编译器申优文档

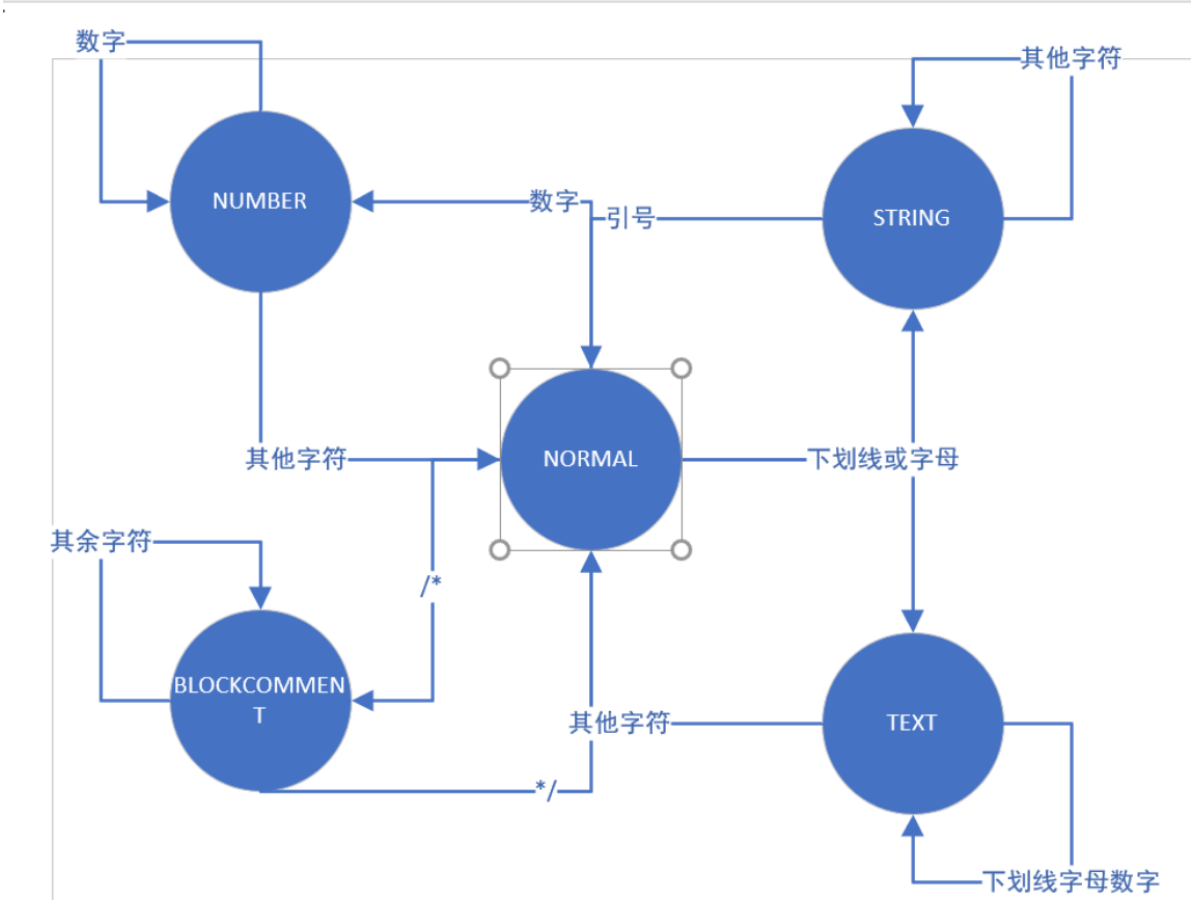
19373736 赵慧雅

词法分析

词法分析部分我通过构造有限状态机完成，其中主要有如下这些状态

```
NORMAL
BLOCK_COMMENT
STRING
TEXT
NUM
```

分别代表着正常状态，多行注释，字符串常量，标识符和数字，其状态转移图如下所示：



词法分析部分无显著难点和设计点，比较细心的设计多个状态和他们之间的转移即可。需要在每个token中记载足够的信息，包括符号类型，符号原文（尤其是标识符）和所在行号（用于错误处理）。另外对于既能作为单字符运算符开头也能作为多字符运算符开头的字符（如`<`和`<=`）要进行预读，判断和下一个字符是否可整体作为一个运算符。

语法分析

语法部分使用递归下降进行实现，在SyntaxAnalysis类中设计了`gen[GrammerElement]`函数系列，完全按照语法成分进行分析。互相递归调用进行语法部分的处理。其中使用了index位置标注了当前读取的位置，并且调用token(0),token(1)获取当前或者下一个单词进行分析。

主要实现和上述一致，在建立语法树的时候建立了部分抽象，其中所有类的基类是RootNode，关于表达式的基类为Exp，语句的基类是Stmt，这样的抽象有利于容器中的类型一致，有相同的接口。在递归下降子程序中返回生成的对应语法成分的指针，在每个对应的递归下降子程序中，分别读取每一个部分，存入node中。

最后，有一个地方特殊设计的地方。在语句`Ival=EXP;`和`EXP;`中，`EXP`的开头部分可以是`Ival`，也就是两者的首部完全相同，且一旦尝试读取 va1 但实际是 Exp 的话就无法跳回递归下降的队规环境，此处采用了回溯的方法进行处理——在涉及方法中采用"预读"的措施，并记录之前进入预读装态中位置，定义getRoll0方法将状态回滚回开启预读之前，在预读过程中禁用所有输出。这样通过一个简单的回溯解决了这一首部相同问题。

错误处理

在错误处理中我改造了之前的递归下降子程序，生成了语法树。而具体错误处理中我将错误分为两类，第一类是语义错误如语句缺少分号，括号等，这一部分将在语法分析中进行处理，第二部分是和语义相关的内容，如重定义等。通过符号表的建立处理语义错误，采用栈式符号表进行管理，每个块内使用hash方式存储、分别给函数和变量建立不同的符号表，存储它们的类型信息，函数包括返回值，参数数量和各自类型。变量只有类型，通过查表判定是否匹配。

在输出中需要建立一个全局的公共数组，在两阶段处理中一起收集所有的错误，并且将他们排序后进行输出。

中间代码生成（SSA构建）

前三部分基本按部就班的按照最佳实践进行即可。从中间代码生成部分开始进入了设计的重点。

类型系统

类型系统设计

首先是中间代码的类型系统的设计，此处我仿照llvm进行设计，将所有的类型都设计为了Value

其中所有类的公共基类都为value，value中主要包含各种类型的指令（Instr），如声明（Decl），表达式（Exp），指针（Pointer），获取地址（GenAddress），分支跳转（Branch）等，它们大多会被后端翻译为具体的指令。还包括一些特殊类型，如phi节点（Phi）,函数（Function），基本块（Block），常量（Constant）等，它们不会被在后端中翻译成一条具体的。所有的这些都有id进行标识。

类型系统引用

Value之间互相的引用构成了整个中间代码的结构，其中主要包含两种引用方式：

- 直接持有，某一类型直接拥有其他类型（或他的容器）作为成员，适用于简单持有类型，如每个block持有入口点和block的数组。
- 通过use关系间接持有，本质为value-user的pair。如每个二元表达式的左右两侧持有都是Use，而在值一侧会记录下持有它的user。这样可以方便的完成def-use关系的追踪和替换，由于java的引用特性，只要将对应use对象内的value替换为新的value即可完成值的中心指向，方便的完成replaceallusewith等操作

SSA构造

在ssa构造环节，我参考了《Simple and Efficient Construction of Static Single Assignment Form》进行实现，在语法分析部分的语法树上进行先序遍历逐条生成，phi诞生于（有多个前驱基本块）变量的读取，因此有如下算法

- ```
  writeVariable(variable, block, value): //向某一块中写入定义
      currentDef[variable][block] ← value
  ```

- ```
  readVariable(variable, block):
      if currentDef[variable] contains block:
          # local value numbering
          return currentDef[variable][block]
      # global value numbering
      return readVariableRecursive(variable, block)#从前驱基本块中读值
  ```

- ```
  readVariableRecursive(variable, block):
      if block not in sealedBlocks:
      # Incomplete CFG
          val ← new Phi(block)
          incompletePhis[block][variable] ← val
      else if |block.preds| = 1:
          # Optimize the common case of one predecessor: No phi needed
          val ← readVariable(variable, block.preds[0])
      else:
          # Break potential cycles with operandless phi
          val ← new Phi(block)
          writeVariable(variable, block, val)
          val ← addPhiOperands(variable, val)
      writeVariable(variable, block, val)
      return val
  ```

- ```
  sealBlock(block):
      for variable in incompletePhis[block]:
          addPhiOperands(variable, incompletePhis[block][variable])
      sealedBlocks.add(block)
  ```

以上简单的算法已经足以生成ssa形式的代码。在中间代码生成的部分主要生成各种形式的instr，并建立基本块之间的连接关系。

中端优化

- 代码内联：对于所有非递归的函数均采用内联操作。直接调用上述的writeVariable函数将参数写入定义之中，同时调整了符号表，需要注意符号表的级别为最外层。该优化可以让常量能传播进函数中，起到了过程间优化的效果。
- 移除简单phi节点，使用上述论文中的算法进行移除，此步骤可以消除大量无用phi节点

```
tryRemoveTrivialPhi(phi):
    same ← None
    for op in phi.operands:
        if op = same || op = phi:
            continue # Unique value or self−reference
        if same 6= None:
            return phi # The phi merges at least two values: not trivial
        same ← op
    if same = None:
        same ← new Undef() # The phi is unreachable or in the start block
    users ← phi.users.remove(phi) # Remember all users except the phi itself
    phi.replaceBy(same)
```

- 常量传播：朴素的常量传播可以通过readVariable和WriteVarible实现，对于那些两端都是常量的表达式，也可进行进一步的优化，在中端遍历所有的value，判断它是否是二元表达式且左右两侧均为常量，如果是的话直接将结果替换为常量计算结果。
- 死代码删除：借鉴了LLVM中aggressive dead code elimination Pass的思路，从必要的访问点（返回值，函数调用，存储值）出发，反向标记代码是否可达，最后一次性消除不可达的边
- 局部公共子表达式消除：（本来想做全局的，看到LLVM2000行代码望而生畏）对于每个基本块的生成value使用hashmap进行保存，这样可以起到去重的作用，当新的值已经存在在hashmap中时，只要使用再次使用替换操作变成已有的value即可。
- 数组访问优化：专门针对testfile1中的数组访问进行了优化，可以将数组基地址设为一种Value，数组偏移量设为另一种，这样可以将数组基地址作为公共子表达式进行优化（一次la有两条指令），针对testfile优化效果很好，另外偏移量也可以作为value进行公共优化。

后端代码生成

寄存器分配

寄存器分配是代码质量的决定性因素，我查阅资料，使用了《Linear Scan Register Allocation on SSA Form》论文中的算法，使用linear scan（这一决定借鉴于LLVM）算法进行寄存器分配，兼顾了寄存器分配的速度与质量，其中主要有两个步骤：

- ```
  BUILDINTERVALS（生命周期计算）
      for each block b in reverse order do
      live = union of successor.liveIn for each successor of b
      for each phi function phi of successors of b do
          live.add(phi.inputOf(b))
      for each opd in live do
          intervals[opd].addRange(b.from, b.to)
      for each operation op of b in reverse order do
          for each output operand opd of op do
              intervals[opd].setFrom(op.id)
              live.remove(opd)
          for each input operand opd of op do
              intervals[opd].addRange(b.from, op.id)
              live.add(opd)
      for each phi function phi of b do
          live.remove(phi.output)
      if b is loop header then
          loopEnd = last block of the loop starting at b
          for each opd in live do
  ```
  ```

```
            intervals[opd].addRange(b.from, loopEnd.to)
    b.liveIn = live
```

- ```
  TRYALLOCATEFREEREG
      set freeUntilPos of all physical registers to maxInt
      for each interval it in active do
      freeUntilPos[it.reg] = 0
      for each interval it in inactive intersecting with current do
          freeUntilPos[it.reg] = next intersection of it with current
      reg = register with highest freeUntilPos

  ALLOCATEBLOCKEDREG
      set nextUsePos of all physical registers to maxInt
      for each interval it in active do
          nextUsePos[it.reg] = next use of it after start of current
      for each interval it in inactive intersecting with current do
          nextUsePos[it.reg] = next use of it after start of current
      reg = register with highest nextUsePos
  ```

phi节点消除

使用经典的算法进行消除，首先将non-conventional的ssa转化为conventional的ssa，并且插入了若干parallel copy，第二步再进行删除，以上算法借鉴于《Static Single Assignment Book》

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 ¬[∀(b ← a) ∈ pcopy, a = b] do
4      if ∃(b ← a) ∈ pcopy s.t. ∄(c ← b) ∈ pcopy then        ▷ b is not live-in of pcopy
5          append b ← a to seq
6          remove copy b ← a from pcopy
7      else                                                  ▷ pcopy is only made-up of cycles; Break one of them
8          let b ← a ∈ pcopy s.t. a ≠ b
9          let a′ be a freshly created variable
10         append a′ ← a to seq
11         replace in pcopy b ← a into b ← a′
```

后端代码优化

最后进行的就是真正的后端代码生成，将每条value逐一翻译成后端语句，由于时间原因我没有构造mips形式的LIR进一步优化，而是直接边生成便进行了如下优化。

- 运算强度消减，将乘除法指令进行优化，参考《Division by Invariant Integers using Multiplication》进行操作。将所有的常量乘除法进行了消除。
- 窥孔优化1：对于生成的block列表，由于在linear时已经进行了较好的线性化，因此跳转目标大概率就是下一基本块，如果是此类型的跳转可以进行优化
- 窥孔优化2：对于生成的move等指令，判断两侧的值是否是一样的，如果一样直接跳过生成
- 窥孔优化3：对于函数调用，并不保存所有寄存器，而是保存当前函数分配的变量和目标函数的交集
- 窥孔优化4：对于立即数，尽量让它直接作为指令的操作数而不是一个临时寄存器。

总结
```

通过上述优化，最后我的成绩是17名，还是非常令人满意的，我总结出了两点经验进行分享

- 多多查询资料，我认为我复现论文中的ssa构造方法要比经典方案更好。对于ssa和其他编译器优化，可以多上llvm或者知乎（RednaxelaFX）或者谷歌学术查询论文等来源进行搜索和学习。
- 多看自己生成的代码，看看为什么性能差。可以琢磨出很多有用的窥孔优化，或者琢磨出要去学习某一个经典优化来提高性能。