算每个基本块的in集合，检查冲突情况，为其分配全局寄存器，实现寄存器和变量之间的一一对应。

之后第二个问题则是在实现冲突图着色算法时的问题，但看in集合并不能保证冲突图的正确性，但是其他变量未必跨块又不一定会为其分配全局寄存器。其实这个问题一直存在，而且理论课的说明也并不详细，为了保证其正确性我进行了大量的论证，实现了一个正确且较为有效的冲突图着色算法。

总之，编译技术课程设计给我的收获是很大的，我觉得敢于创新探索，遇到问题积极解决，也是未来在科研道路上必备的素质。

1. 本部分和郑明悟（16231019）同学讨论完成。 [↑](#footnote-ref-1)