

申优文档

18375299 刘传

本文按照本人编译器设计的步骤，进行分析，介绍相应的重点，难点，并着重介绍了基于SSA的代码优化策略和一些未来的展望

词法分析

词法分析时，读取testfile.txt并拼接成一个String对象，根据当前位置判断读取的下一个单词是什么类型并读取相应词法成分，根据文档中提到的词法分析类别，将词法类型分为：

Ident, Num, FormatString, SeparationChar, Note

为实现相应功能添加两个类，分别记录所有的保留字和分隔符，记录方式为存入 HashMap，此 HashMap 的对应关系为单词名称--类别码

判断和读取方法如下：

- 如果当前字符为**大小写字母**或'_'则接下来的词法成分是**ident**(此时的Ident包含了保留字，后面会做出区分)，开始读取字符直到读到的字符**不是大小写字母，'_'，数字**中的任意一个，随后调用 `src/Run/ReserveWord.java` 的 `contain` 方法，判断是否为保留字
- 如果当前字符为**数字**则接下来的词法成分是**数值常量**，开始读取字符直到读到的字符**不是数字**
- 如果当前字符为"则接下来的词法成分是**FormatString**，开始读取字符直到读到"
- 判断当前字符或者当前和下一字符构成的串，是否在 `src/Run/SeparationChar.java` 定义的分隔符表中，如果在，则接下来的词法成分为**分隔符**（此时包含了注释的情况，后面会做出区分），注意到分隔符包含了除号/，而单行注释和多行注释的开头也是/，因此在读取分隔符前应判断当前和下一字符构成的字符串是否为//或/*，如果是则读取一个注释，否则读取上述分隔符

重复上述过程，直到结束，即可得到源代码的所有词法成分

适配整体设计做出的修改

- 添加一个特殊的词法成分，记为End，标志文件结束。
- 在遍历上述testfile.txt拼接成的String时，随时记录当前的行数，读取到\n则当前行数加1，并在词法成分的类**LexicalData.java**中记录该词法成分处在的行数，方便后续错误处理时输出行数
- 后续发现**注释**对后面的工作没有任何作用，因此不再把注释成分添加到存储所有词法成分的 `ArrayList`

语法分析

- 语法的整体设计

首先改写文法，**消除左递归**，采用递归下降分析法，对文法中除去**BlockItem, Decl, BType**外的所有非终结符编写分析程序，并根据当前的词法成分，以及一定的**超前扫描**，判断接下来是什么语法成分，并调用相应的语法分析程序，在每个语法分析程序结束时输出该语法成分的名称即可完成题目要求的输出。

- **左递归的处理方法**

改写文法消除左递归，但是注意到消除左递归后，语法成分的输出发生了变化，以语法成分 `AddExp` 为例，

其文法改写后为 `AddExp -> MulExp {('+' | '-') MulExp}`，如果按照此文法调用语法分析程序会导致少输出一些 `<AddExp>`

考虑到上述问题，其分析程序为(不建立语法树的情况下)

```
private void addExp() {
    mulExp();
    while (isOpLevel2()) { // isOpLevel2判断当前词法成分是否为'+'或'-'
        if (printGrammarData) {
            print("<AddExp>");
        }
        getSym();
        mulExp();
    }
    if (printGrammarData) {
        print("<AddExp>");
    }
    return addExp;
}
```

- 回溯问题的解决

回溯问题的存在是因为对于某个非终结符号的规则其右部有多个选择，且其first集相交，从而导致分析到此语法成分时，不能仅根据当前词法成分判断接下来的语法成分是什么。

采用**超前扫描**，向前多看几个词法成分，直到可以确定接下来需要分析的语法成分，以**函数调用和函数的定义**的判断为例：

```
private boolean isFuncCall() {
    //当前词法成分是ident，并且下一词法成分为左小括号
    return nowSym().isIdent() && nextSym().isLeftParent();
}
```

此判断根据当前和下一词法成分判断接下来的语法成分是否是函数调用

```
private boolean isFuncDef() {
    return (nowSym().isInt() || nowSym().isVoid()) &&
        nextSym().isIdent() && nextTwoSym().isLeftParent();
}
```

此判断根据当前和接下来的两个词法成分判断接下来的语法成分是否是函数定义

适配整体设计做出的修改

- 语法分析的初步设计阶段，并没有建立语法树，但是后面发现，如果不建立语法树，递归下降分析程序结束后，源代码信息就都没有了，就意味着需要在递归下降的同时进行错误处理和中间代码的生成，所以为了结构上的解耦，需要建立语法树

在每个语法成分的分析程序运行时，同时建立语法树的节点，并把该节点作为程序的返回值，以AddExp为例：

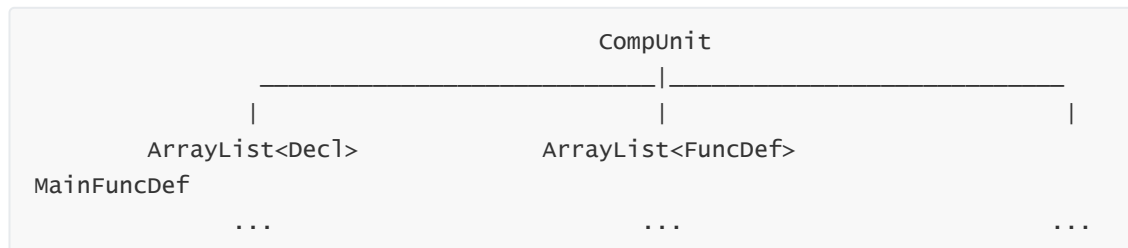
```
//注释的部分显示了建立语法树与否的区别
private AddExp addExp() {
    // mulExp();
    MulExp temp = mulExp();
    AddExp addExp = new AddExp(temp);
    while (isOpLevel2()) {
        if (printGrammarData) { //isOpLevel2判断当前词法成分是否为'+'或'-'
            print("<AddExp>");
        }
    }
}
```

```

        // getSym();
        // mulExp();
        String ope = ope();
        temp = mulExp();
        addExp = new AddExp(addExp, ope, temp);
    }
    if (printGrammarData) {
        print("<AddExp>");
    }
    return addExp;
}

```

最终的语法树结构如下



• 语法树节点的设计

对所有的非终结符，定义一个节点类，注意到某些非终结符号的规则其右部有多个选择，对于这样的节点，提供多个构造方法，并提供方法来判断其类型，以 `AddExp` 为例：

```

//1:MulExp
public AddExp(AddExp addExp, String ope, MulExp mulExp) {
    this.addExp = addExp;
    this.ope = ope;
    this.mulExp = mulExp;
}

//2:AddExp ('+' | '-') MulExp
public AddExp(MulExp mulExp) {
    this.mulExp = mulExp;
}

//判断是此AddExp是否为MulExp
public boolean issingle() {
    return addExp == null;
}

```

错误处理

• 建立符号表

符号表中的记录 `TableRecord` 分为三种类型，以枚举的形式给出

```

public enum RecordType {
    constInt, varInt, funcDef
}

```

并记录维数，来区分普通变量，一维数组和二维数组。

对与初始值可以在编译期计算出的变量，记录变量的初始值，对于函数记录其返回值类型，参数个数和参数表

- **对语法树节点的修改**

因为错误类型中包含函数参数类型不匹配，因此定义数据类型枚举：

```
public enum DataType {  
  
    //      每个Exp都要判断类型，因为涉及数组维数的判断和一些数组/部分数组引用  
    //      需要在建立符号表的过程中判断  
    Undefined, //如果Exp中存在未定义的变量/函数，则定义其类型为Undefined  
    Void,  
    Int,  
    OneDimPointer,  
    TwoDimPointer  
}
```

上述Undefined类型是为了解决如下问题，对于如下代码

```
void func(int a[]) {  
    return;  
}  
int main() {  
    func(x);  
    return 0;  
}
```

对第五行的函数调用，首先会报错**未定义的名字**，如果没有定义Undefined类型，而是给x这个Exp一个随意的类型，或者缺省认为类型是int，则可能导致报参数类型不匹配的错误，但是实际上，这一行只是未定义的名字这一个错误，所以引入Undefined类型，在进行类型匹配的检查时，忽略掉Undefined类型的实参，以下是判断一个实参和形参的类型是否匹配的方法

```
private boolean match(Exp exp, FuncFParam funcFParam) {  
    boolean ret = false;  
    //只判断维数即可 实参:exp 形参:funcFParam  
    if (exp.getDataType() == DataType.Int  
        && funcFParam.getDimension() == 0) {  
        ret = true;  
    } else if (exp.getDataType() == DataType.OneDimPointer  
        && funcFParam.getDimension() == 1) {  
        ret = true;  
    } else if (exp.getDataType() == DataType.TwoDimPointer  
        && funcFParam.getDimension() == 2) {  
        ret = true;  
    }  
    return ret;  
}
```

- **栈式符号表的架构**

首先定义符号表的类型：

```
public enum TableType {
    whileBlock, funcBlock, mainFuncBlock, mainBlock, block, ifDoStmt,
    elseDoStmt
}
```

在 MainTable 中定义 `ArrayList<Table> tableStack` 表示符号表的栈，每次进入一个block根据block类型建立新的符号表并加入栈中，该block结束时把栈顶符号表弹出栈。

- **错误处理的具体实现**

- **非法符号**

单独定义一个方法来检查formatString中的字符。

- **名字重定义**

读取到常量/变量/函数的定义的时候，查**当前符号表**，如果名字已经存在，则说明改名字被重复定义。

- **未定义的名字**

对于常量/变量/函数的调用，查当前符号表，如果存在则找到相应的变量，不存在则查询上一级符号表，如果不存在上一级符号表，则说明该名字未定义。

- **函数参数个数不匹配**

函数调用时，根据传入实参的个数，和符号表中记录的相应函数的形参个数，判断函数个数是否匹配。

- **函数参数类型不匹配**

函数调用时，根据传入的实参和符号表中记录的形参，判断类型是否相同。

- **无返回值的函数存在不匹配的return语句**

对于无返回值的函数，即返回值类型为 `void` 的函数，遍历函数中的stmt，如果stmt为 `ReturnStmt`且该返回语句的Exp不为null，则存在不匹配的return语句。

- **有返回值的函数缺少return语句**

对于有返回值的函数，即返回值为 `int` 的函数，遍历函数，如果函数的最后一个stmt不是具有Exp的返回语句，则报此错误。

- **不能改变常量的值**

对于赋值语句和getint语句，如果语句的左值为常量，则报此错误。

- **缺少分号**

在**语法分析**的时候，调用 `semicolon()` 方法，来读取分号，并添加错误处理

```
private void semicolon() {
    if (!nowSym().isSemicolon()) {
        // error();
        // 报错
        ErrorHandler.handleError(
            new Error(lexicalDataArrayList.get(pos -
1).getLineNum(), "i"));
        return;
    }
    getSym();
}
```

- **缺少右小括号**

实现方法与缺少分号的判断类型，在**语法分析**时进行。

- 缺少右中括号

实现方法与缺少分号的判断类型，在语法分析时进行。

- printf中格式字符与表达式个数不匹配

定义一个方法，读取printf语句中formatString中%d的个数，并与Exp个数比较，如果不同报此错误。

- 在非循环块中使用break和continue语句

记录当前循环的层数初始为0，进入循环的时候层数加1，退出时减1，读取到break或者continue语句的时候，如果循环层数为0，则报此错误。

代码生成

代码生成分两部分进行，生成中间代码和生成目标代码(mips)，方便分开调试，发现问题出在中间代码上还是mips的生成阶段

生成中间代码

设计四元式格式

考虑到代码需要进行的操作，设计了如下四元式

转跳类型

- br 转跳到label
- bez $exp = 0$ 转跳到label

比较指令

- slt $exp1 < exp2 \rightarrow reg = 1$ else $reg = 0$
- sle $exp1 \leq exp2 \rightarrow reg = 1$ else $reg = 0$
- sgt $exp1 > exp2 \rightarrow reg = 1$ else $reg = 0$
- sge $exp1 \geq exp2 \rightarrow reg = 1$ else $reg = 0$
- seq $exp1 = exp2 \rightarrow reg = 1$ else $reg = 0$
- sne $exp1 \neq exp2 \rightarrow reg = 1$ else $reg = 0$

表达式运算

具体操作的是寄存器还是先从内存读取值，放到生成目标代码部分考虑，取决于该变量是否被分配了寄存器

- add $c = a + b$
- sub $c = a - b$
- mult $c = a * b$
- div $c = a / b$
- mod $c = a \% b$
- assign $a = b$

数组存取

- load_array $value = array[offset]$
- store_array $array[offset] = value$

整体IO

- `getInt` 读取整数
- `print_str` 写字符串
- `print_int` 写整数

栈空间管理

- `alloc` 为变量申请栈上空间
- `alloc_array` 为数组申请栈上空间
- `FuncParam` 为函数的参数申请栈上空间

然后在遍历符号表的同时，生成对应的中间代码，以运算指令为例

```
private String analysisAddExp(AddExp addExp) {
    String ret = null, leftOpe = null, rightOpe = null;
    if (!addExp.isSingle()) {
        leftOpe = analysisAddExp(addExp.getAddExp());
    }
    rightOpe = analysisMulExp(addExp.getMulExp());
    addExp.makeDataType();

    if (addExp.isSingle()) {
        ret = rightOpe;
    } else {
        ret = newReg();
        MidCodeMaker.makeCalc(addExp.getOpe(), leftOpe, rightOpe, ret);
    }
    return ret;
}
```

生成目标代码

根据中间代码生成目标代码较为简单，因为设计的四元式已经贴近于汇编语言，因此对于转跳指令，表达式类型和比较并赋值类型的指令，可以直接翻译为相应的中间代码，后面将分两部分解析生成目标代码中较困难部分的思路，分别对应从优化前和优化后生成的中间代码

- 优化前中间代码-目标代码
 - `FuncCall` 函数调用的时候，根据函数的参数个数，压相应的参数入栈，随后调用`jal`进入函数，然后压`$ra`入栈，记录函数的返回地址，参数根据形参的编号，来确定相应的位置(位于`$sp`的顶部依次按顺序向下)
 - 所有的变量认为没有寄存器，每次使用前从内存读出，使用后写入内存，使用`$t0 - $t2`作为临时寄存器
- 优化后中间代码-目标代码
 - `FuncCall` 函数调用的时候，根据函数的参数个数，压相应的参数入栈，随后调用`jal`进入函数，然后压`$ra`入栈，记录函数的返回地址，参数根据形参的编号，来确定相应的位置(位于`$sp`的顶部依次按顺序向下)，这部分和优化前的生成器一致
 - 函数调用前根据数据流分析，把当前正在使用的寄存器压入栈中，函数调用结束时，恢复这些寄存器
 - 变量如果有寄存器，则在调用过程中使用该寄存器，否则使用自定义的三个寄存器来处理`$k0, $k1, $a3`

代码优化

代码优化部分在SSA的架构下做了相关的优化，包括图着色寄存器分配，常量传播，复写传播，全局公共子表达式消除，死代码删除，窥孔优化，乘除优化等等

在编译器的设计中，静态单赋值形式（static single assignment form，通常简写为SSA form或是SSA）是中间代码（IR，intermediate representation）的特性，每个变量仅被赋值一次。在原始的IR中，已存在的变数可被分割成许多不同的版本，在许多教科书当中通常会将旧的变数名称加上一个下标而成为新的变数名称，以至于标明每个变数及其不同版本。在SSA，UD链（use-define chain，赋值代表define，使用变数代表use）是非常明确，而且每个仅包含单一元素。

SSA架构下每个变量仅被赋值一次从而数据流分析更加方便，因此数据流分析相关的优化也可更高效高质量的完成，也可以高效率的完成GVN(全局值编号)从而完成相关的代码优化

SSA的生成

phi节点：当数据是从控制流图(CFG)的不同来源到达时，为区分设计的指令，但是phi指令不能解释为机器语言，所以当SSA上的优化完成的时候，需要消除phi节点，再次生成四元式。

参考 SSA_book 中的生成算法

Algorithm 3.1: Standard algorithm for inserting ϕ -functions

```
1 for  $v$ : variable names in original program do
2    $F \leftarrow \{\}$   $\triangleright$  set of basic blocks where  $\phi$  is added
3    $W \leftarrow \{\}$   $\triangleright$  set of basic blocks that contain definitions of  $v$ 
4   for  $d \in \text{Defs}(v)$  do
5     let  $B$  be the basic block containing  $d$ 
6      $W \leftarrow W \cup \{B\}$ 
7   while  $W \neq \{\}$  do
8     remove a basic block  $X$  from  $W$ 
9     for  $Y$ : basic block  $\in \text{DF}(X)$  do
10      if  $Y \notin F$  then
11        add  $v \leftarrow \phi(\dots)$  at entry of  $Y$ 
12         $F \leftarrow F \cup \{Y\}$ 
13        if  $Y \notin \text{Defs}(v)$  then
14           $W \leftarrow W \cup \{Y\}$ 
```

和论文 Efficiently Computing Static Single Assignment Form and the Control
Dependence Graph 中对DF(支配边界的计算)

The *dominance frontier* $DF(X)$ of a CFG node X is the set of all CFG nodes Y such that X dominates a predecessor of Y but does not strictly dominate Y :

$$DF(X) = \{Y \mid (\exists P \in \text{Pred}(Y))(X \geq P \text{ and } X \not\geq Y)\}.$$

完成对phi节点的插入，即可建立SSA形式的四元式

SSA的销毁

SSA的销毁中最重要的是用可以翻译为机器语言的指令替换phi指令，同时保证正确性，使用SSA_book中的方法，先生成非阻塞赋值指令PCopy，再消除PCopy替换成为一串阻塞赋值指令，在这一过程中可能会产生新的基本块

- 删除phi函数，生成PCopy指令

Algorithm 3.5: Critical Edge Splitting Algorithm for making non-conventional SSA form conventional.

```
1 foreach  $B$ : basic block of the CFG do
2   let  $(E_1, \dots, E_n)$  be the list of incoming edges of  $B$ 
3   foreach  $E_i = (B_i, B)$  do
4     let  $PC_i$  be an empty parallel copy instruction
5     if  $B_i$  has several outgoing edges then
6       create fresh empty basic block  $B'_i$ 
7       replace edge  $E_i$  by edges  $B_i \rightarrow B'_i$  and  $B'_i \rightarrow B$ 
8       insert  $PC_i$  in  $B'_i$ 
9     else
10      append  $PC_i$  at the end of  $B_i$ 
11   foreach  $\phi$ -function at the entry of  $B$  of the form  $a_0 = \phi(B_1 : a_1, \dots, B_n : a_n)$  do
12     foreach  $a_i$  (argument of the  $\phi$ -function corresponding to  $B_i$ ) do
13       let  $a'_i$  be a freshly created variable
14       add copy  $a'_i \leftarrow a_i$  to  $PC_i$ 
15       replace  $a_i$  by  $a'_i$  in the  $\phi$ -function
```

- 删除PCopy指令，生成相应的赋值指令

Algorithm 3.6: Replacement of parallel copies with sequences of sequential copy operations.

```
1 let  $pcopy$  denote the parallel copy to be sequentialized
2 let  $seq = ()$  denote the sequence of copies
3 while  $\neg [\forall (b \leftarrow a) \in pcopy, a = b]$  do
4   if  $\exists (b \leftarrow a) \in pcopy$  s.t.  $\nexists (c \leftarrow b) \in pcopy$  then ▷  $b$  is not live-in of  $pcopy$ 
5     append  $b \leftarrow a$  to  $seq$ 
6     remove copy  $b \leftarrow a$  from  $pcopy$ 
7   else ▷  $pcopy$  is only made-up of cycles; Break one of them
8     let  $b \leftarrow a \in pcopy$  s.t.  $a \neq b$ 
9     let  $a'$  be a freshly created variable
10    append  $a' \leftarrow a$  to  $seq$ 
11    replace in  $pcopy$   $b \leftarrow a$  into  $b \leftarrow a'$ 
```

SSA架构下的相应优化

本编译器中SSA相关的优化只对局部变量进行，全局变量不参与SSA相关优化(这里的全局并不是对应基本块的全局，而是源代码中的全局)，数组也不参与SSA的相应优化

SSA质量优化

开始没有做本优化，发现优化效果很差甚至出现负优化，注意到可能是插入phi节点过多导致的，经过优化，phi节点的数目，从数百个减少到几十个，甚至几个，因为phi节点意味着定义新变量，添加赋值指令(增加冲突)，因此此优化较为重要，做优化前后性能也有明显的提升

优化方法为，首先对于在同一个基本块中定义后使用的变量，不需要为其插入phi节点，因为并不会产生跨基本块的数据流信息，其次，phi节点包含 `UNDEFINED` 情况，即变量未被赋值，但在代码中，因为编写的正确性，这样的phi节点其实是多余的，可以删除，如果phi函数的右部完全一致，也可以删除此phi节点，核心代码如下

```
if (is.isPhi()) {
    ArrayList<String> phiRHS = is.getPhiRHS();
```

```

int num = 0;
for (String rhs: phiRHS) {
    if (rhs.equals(undefined)) {
        num++;
    }
    if (num == phiRHS.size() - 1) {
        needRemoveIS.add(is);
        removePhiFunc(is);
        continue;
    }
    if (arrayIsEqual(phiRHS)) {
        needRemoveIS.add(is);
        removePhiFunc(is);
        continue;
    }
}
}

```

配合常量传播和复写传播，可以在保证正确性的前提下，消除大部分的多余phi函数，做到很大程度上的优化

寄存器分配

分配策略分为两种

- 不区分全局和临时寄存器

直接让\$*t0* – \$*t9*, \$*s0* – \$*s7*, \$*a1*, \$*a2*作为全局寄存器，对所有变量进行图着色的寄存器分配（无论变量是否跨基本块活跃）

但是这时候，如果代码中基本块较大，即出现很长的连续的运算指令，会导致代码的冲突非常多，图着色的效果很差，考虑到这一点，我把\$*t0* – \$*t9*作为临时寄存器，用于不跨基本块活跃的变量使用，从而达到了一定的优化效果，即下述方法

- 区分全局和临时寄存器

令\$*s0* – \$*s7*, \$*a1*, \$*a2*为全局寄存器进行图着色，而只在基本块内活跃的变量为其分配临时寄存器，如

```

ADD reg0 reg1 reg2
ADD reg2 reg1 reg3
ADD reg3 reg1 reg4

```

可进行如下的寄存器分配

```

ADD reg0($t0) reg1($s1) reg2($t1)
ADD reg2($t1) reg1($s1) reg3($t0)
ADD reg3($t0) reg1($s1) reg4($t1)

```

从而大大提高基本块内变量的寄存器分配效果

在图着色分配寄存器的时候，使用如下函数计算每个变量的权值，在寄存器不足的时候，可以让某些变量不持有寄存器，且引起最小的损失，同一循环深度下，冲突越多，则可以让他不持有寄存器，从而减少最多的冲突，同时，对于循环深度较浅的变量，其使用次数的期望相对于循环深度较深的变量要少，也可以优先让它不持有寄存器，赋权代码如下，在图着色算法过程中，当前无法分配寄存器时，选择权重最高的变量删除

```

varWeight.put(var, conflictMap.get(var).size());
for (Instruction is: varUsesSA.get(var)) {
    int deep = blockLoopDeep.get(codeInBlock.get(is));
    varWeight.put(var, varWeight.get(var) + (maxDeep - deep) * 1000);
}

```

乘除优化

- 乘法优化

首先对于如下的中间代码

```
MULT a b c
```

当a或者b为常数的时候可以进行优化，不妨设b为常数，具体分为如下两种情况

$b = 2^i$ 时，优化为

```
SLL a i c
```

$b = 2^i + 2^j$ 时，优化为

```

SLL a i temp1
SLL a j temp2
ADD temp1 temp2 c

```

- 除法优化

对于中间代码

```
DIV a b c
```

若 $b = 2^i$ ，可优化为

```
SRA a i c
```

删除冗余的定义

- 对于只有定义没有使用的变量，可以直接删除它的定义指令，这和常量传播，复写传播，全局公共子表达式删除一起做可以起到删除很多中间变量的作用，达到不错的优化效果

常量传播

常量传播可以把SSA架构下定义时赋值为常数的变量直接替换为常量，删除该变量的定义，此算法要递归的进行，可以替换掉很多不必要的变量，如SSA形式下的四元式

```

ASSIGN reg1_0 1
ADD reg1_0 reg1_0 reg2_0
ASSIGN reg3_0 reg2_0

```

递归的进行常量传播，可将上述代码优化为

```
ASSIGN reg3_0 2
```

因为SSA中每个变量只会被赋值一次的特点，该算法可以保证正确性

复写传播

对于下列形式的四元式

```
ASSIGN reg1_0 reg2_0
```

可将中间代码中所有使用reg1_0的指令变为使用reg2_0，删除此中间代码，删除reg1_0这个变量(因为此时它只被定于而没有被使用，使用删除无用定义来删去)

因为SSA中每个变量只会被赋值一次的特点，该算法可以保证正确性

全局公共子表达式删除

对于如下形式的四元式

```
ADD a b c
ADD a b d
ADD c d e
```

可以优化为

```
ADD a b c
ADD c c e
```

具体实现方法为，对于所有的运算指令，将其右部连接成字符串，放入hashMap，对于上述样例，处理第一条指令时，“a+b”会被放入hashMap，映射值为c，处理到第二条指令的时候，发现“a+b”已经出现在hashMap的键值中，所以对于第二条指令定义的d可以把中间代码中所有使用d的指令替换为hashMap中“a+b”对应的值即c，然后删除第二条指令(第二条指令为d的定义指令，因为此时已经没有使用d的指令，所以可以删除)，同时第三条指令被优化为 `ADD c c e`

因为SSA中每个变量只会被赋值一次的特点，该算法可以保证正确性

有关常量传播，复写传播和全局公共子表达式删除

这些优化的正确性由SSA的性质保证，因为SSA中每个变量都只会被赋值一次，然后被多次使用，所以上述三个优化方向均可保证正确性，又因为上述优化均可以减少中间代码中变量的个数，这对寄存器分配很有意义，因为减少了变量间的冲突，可以让图着色算法为更多的变量分配寄存器

这三个优化也被称为全局值编号(GVN)，可以参考如下伪代码

```
DVNT_GVN(block b):
  for each phi node in b:
    remove and continue if meaningless or redundant
    set the value number for the remaining phi node to be the assigned variable name
    add phi node to the hash table

  for each assignment:
    get value numbers for each operand
    simplify the expression if possible
    if the expression has been computed before:
      set the value number for the assigned variable to the expression's value number
    else:
      set the value number for the expression to be the assigned variable name
      add the expression to the hash table

  for each child c of b in the control flow graph:
    replace all phi node operands in c that were computed in this block with their value number

  for each child c of b in the dominator tree:
    DVNT_GVN(c)

  remove all values hashed during this function call
```

窥孔优化

- 可以注意到，因为四元式的使用，代码中存在不少这样的定义

```
ADD a b c
ASSIGN d c
```

因为为减少这种情况，添加窥孔，将上述代码优化为

```
ADD a b d
```

- 观察我的四元式中间代码，注意到代码中存在这样的形式

```
ADD 0 a b
```

显然可以优化为

```
ASSIGN b a
```

配合复写传播，可以起到优化作用

- 在阅读生成的中间代码时本人注意到，有时会出现这样的情况
a用来定义b，b用来定义a，而除了定义彼此，a，b没有其他的应用，比如中间代码形式

```
while_begin:
//if a > 10 BR while_end
...
ADD a 1 b
ADD b 1 a
BR while_begin
while_end:
```

这时候，上述代码中的 `ADD a 1 b` 和 `ADD b 1 a` 可以被删除而不影响正确性，在这样的代码出现在循环中的时候，可以起到更好的优化效果

未来的展望

本编译器的优化基于SSA的中间代码形式，做了SSA形式下效果较好的一些优化，未来还可以做的优化有

- 常数不是2的整次幂的除法，本人在优化过程中，找到了相关论文，但是只能解决除数为大于0的常量，被除数大于零时候的情况，可以把除法转化为乘法，在本人的编译器中以注释形式出现，因需要添加新的跳转指令(判断被除数是否大于零)，最终版本并未提交此优化，后又找到了普遍条件下适用的除法转乘法的算法，未来可以加入到编译器中
- 寄存器分配的效果，和其他同学交流过程中发现，做的优化大方向相同的前提下，寄存器分配的策略，对结果有很大的影响，因此可以考虑更优的启发式算法，来进行图着色的寄存器分配，从而提高寄存器分配的效果，进行进一步的优化
- 对于只在某一个函数中使用的全局变量可以当作局部变量纳入SSA的处理范围之内，对于长度较小的数组，也可以展