技术文档

1 概述

我的编译器主要分为词法分析、语法分析（和错误处理耦合在一起了）、符号表、抽象语法树、寄存器管理、优化、目标代码生成、文件输出这几部分。

使用抽象语法树（AST）作为中间代码，将语法分析和后边的步骤解耦。直接在 AST 上做优化，然后直接由 AST 转为 MIPS 汇编。

我做的优化包括常量传播、循环展开、函数内联、代数变换。

2 词法分析和语法分析

词法分析和语法分析就是按照课上讲的方法，按部就班做的，没有什么特点。词法分析先画自动机，然后写出对应的分析代码。语法分析使用递归下降子程序法。主要把大小写转换和全局唯一符号名称这两个点。

2.1 大小写

词法分析，将单词存入 Word 对象时，在 Word 对象内部存一份全小写字母的版本。通过 getWord()方法获取单词的原样，通过 getSmallword()方法获取单词的小写版本。

2.2 符号全局唯一名称

为了函数内联时的方便，将所有符号设置唯一名称。在原变量前加上其所在函数名称。

3 符号表

我的符号表内，分成了多个子表。每个子表对应一个函数，记录了该函数内的所有参数、变量、常量的信息。第 0 个子表对应全局状态，记录了所有全局变量和函数的信息。

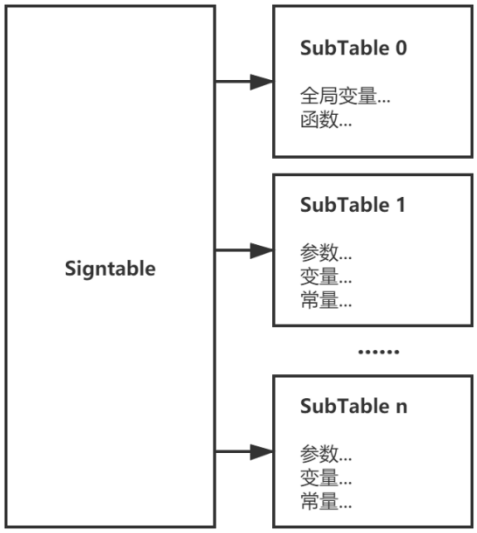

图 1 符号表结构

符号表的全局变量 current 记录当前所处的函数，由语法分析器/优化器/代码生成器在分析过程中赋值给它。这样，在向符号表添加符号时，符号表就可根据current，得知应该将符号扔进哪一个子表。
查询符号信息时，也是先通过 current，确定是哪一个子表，再在子表里查找；但同时也要在 0 号表里查找，因为在函数中遇到的也有可能是全局变量。

子表中存储了“符号”对象。一个“符号”对象，存储了符号的名称、id、级别、类型、类别、维度、第一维长度、第二维长度、初值、基地址（sp 或 gp）、地址偏移量、参数长度、参数类型、返回值类型。一个符号对象可以存储一个变量/常量/数组/函数的信息，实际上一个变量/常量/数组/函数只会用到符号对象的一部分属性。

```
private:
    int id = 0;
    char name[MAX_WORD_LEN];    //名称
    int level = 0;       //层级
    int category = 0;    //类别  函数/变量/常量....
    int type = 0;        //类别  void/int/...
    int dimen = 0;       //维度
    int dimen_n = 0;     //第一维的长度
    int dimen_m = 0;     //第二维的长度
    int f_para_len = 0; //参数个数（<100）
    int f_para_type[100];//参数类别 1代表整数，2代表字符
    int f_return = 0;    //返回值类型
    /*
    若该变量为全局变量，则为相对gp指向地址的偏移量；
    若为局部变量，则为相对sp指向地址的偏移量
    */
    int offset = 0;
    int base = 0;        //为$sp或$gp
    /*
        初值
        若为普通变量，则存储在第0位；
        若为二维数组，则一行一行存。
    */
    vector<int> initValue;
    bool has_init_value = false;
};
```

图 2 Sign 对象中封装的属性

4 抽象语法树

一个 ASTNode 对象代表一个抽象语法树节点，存储了节点类型、子节点地址（构成的数组）、整型值、字符串型值、父节点地址、自己在父节点的数组中的下标。ASTNode 的类别，与文法中的终结符与非终结符基本上是对应的。如下图，图 3 是源代码；图 4 是语法分析后，生成的 AST；图 5 是对 AST 做前序遍历后输出的结果。

```
void main(){
    int a = 1;
    int b;
    b = 1+a;
    printf("b is:",b);
}
```

图 3 源程序

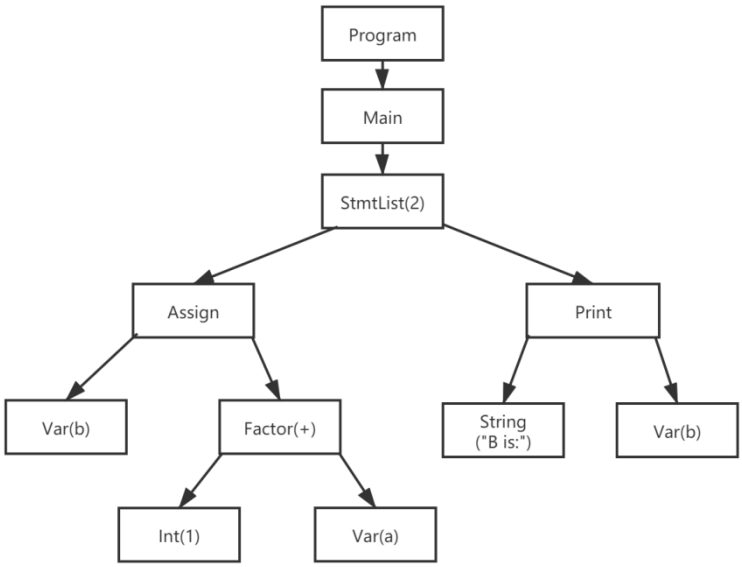

图 4 抽象语法树(AST)

```
Program
Main
StmtList--2{
{
Assign
Var--.main.b
Factor -- +
Int--1
Factor -- *
Var--.main.a
Int--2
},
{
Print
String--b is:
Var--.main.b
},
}
```

图 5 前序遍历输出

通过“父节点地址”和“自己在父节点的数组中的下标”，可以很方便地在优化时实现一些操作，如把自己替换为另一个节点（常量传播，把变量替换为整数）等。

为了方便 AST 的构建，我使用了工厂模式。ASTNodeFactory 类负责创建不同类型的 ASTNode 对象。在语法分析、代码优化阶段，均使用到了 ASTNodeFactory 类。

```
//语句列
ASTNode* makeASTNodeStamentList(vector<ASTNode*> vec_statements);

//Main函数
ASTNode* makeASTNodeMain(ASTNode* stmt_list);

//程序
ASTNode* makeASTNodeProgram(vector<ASTNode*> functions);

//函数
ASTNode* makeASTNodeFunc(char *name,ASTNode* paraListm, ASTNode* stmtList);
ASTNode* makeASTNodeParaList(vector<ASTNode*> paras);
ASTNode* makeASTNodePara(char* name);

//返回语句
ASTNode* makeASTNodeReturn(ASTNode* expression);
ASTNode* makeASTNodeReturn();

//函数调用
ASTNode* makeASTNodeCall(char* name, ASTNode* valuelist);
ASTNode* makeASTNodeValueList(vector<ASTNode*> expressions);

//一维数组和二维数组
ASTNode* makeASTNodeArr(char* name,ASTNode* expression);
ASTNode* makeASTNodeArr2(char* name, ASTNode* expression1, ASTNode* expression2);

ASTNode* makeASTNodeNull();
```

图 6　ASTNodeFactory.h 中的部分方法

5 寄存器管理

5.1 寄存器

我用一个 Register 对象表示一个寄存器，其中封装了寄存器 id、寄存器状态（临时/变量/空闲）、申请使用时间、映射到的内存地址等信息。

5.2 三类寄存器

空闲寄存器：就是空闲的寄存器。

变量寄存器：和某个变量相对应的寄存器，其对象内保存了该变量的内存地址，当被释放时会写回内存。

临时寄存器：不和某个变量相对应，是在计算过程中需要临时用到的寄存器。临时寄存器释放时不需要写回内存。

5.3 寄存器分配

我使用 LRU 算法进行寄存器分配。每当申请寄存器时，在寄存器中记录当前申请

次数（一个递增的变量）。每当没有空闲的寄存器、需要淘汰一个寄存器时，选择最早申请的变量寄存器，进行内存写回并设为空闲状态。
在 RegisterManager 类中统一实现寄存器的分配与管理，其他类通过调用 RegisterManager 类的方法，进行寄存器的申请与释放。

5.4 临时寄存器变为变量寄存器

临时寄存器的一个主要用途是存储表达式计算过程中的中间结果。但如果表达式过长，申请的临时寄存器过多，所有寄存器都占满了，再申请时，就会导致要么临时寄存器被释放（中间结果丢失），要么无法申请到寄存器（程序出错）。
解决办法：就是如果发现当前占着一个临时寄存器，需要去进行其他“不知道需要申请多少个临时寄存器”的操作时，就把当前的临时寄存器转为变量寄存器，为其分配一个内存地址。这样当寄存器不够用时，就可将这个寄存器释放，同时中间结果会被写回内存。

6 优化

6.1 常量传播

在对 AST 做前序遍历的过程中做常量传播。
使用一个 map<string,int>来记录当前可传播的变量/常量名，以及他们的值。
每遍历到一个函数，先将 map 清空，然后将局部变量的初始化值加进 map。
继续遍历，每遍历到赋值语句和 scanf，即可能改变变量的地方，更新 map；每遍历到表达式等需要使用变量值的地方，查询 map，用数值替换掉变量。

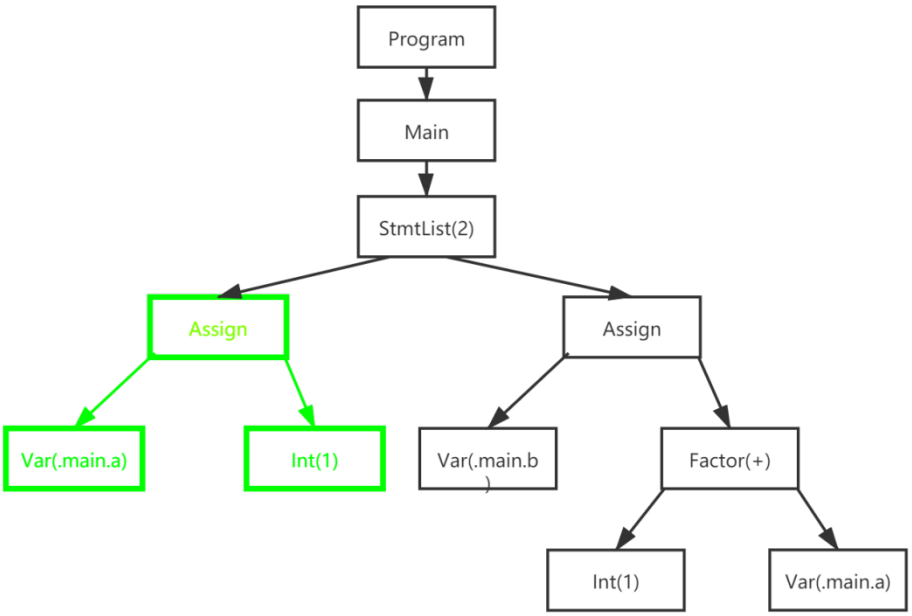

图 7 将键值对(‘.main.a’,1)记录在 map 中。

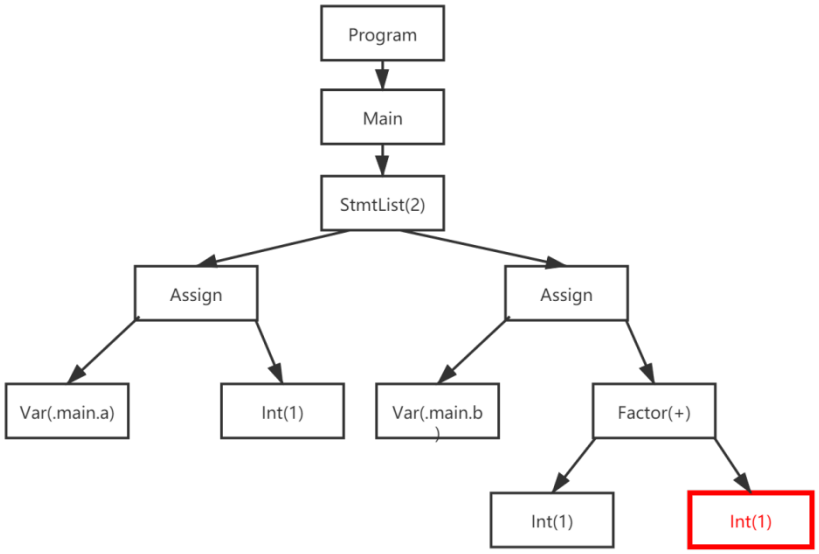

图 8 常量传播 将 Var(.main.a)替换为 Int(1)

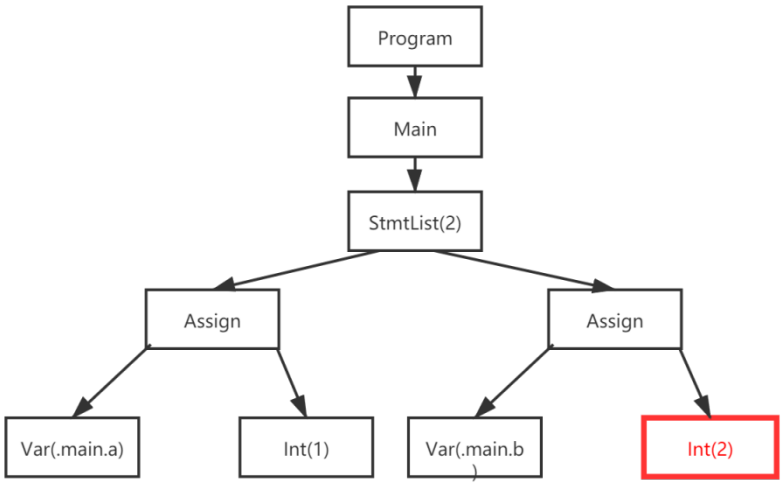

图 9 回溯时直接计算常数结果

遇到 while/for 的时候需要遍历两遍，因为它们的数据流可能由它们内部循环后传过来，在确认了两个数据流并合并之后，才能在 while/for 中传播。

6.2 循环展开

目前我只做了 while 循环的展开。当检测到循环变量有明确的初值、终值、步长时，进行循环展开。

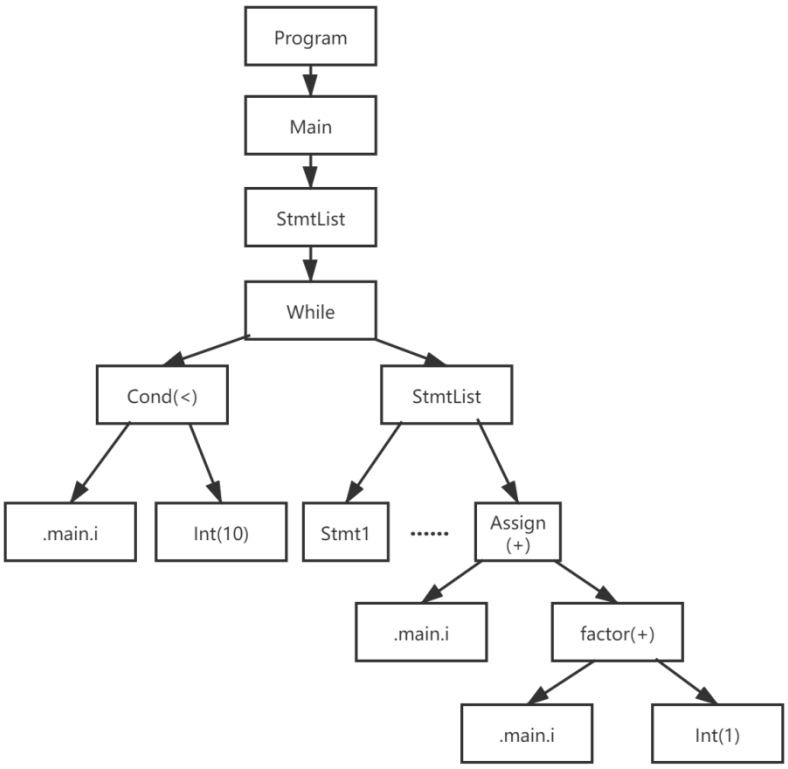

图 10 检测到符合条件的 While，其中.main.i 的初值部分省略了

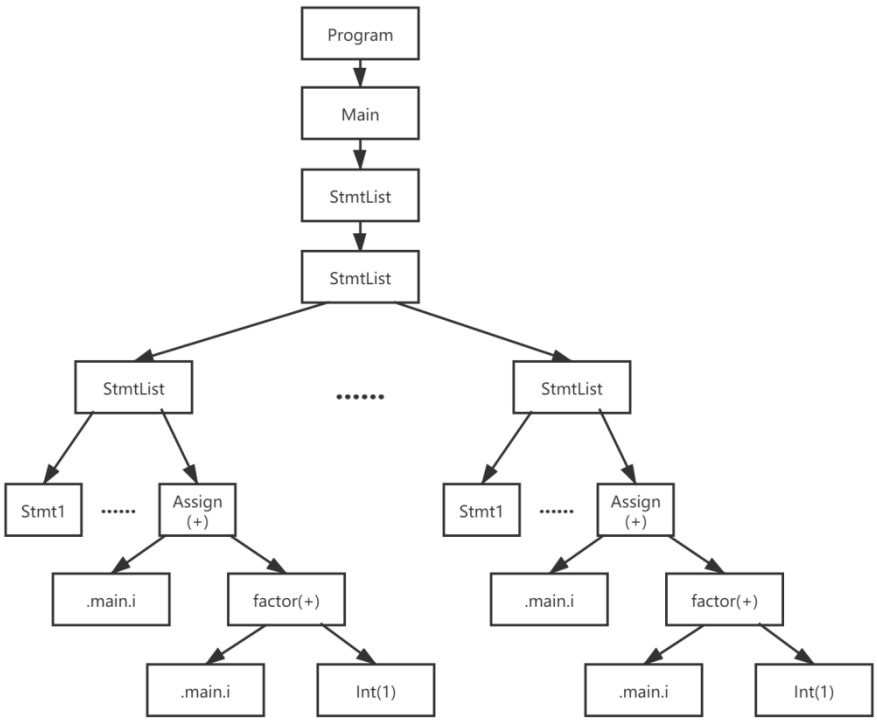

图 11 循环展开后

在循环展开后来一次常量传播可消除大量循环变量 i 的操作。

6.3 函数内联

我判断函数可内联的条件是：至多只有一个 return 语句且在函数最末尾。
内联时，需要导入变量/常量、导入参数、整理代码、修改调用、修改树结构。
导入变量/常量，从符号表中拿到该内联函数的所有局部变量/常量，直接加入到我当前的函数的符号表中。由于在语法分析阶段，实现了符号名的唯一性，所以不用担心变量名冲突的问题。
导入参数，通过创建赋值语句的 ASTNode，将调用函数所用的参数，一个一个赋值给内联函数的参数。
整理代码，将内联函数的语句提出，并检测到 return 语句时将其替换为赋值语句，将返回结果赋值给一个返回变量。
修改调用，将调用位置替换为返回变量。
修改树结构，将上述修改加入到树结构中。

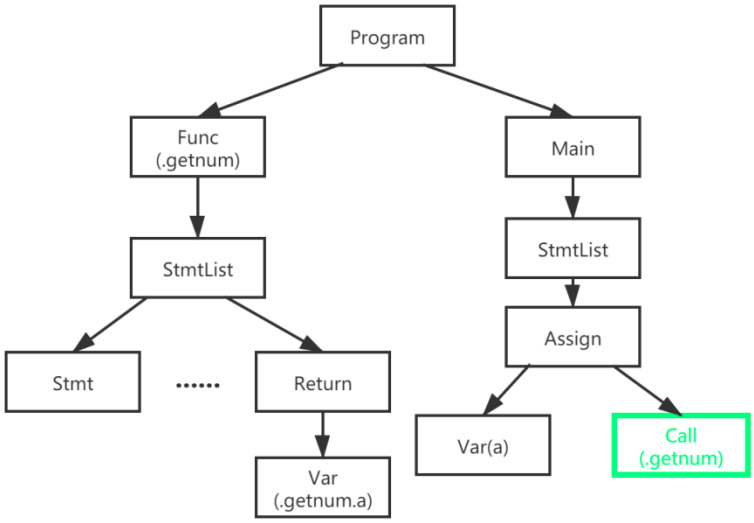

图 12 .getnum 函数可内联，Main 中有调用

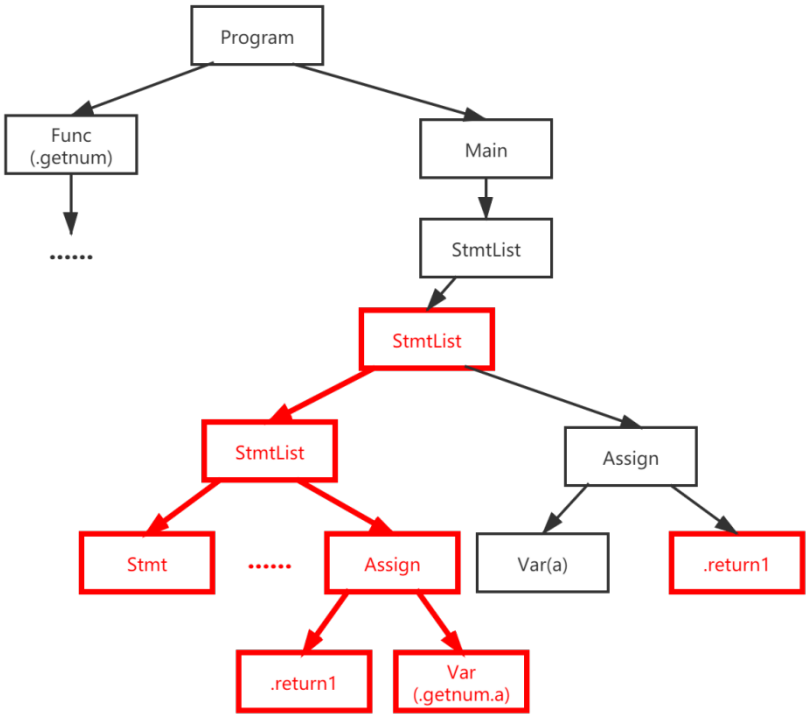

图 13 将函数内联

6.4 代数变换

检测到乘 2 的幂次和除以 2 的幂次时，可替换为位移操作。
在优化阶段，使用“x m y”和“x d y”代表“x 乘 2 的 y 次幂”和“x 除以 2 的 y 次幂”。
然后在代码生成阶段，将“m”和“y”视为和加减乘除一样的二元运算，翻译为为对应的 MIPS 代码。
注意当除以 2 的幂次时，需要考虑除数为负数的情况。

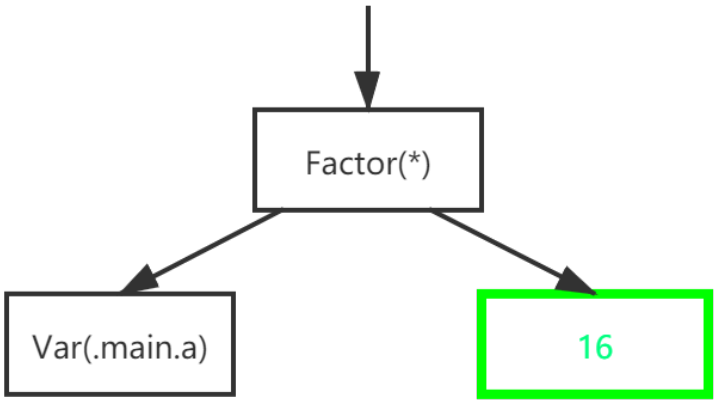

图 14 检测到乘 2 的幂次

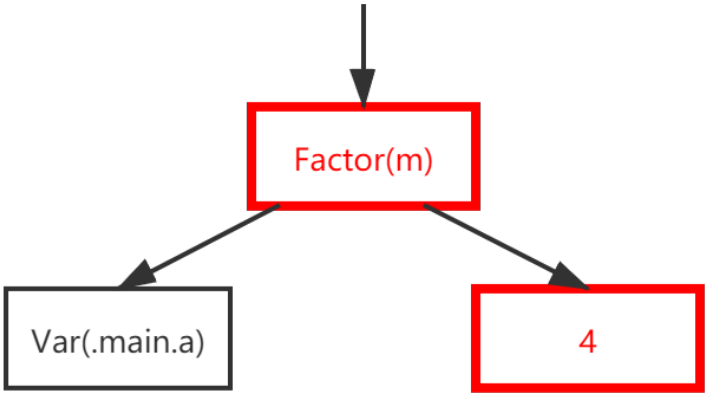

图 15 替换为 m 运算

7 目标代码生成

对 AST 进行前序遍历，综合使用符号表和寄存器管理，生成 MIPS 代码。
大部分是体力活，和 MIPS 做一一对应即可。
大部分的 bug 都出现在分支/循环中数据流合并时，的寄存器分配问题。需要同时思考 MIPS 代码的数据流，以及自己写的 C++代码的模拟的数据流。很多 BUG 出现在这二者的矛盾上。

8 文件输出

在 Output 类中，为了方便调试，使用全局变量 EXP 来判断当前是第几次实验，从而输出不同的内容。如 EXP=0 时，输出词法分析的内容；EXP=1 时输出语法分析的内容....
从 main 函数的 args 参数读数，值赋给 EXP。默认输出代码生成的内容。
编写 python 代码进行自动化测试，通过入口参数来控制编译器输出的内容，并自动比对输出和答案。