编译实验课程申优文档

> 19373118 薛欣

整体架构设计

编译器整体分为四个层次结构。

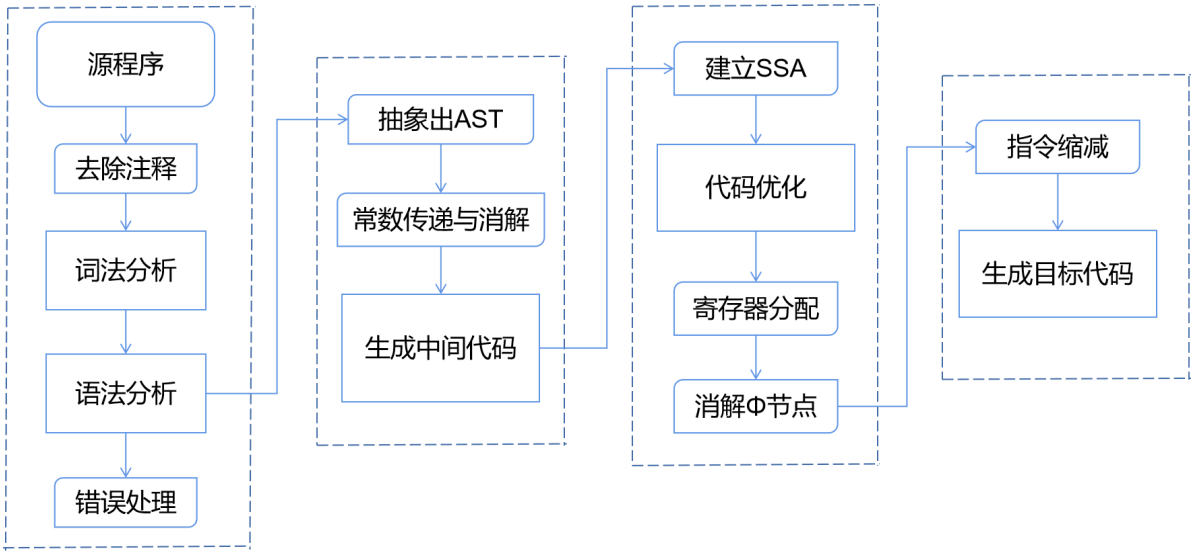
第一层的主要作用是从源程序中提取信息，主要有词法分析、语法分析两部分，在进入词法分析前进行对注释部分发预处理，在语法分析的过程中对错误进行处理。

第二层的主要作用是将源程序信息转化成便于处理的四元式，先从语法分析中抽象出AST树，然后进行一定的常数传递与消解，并生成四元式形式的中间代码。

第三层的主要作用是进行代码优化，首先利用中间代码建立SSA，然后进行一系列优化（例如常数传播、死代码删除），对优化重构后的代码进行寄存器分配，最后消解ϕ节点，生成优化后的代码。

第四层的主要作用是将四元式中间代码转化成mips目标代码，经过乘除优化、指令合并和缩减等，生成最终的目标代码。

整体架构如下图所示。



详细解读难点与解决方案

词法分析

概述

词法分析部分代码的主要任务是将源程序代码划分成若干单词，单词为规定的保留字、字符串常量、标识符、数字、符号等。由于我们已经预先过滤掉了注释，因而不需要考虑注释所带来的问题。

词法分析分为单词识别和存储两个部分。

单词识别分为单字符符号识别、双字符符号识别、保留字识别、标识符识别、字符串常量识别。按照逻辑顺序逐一进行匹配和识别即可。

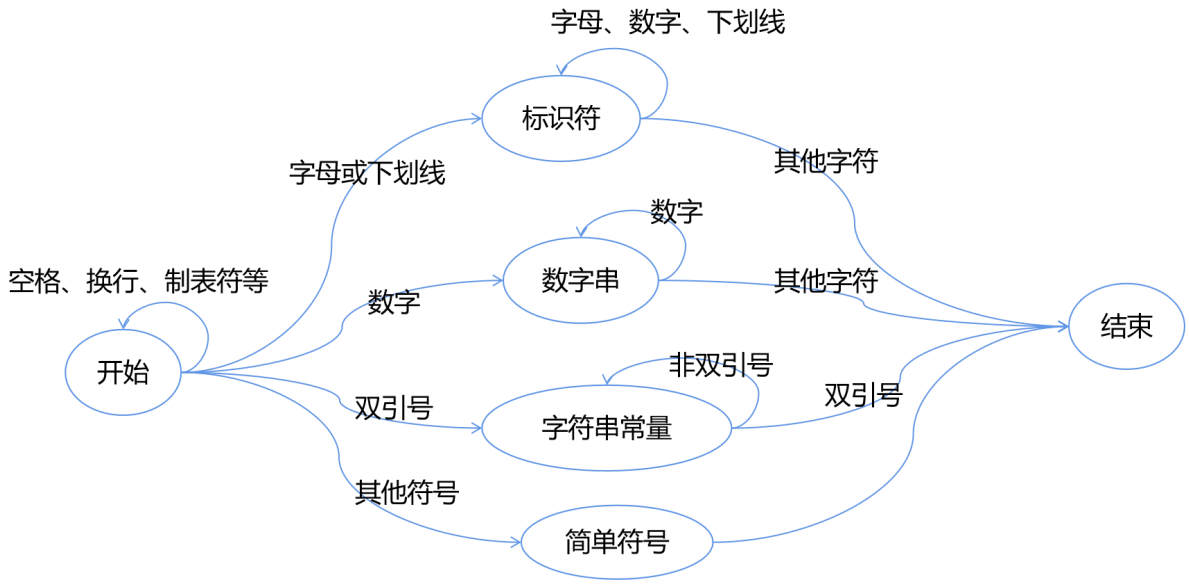
识别后将单词序列按照顺序存储，注意需要同时保存所在行和字符串常量，以备输出和错误处理。

逻辑顺序

词法分析的难点在于如何以正确的逻辑顺序识别单词，识别顺序不当可能造成识别错误。

解决方案

各类符号的识别思路大致如下图所示。需要注意判断顺序依次为：空白符号、标识符、数字串、字符串常量、简单符号。



语法分析

概述

语法分析部分采用递归下降的方法对已经识别的单词序列进行处理。首先，在识别常量定义、变量定义、函数定义、主函数定义前，要进行预读和判断，预读结束后要及时退回。

进入各个非终结符逐层识别，如果某一非终结符含有多种推导方案，优先判断能够用开始第一个字符就能进行判断的方案推导，否则，预读字符进行判断。预读后及时退回。

stmt语句的识别

在stmt中，识别到的语句可能性很多，区分比较困难。

```
语句 Stmt → LVal '=' Exp ';'
        | [Exp] ';'
        | Block
        | 'if' '( Cond ')' Stmt [ 'else' Stmt ]
        | 'while' '(' Cond ')' Stmt
        | 'break' ';' | 'continue' ';'
        | 'return' [Exp] ';'
        | LVal = 'getint''('')'';'
        | 'printf' '('FormatString {',' Exp} ')'';'
```

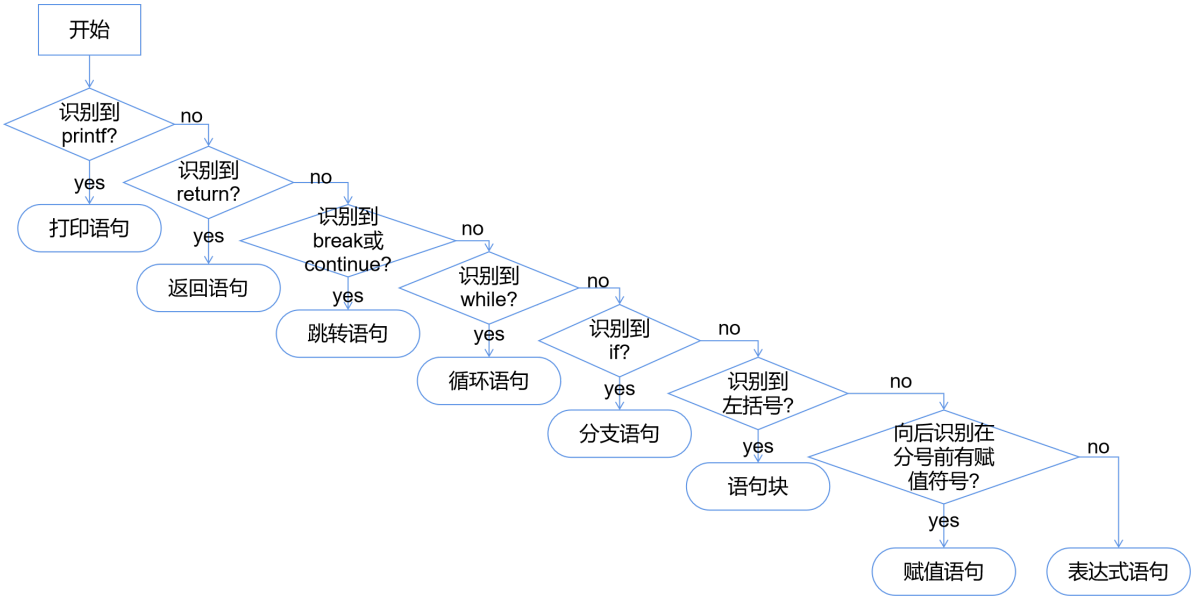
解决方案

首先对于打印语句、返回语句、跳转语句、循环语句、分支语句，可以很方便地通过第一个标识符（保留字）进行识别。

然后，如果第一个符号是左括号，则判断为语句块。

对于赋值语句和表达式语句，本文的识别方法是这样的：从当前位置向后找，判断先遇到分号还是先遇到赋值符号('=')。如果先遇到赋值符号，则说明是赋值语句，否则是表达式语句。

判断思路如下图所示。



错误处理

概述

错误处理首先需要建立变量名表，并记录它的作用域。在函数或变量定义和使用过程中，需要遍历函数表进行查找，查看是否有重名或未定义的情况。

错误处理中最需要注意的是函数传参的判断。函数传参的维数需要逐层递归判断，本编译器采用传入变量指针的方式，将相应维度逐层逐表达式传递。并且需要记录参数栈，来维护多层函数嵌套的情况。

缺少右括号、分号的情况，只要在需要这些符号的位置添加判断即可。

对于函数中return的判断，编译器的做法是，将函数是否需要return的信号向下传递，然后在遇到return的位置进行判断。注意return的位置应是函数体语句块的花括号的前一个有字符的行，而不一定是前一行。

传参验证处理

错误处理过程中，如何判断函数的传参类型、数量、维数是否正确是一个难点。

解决方案

首先，在定义函数时，记录函数参数表，主要记录函数参数的类型、每个参数的维数。在调用函数时，将传入的参数与表中对照。

那么如何获得参数的维数呢?

在递归下降的过程中逐层传递识别到的函数参数，注意，定义参数的维数从0开始记录，每多一个方括号增加一维；调用函数的参数则是查表后，从表中原符号的维数开始，每多一个方括号减少一维。

AST构建

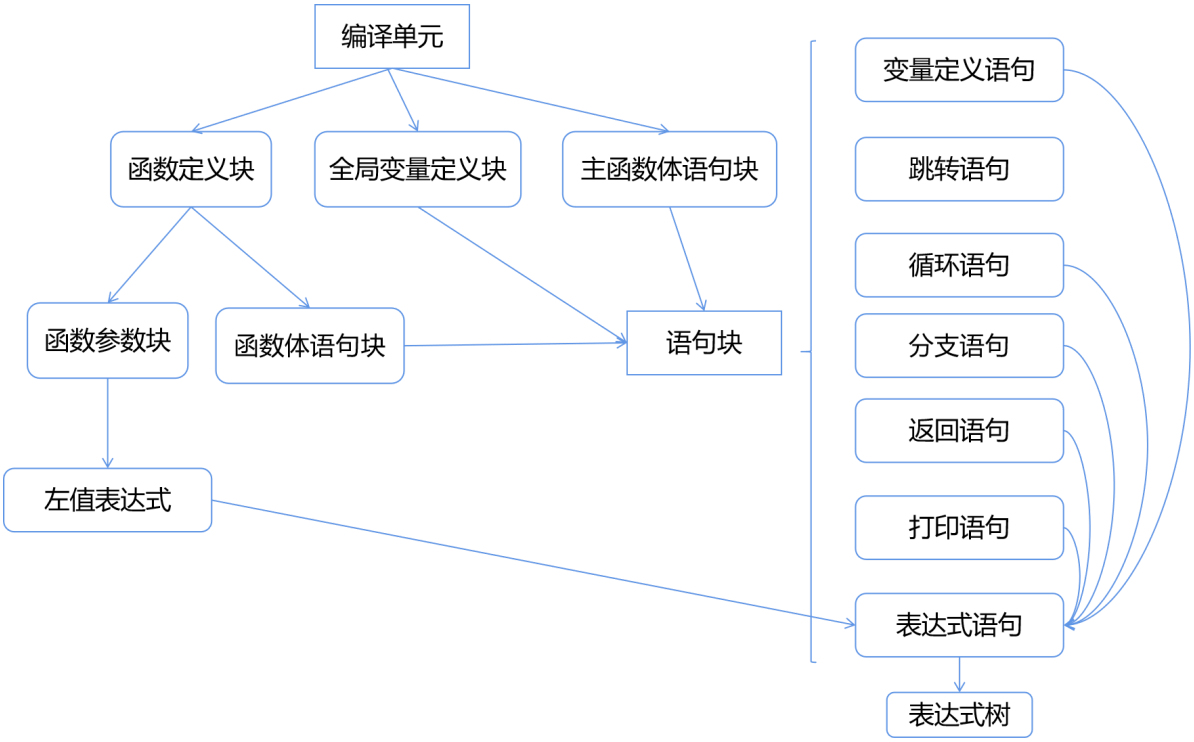
概述

构建AST为方便后续进行常数消解和中间代码的灵活生成。AST设计思路如下：

顶层模块为编译单元，其下包含函数定义块、全局变量块、主函数语句块。这些部分再继续向下细化，产生各种类型的语句块（如变量定义语句块、函数体语句块等）。

这些语句块中集合了各类语句，以线性结构排布，每条语句又包含它所拥有的下一层信息，例如：循环语句包含循环条件、循环语句块等，然后再次向下细分。

最底层将形成表达式树，表达式树是二叉树，由运算符、左操作数、右操作数构成。运算符可以是加、减、乘、除、与、或、非、大于、大于等于、小于、小于等于、等于、不等于。左右操作数可以是左值表达式、立即数和读入语句。



中间代码导出

概述

中间代码导出的目的是为了将原本的信息加工成利于优化的四元式，再利用这些四元式构建SSA和进行代码优化。

这一部分可以看成**无限寄存器**状态。即每次将表达式树上的一个节点，定义一个新的临时变量，赋值为左右操作数运算后的结果。

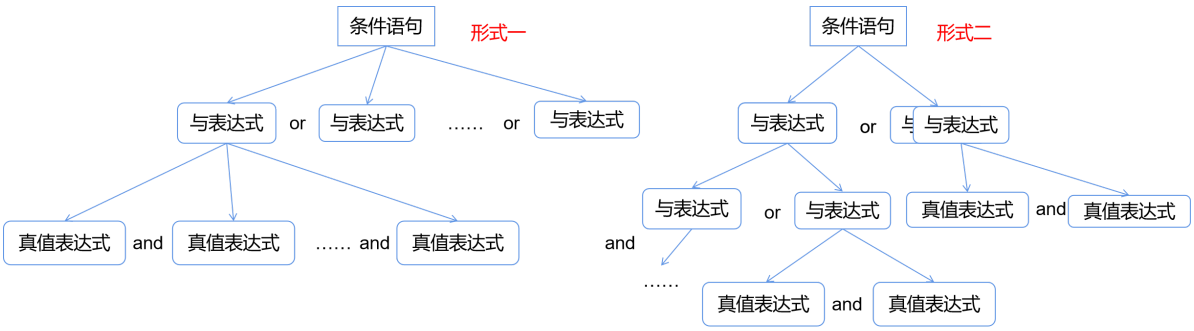
短路求值处理

导出中间代码较难的地方在于如何处理短路求值是一个难点。

对于由AST生成中间代码的方式，短路求值主要有两种处理方法：

1. AST的条件分支中有若干个平行的、由或运算符连接的分支
2. AST的条件分支中是一颗二叉树，或运算的操作数由左至右地对应二叉树上由底向上的节点

两种处理方法如下图所示：



解决方案

考虑我们最终要生成的形式：

连续的条件或：

```
if ( a || b || c) {
    // if 语句块
} else {
    // else语句块
}
```

转化成：

```
bgtz a begin_if
bgtz b begin_if
bgtz c begin_if
j begin_else
begin_if:
    // if语句块
    j end_else
begin_else:
    // else语句块
end_else:
```

连续的条件与：

```
if ( a && b && c) {
    // if 语句块
} else {
    // else语句块
}
```

转化成：

```
beqz a begin_else
beqz b begin_else
beqz c begin_else
begin_if:
    // if语句块
    j end_else
begin_else:
    // else语句块
end_else:
```

考虑到事实上有多级与、或的情况，我们大多数情况不采用直接跳转到标签的方式，而是对条件结果赋真值。

将其拆解为`cond = (a || b || c)`转化成：

```
bgtz a begin_if
bgtz b begin_if
bgtz c begin_if
j begin_else
begin_if:
    cond = 1
    j end_else
begin_else:
    cond = 0
end_else:
```

将`cond = (a && b && c)`转化成：

```
beqz a begin_else
beqz b begin_else
beqz c begin_else
begin_if:
    cond = 1
    j end_else
begin_else:
    cond = 0
end_else:
```

由上述代码可以看出，如果我们按照平行式（方式一）存储，处理将很简单，即在合适的位置插入带有标号的标签即可。不过方式一的构建过程较复杂。

那么层次式（方式二）如何处理呢？本文采用的方式是：由于`and`和`or`不能嵌套，那么我们就可以从一个条件语句递归下去，先处理最左与表达式，然后给这个表达式加入适当标签，再逐层回溯，找到下一个与表达式，给予同样标号的标签。需要注意，不同与表达式直接的标签是需要区分的。

思悟

事实上，方式一和方式二的本质区别是将层次结构**展开**为线性结构的**位置不同**。方式一是在导出AST做出的展开，方式二是在生成中间代码的过程中进行展开。总的来说，在工作量上，并没有太大区别。

SSA构建

SSA的构建主要步骤有：划分代码块、求解支配点集合、求解支配边界、插入ϕ节点、重命名。

划分代码块

划分代码块首先需要找出所有的分支、跳转、标签的位置。将分支和跳转之后、标签之前的位置作为切割点，并将标签的位置做好存储。再次循环，对代码块之间根据跳转、分支等关系建立有向边，需要注意的是，如果某个代码块最后一条不是跳转或分支指令，则它与它紧邻的下一个代码块有一条自然的有向边。

求解必经点集合

这里的必经点的含义为，从0号代码块开始，到达某一个代码块`x`，所以必然要经过的代码块的集合。

求解方法为，考虑删除某一个节点`x`，从0号节点开始bfs搜索，对于每一个不能到达的节点`y`，`x`是`y`的必经点。

求解支配边界

支配边界的含义为，对于一个点`y`和它的一个必经点`x`，设`y`有一个后继节点`z`，`x`不是`z`的必经点，则`z`是`x`的支配边界。简单来说，`z`是`x`刚好支配不了的点。

求解方法为，遍历每一个点和它祖先的所有支配边界，如果能够继续支配，则将继续向下传递，否则将该点去除。

插入ϕ节点

插入ϕ节点的位置，直观上理解，应当某一数据经过不同分支后，第一次交汇的位置。

求解方法为，对于某一个变量`a`，首先找到所有定义了这个变量的基本块，从他们出发，寻找支配边界，将这些支配边界加入队列，这些地方需要插入ϕ。然后再从队列中这些支配边界继续出发，寻找新的支配边界，加入队列，插入ϕ.....直到不再产生新的支配边界。

重命名

重命名需要对所有局部变量、临时变量、函数参数进行重命名操作，每次遇到定义点，即启用新的名字，否则继续沿用前驱到达时的名字。特别的，当有**多个前驱时该如何重命名**呢?

注意到，有多个前驱的节点，必然有ϕ节点，此时我们暂不让ϕ继承任何一个前驱传递来的名字，而是让它像新定义点一样，拥有一个新的名字。然后再此后插入前驱的过程中将所有前驱的名字插入到ϕ的前驱集合中来。

需要注意的是，重命名工作和插入ϕ的前驱集合要分成两次遍历进行，否则将造成向前跳转的边未定义而先插入的情况。

代码优化

常量传播

在SSA结构下，每个变量只有单一定义点，因此，可以直接记录每个变量的值是否为常数，然后替换掉其后的所有使用点。如果某一语句的操作数均为常量，那么即可将这一语句进行简化。

例1:

```
a = 1
b = 2
c = a * b
```

变成

```
a = 1
b = 2
c = 2
```

例2:

```
a = 1
bgtz a, begin_if
// 代码块
begin_if:
```

变成：

```
a = 1
j begin_if
// 代码块
begin_if:
```

观察到这其中有一些代码已经没有存在的必要了，那么我们就可以考虑死代码删除了。

实现方法

利用map存储变量和常数之间的对应关系，即遇到赋值语句`a = 1`时，存入<a, 1>二元组，然后在后续代码中，将所有的`a`换成`1`。

1. 利用map存储变量和常数之间的对应关系，遇到一条语句时，首先查表，尝试将其中的操作数换成常数
2. 如果是运算类指令：
    1. 对于所有能够进行进一步计算的表达式进行计算
    2. 变成结果常数，则存入map中
3. 如果是分支类指令：
    1. 考虑能否删去多余分支，或替换成跳转指令
    2. 重新梳理基本块中边的连接情况
    3. 注意phi节点的依赖情况可能发生改变

逻辑运算的一些技巧

注意到或运算是一真即真的，与运算是一假即假的，这些可以帮助我们提前将一些运算确定为常量。类似的，*1，+-0之类的运算也可以被提前优化掉。

复制传播

复制传播的思路与常量传播相似，主要是利用消除无用的语句中。

例如：

```
a = b + c
d = a
output d
```

可以通过复制传播优化成：

```
a = b + c
output a
```

实现方法：与常量传播相似，利用map存储变量之间的对应关系，即遇到赋值语句`d = a`时，存入<d, a>二元组，然后在后续代码中，将所有的`d`换成`a`。

消除公共子表达式

实现方法：在基本块内，将运算类表达式以集合的方式保存好，每次遇到运算类指令，查找集合中是否有相同的运算，如果有，直接改成赋值语句。

需要注意的是，消除操作要在常量传播、复制传播之后，添加复制传播表之前。由于常量传播和复制传播可能使得表达式与前文有更高的概率重合，所以要在其后完成。因为消除公共子表达式之后可能产生新的可复制变量，因而要在添加复制表之前完成。

最重要的是，要注意消除公共子表达式需要在基本块内部进行，跨越基本块需要清空。

死代码删除

只保留具有后效性的代码，例如：读入、输出、分支、返回值、修改全局变量等，将这些代码中使用的变量标记为有效变量，向前寻找其定义点，将定义点中的所有变量也标注为有效变量，继续寻找……直到所有需要的变量都被定义过。然后删除所有没有被标记过的代码。

例1：

```
a = 1
b = 2
c = 2
```

变成：

```
c = 1
```

例2：

```
a = 1
j begin_if
// 代码块
begin_if:
```

变成：

```
j begin_if
// 代码块
begin_if:
```

此后，然后对代码块重新按照跳转关系建立有向边，将无法到达的代码块删除，变成：

```
j begin_if
begin_if:
```

然后，我们可以利用下文的窥孔优化，直接去掉这个跳转语句，最终变成：

```
begin_if:
```

循环优化

为了减少循环次数，本编译器将`while`改成`if`+`do_while`形式，这样可以节省一次跳转操作：

例如：

```
judge_while:
// while的条件
beqz cond end_while
// while的主体
j judge_while
end_while:
```

修改为：

```
// while的条件
beqz cond end_while
begin_while:
// while的主体
judge_while:
// while的条件
beqz cond begin_while
end_while:
```

值得注意的是，我们额外加了`judge_while`这个在常规while循环中没有被使用的标签。这个标签是给continue语句准备的。在处理break和continue语句时，应当注意：如果while中有break语句，其跳转的位置应该是end_while；而对于continue语句，其跳转的位置应该是judge_while。

冗余的标签会造成跳转语句和跳转目标之间被隔开，从而使得只向下预读一步的窥孔失效，因而我们在下文中采用了向下预读多步的窥孔。

指令缩减与强度削弱

1. 对于除常数的除法：转化为乘法、移位、加减指令，参考《Division by Invariant Integers using Multiplication》
2. 对于除常数的取模：利用伤处方式转化为后，再用商减去除数与商的乘积。
3. 利用addiu 代替subu，减少一条指令
4. 利用div（二目）+mf代替div（三目），以避免div（三目）所带的bne操作

窥孔优化

删去多余的move

当两个变量使用同一个寄存器时，他们之间的复制操作不需要move。

删去多余的分支和跳转

例如：

```
beqz $t0, begin_else_1
label1:
label2:
label3:
j end_else_1
begin_else_1:
end_else_1:
```

当遇到上述情况，分支或跳转与它所要跳转到的标签之间没有实际有效的语句时，可以直接删去这一分支或跳转语句。

实现方法为：在分支（跳转）语句后，向下预读：

1. 如果遇到标签：
    1. 是要跳转到的标签，分支（跳转）语句可以删除，停止预读
    2. 不是要跳转到的标签，继续预读
2. 其他语句，分支（跳转）语句不可以删除，停止预读

寄存器分配

概述

寄存器分配主要步骤包含：活跃变量分析、求解支配树、按支配树dfs序进行分配与冲突处理。

活跃变量分析

活跃变量分析中，我们考虑数组传参、局部变量和临时变量。

先求解每个基本块的`def`和`use`集合，然后进行多次迭代求出`in`和`out`集合。

求解支配树

> 此处求解支配树的方法是笔者自行思考的，定义并不严格，但是在本编译器中具有一定的实用性，最终分配寄存器的效果尚可。

在前面我们提到必经点的概念，那么我们设一个点的必经点中dfs序最大的一个即是它的支配祖先。

实现方法为：首先求出dfs序，然后考虑每个节点，对该节点的所有必经点中dfs序最大的一个进行建边，即得到支配树。

分配与冲突处理

考虑每一个基本块，将所有`in`集合中的点加入活跃集合，然后找每一个没有out的变量的最后一次使用位置。

接下来就可以愉快地分配寄存器了。逐条语句考虑，如果这条语句是最后一次使用，那么将它从活跃集合中去除。对于一个变量的定义点，考虑所有活跃的变量，找到不与之冲突的寄存器。

> 需要注意的是，当两个基本块到达同一个后继节点时，原先针对两个基本块分别进行分配的寄存器之间可能存在冲突，需要及时处理。

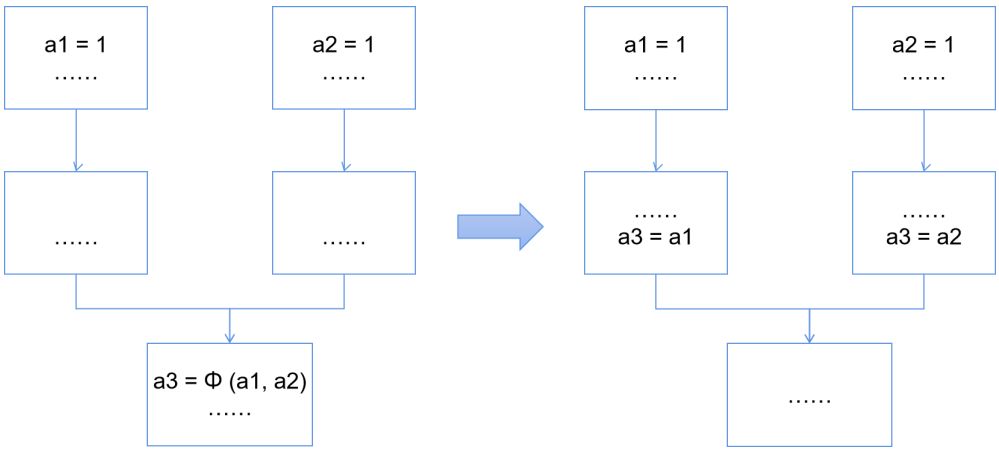
ϕ 节点的共用寄存器

由于一个变量所产生的ϕ节点中不同变量是互相不矛盾的（他们本质上一个变量的不同分身），将它们安排进同一个寄存器更加合理。这样不仅能节省寄存器，而且便于ϕ节点之间的转移。

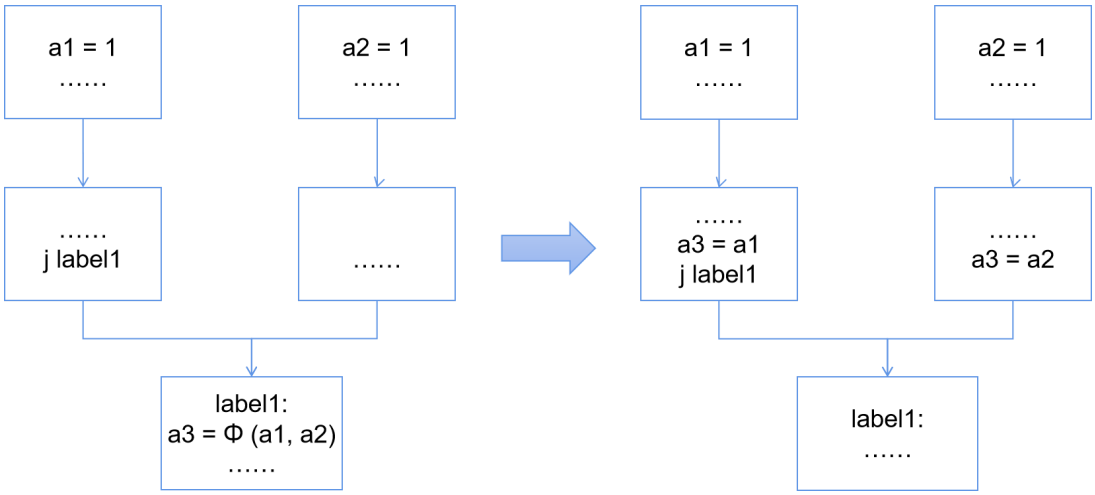
具体实现方法为：在处理ϕ节点时，考虑它的上一级是否有寄存器，如果有则尝试给它这个寄存器，否则不给。

ϕ节点的消解

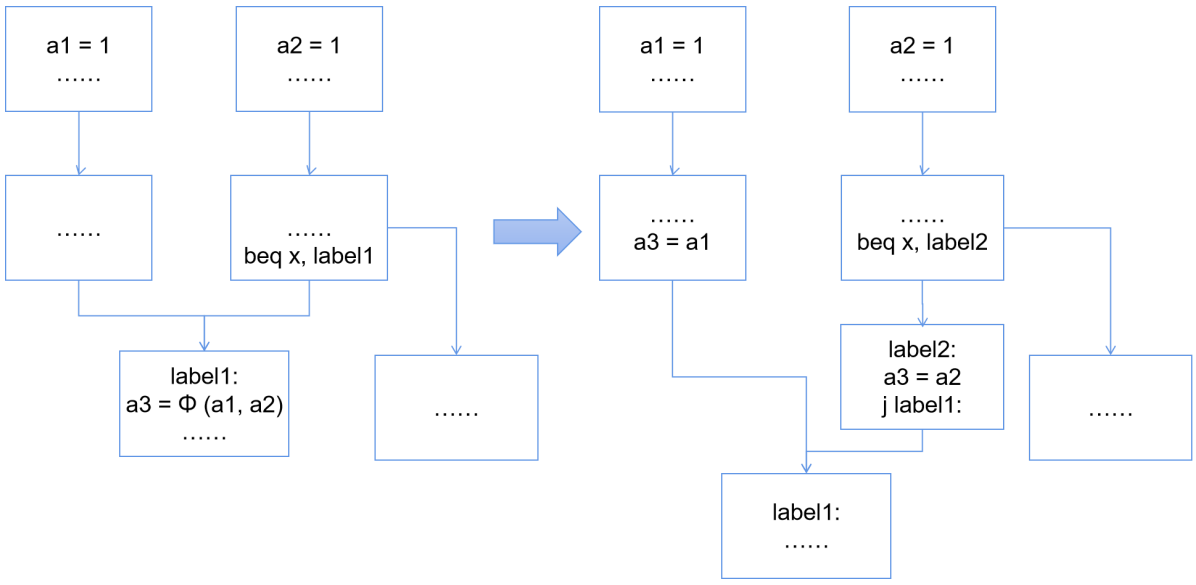
ϕ消解的本质思想是将ϕ函数转化成其前驱基本块中的赋值语句。



特殊地，遇到跳转语句时，赋值加在跳转之前：



遇到分支语句时，需要新建一个跳转块用来单独存储赋值信息：



生成目标代码

生成目标代码的基本思路为：

```
取操作数1
取操作数2
运算
存结果
```

存取变量

其中取操作数有几种情况：

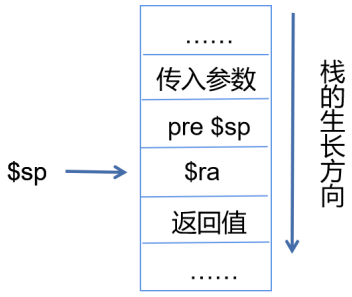
	有寄存器	无寄存器
全局变量	值已经在指定寄存器中	从data段取出，放入临时寄存器
局部变量	值已经在指定寄存器中	从栈中取出（$sp-偏移），放入临时寄存器
函数参数	值已经在指定寄存器中	从栈中取出（$sp+偏移），放入临时寄存器

	定偏移	不定偏移
全局数组元素	从data段取出（变量名+偏移），放入临时寄存器	先取出偏移存入临时寄存器$t3，从data段取出（变量名+$t3），放入临时寄存器
局部数组元素	从栈中取出（$sp-首地址+偏移），放入临时寄存器	先取出偏移存入临时寄存器$t3，从栈取出（$sp-首地址+$t3），放入临时寄存器
参数数组元素	从栈中取出存放数组的首地址（$sp+参数地址）存入$t3，从($t3 + 偏移)得到需要的值，放入临时寄存器	从栈中取出存放数组的首地址（$sp+参数地址）存入$t3，取出偏移存入临时寄存器$t2,从($t3 + $t2)得到需要的值，放入临时寄存器

函数调用

函数调用的过程的主要步骤有：参数压栈、保存现场、记录活动位置、跳转、恢复活动位置、恢复现场。

运行栈的结构示意图如下：



保存现场需要将所有局部变量、临时变量压入对应的位置。记录活动记录则需要存入$sp，$ra，移动$sp指向$ra的首地址，并留出返回值的位置。然后就可以进行跳转了，注意需要使用*jr*指令。恢复活动记录主要是将$sp指针恢复。恢复现场是将所有寄存器重新取回值。

结语

编译实验确实是很花费体力的一项工作，很多过程思考起来并不难，但是处理起来细节很多，很繁琐。能坚持下来很考验毅力，磨练意志。

想要把每一种情况都处理好，就需要在设计之初仔细思考，统筹和规划的能力尤为重要。边写边想往往会导致大量的重构，非常痛苦。不过，即使有比较全面的构思，其实也会有一定的重构，因为随着了解的深入，思考的角度发生变化。所以，在编写代码的过程中要尽可能地降低耦合度，让修改更加方便。

优化部分就是仁者见仁智者见智了，虽然构想了很多情况，但是到测试数据上，并不一定能够有很好的效果。面对这种情况，要学会调整心态，能够实现某一项优化，本身就是一项很有成就感的事，即使没有反映到分数上，也是一件值得兴奋的事。