<<编译技术>>课程设计申优文档

17377416　乐洋

1. 设计中的困难及解决方案

1.1 词法分析

本人设计的是一个LexicalAnalyzer类，对外的接口是analyze和show函数。其中analyze函数对指定的输入流进行词法分析，并把分析结果保存在对象内部属性symList，symTypeList和symRowList中。调用show函数把保存在内部属性的分析结果输出到输出流中。

而进行analyze的内部核心函数是nextChar和nextSym。 nextChar读取一个字符并将放入对象属性chrCurr中；nextSym读取一个单词。

再根据给定的文法画出单词符号的状态图，即可以编码nextSym。

回退问题

在读取的charCurr为'='时，读取第二个字符如果charCurr不是'='，则需要将charCurr回退。还有一些其他情况也需要回退。但因为本人采用的是直接从istream流中读取字符，不方便将已经读取的字符重新放入istream中（其实也可以用seekg和tellg重定位）。这里采用一个简单的办法，**设置一个回退标志needRetract。当需要回退时，将needRetract设为1。调用nextChar时，发现needREtract为True，则不读取新字符，保持CharCurr不变。**

输入的结束标志

一开始本打算用fin强制转化为bool型表示是否到达文件结尾。但是会导致nextSym的return条件比较奇怪。

于是直接用chr = fin.get(), 将EOF读取到chr中，如果chr是EOF，自然无法进入nextSym的任何一个分支，自然就return了。

```
void LexicalAnalyzer::analyzeLexis() {
    nextChar();  // 预读一个字符
    while (fin) {  // cin.get无法读取EOF到chrCurr，但是     bool（cin）可以表示是否能继续读取
        nextSym();
        nextChar();
    }
}
```

1.2 语法分析

语法分析程序的框架采用是递归下降方式进行设 计。为每一个语法成分设计对应语法解析函数，函数直接相互调用，构成整个语法分析程序。这样以语法成分划分函数模块，方便后续的符号表构建，错误处理和语法制导生成中间代码。

FIRST集交集不为空

采用预读方法判断进入哪一个非终结符。（对于某些情况，如变量定义和有返回值的函数声明，需要预读多个才能判断。）

建议增加了curr_sym_type, peek_sym_type, equal, check和error等支持函数简化代码编写。

词法列表的指针移动

为了保证代码编写的正确性和一致性，对每个非终结符的语法分析函数做了如下规定。

- 保证进入每个分程序（指各个非终结符的语法分析函数）时，sym_idx_指向分程序对应的第一个sym
- 在退出分程序前，将sym_idx指向分程序对应语法成分后的第一个sym
- 不论在进入分程序前有没有进行过sym类型检查，在进入分程序后都要对对应的sym进行类型检查。

1.3 符号表

错误处理

按照教材上的说法，错误处理和符号表这两个部分，是贯穿于整个编译过程的，这么看这两个部分应该处于“平等的”地位。那为什么笔者要把符号表放在错误处理下面呢？**原因是笔者认为做好错误处理的关键就在于合理的设计符号表。**

我们知道编译错误分为语法错误和语义错误。语法错误直接由递归下降子程序就能得出。而发现语义错误（如表达式类型不匹配，初始化元素个数不匹配等）的关键就在于将当前信息和之前已经存储到符号表中的信息进行对比。

因此设计符号表的一个重点是如何存储我们需要的信息。为了便于后续的扩展，笔者采用了类和继承的方式。**父类TableEntry有五个子类: ConstENtry, VarENtry, FUnctionEntry, LabelEntry, ImmediateEntry(立即数)**

```
class TableEntry {
private:
    EntryType entry_type_;
    ValueType entry_value_type_;
    string identifier_;
};

class ConstEntry : public TableEntry {};

class VarEntry : public TableEntry {
private:
    vector<int> shape_;
};

class FunctionEntry : public TableEntry
private:
    vector<ValueType> formal_param_list_;
};
```

定位和重定位

定位和重定位操作的实现非常简单，因为按照现有文法，只有两个作用域，全局作用域和函数作用域。**当进入函数**，将curr_block_head指向符号栈栈顶（表示当前局部作用域）；**当离开函数，**将curr_block_head设为符号栈栈底（全局变量在栈底，表示全局作用域）.

1.4 中间代码生成

中间代码的设计问题

数组取值和复制

课程组给出的数组取值或者赋值的中间代码如下:

```
# 源码形如：
a[i] = b * c[j]

# 中间代码：
t1 = c[j]
t2 = b * t1
a[i] = t2
```

问题是我们的文法中不仅有一维数组，还有二维数组。对于二维数组，数组变量，两个索引，操作符，结果，一共有五个元，显然说明不能用一个四元式表示二维数组的取值或者赋值。因此将数组的取值或者赋值拆分为两步:

```
#源代码
t = c[i][j]
# 中间代码
pushArrayIndex i
pushArrayIndex j
t = GetArrayElem(c)

#源代码
a[i][j] = b
# 中间代码
pushArrayIndex i
pushArrayIndex j
a = setArrayElem(n)
```

变量和常量声明

变量声明可能有初始化，也可能没有初始化，因此变量声明四元式的opA可能是一个立即数，也可能是None。而常量声明一定有右值。

printf

prinf有三种形式如下

```
printf("str")    # 输出字符串
printf(expr)     # 输出表达式的值
printf("str", expr)  # 输出字符串+表达式的值
```

可以用一种四元式同一表示， opA表示str， opB表示expr，没有的项置Null.

元的位置问题

在最初设计四元式时，当四元式的实际的元的数目小于3(比如变量声明，设置标签等语法)，我并没有精心设计实际的元应该与result域还是opA或者opB域进行对应。

```
# 比如对于变量声明 int a;
# 如果四元式的四个域从左往右依次为 （操作符，result，opA，opB），那么有两种表示
(VarDeclare, a, Null, Null)
(VarDecalre, Null, a, Null)
```

但这样会给代码编写工作带来混来，后续的优化还有Mips生成时经常搞不清某个域的实际含义。

因此我对中间代码设计进行了修改，**凡是具有取值含义的 (如FuncParamPush的param, FUncCall的func)，全部设为opA.**

凡是具有被赋值含义或者作为某个动作的结果含义的(如setLabel的label， scanf(a)的a， 函数形式参数声明的param)。全部设为opB、

最终设计出来的四元式如下（部分）

四元式的具体细节如下:

操作	result	opA	opB	含义
pushArrayIndex	index			
getArrayIndex	var	array		var = array[i][j]
setArrayIndex	array	var		array[i][j] = var
VarDecalre	var	inum(立即数)/None		变量声明
ConstDecalre	const	inum		常量声明
printf		str / None	expr/None	
BNE	label	expr1	expr2	Goto label if expr1 != expr2
BEQ	label	expr1	expr2	Goto label if expr1 == expr2
BLT	label	expr1	expr2	Goto label if expr1 < expr2

操作	result	opA	opB	含义
BLE	label	expr1	expr2	Goto label if expr1 <= expr2
BGT	label	expr1	expr2	Goto label if expr1 > expr2
BGE	label	expr1	expr2	Goto label if expr1 >= expr2

1.5 MIPS目标代码生成

全局变量的load与save对应指令数过多的问题

开始的内存布局

常量和立即数	采用硬编码
全局变量和String	存储在.data区
局部变量和临时变量	存储在栈中

笔者发现加载和存储.data段数据都需要三条指令。

```
# 原MIPS
.data
a: .space 1

.text
lw $s0, a($zero)

# 经MARS展开后的MIPS
lui $1, 0x0000 1001
addu $1, $1, $0
lw $16, 0($1)
```

查阅资料发现，**可以使用$gp寄存器来减少部分全局变量的load,save指令条数。**具体做法如下

- 数组类型的全局变量存储在.data段，这一部分变量加载存储还是三条指令
- 小变量存储在$gp指向的地址，这一部分变量加载存储变为一条指令

至于为什么要分开存储? 因为$sp + 16位offset能访问到的地址有限（这一部分地址才能做到一条指令访问），如果大数组也存储在这里，很容易就超出了这个范围，减少了其他变量被一条指令就能访问到的机会。

fp寄存器的使用

实际上只使用sp就足够为函数分配和管理栈空间了。但是为了方便管理，还是让fp指向帧顶，sp指向帧低。

```
    栈布局:
        a,v,t   (如果要求函数调用前后保持一致，则由调用方保护,simple情况无需考虑这一点)
        |
        func param
        |
    -------------------------<<<<<<  fp:当前函数的帧开始的地方
        |
        ra
```

```
          |
          prev fp
          |
          s0-s7  (如果子函数使用，则必须保护,simple情况无需
考虑这一点)
          |
          局部变量
          |
          临时变量
          |
     ---------------------------<<<<<  sp:当前函数的帧结
束的地方
```

具体实现在每个函数的开头和结尾都加上如下代码:

```
# 开头
sw $ra, -4($sp)    # 保护ra寄存器
sw $fp, -8($sp)    # 保存上一个函数的帧顶
move $fp, $sp      # 将上一个函数的帧底作为这个函数的帧顶

# 结尾
lw $ra, -4($fp)    # 恢复ra
move $sp, $fp      # 恢复上一个函数的帧低
lw $fp, -8($fp)    #   恢复上一个函数的帧顶
```

2. 优化

2.1 常数合并和传播

第一种情况是opA和opB都是常数，直接合并. 如t0 = 1 + 2, t1 = t0 + a直接化简为t1 = 3 + a.

第二种情况是加减乘0, 或者乘除1，也可以直接化简。

```
t1 = t0 + 0
t1 = t0 - 0
t1 = t0 * 0
t1 = t0 * 1
t1 = t0 / 1
```

第三种情况比较复杂。想象一下我们有一个原始表达式， b = 1 + a - 2

对应的没有优化的中间代码是

```
t0 = 1 + a
b = t0 - 2
```

其实可以化简为b = -1 + a. 那么如何做到呢，对于当前四元式，如果opA，opB中只有一个是立即数，那么就**看一下上一个四元式**，如果上一个四元式也有立即数，就可以尝试进行合并。不过这里要注意只有加法和乘法有交换律，减法和乘法不能交换，还有运算符的优先级问题。可以对可化简的情况进行打表，这样不容易出错。

2.2 函数内联

函数内联是一个非常重要的优化，所以要尽量把能够内联的函数全部内联。

能内联的函数

那么有哪些函数是不能内联的呢？一是大函数，可以设置一个阈值，超过了就不内联。

二是直接或间接递归调用的函数。直接递归非常好判断。一般来说，判断是否间接递归，要首先建立以函数为结点的调用图，再通过DFS判断是否有环。但在这里，因为只能调用前面已经定义过的函数，所以不可能出现间接递归的情况。

嵌套调用的情况

如果函数f1调用f2,f2调用f3,最理想的内联情况是最后只剩f1。 做到这一点只需要设置一个新的Function_list，把f1内联到f2后，就把内联后的f2加入新的Function_list, 然后判断f2能内联到f3，就把新的Function_list中的f2拿去内联。

实现细节

- callee函数的参数和变量都变为caller的局部变量
- 每次进入被内联的代码区时，都要为callee的参数和有初始化的变量（现在它们都是caller的局部变量）赋值
- 更换重名的变量，可以直接在后面加一个后缀“_funcname”
- 去掉callee的函数头；对于有返回值的四元式，ReturnAssign语句变为普通Assign语句

2.3 窥孔优化

跳转语句

如果一个跳转语句的下一句就是它的标签，就可以删去这个跳转语句。需要注意的是便签语句不能删除，因为还有其他跳转语句也可能跳转到这个标签，而且标签也不增加MIps指令数。

冗余的临时变量

如果一个临时变量的功能只是单纯的作为缓存或者中转站，那么可以把它删除。如a = b + 1会产生如下的中间代码。就可以把t0删除，合并为一条四元式。

```
t0 = b + 1
a = t0
```

另外要注意的是，如果下语句是标签语句，要继续向下寻找，直到不是标签语句。如下面这种情况也可以合并。

```
t0 = b + 1
label1:
label2:
a = t0
```

数组的偏移量计算

其实这儿可以算常数合并，但由于我的中间代码没有用乘法计算偏移量这一步（而是直接采用ArrayIndexPush做了一个标记），所以这里单独提出来。对于二维数组，偏移量 $offset = index1 * c1 + index2 * c2$. c1,c2是两个常数，如果index1和index2中至少有一个常数，就可以提前计算一部分。

2.4 临时寄存器分配

LRU算法

临时寄存器分配采用**LRU算法**，对于请求临时寄存器的变量（一般是临时变量）：

1. 如果存在某个寄存器保存的就是该变量的值，返回寄存器编号，寄存器移至队首
2. 如果不存在，且有空余寄存器，取一个空余寄存器， load该变量，寄存器移至队首
3. 如果不存在空闲寄存器，取出队尾寄存器，**先将寄存器内的值写回内存**，再load新变量，移至队首

需要注意的是，进入新的基本块时，要把所有临时寄存器里的值清空。对于活跃的变量，要回写到内存中。要**注意回写的时机**。对于以跳转类指令结束的基本块，回写发生在跳转指令的前面（不然可能跳转走了就不能回写了；而且跳转指令只读不写，不影响寄存器里的值）；对于以其他类型指令结尾的基本块，回写发生在整个基本块的最后面。

buffer寄存器

只拿出$t0-$t7做为临时寄存器去分配给变量。留出$8和$t9作为缓存寄存器。缓存寄存器的作用是存储一些没有变量对应的运算结果的中间值。如计算数组偏移量时会用到。

基本块的划分，流图的建立和活跃变量分析

关于判断变量在基本块后是否仍然活跃，当然就是划分基本块，建立流图做活跃变量分析。

划分基本块就对四元式序列扫描一遍，找到所有的入口。然后入口作为隔断，就得到各个基本块。

但要注意，建立流图有一个坑点。**比如说无条件跳转语句，它的下一句和能跳转到的语句都是入口，但是考虑到实际情况，它的后继块只有能跳转到的语句，没有下一句。**Return语句也是类似的情况。

活跃变量分析按照课本做就行了，比较简单。

2.5 全局寄存器优化

做好了活跃变量分析后，可以选择已经冲突图，采用图着色算法分配全局寄存器。但由于时间比较紧，目前只做了比较简单的引用计数法，接下来一个星期再补充图着色算法。

引用计数法中，对于循环体内的变量加大权重，增加其分配到寄存器的可能性。

需要注意的是:

- 函数调用时要save现场（s寄存器和t寄存器里面只要有变量，都要save）
- 函数结束时，s寄存器里面的变量要清空

全局变量的寄存器分配

对于全局变量，书上说不能给它分配全局寄存器，理由如下。

> 处于线程安全的考虑，全局变量和静态变量一般不参与全局寄存器的分配，这是因为当一个全局变量或静态变量的值保存于寄存器中时，如果发生线程切换，当前的寄存器状态将作为线程现场被保存，切入线程将恢复其此前保存的寄存器状态，这就导致了其他线程无法得到该寄存器在此前线程中的值，程序运行可能发生不可预知的错误。

但是因为我们的C0文法不支持多线程，因此实际上可以给全局变量分配全局寄存器（虽然这样做不利于后续拓展到多线程）。

2.6 指令选择

立即数乘法指令

对立即数乘法指令，如果立即数是正数且是2的幂，那么可以改成左移sll。如果是负数且绝对值是2的幂，可以改为左移和取反（减法）。

除法指令

div result, opA, opB指令会被扩展成除0检查的相关指令，在我们的C0文法里面不进行除0检查。因此可以换成div opA, opB 和 mflo.

对立即数除法x/y，如果立即数绝对值是2的幂，不能直接右移，因为有取整的问题，要分x是正数和负数讨论，应该采用以下公式。

$$x/y = (x < 0 ? x + (1 << k) - 1 : x) >> k$$

其中x,y是整数且除法是取整除法， k = log2 (y)

立即数减法指令

subiu是扩展指令，实际对应两条指令（addi和sub),为了减少指令条数，改为addiu立即数的相反数。

2.7 循环优化

按照一般的写法，for循环和while循环都是两条跳转指令，虽然我们的C0文法没有要求实现do while，但是容易知道do while可以一条跳转指令实现。我们可以借鉴do while的结构，把for和while的跳转指令数降低到“近似1条”.

```
# 没有优化的while
WHILE_BEGIN:
BNE expr1, expr2, WHILE_END
    ...
GOTO WHILE_BEGIN
WHILE_END:


#优化后
BNE expr1, expr2, WHILE_END
WHILE_BEGIN:
    ...
```

```
BEQ expr1, expr2, WHILE_BEGIN
WHILE_END:
```

看起来还是两条跳转，但是WHILE_BEGIN标签移到了第一条跳转之后，所以第一条跳转最多执行一次，所以实际上近似只有一条跳转。