Lab 实验报告

SysY Compiler

丁睿
dromniscience@gmail.com

更新:2021 年 6 月 4 日

Source Code Available at *this repo*

目录

1 总体设计

编译器的总体框架如下图所示（以下简称该编译器为 Tiny）。Tiny 总是以 SysY 源文件为输入，由 `-e`，`-t`，`-S` 指示分别将它编译为 Eeyore, Tigger 和 RISC-V 文件。

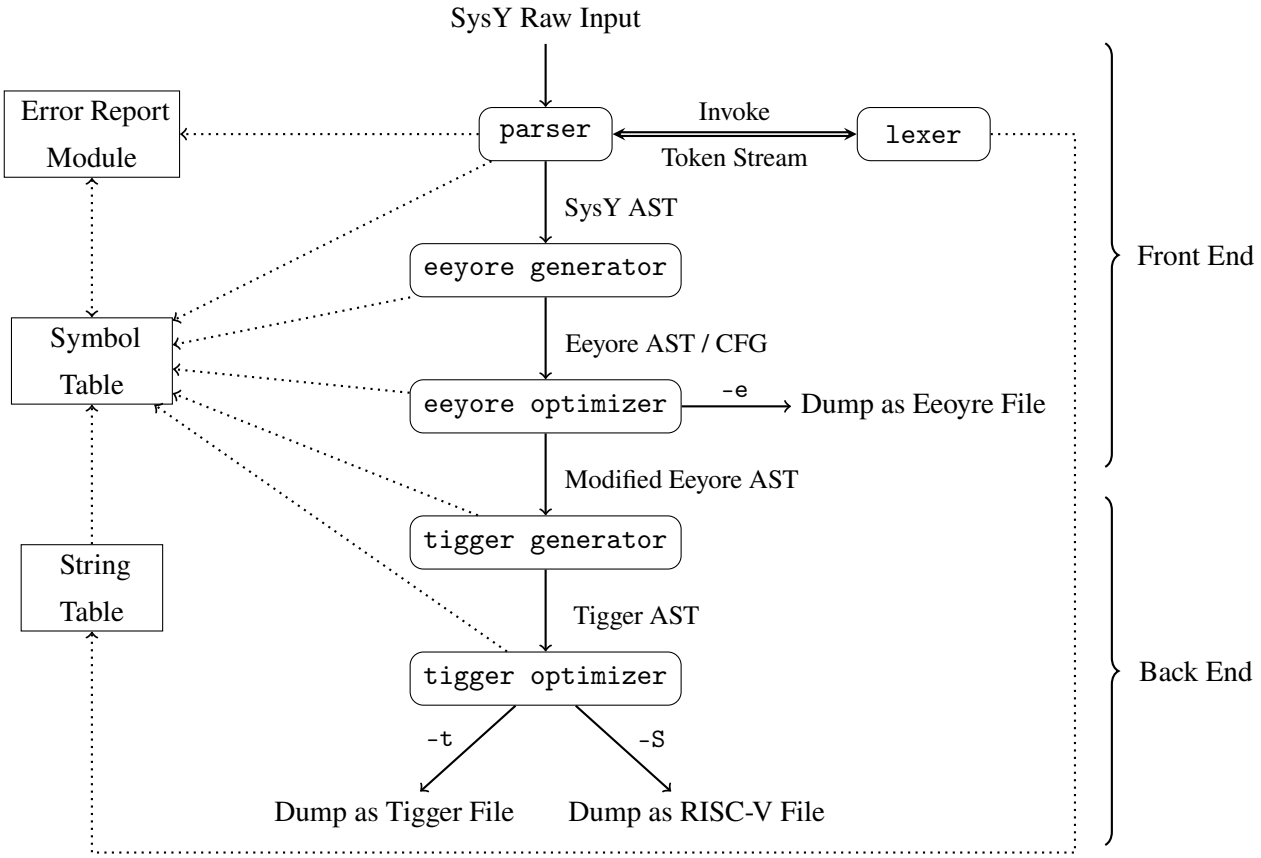

图 1: Tiny 的基本结构

1.1 符号表 & 字符串表

为了避免字符串的频繁移动甚至拷贝，采用字符串表存放一切变量名和函数名。它需要支持的操作仅仅是 `Insert(std::string &&)`，因此使用链表存储。该操作仅在 lexer 中执行。

符号表包含了作用域的语义信息，因此由 parser 维护[1]。其中涉及到名称的存储，皆由到字符串表的指针指明。具体的数据结构在节2.1谈及。

1.2 前后端

由于 Eeyore 的三地址代码的特性，Tiny 将以 Eeyore 优化器的输出为界划分为前端和后端。parser 将采用 1 pass 的方式得到源程序的 AST，然后先序遍历该树得到 Eeyore 的 AST 表示。注意 Eeyore 除了寄存器和过程栈帧外已与目标语言无差。其 AST 只需要一个线性表来表示，因为所有源文件中的复杂数据结

[1]请注意 SysY 规则的简单性使得在 lexer 中分析 scoping 的信息也非常轻松。

构以及控制结构均被翻译到低层次的三地址代码。较之于线性表，为了有效优化，我们很容易基于此构造其 CFG 表示。分析器需要进行的分析在 basic block 内部只有两种顺序，例如正向的 Available Expression Analysis 以及逆向的 Liveness Analysis，因此链表是 basic block 最合适的数据结构。

同样 Tigger 的 AST 也是简单的线性表。其内容可以通过宏展开的方式翻译到 Tigger 语法或者 RISC-V 语法[2]。寄存器分配的重任实际上落到了 tigger generator 头上，而 tigger optimizer 的工作包括和 Eeyore 类似的优化。

1.3 报错

目前实现的报错可分为以下三个层次：

- **非法的命令行调用和无效参数**
- **语法错误**。这里我们仅使用 bison 提供的默认报错机制，不再添加 error production 或者针对常见语法错误进行补救。
- **语义错误**。所有基本的语义错误均被涵盖（除未检查是否函数的每个可能分支上的返回类型均与声明类型匹配）。具体类型请见节2.3。

报错界面借鉴了 clang 风格。如下是两个简单示例。

Source file *Error Report*

```
int a = 2;
int b = 2;

int main(){
  int a = 0;
  a = a + 1;
  return;
}
```

```
7:3: error: non-void function 'main' should return a value
  return;
  ^~~~~~
note: declaration was here at line 4, column 5:
int main(){
    ^~~~
SysY compiler: 1 errors generated.
compilation terminated.
```

```
int a = 2;

int main(){
  int b[a
        + 10]
  = {};
  return 0;
}
```

```
4:9: error: constant expression is expected, but got a non-
const one
  int b[a
        ^
        + 10];
          ~
SysY compiler: 1 errors generated.
compilation terminated.
```

图 2: Tiny 报错示例

错误类型和错误范围需要在 AST 中动态维护。值得注意的是因为错误可以跨行，因此其范围至少需要四个参数描述：⟨bgnlno, bgnpos, endlno, endpos⟩，即起始行号、起始位置、终止行号、终止位置。第二个示例展示了错误跨行的情况。

[2]其中仍有优化空间，但我们仍采取最简单的做法，因为最有效的优化未必在此。

2 前端

2.1 作用域 & 符号表

作用域部分的代码集成在 [front/global.cpp:27-28] 中，符号表的实现在 [front/symtab.cpp] 中。

SysY 的作用域实际只有两种。从语法的角度看，一种是普通的 $\underbrace{\{...\}}$，另一种则是函数定义中的 `RetType FuncName` $\underbrace{\texttt{(ParamsList)}\{...\}}$。一般看来，作用域作为 Token 属性是 L-attributed 的，但是这里很容易修改文法将其变为 R-attributed，从而适合自底向上的情况。每当进入一个子作用域，则向栈中压入一个

Grammar Rules	Syntax-Directed Definition
`Block := '{' NewScope Body '}'`	Exit Current Scope
`FunDef := FunHead '(' NewScope [FunFparams] ')' '{' Body '}'`	Exit Current Scope
`NewScope` := ϵ	Enter a New Scope

表 1: 修改后的 R-attributed 语法制导定义

空符号表。每当退出一个作用域，则弹出当前符号表。为了以后的方便，所有的符号表我们都保留，进而它形成了树结构。符号表需要支持的基本操作有：

- 插入新表项。
- 查找当前作用域内某变量是否已定义。
- 沿根路径查找某变量定义位置，返回定义时信息。

一个合适的选择是哈希表，从而符号表整体是一个哈希表形成的树。期望的查找时间是 $O(h)$ 的，其中 h 是作用域的深度，而期望的插入时间是 $O(1)$ 的。

对于变量的符号表，我们考虑的问题是如何区分标量 (scalar) 和数组 (array)。在我的设计中，以如下规则区分。这些规则为我后面生成代码带来了极大的便利，而且比避免了数组偏移量的重复计算。注意，不同类型有着不同的表示。

Data Type	Vector Representation	Examples
`int`	$\langle 1\rangle$	`int` $\mapsto \langle 1\rangle$
`int` $[c_k]...[c_0]$	$\langle 1, c_0, \cdots, \prod_{i=0}^{k} c_i\rangle$	`int [1]` $\mapsto \langle 1,1\rangle$; `int [2][3]` $\mapsto \langle 1,3,6\rangle$
`int []` $[c_k]...[c_0]$	$\langle 1, c_0, \cdots, \prod_{i=0}^{k} c_i, 0\rangle$	`int []` $\mapsto \langle 1,0\rangle$; `int [][1]` $\mapsto \langle 1,1,0\rangle$; `int [][3]` $\mapsto \langle 1,3,0\rangle$

表 2: 变量在符号表中的统一表示形式

对于函数的符号表，其中开一向量记录其形式参数所在的符号表指针。这样，检查函数调用时参数个数、常量属性和类型是否匹配时就可以先找到函数定义的符号表条目，进而找到形式参数的条目。

有趣的一点是，我们始终在 AST 中维护语法元素的文件位置（用于报错），同时对于有初始值的变量在其 AST 中记录对应的初值（即表达式的 AST 的根节点）。因此标志函数或者变量定义的 AST 节点需要有指针指向对应的符号表条目。同时符号表条目也需要有指针指回标志其定义的 AST 节点。

2.1.1 库函数

为了对库函数的解析能正常进行,我们预先在符号表中插入对它们的声明。同时为其分配一个悬空的 AST 节点[3]。注意,这些节点并不在解析 SysY 程序产生的 AST 中。它们的作用仅仅是制造声明已存在的假象,以便和已有的符号解析与函数参数检查机制适配。

2.2 SysY AST

基类 `BaseAST` 实现在 [front/baseast.cpp] 中,派生类的代码实现在 [front/ast.cpp] 中。

SysY 节点的异质性表明我们采用基类加继承的方式来实现。`class BaseAST`是公共的基类, 其中记录该节点的类型(方便 down casting),以及该节点对应的源码中的四元组范围。由它派生出的类总结如下:

AST Type	Functions	Specs
`BinaryAST`	A Binary / Logic Op	Data Type
`UnaryAST`	A Unary Op	Data Type
`FunCallAST`	Function Call	Data Type
`LvalAST`	Ref of a variable	Data Type
`DeclAST`	Def of a Variable	Symtab Type
`BlockAST`	Statements Within a Block	Vector Type
`FunDefAST`	Def of a Function	Symtab Type

AST Type	Functions	Specs
`AssignAST`	Assignment	Copy Type
`IfAST`	If Structs	Control Type
`WhileAST`	While Structs	Control Type
`BreakAST`	Break Statements	Control Type
`ContAST`	Continue Statements	Control Type
`RetAST`	Return Statements	Control Type

表 3: SysY AST Node Type

具体的类型规约 (Specifications) 如下:

Data Type 代表一个数据。其属性包括 `bool isevaled`, 即是否已尝试编译时求值以及 `bool isconst`, 即是否是编译时常量。其方法包括 `void Eval()`, 即在编译时尝试求值。求值结果写入`int cval`字段。对于`class LvalAST`[4] ,其还有`bool isleft`字段表明是否是一个真正的左值。

Symtab Type 代表一个符号表引用。对于变量,其还保存初始值的字典。该字典的每个键值对形如 ⟨`offset`, `AST`⟩。其中`offset`是已将多维数组展平 (Flattened) 后计算的常量偏移, `AST`表明该初始值对应的表达式树的根节点。对于函数,其孩子是函数体对应的`BlockAST`。

Vector Type 即 AST 的有序列表。用于收集一系列 AST 子树。

Copy Type 表明是一个赋值。

Control Type 表明一个控制结构。对于`BreakAST, ContAST, RetAST`, 其`ASTptr`[5] `back`字段指明所在的逻辑流的下一个位置。

实际上 AST 还有两个很有用的工具类 (Helper Class), 它们仅仅用于在建树时将高级结构翻译到低级结构。当该翻译完成时,它将自己的数据交给对应的普通 AST 节点,然后其资源就被释放,不再存在。

[3]当然也需要为它们的形式参数分配符号表表项和 AST 节点。它们的符号表表项实际在深度是 1 的层上。

[4]这个名字具有一定误导性,实际上叫做`RefAST`可能更为合适。

[5]`using ASTptr = BaseAST *;` 即 AST 类的 generic pointer。

`VecAST` Specifications: Vector Type

Functions: 临时收集序列化的 AST。例如在声明函数的形参 / 实参列表时，或者收集列表初始化的元素，或者收集多维的数组下标时。

`ListTree` Specifications: Special Type

Functions: 建立列表初始化的树结构。检查它是否与变量的类型定义匹配。一旦匹配，将嵌套的`{}`翻译成一维的偏移。例如`int a[3][2] =`$\{\texttt{Exp}_0, \texttt{Exp}_1, \{\}, \texttt{Exp}_4\}$被翻译为$\{\langle 0, \texttt{Exp}_0\rangle, \langle 1, \texttt{Exp}_1\rangle, \langle 4, \texttt{Exp}_4\rangle\}$。其余缺省下标皆默认其值为 0。

2.2.1 编译时求值

一旦规约到 Exp 这个 SysY 语法元素，就尝试对表达式进行递归求值。是否能够求值是编译时定值，因此通过`bool isevaled`标记的技术可以使得由此带来的总额外开销不超过遍历整棵 AST 的时间。常见的可求值情况是常量引用，或者运算数都可求值。求完值后，该树即可释放。

2.2.2 代数恒等变形

SysY 语法的书写使得对某个运算符的规约要么左结合，要么右结合。以+为例，Tiny 中它左结合。如果一个程序员很笨拙地书写`a+1+2+3`，假设`a`不是常量，那么 Tiny 会错失掉编译时简化计算的机会。因此可以对表达式的子树进行旋转，如图 3 所示。另外，对于复合的单目运算符，也有代数化简的空间。图 4 展示

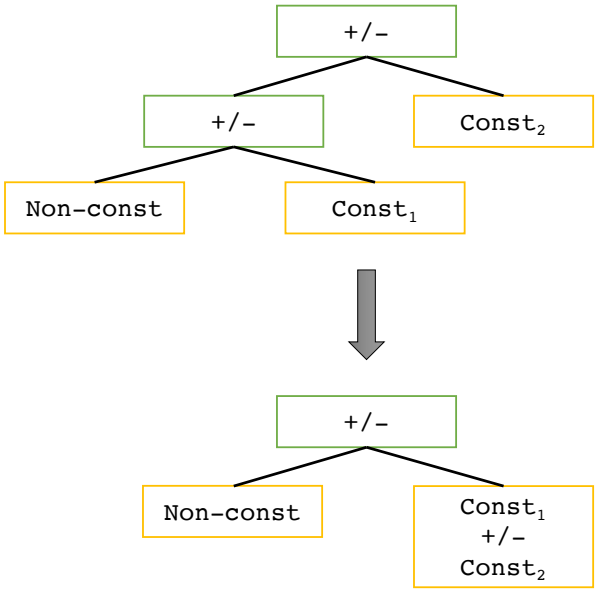

图 3: 对子树进行旋转

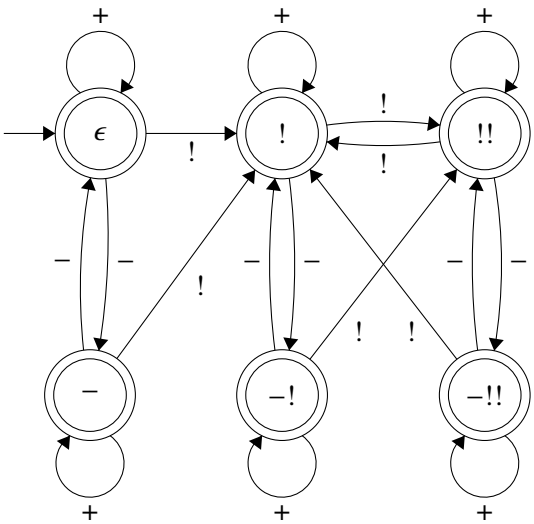

图 4: 合并单目操作

了这种情况，其中自动机的标号是单目运算符栈中的符号，栈顶在左侧。

2.3 编译时异常

异常模块实现在 [front/error.cpp] 中。

我们一共提供三种类型的编译时异常 (Compiling-time Exception)。

Error 表示严重的错误。检测到某些错误时,编译器可以暂时忽略继续编译。而某些错误将使得它立即抛出异常信息退出。

Warning 表示代码中存在潜在的隐患。源程序可以正常编译,但可能有非期望或者未定义的行为。

Exception 编译器保留给自己使用。主要用于编写 Tiny 时的本地调试。一旦触发,表明 Tiny 触发了一个不正确的行为。该异常不应当在使用 Tiny 编译源程序时触发。

对于每种类型的异常,我们像 Unix 类操作系统那样提供一个统一的异常入口 (Entry Point)。对于特定类型的以异常, 其处理手柄 (Handler) 注册在一个异常向量表中。所有的处理手柄具有原型`void (*)(const std::string &, ASTptr)`。一旦异常被检测到, 调用者主动按类型调用异常入口函数并传入错误的地址和提示信息 (即对应原型的两个参数)。入口函数将根据异常号选择正确的处理函数。大部分处理函数会向标准错误打印一个用户友好的详细的错误信息,然后返回到调用位置。少数处理函数会在打印错误信息后直接退出。目前一共有 29 种 Error,2 种 Warning,7 种 Exception。最重要的一些异常展示在表 4 中。

Exception Type	Descriptions	Return Behavior
`Error::Arrayinitmismatch`	不匹配的类型赋值	Always Return
`Error::Arraynegindex`	负的数组索引	Always Return
`Error::Arraybigindex`	数组索引上越界	Always Return
`Error::Identdeclared`	重复定义的标识符	Exit Immediately
`Error::Identnotdeclared`	未定义的标识符	Exit Immediately
`Error::Unexpectedfuncall`	以变量形式使用函数	Return On Revocery
`Error::Varnotcallable`	以函数形式调用变量	Return On Recovery
`Error::Toomanyindices`	索引数量过多	Return On Recovery
`Error::Pointerdimmismatch`	实参中的指针类型失配	Exit Immediately
`Error::Nowhiletoescape`	悬空的`break`或者`continue`	Always Return
`Error::Notanumber`	非数值类型,例如`void`	Always Return
`Error::Assignconstant`	非定义时的常量赋值	Always Return
`Error::Constarrtoptr`	向函数传入常量数组的指针	Always Return
`Error::Nomain`	找不到`main`函数	Always Return
`Warning::Intoverflow`	整型溢出	Always Return
`Exception::Unexpectedbasecall`	不期望的`class BaseAST`虚函数调用	Always Return

表 4: Tiny 异常举隅

2.4 Eeyore AST

Eeyore 的 AST 定义在 [front/eerep.hpp] 中。其 Dump 函数实现在 [front/eerep.cpp] 中。

由于 Eeyore 的文法相当简单, 不同类型语句间的同质性内容较多, 因此不必使用复杂的基类-子类派生。如前述所言,其操作只有前向或者逆向的增删,因此应当使用链表存储该 AST。

2.5 自顶向下的增量式 Eeyore AST 生成

本部分代码封装在 [front/geneey.cpp] 中。

出于易维护性和简介性的考虑，我们选择在建好 SysY AST 后自顶向下地增量式生成 Eeyore AST。我们生成的 Eeyore 具有如下性质，它们在优化中是极为有用的。

- 关于临时变量是静态单赋值 (Single Static Assignment, SSA)。
- 任意临时变量的活跃区间恰好只在一个基本块中。

最典型的例子是一个名为 Exp 的 SysY 语法元素，由其被使用的上下文决定最简的 Eeyore 翻译方案，如下例所示。简单起见，我们仅使用当前节点的信息来为它生成代码，而不使用上下文的信息。由此造成的冗余是高度局部的，它将被 Eeyore optimizer 消除。

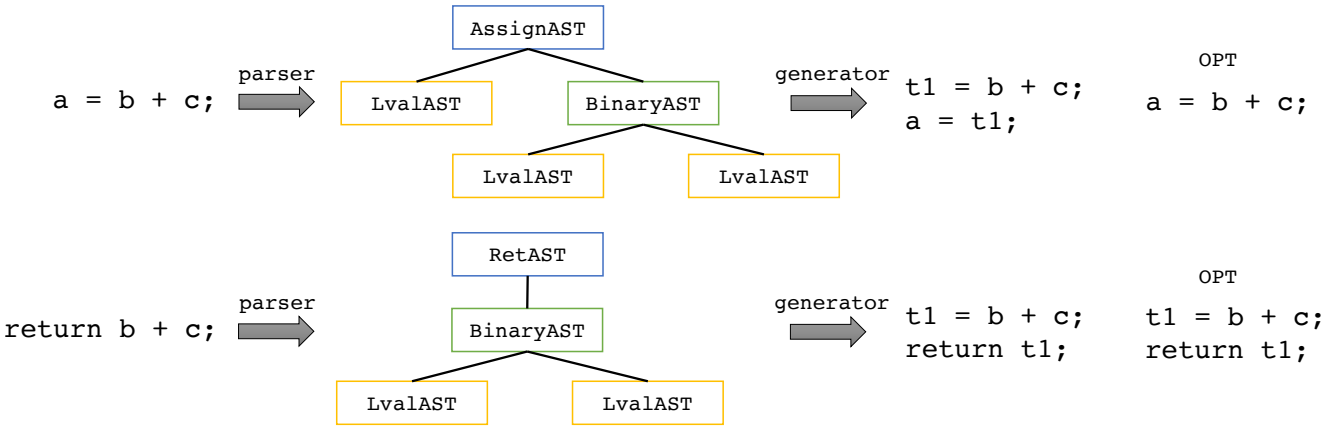

图 5: 是否需要引入临时变量由表达式所在的上下文决定

2.5.1 短路表达式

由于是自顶向下，L-attributed SDD 可以被很好地支持。因此我们无需回填，直接增量生成即可。这一部分的代码一经理解其实相当机械，参见 [front/geneey.cpp:153-174]。

2.5.2 布尔表达式

当一个原本期待是布尔表达式的地方表达式以算术运算或者函数调用为根，那么必须特别地为此生成一个以 `!= 0` 为条件的条件跳转。何时判断需要跳转是一个很容易解决的问题。一种解决方法是生成 Cond 的 Eeyore 语句块后按 `&&` 和 `||` 对表达式进行切分，逐个在根子式的位置添加条件跳转。但它依赖 SysY 的一个落后的语法特性——短路表达式是不能嵌套的。我们采用的方法是在生成时增加一个`bool endpoint`的标志位。它由 Cond 设置，并由根子式清除。该方法适用于哪怕嵌套的短路表达式。具体实现请见 [front/geneey.cpp:40]。

2.5.3 全局变量初始化

一个 SysY 全局变量的初始值当然可以是函数调用的返回值，但是 Eeyore 的文法却不支持函数体外的函数调用。为此，我们统一地将全局变量的初始化移动到 `main` 函数体中。这可以通过在遍历 AST 前首先遍历深度为 1 的 `DeclAST` 节点做到。解析到 `main` 函数时，将这些预先生成的语句直接插入即可。

2.6 Eeyore 优化

本部分代码封装在 [front/optimee.cpp] 中。根据 Eeyore 生成器之性质，实际在建立基本块之前可以有许多高度局部但有效的优化，它们包括表达式的化简（例如数组下标运算时常会在最低位产生 `* 1` 的操作），常量替换（例如某些临时变量初始值即是常量，由 SSA 之性质其每次出现均可由常值替换），以及 Cond 的合并（AST 生成 Eeyore 时实际每次是将 Cond 对应的 Exp 值保存在一个临时变量中，再判断该临时变量是否是零。如果表达式的根恰好是逻辑运算符，那么应消去该临时变量）。这部分的代码统一实现在函数 `void NaiceEEOpt()` [front/optimee.cpp:394-494] 中。建立基本块和 CFG 之后，我们实现了公共子表达式消除 (Common Subexpression Elimination)、复制传播 (Copy Propagation)、死代码消除 (Dead Code Elimination)。其中前两者基于 Available Expression Anaysis，第三者基于 Liveness Analysis。它们分开被实现，统一地在函数 `EEptr InFuncBBEE(EEptr)` [front/optimee.cpp:52-128] 中被交错地调用。

2.6.1 CFG 之前：PeepHole 优化

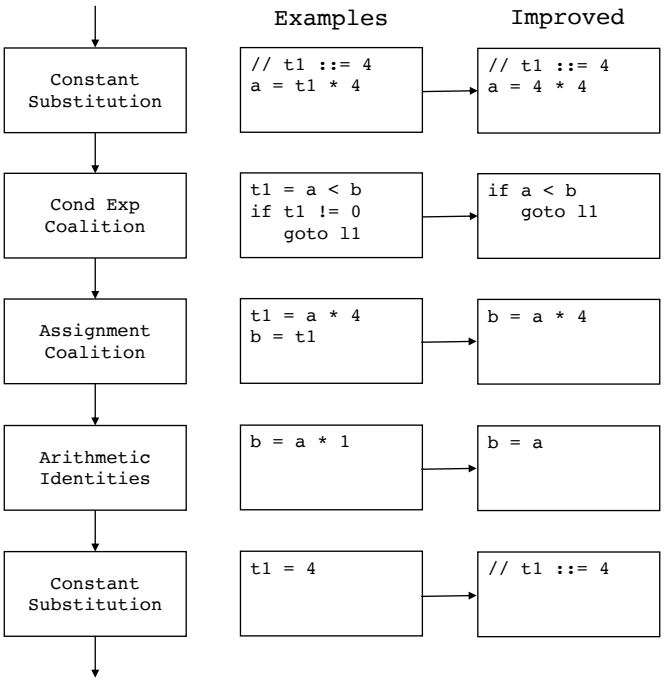

图 6: PeepHole 优化流程

细致的考虑可以发现上述提到的一切基本块之前的优化可以集中在 one pass 中实现。为了使优化器不错过优化的机会，每种优化的先后次序需要加以限制。一种可能的流程图如图 6 所示。这里我们省略分析，请读者自明。

2.6.2 建立 CFG

本部分代码位于 [front/optimee.cpp:269-355] 中。容易看出，由于 Eeyore 中跳转目前需要显式的标签，基本块的划分可以通过一遍顺序的扫描完成。一个基本块的范围由以下若干种情况标识。

基本块开始（当且仅当满足下述之一）：

- 前一条语句是函数头
- 前一条语句是跳转或者返回语句
- 本语句声明一个标签

基本块结束（当且仅当满足下述之一）：

- 后一条语句是函数尾
- 后一条语句是标签
- 本语句是跳转或者返回语句

根据基本块的最后一条语句，很容易构建出 CFG。注意，每个基本块的出度最大是 2。

2.6.3 CFG 之后：分析与局部优化

本部分代码请见 [front/optimee.cpp:39-266]。由于目前只实现了局部优化，实际能达到的效果有限。

可用表达式分析 此分析将帮助完成公共子表达式优化和复制传播。注意，分析和优化是同步进行的——在每一个程序点，首先检查是否有可用的表达式进行优化，然后对该语句（优化前的形式）分析，更新可用的表达式列表。我们需要执行三种操作：

- 给定一个代表表达式的 EEyore 节点[6]，查找当前可用的表达式列表有无该右值表达式。
- 加入新的可用表达式。
- 设置所有包含给定变量的表达式为不可用。

基于此，我们并不显式地维护当前可用表达式的列表。相反，我们在 Eeyore 节点中维护一个 `bool available` 的位区分目前该表达式是否可用。一个哈希表 `eeexpright` 用于存放右值表达式到 Eeyore AST 节点的拉链，而另一个哈希表 `eeexpleft` 存放变量到所有包含它的右值表达式所在的 Eeyore AST 的拉链。拉链在此处的含义是开散列，因为一个键可以有多个 Eeyore AST 对应。标记失效时，我们并不需要去 `eeexpright` 中删除所有关联的表达式，而是在使用时通过 `available` 字段区分 (即哈希表中的值可能是假的)。有几个特殊情形需要注意。

- 如果表达式中左值变量也出现在右值表达式中，那么该表达式在以后的程序点失效。
  e.g. `a = b + c` 表明 `b + c` 在本句后的程序点有效，同时所有包含 `a` 的表达式失效。
  e.g. `a = a + b` 表明 `a + b` 在本句后的程序点为失效。
- 跨函数调用后所有包含全局变量的、访问全局数组元素的、访问传入函数参数的局部数组的元素的表达式失效。
  e.g. `param T2 \param T3 \call f_bar` 假设 `var T0` 和 `var 40 T1` 是全局的，而 `var T2` 和 `var 40 T3` 是局部的。那么函数调用后的程序点所有形如 `T1[...]` 和 `T3[...]` 的表达式失效。另外，包含 `T0` 的表达式亦失效，但包含 `T2` 的表达式不会因此失效。

类似可完成复制传播。此处省略分析，具体请见本节的代码片段。

[6]一共有四种情况，`EERecord::Binary`, `EERecord::Unary`, `EERecord::Copy`, `EERecord::RArr`。

活性分析 同前，分析和死代码消除可以同步完成。由于缺少全局信息，在基本块的出口，需要假定基本块中所有定义的变量存活。另外，仅针对标量类型进行此分析。经过函数调用，总认为全局变量以及函数传入的局部变量存活。具体请见本节代码片段。

交替执行 目前的实现中，每个基本块交错地进行以上两种分析各两次。后期可以引入启发函数。

3 后端

3.1 Tigger 生成和优化

[back/tgrep.cpp] 包含了从 Tigger AST 导出目标代码的例程，[back/gentg.cpp] 包含了从 Eeyore AST 生成 Tigger AST 的例程，而 [back/optimtg.cpp] 封装了针对 Tigger AST 的优化代码。

从 Eeyore 到 Tigger 仅有的工作是寄存器分配和过程时栈的分配。这两个问题本质上又是相互联系的。尽管 Tigger 只是 RISC-V 的粗糙抽象，使得指令选择不再是重要的问题，但是从 Tigger 翻译到 RISC-V 时一个有效的优化仍然是指令选择。这一部分我们完成得非常粗糙，寄存器分配采取了最简单的 load-store 算法[7]，同时关于 Tigger 的优化也局限于消除局部冗余的 load 和 store 指令。关于 Tigger 生成 RISC-V 的部分，采用的是最基本的宏展开翻译策略。

4 未来的工作

1. Eeyore 的全局分析和优化。
2. 变量活跃期分析以及图染色的寄存器分配策略。
3. 简单的跨过程分析。

[7]不分配也是一种分配。