编译实习实验报告

作者：陈梓立(wander4096@gmail.com) 学号：1500012726

编译器概述

基本功能

- 核心功能: 把合法的 MiniC 代码编译为 RISC-V 汇编代码, 使用 RISC-V 的汇编器生成机器代码, 可运行在 RISC-V qemu 模拟器上.

- 错误报告：在将 MiniC 翻译到中间代码 Eeyore 前做类型检查，能报告出表达式中的类型错误；调用函数前检查签名，能报告出参数不匹配(包括数量，类型)的函数调用.

个性特点

1. 基本功能提到的错误报告功能.
2. 在 MiniC Base 集上，还支持以下的语法.
    1. 沉没表达式，即表达式的返回值可以不被接收. 例如，`42 + 34;`，`func(arg);`等. 这一点解放了函数调用必须以 `var = func(args);` 的形式出现的约束.
    2. 复杂的函数调用. 例如，`func(x + 1, x + y % 3);`，即函数的参数可以是表达式，特别的，支持函数嵌套调用，`f(g(x))`.
    3. 自由的外层定义. main 函数不必定义在最后，所有的外层定义(函数定义，函数声明，变量声明)是平等的.
    4. 连续赋值及赋值表达式. `a = b = c = 42;` 或 `a = b[2] = c + d;`，以及 `a = b + (c = 4)`. 因为赋值表达式不依赖于赋值语句.
3. Eeyore 变量实现为 `t` 类，`g` 类和 `p` 类，`p` 类仍为函数参数，`t` 类为局部变量，不再区分 Eeyore 中的辅助临时变量和原生变量，`g` 类为全局变量. 这是因为全局变量的区别比起原生变量更为重要.
4. 在中间代码处理的步骤，进行了额外的处理/优化.
    1. 标签与跳转压缩. 在翻译 if 和 while 的过程中，可能出现连续的标签，可以压缩为一个标签；可能出现跳转到某一标签后下一句是跳转到另一个标签，可以压缩为跳转到最终目的地.
    2. 死代码消除. 应用激进的死代码消除策略，初始假定只有第一条代码可达，往后标记所有可达语句，翻译为汇编代码时无视不可达代码. 这一步没有进行 Guard 判定，即语义上不可达的代码无法识别，只是简单的消除在语法上不可达的代码.
    3. 常量传播和折叠. 压缩表达式，使得一元表达式不包含数字字面量(若有，转换为赋值)，二元表达式最多有一个数字字面量.
    4. 选择 Callee. 在线性扫描活性分析的基础上，将寄存器划分为 Caller Save 和 Callee Save 两部分，对于在函数调用时活跃的变量，尽量将其分配到 Callee Save 的寄存器中，以此减少甚至去除函数调用时的 Caller Save 工作.
    5. 指令选择. 对于数组操作中常见的 `*4` 操作，改用左移操作代替，减轻运算强度.
5. 跳过了 Tigger 代码的生成过程. 实际上，Tigger 代码与 RISC-V 代码的区别很小，且全局变量的设计比较复杂. 在汇编代码中，将全局标量设计为单元素数组.
6. 主要使用 Perl 6 语言作为开发语言. 一方面可以使用更高阶的抽象，另一方面 Perl 6 原生的 grammar 语法可以方便地解析文本，比 lex/yacc 更轻松一点.

实验假设和未实现的功能

1. 所有数据只有 int 和 int 数组这两种类型.
2. 所有涉及的数字值绝对值都较小，即不考虑运算时溢出的问题.
3. 函数内的函数声明语法不支持.
4. 逻辑判断语句没有实现为短路.

编译器设计与实现

工具软件的介绍

主要使用 Perl 6 语言作为开发语言. 原因有两个, 主要的原因是它原生的 grammar 语法可以方便地解析文本, 另外的原因是它提供了更多可用的高阶抽象. 关于 grammar 语法和使用它做 parsing 的材料, 可以参考 Moritz Lenz 的 Parsing with Perl 6 Regexes and Grammars, A Recursive Descent into Parsing. 在下面的报告中, 涉及到 Perl 6 语法时会相应地做简要的介绍.

特别地, 由于 Perl 6 是一门正在活跃开发的语言, 并且由于它希望支持庞大的功能集的原因, 它的迭代速度非常的快, 请保证在和下面给出的版本相同的版本号的环境下测试和使用实验代码, 如果环境不同可能出现各种问题.

```
$ perl6 -v
This is Rakudo version 2017.11-44-g4a32089fd built on MoarVM version
2017.11-20-gd23f5ca1
implementing Perl 6.c.
```

MiniC 的解析

词法分析

在第一次编写 Eeyore 生成代码时, 我是直接使用 grammar 将输入文本完整匹配来解析 MiniC 的. 但是由于 Perl 6 使用 LTM 算法(Longest Token Match), 并且在用户自定义空白字符集 `<ws>` 上存在 BUG, 为了正确匹配需要非常多的补丁代码, 因此增加了一个 Lexing 的过程, 去除 MiniC 代码中的注释, 并将字符流转化为标准的格式. 代码如下(Lexer.pm6): (`subst` 函数即字符串替换函数, `unit`, `module`, `is export` 等内容与模块化相关, 主要用于支持在其他代码中调用这段代码)

```
#!/usr/bin/env perl6

unit module Lexer;

my token comment {
  | '//'.*?\n\s*
  | '/*'.*?'*/'\s*
}

my token whiteSpace {
  <!ww> \s* <comment>*
}

our $TOKENS is export =
  $*IN.slurp.subst(/<whiteSpace>/, '$', :g)
            .subst(/\$+/, '$', :g)
            .subst(/^\$/, '')
            ;
```

词法分析的典型结果如下:

```
// test/cmt.c

/* comment
 * comment
 */
int getint(); /* comment */
int putint(int x); // comment
int putchar( int   x); /* comment */
/* /*
  comment // xxx
  ?? */
//*/
int main(/*nothing*/)
{
    int a;
    a    = getint();//
    //a = getint()+1;
    int b;///*
    b=getint(     );/*inline comment*/ putint(a + b)/*xxx*/;
putchar(10);// putint(1);
    return 0;
}
--------------------
# OUTPUT:

int$getint$($)$;$int$putint$($int$x$)$;$int$putchar$($int$x$)$;$int$main
$($)${$int$a$;$a$=$getint$($)$;$int$b$;$b$=$getint$($)$;$putint$($a$+$b$
)$;$putchar$($10$)$;$return$0$;$}$
```

在下面语法分析的阶段中，使用了如下的基本 Token 集合：

```
# ====================
  # Basic Tokens
  # ====================
  token _DEBUG_BASIC_TOKEN {
    [
    |<IF>|<GOTO>|<END>|<RETURN>|<CALL>|<PARAM>|<VAR>
    |<LBRACK>|<RBRACK>|<COLON>
    |<ASSIGN>
    |<OR>|<AND>
    |<EQ>|<NE>
    |<LT>|<GT>
    |<ADD>|<SUB>
    |<MUL>|<DIV>|<MOD>
    |<NEG>|<NOT>
    |<FUNCTION>|<VARIABLE>|<LABEL>
    |<INTEGER>|<NEWLINE>
    ]+
```

```
  }
  token IF     { 'if'<DELIM> }
  token GOTO   { 'goto'<DELIM> }
  token END    { 'end'<DELIM> }
  token RETURN { 'return'<DELIM> }
  token CALL   { 'call'<DELIM> }
  token PARAM  { 'param'<DELIM> }
  token VAR    { 'var'<DELIM>}
  token LBRACK { '['<DELIM> }
  token RBRACK { ']'<DELIM> }
  token COLON  { ':'<DELIM> }
  token ASSIGN { '='<DELIM> { make '=' } }
  token OR     { '|'<DELIM>'|'<DELIM> { make '||' } }
  token AND    { '&'<DELIM>'&'<DELIM> { make '&&' } }
  token EQ     { '='<DELIM>'='<DELIM> { make '==' } }
  token NE     { '!'<DELIM>'='<DELIM> { make '!=' } }
  token LT     { '<'<DELIM> { make '<' } }
  token GT     { '>'<DELIM> { make '>' } }
  token ADD    { '+'<DELIM> { make '+' } }
  token SUB    { '-'<DELIM> { make '-' } }
  token MUL    { '*'<DELIM> { make '*' } }
  token DIV    { '/'<DELIM> { make '/' } }
  token MOD    { '%'<DELIM> { make '%' } }
  token NEG    { '-'<DELIM> { make '-' } }
  token NOT    { '!'<DELIM> { make '!' } }
  token FUNCTION { (f_<[_A..Za..z]><[_A..Za..z0..9]>*) <DELIM> { make
$0.Str } }
  token VARIABLE { (<[tpg]><[0..9]>+) <DELIM> { make $0.Str } }
  token LABEL    { (l<[0..9]>+) <DELIM> { make $0.Str } }
  token INTEGER  { (\-?<[0..9]>+) <DELIM> { make $0.Str } }
  token NEWLINE  { \n <DELIM> }
  token DELIM { " "? }
```

其中 `_DEBUG_BASIC_TOKEN` 用于调试，其他为基本 Token，`{ make … }` 为解析时附加的动作，用于将值与这个 AST 节点关联起来，类似于 yacc 中的 `{ $$ = … }`.

核心过程：语法分析与语法制导翻译

实现的 MiniC 文法在 Base 集上有所拓展和转化，主要包括编译器概述一节个性特点中提到的支持沉没表达式和复杂的函数调用. 此外，不支持函数内的函数声明语法，且 main 函数与其他函数在语法分析时无异. 下面介绍实现的 MiniC 语法以及语法制导翻译方案.

首先介绍符号表的设计(MiniC.p6). 这里只给出类的结构，实现内容代码在涉及时给出.

```
class SymbolTable {
  has @!scopes = {}, ;

  method enterScope() { @!scopes.push({}); }
  method leaveScope() { @!scopes.pop();}
```

```
  method declare(Str $var, %info);
  method getInfo(Str $var);

  # ===============
  # Check declare
  # ===============
  method !checkReserved(Str $var);
  method !checkDefined($var, %info);
  method !dieDefinedVariable($var, %info);
  method !checkDefinedFunction($var, %info);

  method _getScope();
  method _getScopes();
}
```

实现上，将符号表实现为若干个字典的列表，列表可以想象成栈，除了可以访问任意列表元素．栈顶的字典即为当前作用域，往下依次是上一层作用域．字典的结构为，键为标识符名称，包括变量名和函数名，值为一个信息映射，包含标识符关联的所有信息，包括它在 Eeyore 中对应的 `resolveId`，对于数组，数组的长度 `size`，对于函数，函数的参数列表信息以及是否定义过（我们允许多次相同签名的签名，但只能有一次定义）．

技术上，附加实现了一个匿名的 `Counter` 类，用于计数出现的变量个数，这是因为 Eeyore 中的变量约定为特殊的抬头后跟一个数字编号．同时，它还能计数标签的编号．

```
my $counter = class {
  has $!labelCounter = 0;
  has $!globalCounter = 0;
  has $!localCounter = 0;

  method yieldLabel() {
    my $res = "l$!labelCounter";
    $!labelCounter += 1;
    return $res;
  }
  method yieldGlobal() {
    my $res = "g$!globalCounter";
    $!globalCounter += 1;
    return $res;
  }
  method yieldLocal() {
    my $res = "t$!localCounter";
    $!localCounter += 1;
    return $res;
  }
}.new;
```

> Perl 6 的面向对象系统继承自 Perl 中 CPAN 上的模块 Moose.

> 其中，使用 `has` 关键字定义实例变量，`method` 关键字定义方法，第二魔符(tigil，相对比 sigil) `!` 指示私有变量和私有方法.

下面依次介绍 MiniC 语法的非终结符.

顶层非终结符(TOP)

```
token TOP { # translateUnit
    :my $*ST = SymbolTable.new;
    <externalDeclaration>+
  }
```

本质是将编译单元 `translateUnit` 作为顶层的非终结符展开，由于 Perl 6 默认的 grammar 顶层符号为 `TOP`，也就叫 `TOP` 了，实际上可以叫 `translateUnit` 并在解析时指定顶层非终结符.

内容上，编译单元由一个或多个外层定义构成，实际上，C 程序的最外层(全局)就是一组声明和定义. 特别地，不允许内容为空的编译单元.

这里有一句 `:my $*ST = SymbolTable.new;`，定义了一个与解析过程关联的动态作用域变量，符号表 `$*ST`，使用动态作用域变量，避免维护一个全局变量，并且非常契合解析时一层层调用下一个 token 函数时的运行状态.

外层定义(externalDeclaration)

```
token externalDeclaration {
    | <functionDeclaration>
    | <functionDefinition>
    | <variableDefinition> {
      my %info = $<variableDefinition>.made.Hash;
      %info<resolvedId> = $counter.yieldGlobal;
      $*ST.declare(
        $<variableDefinition>.made<id>,
        %info,
      );
      given %info<type> {
        when 'Scalar' { say "var {%info<resolvedId>}" }
        when 'Array'  { say "var {%info<size> * 4} {%info<resolvedId>}"
}
      }
    }
  }
```

一个外层定义是函数定义，函数声明或变量声明其中一种. 这里对变量声明附加动作以在符号表中注册全局变量，这是因为在变量声明中不好知道自己是在全局环境还是局部环境.

> given/when 语法类似于 switch/case.
>
> .made 用于提取与 AST 节点关联的值，类似于 yacc 中的 $1 等.

函数声明(functionDeclaration)

```
  token functionDeclaration {
    | <INT> <IDENTIFIER> <LPAREN> [<variableDeclaration>+ % <COMMA>]?
<RPAREN> <SEMI>
      :my %info; {
        %info<id> = $<IDENTIFIER>.made;
        %info<resolvedId> = "f_{%info<id>}";
        %info<isDefine> = False;
        %info<type> = 'Function';
        %info<typeList> = $<variableDeclaration>.Array.map(*.made.
<type>);
        $*ST.declare(%info<id>, %info);
      }
  }
```

函数返回值类型硬编码为 `<INT>`，`<variableDeclaration>` 是一个形如 `int id` 的变量声明，附加的动作为在符号表中注册函数，注意信息中标记 `%info<isDefine> = False;`，这样就将函数定义和声明区分开来，前面提到，我们允许多次相同签名的签名，但只能有一次定义.

> Perl 6 中的正则表达式语法 `<variableDeclaration>+ % <COMMA>` 表示一组 `<variableDeclaration>` 被 token `<COMMA>` 划分，典型的文本为 `int a, int b, int c`.

函数定义(functionDefinition)

```
token functionDefinition {
    | <INT> <IDENTIFIER> <LPAREN> [<variableDeclaration>+ % <COMMA>]?
<RPAREN>
      :my %info; {
        %info<id> = $<IDENTIFIER>.made;
        %info<resolvedId> = "f_{%info<id>}";
        %info<isDefine> = True;
        %info<type> = 'Function';
        %info<typeList> = $<variableDeclaration>.Array.map(*.made.
<type>);
        $*ST.declare(%info<id>, %info);

        say "{%info<resolvedId>} [{%info<typeList>.elems}]";

        $*ST.enterScope;
        for $<variableDeclaration>.Array Z (0...*) {
          my %parameterInfo = .[0].made;
          %parameterInfo<resolvedId> = "p{.[1]}";
          $*ST.declare(%parameterInfo<id>, %parameterInfo);
        }
      } <block> {
        $*ST.leaveScope;
        say "end {%info<resolvedId>}";
      }
  }
```

头部类似于函数声明. 在注册函数是标记这是一个函数定义，为参数定义建立一个新的作用域并注册参数，在结束函数体时，记得离开参数这一层作用域. 中间包括产生相应的 Eeyore 代码. 函数包括函数体(一个语句块 `<block>`).

变量声明(variableDefinition)

```
token variableDefinition {
    | <variableDeclaration> <SEMI> {
      make $<variableDeclaration>.made;
    }
  }
```

通过结尾的分号 `<SEMI>` 判断 `<variableDeclaration>` 是一个变量声明而不是函数签名中的参数声明，语义动作仅仅是简单的传递 `<variableDeclaration>` 中的信息.

变量声明'(variableDeclaration)

```
token variableDeclaration {
    | <INT> <IDENTIFIER> <LBRACK> <INTEGER> <RBRACK> {
      make %(
        id => $<IDENTIFIER>.made,
        size => $<INTEGER>.made,
        type => 'Array',
      );
    }
    | <INT> <IDENTIFIER> {
      make %(
        id => $<IDENTIFIER>.made,
        type => 'Scalar',
      );
    }
  }
```

区分为定义变量和定义数组，相应的记录定义信息.

语句块(block)

```
token block {
    | <LBRACE> { $*ST.enterScope; } <statement>* { $*ST.leaveScope; }
<RBRACE>
  }
```

由大括号括起来的一组语句，一个语句块引起一个新的作用域.

语句(statement)

```
 token statement {
    | <block>
```

```
    | <ifStatement>
    | <whileStatement>
    | <returnStatement>
    | <variableDefinition> {
      my %info = $<variableDefinition>.made.Hash;
      %info<resolvedId> = $counter.yieldLocal;
      $*ST.declare(
        $<variableDefinition>.made<id>,
        %info,
      );
      given %info<type> {
        when 'Scalar' { say "var {%info<resolvedId>}" }
        when 'Array'  { say "var {%info<size> * 4} {%info<resolvedId>}"
}
      }
    }
    | <expression> <SEMI>
    | <SEMI>
  }
```

语句分为以下几种，语句块，`if` 语句，`while` 语句，`return` 语句，表达式语句，空语句和变量定义，只为变量定义添加动作，理由同全局中的变量定义．其他语句自己有自己的动作，在解析具体语句时执行．

`if` 语句(`ifStatement`)

```
token ifStatement {
    | <IF> <LPAREN> <expression> <RPAREN>
      :my $endLabel = $counter.yieldLabel; {
        say "if {$<expression>.made<id>} == 0 goto $endLabel";
      } <statement> [<ELSE> {
        my $resolvedEndLabel = $counter.yieldLabel;
        say "goto $resolvedEndLabel";
        say "$endLabel:";
        $endLabel = $resolvedEndLabel;
      } <statement>]? {
        say "$endLabel:"
      }
  }
```

`if` 语句包括可选的 `else` 子句．在这一层中主要产生 `Eeyore` 中对应的控制逻辑．

`while` 语句(`whileStatement`)

```
token whileStatement {
    | <WHILE>
      :my $testLabel = $counter.yieldLabel; {
        say "$testLabel:";
      } <LPAREN> <expression>
      :my $endLabel = $counter.yieldLabel; {
        say "if {$<expression>.made<id>} == 0 goto $endLabel";
      } <RPAREN> <statement> {
        say "goto $testLabel";
        say "$endLabel:";
      }
  }
```

类似的，产生 Eeyore 中对应的控制逻辑.

return 语句(returnStatement)

```
token returnStatement {
    | <RETURN> <expression> <SEMI> {
      say "return {$<expression>.made<id>}";
    }
  }
```

简单地生成 return 对应的 Eeyore 语句.

表达式(expression)

```
token expression {
    | <assignmentExpression> {
      make $<assignmentExpression>.made;
    }
  }
```

解析表达式的内容，由于 Perl 6 的解析器是 LL(k) 的，使用不同的非终结符以实现运算符优先级.

在具体的表达式解析时，包括了类型检查的内容，这是因为在类型限定下，某个位置能出现的变量类型是固定的，例如，`a + b` 中 `a` 与 `b` 只能是标量，`a[idx]` 中 `a` 只能是数组，`f(arg)` 中 `f` 只能是函数. 此外，函数调用还必须匹配签名，包括参数数量和类型.

赋值表达式(assignmentExpression)

```
token assignmentExpression {
    | <IDENTIFIER> <LBRACK> <expression> <RBRACK> <ASSIGN>
<assignmentExpression> {
      my %info = $*ST.getInfo($<IDENTIFIER>.made);
      checkType(%info<type>, 'Array');
      my $expression = $<expression>.made;
      checkType($expression<type>, 'Scalar', 'Number');
```

```
      my $assignmentExpression = $<assignmentExpression>.made;
      checkType($assignmentExpression<type>, 'Scalar', 'Number');

      my $offset;
      given $expression<type> {
          when 'Number' { $offset = $expression<id> * 4 }
          when 'Scalar' {
            my $temp = $counter.yieldLocal;
            say "var $temp";
            say "$temp = {$expression<id>} * 4";
            $offset = $temp;
          }
      }
      say "{%info<resolvedId>} [$offset] = {$assignmentExpression<id>}";
      make $assignmentExpression;
    }
    | <IDENTIFIER> <ASSIGN> <assignmentExpression> {
      my %info = $*ST.getInfo($<IDENTIFIER>.made);
      checkType(%info<type>, 'Scalar');
      my $assignmentExpression = $<assignmentExpression>.made;
      checkType($assignmentExpression<type>, 'Scalar', 'Number');
      say "{%info<resolvedId>} = {$assignmentExpression<id>}";
      make $assignmentExpression;
    }
    | <logicOrExpression> {
      make $<logicOrExpression>.made;
    }
  }
```

优先级最低的表达式，在语法中处于解析的最顶层以最后处理．写出右递归的语法以实现赋值运算符的右结合性．特别地，最后一个分支即当前赋值表达式是一个逻辑或表达式(平凡情况)，后面的情况类似，平凡情况即当前表达式为下一优先级的表达式．

下面为一系列的二元运算表达式，先统一给出语法，然后在解释语义动作 `emitBinOpCode`．其中普遍使用 `<logicAndExpression>+ % (<OR>)` 语法，这是因为这样可以将下一级的表达式作为列表从左往右处理，由于 Perl 6 的自动机是 LL(k) 的，难以用左递归实现左结合性，实现为右递归将会导致右结合性．

逻辑或表达式(logicOrExpression)

```
token logicOrExpression {
    | <logicAndExpression>+ % (<OR>) {
      emitBinOpCode($/, $<logicAndExpression>, $0);
    }
  }
```

逻辑与表达式(logicAndExpression)

```
token logicAndExpression {
    | <equalityExpression>+ % (<AND>) {
      emitBinOpCode($/, $<equalityExpression>, $0);
    }
  }
```

判等表达式(equalityExpression)

```
token equalityExpression {
    | <relationalExpression>+ % (<EQ>|<NE>) {
      emitBinOpCode($/, $<relationalExpression>, $0);
    }
  }
```

关系表达式(relationalExpression)

```
token relationalExpression {
    | <additiveExpression>+ % (<LT>|<GT>) {
      emitBinOpCode($/, $<additiveExpression>, $0);
    }
  }
```

支持 `>` 和 `<`.

加法优先级表达式(additiveExpression)

```
token additiveExpression {
    | <multiplicativeExpression>+ % (<ADD>|<SUB>) {
      emitBinOpCode($/, $<multiplicativeExpression>, $0);
    }
  }
```

乘法优先级表达式(multiplicativeExpression)

```
token multiplicativeExpression {
    | <unaryExpression>+ % (<MUL>|<DIV>|<MOD>) {
      emitBinOpCode($/, $<unaryExpression>, $0);
    }
  }
```

下面介绍二元运算表达式的语义动作 `emitBinOpCode`.

```
sub emitBinOpCode($/, $operands, $operator) {
  my @operands = $operands.Array.map(*.made);
  my @operators = $operator.Array.map(*.hash.values.[0].made);
  if @operands.elems == 1 {
    make @operands[0];
```

```
    return;
  }

  checkType(@operands[0]<type>, 'Scalar', 'Number');
  checkType(@operands[1]<type>, 'Scalar', 'Number');
  my $temp = $counter.yieldLocal;
  say "var $temp";
  say "$temp = {@operands[0]<id>} {@operators[0]} {@operands[1]<id>}";
  for 2..^@operands.elems -> $id {
    checkType(@operands[$id]<type>, 'Scalar', 'Number');
    say "$temp = $temp {@operators[$id - 1]} {@operands[$id]<id>}";
  }
  make %(
    id => $temp,
    type => 'Scalar',
  );
}
```

前面提到，引入这样的设计主要是为了实现左结合性，可以看到语义动作包含平凡情况(只有一个操作数)的处理，类型检查和 Eeyore 代码生成，对于连续的同一种二元运算，只产生一个中间变量，这样相对每做一次运算产生一个中间变量，可以减少中间变量的数量.

> Perl 6 中，`$/` 是当前匹配的默认变量，这里简单的认为是实现语义动作需要的语法即可.
> 具体内容涉及 grammar 语法的底层实现，即实际上每个 token 都是一种函数的简写.

一元运算表达式(unaryExpression)

```
token unaryExpression {
    | (<NEG>|<NOT>) <unaryExpression> {
      my $unaryExpression = $<unaryExpression>.made;
      my $unaryOp = $0.hash.values.[0].made;
      checkType($unaryExpression<type>, 'Scalar', 'Number');

      given $unaryExpression<type> {
        when 'Number' {
          if $unaryOp eq '-' {
            make %(
              id => (-$unaryExpression<id>).Int,
              type => 'Number',
            );
          } else {
            make %(
              id => (!$unaryExpression<id>).Int,
              type => 'Number',
            );
          }
        }
        when 'Scalar' {
          my $temp = $counter.yieldLocal;
```

```
          say "var $temp";
          say "$temp = $unaryOp {$unaryExpression<id>}";
          make %(
            id => $temp,
            type => 'Scalar',
          );
        }
      }
    }
    | <postfixExpression> {
      make $<postfixExpression>.made;
    }
  }
```

比乘法优先级表达式更高优先级的就是一元运算符表达式了，包括一元的 `-` 和 `!`.

后缀表达式(postfixExpression)

```
token postfixExpression {
    | <IDENTIFIER> <LBRACK> <expression> <RBRACK> {
      my %info = $*ST.getInfo($<IDENTIFIER>.made);
      checkType(%info<type>, 'Array');
      my $expression = $<expression>[0].made;
      checkType($expression<type>, 'Scalar', 'Number');

      my $offset;
      given $expression<type> {
          when 'Number' { $offset = $expression<id> * 4 }
          when 'Scalar' {
            my $temp = $counter.yieldLocal;
            say "var $temp";
            say "$temp = {$expression<id>} * 4";
            $offset = $temp;
          }
      }

      my $temp = $counter.yieldLocal;
      say "var $temp";
      say "$temp = {%info<resolvedId>} [$offset]";
      make %(
        id => $temp,
        type => 'Scalar',
      );
    }
    | <IDENTIFIER> <LPAREN> [<expression>+ % <COMMA>]? <RPAREN> {
      my %info = $*ST.getInfo($<IDENTIFIER>.made);
      checkType(%info<type>, 'Function');

      my @typeList = [];
```

```
      for $<expression>.Array {
        my $expression = .made;
        @typeList.push($expression<type>);
        say "param {$expression<id>}";
      }
      checkFunctionTypeList(@typeList, %info<typeList>);

      my $temp = $counter.yieldLocal;
      say "var $temp";
      say "$temp = call {%info<resolvedId>}";
      make %(
        id => $temp,
        type => 'Scalar',
      );
    }
    | <primaryExpression> {
      make $<primaryExpression>.made;
    }
  }
```

包括数组取值，函数调用，以及平凡情况(基本表达式)．前面提到过，函数调用还会对实参类型进行检查．

基本表达式(primaryExpression)

```
token primaryExpression {
    | <LPAREN> <expression> <RPAREN> {
      make $<expression>.made;
    }
    | <IDENTIFIER> {
      my %info = $*ST.getInfo($<IDENTIFIER>.made);
      make %(
        id => %info<resolvedId>,
        type => %info<type>,
      );
    }
    | <INTEGER> {
      make %(
        id => $<INTEGER>.made,
        type => 'Number',
      );
    }
  }
```

包括标识符，数字以及括号括起来的表达式．注意标识符不一定是标量，这是因为函数调用可以使用数组基本表达式来作为参数，而且在上层用到的位置会有类型检查．当然这也说明语句 `a;` 在 `a` 是数组或函数的时候也是合法的语句，这没什么问题，所以并不报错(gcc 对这种情况也不报错)．

以上就是语法分析以及语法制导翻译的内容，下面介绍类型检查和函数调用中的签名检查的实现．

类型检查

检查函数为：

```
sub checkType(Str $checked, *@checker) {
  die qq:to/END/ unless $checked (elem) @checker
    Type check fails!
      Get $checked,
      expecting @checker[].
    END
    ;
}
```

即检查所给类型是否在允许的类型集合中．所有的调用点都在表达式语句的翻译中，在那里，对于特定的变量有类型限定，可以做类型检查．例如，`a + b` 中 `a` 与 `b` 只能是标量，`a[idx]` 中 `a` 只能是数组，`f(arg)` 中 `f` 只能是函数．

函数调用检查

检查函数为：

```
sub checkFunctionTypeList(@checked, @checker) {
  die qq:to/END/ unless @checked.elems == @checker.elems;
    Parameters unfit!
      Call with { @checked.elems } parameters,
      expecting { @checker.elems } parameters.
    END
    ;

  for ^@checked.elems -> $i {
    if @checker[$i] eq 'Array' { next if @checked[$i] eq @checker[$i] }
    elsif @checker[$i] eq 'Scalar' { next if @checked[$i] (elem) [@checker[$i], 'Number'] }
    die qq:to/END/;
      Parameters unfit!
        Parameter $i has type @checked[$i],
        expecting @checker[$i].
      END
      ;
  }
  return True;
}
```

首先检查参数个数是否符合，如果符合，对每一个参数，检查对应位置的类型限制．调用点在解析函数调用时．

符号表注册

符号表定义在本节最前面已经给出，下面进行实现的说明.

```
has @!scopes = {}, ;

method enterScope() {
  @!scopes.push({});
}

method leaveScope() {
  @!scopes.pop();
}
```

本身保持一个作用域列表，列表尾是当前作用域，往前依次是上一层作用域，在进入新作用域时调用 `enterScope`，离开作用域时调用 `leaveScope`.

```
method getInfo(Str $var) {
    for @!scopes.reverse -> %scope {
      return %scope{$var} if defined %scope{$var};
    }

    die qq:to/END/;
      Cannot refer to identifier $var!
        $var has not been defined.
      END
      ;
  }
```

`getInfo` 方法根据变量名，逐个作用域的查找变量定义，使用静态作用域，在找不到变量定义时报错.

```
method declare(Str $var, %info) {
    self!checkReserved($var);
    self!checkDefined($var, %info);
    @!scopes[*-1]{$var} = %info;
  }

method !checkReserved(Str $var) {
    state %reseverdWords =
      %(
        "int"    => 0,
        "if"     => 1,
        "else"   => 2,
        "while"  => 3,
        "return" => 4,
      );
    die qq:to/END/ if defined %reseverdWords{$var};
      Cannot defined identifier $var!
        Conflict with reseverd word $var.
```

```
      END
      ;
  }

method !checkDefined($var, %info) {
    return unless defined @!scopes[*-1]{$var};
    return self!dieDefinedVariable($var, %info) unless %info<type> eq
'Function';
    return self!checkDefinedFunction($var, %info);
  }

method !dieDefinedVariable($var, %info) {
    die qq:to/END/;
      Cannot defined variable $var!
        $var has already been defined in this scope.
      END
      ;
  }

method !checkDefinedFunction($var, %info) {
    die qq:to/END/ if @!scopes[*-1]{$var}<isDefine>;
      Cannot defined function $var!
        $var has already been defined in this scope.
      END
      ;
    checkFunctionTypeList(%info<typeList>, @!scopes[*-1]{$var}
<typeList>)
  }
```

`declare` 函数提供注册变量到符号表的逻辑，包括一系列的检查函数：是否是保留字，是否已经定义，对于函数，允许签名相同的多次签名，不允许多次定义.

Eeyore 的解析

主要流程包括将 Eeyore 代码转换为类似于四元式的带编号的 `$instruction` 列表或字典(以编号为键)，然后针对这一个指令集合进行操作，包括代码优化，转化，活性分析，寄存器分配和汇编代码生成.

中间指令

应用类似于解析 MiniC 的方式，将 Eeyore 代码转化为 Perl 6 的内部数据结构，显示为一个中间指令集合，即 `$instruction` 为一组与该指令关联的信息，例如编号，使用变量，指令类型，前驱和后继等.

在 Eeyore.pm6 文件中解析 Eeyore 代码，导出为:

```
our %SYMBOLS is export;
our %FUNCTIONS is export;
```

即一个符号表和一个函数字典，函数字典以函数名为键，函数内的指令数组为值．注意到 Eeyore 中每个标识符都有唯一的名称，查找名称时不存在作用域问题，因此可以使用一个统一的符号表．

由于 Eeyore 代码格式规整，因此不用额外的 Lexer 来绕过 Perl 6 有点问题的 LTM 解析算法．

在这一步中，默认输入来自于编译器前端生成的 Eeyore 代码，因此不支持注释，且认为代码正确．

下面介绍中间指令的产生方法，包括解析 Eeyore 的过程．

首先是基本的 Token 集合，这一点和解析 MiniC 时非常类似．

```
# ====================
  # Basic Tokens
  # ====================
  token _DEBUG_BASIC_TOKEN {
    [
    |<IF>|<GOTO>|<END>|<RETURN>|<CALL>|<PARAM>|<VAR>
    |<LBRACK>|<RBRACK>|<COLON>
    |<ASSIGN>
    |<OR>|<AND>
    |<EQ>|<NE>
    |<LT>|<GT>
    |<ADD>|<SUB>
    |<MUL>|<DIV>|<MOD>
    |<NEG>|<NOT>
    |<FUNCTION>|<VARIABLE>|<LABEL>
    |<INTEGER>|<NEWLINE>
    ]+
  }
  token IF     { 'if'<DELIM> }
  token GOTO   { 'goto'<DELIM> }
  token END    { 'end'<DELIM> }
  token RETURN { 'return'<DELIM> }
  token CALL   { 'call'<DELIM> }
  token PARAM  { 'param'<DELIM> }
  token VAR    { 'var'<DELIM>}
  token LBRACK { '['<DELIM> }
  token RBRACK { ']'<DELIM> }
  token COLON  { ':'<DELIM> }
  token ASSIGN { '='<DELIM> { make '=' } }
  token OR     { '|'<DELIM>'|'<DELIM> { make '||' } }
  token AND    { '&'<DELIM>'&'<DELIM> { make '&&' } }
  token EQ     { '='<DELIM>'='<DELIM> { make '==' } }
  token NE     { '!'<DELIM>'='<DELIM> { make '!=' } }
  token LT     { '<'<DELIM> { make '<' } }
  token GT     { '>'<DELIM> { make '>' } }
  token ADD    { '+'<DELIM> { make '+' } }
  token SUB    { '-'<DELIM> { make '-' } }
  token MUL    { '*'<DELIM> { make '*' } }
```

```
  token DIV    { '/'<DELIM> { make '/' } }
  token MOD    { '%'<DELIM> { make '%' } }
  token NEG    { '-'<DELIM> { make '-' } }
  token NOT    { '!'<DELIM> { make '!' } }
  token FUNCTION { (f_<[_A..Za..z]><[_A..Za..z0..9]>*) <DELIM> { make $0.Str } }
  token VARIABLE { (<[tpg]><[0..9]>+) <DELIM> { make $0.Str } }
  token LABEL    { (l<[0..9]>+) <DELIM> { make $0.Str } }
  token INTEGER  { (\-?<[0..9]>+) <DELIM> { make $0.Str } }
  token NEWLINE  { \n <DELIM> }
  token DELIM { " "? }
```

下面是一系列的非终结符，这一次的介绍会快一点.

顶层非终结符(TOP)

```
token TOP { # translateUnit
    <externalDeclaration>+
  }
```

外层定义(externalDeclaration)

```
token externalDeclaration {
    | <functionDefinition>
    | <variableDeclaration>
  }
```

变量定义(variableDeclaration)

```
token variableDeclaration {
    | <VAR> <INTEGER> <VARIABLE> <NEWLINE> {
      my %info = %(
        id => $<VARIABLE>.made,
        size => $<INTEGER>.made,
        type => 'Array',
      );
      %SYMBOLS{%info<id>} = %info;
    }
    | <VAR> <VARIABLE> <NEWLINE> {
      my %info = %(
        id => $<VARIABLE>.made,
        size => 4,
        type => 'Scalar',
      );
      %SYMBOLS{%info<id>} = %info;
    }
  }
```

类似于 MiniC 的解析，将变量定义注册到符号表中，注意由于名称包含了变量的信息(是不是全局变量)，所以可以直接在这里注册．把标量的 `size` 属性标记为 `4` 是为了统一处理全局数组和标量，即将全局标量视作一个元素的数组，当然，由于在为全局标量分配寄存器时(比起全局数组)可以有更好的策略，仍然区分全局数组和标量．

函数定义(functionDefinition)

```
token functionDefinition {
    | <FUNCTION> :my $*FUNCTION; {
      $*FUNCTION = $<FUNCTION>[0].made;
    } <LBRACK> <INTEGER> <RBRACK> <NEWLINE> {
      %SYMBOLS{$*FUNCTION} = %(
        id => $*FUNCTION,
        type => 'Function',
        nParam => $<INTEGER>.made,
      );
      %FUNCTIONS{$*FUNCTION} = [];
    } <lineStatement>*? <END> <FUNCTION> <NEWLINE>
  }
```

注册函数并定义动态作用域变量 `$*FUNCTION`，这是为了在接下来的解析过程中将指令数组与当前函数的函数名关联起来，同样，使用动态作用域变量避免了全局变量，并且更符合程序逻辑．函数体由一系列行语句(lineStatement)组成．

行语句(lineStatement)

```
token lineStatement {
    | <variableDeclaration>
    | <expression> <NEWLINE>
  }
```

包括变量定义和表达式．

表达式(expression)

```
token expression {
    | <binaryExpression>
    | <unaryExpression>
    | <directAssignExpression>
    | <ifExpression>
    | <gotoExpression>
    | <labelExpression>
    | <paramExpression>
    | <callExpression>
    | <returnExpression>
  }
```

包括 if 表达式，goto 表达式，label 表达式，param 表达式，call 表达式，return 表达式，一元/二元运算表达式和直接赋值表达式．

if 表达式(ifExpression)

```
token ifExpression {
    | <IF> <rightValue> <binaryOp> <rightValue> <GOTO> <LABEL> {
      # Assert the form is 'if <VARIABLE> == 0 goto <LABEL>'
      my %instruction;
      %instruction<id> = %FUNCTIONS{$*FUNCTION}.elems;
      %instruction<type> = 'ifFalse';
      %instruction<use> = [$<rightValue>[0].made];
      %instruction<label> = $<LABEL>.made;
      %FUNCTIONS{$*FUNCTION}.push: %instruction;
    }
  }
```

根据前端的逻辑，所有 if 表达式都有 `if <VARIABLE> == 0 goto <LABEL>` 的形式，因此记录相关信息，并标记为 `ifFalse` 类型的语句，将信息登记到指令数组中.

goto 表达式(gotoExpression)

```
token gotoExpression {
    | <GOTO> <LABEL> {
      my %instruction;
      %instruction<id> = %FUNCTIONS{$*FUNCTION}.elems;
      %instruction<type> = 'goto';
      %instruction<label> = $<LABEL>.made;
      %FUNCTIONS{$*FUNCTION}.push: %instruction;
    }
  }
```

label 表达式(labelExpression)

```
token labelExpression {
    | <LABEL> <COLON> {
      my %instruction;
      %instruction<id> = %FUNCTIONS{$*FUNCTION}.elems;
      %instruction<type> = 'label';
      %instruction<label> = $<LABEL>.made;
      %FUNCTIONS{$*FUNCTION}.push: %instruction;
      my %info = %(
        id => $<LABEL>.made,
        type => 'Label',
        location => %instruction<id>,
      );
      %SYMBOLS{%info<id>} = %info;
    }
  }
```

同时登记到指令列表和符号表，标签需要作为符号被记录，这涉及到后面的标签/跳转压缩.

param 表达式(paramExpression)

```
token paramExpression {
    | <PARAM> <rightValue> {
      my %instruction;
      %instruction<id> = %FUNCTIONS{$*FUNCTION}.elems;
      %instruction<type> = 'param';
      %instruction<use> = [$<rightValue>.made];
      %FUNCTIONS{$*FUNCTION}.push: %instruction;
    }
  }
```

call 表达式(callExpression)

```
token callExpression {
    | <VARIABLE> <ASSIGN> <CALL> <FUNCTION> {
      my %instruction;
      %instruction<id> = %FUNCTIONS{$*FUNCTION}.elems;
      %instruction<type> = 'call';
      %instruction<def> = $<VARIABLE>.made;
      %instruction<function> = $<FUNCTION>.made;
      %FUNCTIONS{$*FUNCTION}.push: %instruction;
    }
  }
```

return 表达式(returnExpression)

```
token returnExpression {
    | <RETURN> <rightValue> {
      my %instruction;
      %instruction<id> = %FUNCTIONS{$*FUNCTION}.elems;
      %instruction<type> = 'return';
      %instruction<use> = [$<rightValue>.made];
      %FUNCTIONS{$*FUNCTION}.push: %instruction;
    }
  }
```

二元运算表达式(binaryExpression)

```
token binaryExpression {
    | <VARIABLE> <ASSIGN> <rightValue> <binaryOp> <rightValue> {
      my %instruction;
      %instruction<id> = %FUNCTIONS{$*FUNCTION}.elems;
      %instruction<type> = 'binary';
      %instruction<op> = $<binaryOp>.made;
      %instruction<def> = $<VARIABLE>.made;
      %instruction<use> = [$<rightValue>[0].made, $<rightValue>
[1].made];
      %FUNCTIONS{$*FUNCTION}.push: %instruction;
    }
  }
```

由于操作数只有两个，不存在优先级问题.

一元运算表达式(unaryExpression)

```
token unaryExpression {
    | <VARIABLE> <ASSIGN> <unaryOp> <rightValue> {
      my %instruction;
      %instruction<id> = %FUNCTIONS{$*FUNCTION}.elems;
      %instruction<type> = 'unary';
      %instruction<op> = $<unaryOp>.made;
      %instruction<def> = $<VARIABLE>.made;
      %instruction<use> = [$<rightValue>.made];
      %FUNCTIONS{$*FUNCTION}.push: %instruction;
    }
  }
```

直接赋值表达式(directAssignExpression)

```
token directAssignExpression {
    | <VARIABLE> <LBRACK> <rightValue> <RBRACK> <ASSIGN> <rightValue> {
      my %instruction;
      %instruction<id> = %FUNCTIONS{$*FUNCTION}.elems;
      %instruction<type> = 'arrayScalar';
      %instruction<use> = [$<VARIABLE>[0].made, $<rightValue>[0].made,
$<rightValue>[1].made];
      %FUNCTIONS{$*FUNCTION}.push: %instruction;
    }
    | <VARIABLE> <ASSIGN> <VARIABLE> <LBRACK> <rightValue> <RBRACK> {
      my %instruction;
      %instruction<id> = %FUNCTIONS{$*FUNCTION}.elems;
      %instruction<type> = 'scalarArray';
      %instruction<def> = $<VARIABLE>[0].made;
      %instruction<use> = [$<VARIABLE>[1].made, $<rightValue>[0].made];
      %FUNCTIONS{$*FUNCTION}.push: %instruction;
    }
```

```
    | <VARIABLE> <ASSIGN> <rightValue> {
      my %instruction;
      %instruction<id> = %FUNCTIONS{$*FUNCTION}.elems;
      %instruction<type> = 'scalarRval';
      %instruction<def> = $<VARIABLE>[0].made;
      %instruction<use> = [$<rightValue>[0].made];
      %FUNCTIONS{$*FUNCTION}.push: %instruction;
    }
  }
```

注意数组名被标记为 use 而不是 def.

以上就是解析 Eeyore 代码的逻辑，后面我们会在导出的指令集合和符号表上进行分析和操作.

中间指令的解析

标签/跳转压缩

首先重定位标签的位置.

```
# ====================
# Fix LABEL Location
# ====================

for %FUNCTIONS.kv -> $function, @instruction {
  for ^@instruction.elems -> $id {
    if @instruction[$id]<type> eq 'label' {
      next unless $id + 1 < @instruction.elems;
      if @instruction[$id + 1]<type> eq any('label', 'goto') {
        %SYMBOLS{@instruction[$id]<label>}<location> = @instruction[$id
+ 1]<label>;
      }
    }
  }
}
```

通过上面的函数将标签的 `<location>` 属性重定位到下一个位置，在后面的压缩过程中可以根据下一个，一个一个的找到 `<location>` 为数字的地方. 这里，`<location>` 属性即当前标签指示的位置，由于在翻译过程中可能产生标签下边紧跟着一个标签或者紧跟着一个 `goto` 语句的情况，实际上跳转到该标签的语句很快就要跳转到别的位置或者即将执行什么也不做的标签语句，重定位后压缩可以减少无用的汇编代码.

压缩逻辑包含在下面生成基本块的代码中.

构造可达基本块(死代码消除)

```
# ====================
# Build Reachable BLOCKS
# ====================
```

```
4
5  my %BLOCKS;
6  for %FUNCTIONS.kv -> $function, @instruction {
7    %BLOCKS{$function} = Hash.new;
8
9    my $instructionId = 0;
10   my %reachedInstruction = %(0 => True);
11   my $blockId = 0;
12   %BLOCKS{$function}{$blockId} = [@instruction[$instructionId]];
13
14   while $instructionId < @instruction.elems {
15     given @instruction[$instructionId]<type> {
16       when 'call' {
17         $blockId += 1;
18         %BLOCKS{$function}{$blockId} = [];
19         %reachedInstruction{$instructionId + 1} = True;
20       }
21       when 'return' {
22         $blockId += 1;
23         %BLOCKS{$function}{$blockId} = [];
24       }
25       when 'ifFalse' {
26         $blockId += 1;
27         %BLOCKS{$function}{$blockId} = [];
28         %reachedInstruction{$instructionId + 1} = True;
29         %reachedInstruction{resolveLabel(@instruction[$instructionId]
   <label>)} = True;
30         @instruction[$instructionId]<label> =
   fixLabel(@instruction[$instructionId]<label>);
31       }
32       when 'goto' {
33         $blockId += 1;
34         %BLOCKS{$function}{$blockId} = [];
35         %reachedInstruction{resolveLabel(@instruction[$instructionId]
   <label>)} = True;
36         @instruction[$instructionId]<label> =
   fixLabel(@instruction[$instructionId]<label>);
37       }
38       default {
39         %reachedInstruction{$instructionId + 1} = True;
40       }
41     }
42
43     repeat { $instructionId += 1 } until
   %reachedInstruction{$instructionId} or $instructionId >=
   @instruction.elems;
44     last if $instructionId >= @instruction.elems;
45
46     if @instruction[$instructionId]<type> eq 'label' {
```

```
      if %BLOCKS{$function}{$blockId}.elems > 0 {
        $blockId += 1;
        %BLOCKS{$function}{$blockId} = [];
      }
    }

    %BLOCKS{$function}{$blockId}.push(@instruction[$instructionId]);
  }

}
```

没什么特别的，按照基本块的构建算法逐步挑出来．在解析到跳转语句时压缩跳转目标，即多次连续的跳转压缩为一次，或者跳过无意义的标签语句．这里的辅助函数 `resolveLabel` 和 `fixLabel` 实现如下：

```
sub resolveLabel(Str $label is copy) {
  until isInteger(%SYMBOLS{$label}<location>) {
    $label = %SYMBOLS{$label}<location>;
  }
  return %SYMBOLS{$label}<location>;
}

sub fixLabel(Str $label is copy) {
  until isInteger(%SYMBOLS{$label}<location>) {
    $label = %SYMBOLS{$label}<location>;
  }
  return $label;
}
```

> Perl 6 中可以标记参数具有 `is copy` 属性以在函数体中获得参数的一份拷贝，这是因为参数默认是不可变的，我们既不想改变参数，又不想写 `my $localLabel = $label;`，这就是一种绕过方式．

注意这一步从第一条语句开始一个一个基本块的挑，实际上，不可达的基本块会被丢弃，也就是说实现了死代码消除．

常量传播和折叠

```
for %BLOCKS.kv -> $function, %blocks {
  for ^%blocks.elems -> $blockId {
    my %VALUE;
  loop {
    my $modified = False;
    for ^%blocks{$blockId}.Array.elems -> $instructionId {
      my $instruction := %blocks{$blockId}[$instructionId];
      given $instruction<type> {
        when 'scalarRval' {
          %VALUE{$instruction<def>} = $instruction<use>[0];
          if defined %VALUE{%VALUE{$instruction<def>}} {
```

```
            %VALUE{$instruction<def>} =
%VALUE{%VALUE{$instruction<def>}};
          }
        }
      }
    }
    for ^%blocks{$blockId}.Array.elems -> $instructionId {
      my $instruction := %blocks{$blockId}[$instructionId];
      given $instruction<type> {
        when 'unary' {
          if isInteger($instruction<use>[0]) {
            my %instruction;
            %instruction<id> = $instruction<id>;
            %instruction<type> = 'scalarRval';
            %instruction<def> = $instruction<def>;
            %instruction<use> = [resolveUnary($instruction<op>,
$instruction<use>[0].Int)];
            $instruction = %instruction;
            $modified = True;
          } elsif isInteger(%VALUE{$instruction<use>[0]} // '#') {
            my %instruction;
            %instruction<id> = $instruction<id>;
            %instruction<type> = 'scalarRval';
            %instruction<def> = $instruction<def>;
            %instruction<use> = [resolveUnary($instruction<op>,
%VALUE{$instruction<use>[0]}.Int)];
            $instruction = %instruction;
            $modified = True;
          }
        }
        when 'binary' {
          if isInteger($instruction<use>[0]) and
isInteger($instruction<use>[1]) {
            my %instruction;
            %instruction<id> = $instruction<id>;
            %instruction<type> = 'scalarRval';
            %instruction<def> = $instruction<def>;
            %instruction<use> = [resolveBinary($instruction<op>,
$instruction<use>[0].Int, $instruction<use>[1].Int)];
            $instruction = %instruction;
            $modified = True;
          } elsif isInteger(%VALUE{$instruction<use>[0]} // '#') and
isInteger($instruction<use>[1]) {
            my %instruction;
            %instruction<id> = $instruction<id>;
            %instruction<type> = 'scalarRval';
            %instruction<def> = $instruction<def>;
            %instruction<use> = [resolveBinary($instruction<op>,
%VALUE{$instruction<use>[0]}.Int, $instruction<use>[1].Int)];
```

```
            $instruction = %instruction;
            $modified = True;
          } elsif isInteger($instruction<use>[0]) and
isInteger(%VALUE{$instruction<use>[1]} // '#') {
            my %instruction;
            %instruction<id> = $instruction<id>;
            %instruction<type> = 'scalarRval';
            %instruction<def> = $instruction<def>;
            %instruction<use> = [resolveBinary($instruction<op>,
$instruction<use>[0].Int, %VALUE{$instruction<use>[1]}.Int)];
            $instruction = %instruction;
            $modified = True;
          } elsif isInteger(%VALUE{$instruction<use>[0]} // '#') and
isInteger(%VALUE{$instruction<use>[1]} // '#') {
            my %instruction;
            %instruction<id> = $instruction<id>;
            %instruction<type> = 'scalarRval';
            %instruction<def> = $instruction<def>;
            %instruction<use> = [resolveBinary($instruction<op>,
%VALUE{$instruction<use>[0]}.Int, %VALUE{$instruction<use>[1]}.Int)];
            $instruction = %instruction;
            $modified = True;
          }
        }
      }
      last if $modified;
    }

    last unless $modified;
  }
  }
}
```

通过一个临时的 `VALUE` 字典记录变量当前的值，只对标量的赋值和运算进行常量传播和折叠．由于不是静态单赋值，所以需要按照指令顺序进行折叠，在进行每一次折叠后都要刷新 `VALUE` 字典．常量传播和折叠只对每一个基本块进行，不做跨基本块分析．

活性分析与寄存器分配

在上面的操作完成后，我们先将基本块恢复为线性代码．

```
my %LINEAR;
for %BLOCKS.kv -> $function, %blocks {
  %LINEAR{$function} = Hash.new;
  for ^%blocks.elems -> $blockId {
    for %blocks{$blockId}.Array -> $instruction {
      %LINEAR{$function}{$instruction<id>} = $instruction;
    }
  }
}
```

注意，由于前面可能确有死代码被消除，这里应当使用字典来保存指令，即使用指令编号来索引指令，因为可能有某个编号的指令因为不可达而不再被保留.

可以逐个函数的完成活性分析与寄存器分配，下面的代码全都在这个大循环内:

```
my %livenessAnalyse;
for %LINEAR.kv -> $function, %instruction {
    # ...
}
```

其中，`%livenessAnalyse` 保存函数中变量的活性分析结果，包括活跃区间和分配的寄存器等，这些信息在后面的代码生成中还要用到.

活性分析

初始化

首先，初始化指令的前驱后继关系和活跃变量集合.

```
.value.<prev> = [] for %instruction;
.value.<succ> = [] for %instruction;
.value.<live> = [] for %instruction;
%instruction{0}<prev>.push("-1");
my $maxInstructionId = max(%instruction.keys.map(*.Int));
```

前驱和后继可能不止一个，所以使用列表来保存. 因为首指令没有前驱，特殊处理首指令，避免后面代码因为变量未定义出错. 另外，记录最大的指令编号，避免后继越界导致一些奇怪的问题.

前驱/后继

下面生成前驱/后继关系.

```
for %instruction.kv -> $id, $instruction {
    given $instruction<type> {
      when 'return' {
        ;
      }
      when 'ifFalse' {
        $instruction<succ>.push(resolveLabel($instruction<label>));
```

```
        next if $id + 1 > $maxInstructionId;
        $instruction<succ>.push($id + 1);
        $instruction<succ>.unique;
      }
      when 'goto' {
        $instruction<succ>.push(resolveLabel($instruction<label>));
      }
      default {
        next if $id + 1 > $maxInstructionId;
        $instruction<succ>.push($id + 1);
      }
    }
    for $instruction<succ>.Array {
      %instruction{$_}<prev>.push($instruction<id>);
    }
  }
```

没什么特别的，后继的编号直接可以得到，先标记后继，再遍历后继标记前驱.

活跃范围(Live Range)

接着计算出变量的活跃范围(Live Range).

```
  # =================
  # Generate Live Range
  # =================
  for %instruction.kv -> $id, $instruction {
    next unless defined $instruction<use>;
    for $instruction<use>.Array -> $usedRval {
      next if isInteger($usedRval);

      my @notifyLiveQueue = [];
      unless $usedRval (elem) $instruction<live> {
        $instruction<live>.push($usedRval);
      } # Do not forget defined and used in the same instruction

      @notifyLiveQueue.append($instruction<prev>.Array);
      while @notifyLiveQueue.elems > 0 {
        my $notifiedId = @notifyLiveQueue.shift;
        next if $notifiedId < 0;
        next if $usedRval (elem) %instruction{$notifiedId}<live>;
        next if $usedRval (elem) %instruction{$notifiedId}<def>;
        %instruction{$notifiedId}<live>.push($usedRval);
        @notifyLiveQueue.append(%instruction{$notifiedId}<prev>.Array);
      }
    }
  }
```

由于包装器的未知 Bug，Perl 6 中的 `Set` 和我为此实现的 `Algorithm::BitMap` 在循环中会丢失包装内容，简单地说就是不好实现并行的数据流方程活性分析．因此，按照虎书上的方法，逐个变量从 use 点开始往前逐句标记 live，直到遇见 def 点或者上一个活跃点或者超出函数范围(未定义使用，在实验过程中是个非常恶心的情况，但是全局变量和数组确实可以未定义使用，因此是合法的)．注意处理在同一语句中定义和使用的情况，只要在当前语句中使用了，它就是活跃的．

活跃区间(Live Interval)

```
  my %usedOnCall = Hash.new;
  # =================
  # Generate Live Interval
  # =================
  %livenessAnalyse{$function} = Hash.new;
  for %instruction.kv -> $id, $instruction {
    for $instruction<live>.Array -> $variable {
      unless defined %livenessAnalyse{$function}{$variable} {
        %livenessAnalyse{$function}{$variable} = { };
        %livenessAnalyse{$function}{$variable}<start> = Inf;
        %livenessAnalyse{$function}{$variable}<end> = -Inf;
        %livenessAnalyse{$function}{$variable}<reg> = "";
      }
      %livenessAnalyse{$function}{$variable}<start> min=
$instruction<id>-1;
      %livenessAnalyse{$function}{$variable}<start> max= 0;
      %livenessAnalyse{$function}{$variable}<end> max= $instruction<id>;
      next if isGlobal($variable) and !isArray($variable);
      %usedOnCall{$variable} = True if $instruction<type> eq 'call';
    }
  }
```

把活跃范围连起来，生成活跃区间，标记好在函数调用时活跃的变量．这些变量如果分配了 Caller Save 寄存器，在函数调用时就需要保存，因此尽量把它们分配到 Callee Save 上．

这种方法有一种情况会遗漏，即 `t0 = call f_func`，而且 `t0` 的活跃区间跨过了此处的 call，这样由于这一句定义了 `t0`，它在这一句上不活跃，为了解决这种问题需要先生成活跃区间，再判断是否跨过了这个函数调用，实际上，对于每个函数调用，可能忽略的这种变量最多只有一个，因此我选择不做处理，就分配给 `t0` Caller Save，假装不知道它的活跃区间覆盖了这里．

另外，由于全局标量在每次函数调用时都必须写回，优先给它分配 Caller Save，不占用 Callee Save．

寄存器分配

寄存器分类

```
my @callerSave = [
  "t0", "t1", "t2", "t3", "t4", "t5", "t6",
];
my @calleeSave = [
  "s0", "s1", "s2", "s3", "s4", "s5", "s6",
  "s7", "s8", "s9", "s10", "s11",
];
my %callerSave = @callerSave.map(* => True);
my %calleeSave = @calleeSave.map(* => True);
```

活跃区间排序

```
my @variables = %livenessAnalyse{$function}.Hash;
@variables.=sort({
  $^a.value<start> != $^b.value<start>
  ?? $^a.value<start> <=> $^b.value<start>
  !! $^b.value<end> <=> $^a.value<end>
});
```

分配寄存器

```
 my %registers;
 %SYMBOLS{$function}<usedCallee> = Hash.new;
 for @variables -> %variableInfo {
   my $variable = %variableInfo.key;

   # =================
   # Expire Variable
   # =================

   for %registers.kv -> $register, $holdVariable {
     if %livenessAnalyse{$function}{$holdVariable}<end> <
%livenessAnalyse{$function}{$variable}<start> {
       %registers{$register} :delete;
       @calleeSave.unshift($register) if %calleeSave{$register};
       @callerSave.unshift($register) if %callerSave{$register};
     }
   }

   # =================
   # Registe Register
   # =================

   if %usedOnCall{$variable} {
     if @calleeSave.elems > 0 {
       my $register = @calleeSave.shift;
       %livenessAnalyse{$function}{$variable}<reg> = $register;
       %registers{$register} = $variable;
```

```
        %SYMBOLS{$function}<usedCallee>{$register} = True;
        next;
      }
    }

    if @callerSave.elems > 0 {
      my $register = @callerSave.shift;
      %livenessAnalyse{$function}{$variable}<reg> = $register;
      %registers{$register} = $variable;
      next;
    }

    next if isGlobal($variable) and !isArray($variable);

    if @calleeSave.elems > 0 {
      my $register = @calleeSave.shift;
      %livenessAnalyse{$function}{$variable}<reg> = $register;
      %registers{$register} = $variable;
      %SYMBOLS{$function}<usedCallee>{$register} = True;
      next;
    }
  }
```

按照 Linear Scan 算法执行，特别地，记录使用的 Callee Save 寄存器的信息，后面的代码生成中，函数开头要保存 Callee Save 寄存器，离开前要恢复．优先为变量分配寄存器，不管它被建议分配到 Caller Save 还是 Callee Save，但是全局标量不要分配到 Callee Save，因为 Callee Save 在函数调用时不写回，我不想针对分配到 Callee Save 的全局标量在函数调用时再做讨论．

代码生成

定义全局变量

```
for %SYMBOLS.kv -> $id, %info {
  next unless isGlobal($id);
  say "\t.comm\t$id,{%info<size>},4";
}
```

特别的，把全局标量生成为单个元素的全局数组，但在后面的操作上仍然区分开来．

下面逐个函数生成代码．汇编代码并不依赖函数定义的顺序．

函数头

```
# ====================
# Generate Function Head
# ====================

@riscvCode.push("\t.text");
@riscvCode.push("\t.align\t2");
@riscvCode.push("\t.global\t{$function.substr(2)}");
@riscvCode.push("\t.type\t{$function.substr(2)}, \@function");
@riscvCode.push("{$function.substr(2)}:");
@riscvCode.push("\tadd\tsp,sp,-STK");
@riscvCode.push("\tsw\tra,STK-4(sp)");
```

保存 Callee Save

```
# ====================
# Store Used Callee
# ====================
my @usedCallee = %SYMBOLS{$function}<usedCallee>.keys;
for @usedCallee -> $register {
  @riscvCode.push("\tsw\t$register,{$stackSize*4}(sp)");
  $stackSize += 1;
}
```

把 Callee Save 的保存区放在函数帧的头部.

注册函数参数

```
# ====================
# Handle Parameters
# ====================
for ^%SYMBOLS{$function}<nParam> -> $i {
  registVariable("p$i");
  my $reg = getRegister("p$i", "a0");
  @riscvCode.push("\tmv\t$reg,a$i");
  storeIfSpilled("p$i", $reg);
}
```

函数体

1. 变量超出活跃期，对于全局标量，写回.（所有的数组地址都是常量，不需要写回.）对于标签语句，因为执行这一句在标签之后，写回逻辑也要出现在标签之后，其他情况下要在这一句之前超出活跃期，否则寄存器表的状态不对，对于标签语句，因为除了生成标签什么也不做，所以暂时状态不对也不会出问题.

```
  # ==================
  # Expire Register
  # ==================
  my @saveBackGlobal;
```

```
    for %registers.kv -> $register, $holdVariable {
      if %livenessAnalyse{$function}{$holdVariable}<end> <
$instruction<id> {
        @saveBackGlobal.push(($register, $holdVariable)) if
isGlobal($holdVariable);
        %registers{$register} :delete;
      }
    }

    if $instruction<type> ne 'label' {
      for @saveBackGlobal -> ($register, $holdVariable) {
        next if isArray($holdVariable);
        @riscvCode.push("\tlui\ta5,\%hi($holdVariable)");
        @riscvCode.push("\tsw\t$register,\%lo($holdVariable)(a5)");
      }
    }
    # ...
    if $instruction<type> eq 'label' {
      for @saveBackGlobal -> ($register, $holdVariable) {
        next if isArray($holdVariable);
        @riscvCode.push("\tlui\ta5,\%hi($holdVariable)");
        @riscvCode.push("\tsw\t$register,\%lo($holdVariable)(a5)");
      }
    }
```

2. 注册活跃变量，主要处理数组地址的加载. `registVariable` 中不处理局部标量，假定局部标量不存在未定义使用.

```
    for $instruction<live>.Array -> $variable {
      registVariable($variable);
    }
```

3. 按类别处理语句
    1. 跳转相关的语句

```
      when 'ifFalse' {
        registVariable($instruction<use>.Array[0]);
        my $reg = getRegister($instruction<use>.Array[0], "a0");
        @riscvCode.push("\tbeq\t$reg,zero,.{$instruction<label>}");
      }
      when 'label' {
        @riscvCode.push(".{$instruction<label>}:");
      }
      when 'goto' {
        @riscvCode.push("\tj\t.{$instruction<label>}");
      }

```

2. 返回语句

```
      when 'return' {
        registVariable($instruction<use>.Array[0]);
        if isInteger($instruction<use>.Array[0]) {
          @riscvCode.push("\tli\ta0,{$instruction<use>.Array[0]}");
        } else {
          my $register = getRegister($instruction<use>.Array[0],
"a0");
          @riscvCode.push("\tmv\ta0,$register");
        }
        for @usedCallee Z (0..*) -> ($register, $location) {
          @riscvCode.push("\tlw\t$register,{$location*4}(sp)");
        }
        @riscvCode.push("\tlw\tra,STK-4(sp)");
        @riscvCode.push("\tadd\tsp,sp,STK");
        @riscvCode.push("\tjr\tra");
      }
```

注意在每个返回的位置都要写上离开函数的逻辑，在函数的自然尾部并不写上离开函数的逻辑，这是因为假定每个函数都正确的写上了 `return` 语句. 此外，针对返回值为整数的情况作了细微的优化，不需要先将整数加载到某个寄存器再 `mv` 到 `a0`.

3. 直接赋值语句

```
      when 'scalarRval' {
        if isDefineValid($instruction<def>, $instruction) {
          registVariable($instruction<def>);
          registVariable($instruction<use>.Array[0]);
          my $reg2 = getRegister($instruction<def>, "a2");
          if isInteger($instruction<use>.Array[0]) {
            @riscvCode.push("\tli\t$reg2,
{$instruction<use>.Array[0]}");
          } else {
            my $reg0 = getRegister($instruction<use>.Array[0],
"a0");
            @riscvCode.push("\tmv\t$reg2,$reg0");
          }
          storeIfSpilled($instruction<def>, $reg2);
        }
      }
      when 'arrayScalar' {
        registVariable($instruction<use>.Array[0]);
        registVariable($instruction<use>.Array[1]);
        registVariable($instruction<use>.Array[2]);
        my $reg2 = getRegister($instruction<use>.Array[0], "a2");
        my $reg0 = getRegister($instruction<use>.Array[1], "a0");
        my $reg1 = getRegister($instruction<use>.Array[2], "a1");
        @riscvCode.push("\tadd\ta5,$reg2,$reg0");
```

```
        @riscvCode.push("\tsw\t$reg1,0(a5)");
      }
      when 'scalarArray' {
        if isDefineValid($instruction<def>, $instruction) {
          registVariable($instruction<def>);
          registVariable($instruction<use>.Array[0]);
          registVariable($instruction<use>.Array[1]);
          my $reg2 = getRegister($instruction<def>, "a2");
          my $reg0 = getRegister($instruction<use>.Array[0], "a0");
          my $reg1 = getRegister($instruction<use>.Array[1], "a1");
          @riscvCode.push("\tadd\ta5,$reg0,$reg1");
          @riscvCode.push("\tlw\t$reg2,0(a5)");
          storeIfSpilled($instruction<def>, $reg2);
        }
      }
```

针对标量赋值数字做了优化，关于数组元素赋值数字，因为最后也要从寄存器中 `sw`，所以不需要讨论.

4． 函数调用语句

```
      when 'param' {
        registVariable($instruction<use>.Array[0]);
        loadParameter($currentParameterId,
$instruction<use>.Array[0]);
        $currentParameterId += 1;
      }
      when 'call' {
        $currentParameterId = 0;
        for %registers.kv -> $register, $variable {
          storeCaller($register, $variable);
        }
        @riscvCode.append(@parameterCode);
        @parameterCode = [];

@riscvCode.push("\tcall\t{$instruction<function>.substr(2)}");
        for %registers.kv -> $register, $variable {
          loadCaller($register, $variable);
        }
        if isDefineValid($instruction<def>, $instruction) {
          registVariable($instruction<def>);
          my $register = getRegister($instruction<def>, "a2");
          @riscvCode.push("\tmv\t$register,a0");
          storeIfSpilled($instruction<def>, $register);
        }
      }
```

顺序非常重要，由于将几乎不用到的 `a0` 到 `a7` 作为临时寄存器，在 `param` 时，这些寄存器有特殊的作用，因此 `call` 的写回工作一定要在准备参数之前完成，但是 Eeyore 的逻辑是先出现 `param` 语句，再出现 `call` 语句，因此生成 `param` 代码时先保存到 `@parameterCode` 中，遇到 `call` 时再写入到写回寄存器的逻辑之后．（主要是写回全局标量时涉及到 `a5`）

5． 一元运算语句

```
      when 'unary' {
        if isDefineValid($instruction<def>, $instruction) {
          registVariable($instruction<def>);
          registVariable($instruction<use>.Array[0]);
          my $reg2 = getRegister($instruction<def>, "a2");
          my $reg0 = getRegister($instruction<use>.Array[0], "a0");
          given $instruction<op> {
            when '-' {
              @riscvCode.push("\tsub\t$reg2,zero,$reg0");
            }
            when '!' {
              @riscvCode.push("\tseqz\t$reg2,$reg0");
            }
          }
          storeIfSpilled($instruction<def>, $reg2);
        }
      }
```

经过常量传播和折叠，一元运算语句的操作数必然是变量．

6． 二元操作语句

```
      when 'binary' {
        if isDefineValid($instruction<def>, $instruction) {
          registVariable($instruction<def>);
          registVariable($instruction<use>.Array[0]);
          registVariable($instruction<use>.Array[1]);
          my $reg2 = getRegister($instruction<def>, "a2");
          given $instruction<op> {
            when '||' {
              my $reg0 = getRegister($instruction<use>.Array[0],
"a0");
              my $reg1 = getRegister($instruction<use>.Array[1],
"a1");
              @riscvCode.push("\tor\t$reg2,$reg0,$reg1");
              @riscvCode.push("\tsnez\t$reg2,$reg2");
            }
            when '&&' {
              my $reg0 = getRegister($instruction<use>.Array[0],
"a0");
              my $reg1 = getRegister($instruction<use>.Array[1],
"a1");
```

```
              @riscvCode.push("\tand\t$reg2,$reg0,$reg1");
              @riscvCode.push("\tsnez\t$reg2,$reg2");
            }
            when '==' {
              my $reg0 = getRegister($instruction<use>.Array[0],
"a0");
              my $reg1 = getRegister($instruction<use>.Array[1],
"a1");
              @riscvCode.push("\txor\t$reg2,$reg0,$reg1");
              @riscvCode.push("\tseqz\t$reg2,$reg2");
            }
            when '!=' {
              my $reg0 = getRegister($instruction<use>.Array[0],
"a0");
              my $reg1 = getRegister($instruction<use>.Array[1],
"a1");
              @riscvCode.push("\txor\t$reg2,$reg0,$reg1");
              @riscvCode.push("\tsnez\t$reg2,$reg2");
            }
            when '<' {
              if isInteger($instruction<use>.Array[1]) {
                my $reg0 = getRegister($instruction<use>.Array[0],
"a0");
                @riscvCode.push("\tslt\t$reg2,$reg0,
{$instruction<use>.Array[1]}");
              } else {
                my $reg0 = getRegister($instruction<use>.Array[0],
"a0");
                my $reg1 = getRegister($instruction<use>.Array[1],
"a1");
                @riscvCode.push("\tslt\t$reg2,$reg0,$reg1");
              }
            }
            when '>' {
              if isInteger($instruction<use>.Array[0]) {
                my $reg1 = getRegister($instruction<use>.Array[1],
"a1");
                @riscvCode.push("\tslt\t$reg2,$reg1,
{$instruction<use>.Array[0]}");
              } else {
                my $reg0 = getRegister($instruction<use>.Array[0],
"a0");
                my $reg1 = getRegister($instruction<use>.Array[1],
"a1");
                @riscvCode.push("\tsgt\t$reg2,$reg0,$reg1");
              }
            }
            when '+' {
              if isInteger($instruction<use>.Array[1]) {
```

```
54                my $reg0 = getRegister($instruction<use>.Array[0],
"a0");
55                @riscvCode.push("\tadd\t$reg2,$reg0,
{$instruction<use>.Array[1]}");
56              } elsif isInteger($instruction<use>.Array[0]) {
57                my $reg1 = getRegister($instruction<use>.Array[1],
"a1");
58                @riscvCode.push("\tadd\t$reg2,$reg1,
{$instruction<use>.Array[0]}");
59              } else {
60                my $reg0 = getRegister($instruction<use>.Array[0],
"a0");
61                my $reg1 = getRegister($instruction<use>.Array[1],
"a1");
62                @riscvCode.push("\tadd\t$reg2,$reg0,$reg1");
63              }
64            }
65            when '-' {
66              my $reg0 = getRegister($instruction<use>.Array[0],
"a0");
67              my $reg1 = getRegister($instruction<use>.Array[1],
"a1");
68              @riscvCode.push("\tsub\t$reg2,$reg0,$reg1");
69            }
70            when '*' {
71              if $instruction<use>.Array[1] eq 4 {
72                my $reg0 = getRegister($instruction<use>.Array[0],
"a0");
73                @riscvCode.push("\tsll\t$reg2,$reg0,2");
74              } else {
75                my $reg0 = getRegister($instruction<use>.Array[0],
"a0");
76                my $reg1 = getRegister($instruction<use>.Array[1],
"a1");
77                @riscvCode.push("\tmul\t$reg2,$reg0,$reg1");
78              }
79            }
80            when '/' {
81              my $reg0 = getRegister($instruction<use>.Array[0],
"a0");
82              my $reg1 = getRegister($instruction<use>.Array[1],
"a1");
83              @riscvCode.push("\tdiv\t$reg2,$reg0,$reg1");
84            }
85            when '%' {
86              my $reg0 = getRegister($instruction<use>.Array[0],
"a0");
87              my $reg1 = getRegister($instruction<use>.Array[1],
"a1");
```

```
88                @riscvCode.push("\trem\t$reg2,$reg0,$reg1");
89              }
90            }
91            storeIfSpilled($instruction<def>, $reg2);
92          }
93        }
```

针对几种常见的操作做了讨论和优化，包括加法允许常数出现，小于允许常数出现，`num > rval` 可以转换为带常数出现的小于，以及对于数组操作常见的 `* 4`，使用 `sll reg, reg, 2` 来处理.

由于 writeup 的内容不太清楚，下面给出实现非短路的逻辑运算符和一元运算符的对应表：

```
x = y || z => or reg(x),reg(y),reg(z)
              snez reg(x),reg(x)
  # without short circuit, it's same as 'x = Bool(y | z)'

x = y && z => and reg(x),reg(y),reg(z)
              snez reg(x),reg(x)
  # without short circuit, it's same as 'x = Bool(y & z)'

x = y == z => xor reg(x),reg(y),reg(z)
              seqz reg(x),reg(x)

x = y != z => xor reg(x),reg(y),reg(z)
              snez reg(x),reg(x)

x = y < z => slt reg(x),reg(y),reg(z)

x = y > z => sgt reg(x),reg(y),reg(z)

if x == 0 goto L => beqz reg(x),.L

x = !y => seqz reg(x),reg(y)

x = -y => sub reg(x),zero,reg(y)
```

函数尾

```
  @riscvCode.push("\t.size\t{$function.substr(2)}, .-{$function.substr(2)}");

  my $riscvCode = @riscvCode.join("\n");
  my $STK = (($stackSize div 4) + 1) * 16;
  say $riscvCode.subst(/STK\-4/, $STK - 4, :g)
                .subst(/STK/, $STK, :g);
```

生成函数尾，并重写栈帧大小. 这是因为栈帧大小是在生成代码的时候同时确定的.

寄存器约定

上边代码中还有一些辅助函数的内容没有给出，大多数可由其名称得知作用，使用这些辅助函数时唯一非平凡的点就是需要知道对于溢出变量的处理和临时寄存器的约定．

借鉴蔡佳晋同学的想法，用于参数和结果传递的 `a0` 到 `a7` 这八个寄存器在此外的大多数情况下没有使用，因此可以用来加载溢出变量，加载数字和加载数组地址．由于前面的工作，同时需要的溢出变量不超过三个，数字不超过一个，地址不超过一个，因此这八个寄存器是够用的．在大多数语句下也无需担心错误使用．只有在参数准备时，需要将所有参数在原寄存器准备完毕后在一次性加载到 `a` 系列寄存器，这是因为准备参数的过程可能修改 `a` 系列寄存器．Eeyore 代码生成时，所有 `param` 语句紧跟在 `call` 的前面，因此可以像前面代码那样编写．

我使用的寄存器约定是，`a0` 到 `a2` 作为溢出变量加载区，`a4` 用于加载数字，`a5` 用于加载数组地址．

测试

测试用例

1. 课程改革时的测试文件，包括 aarg.c，ab.c，arr.c，cmt.c，expr.c，fac.c，func.c，funny.c，id.c，if.c，ifelse.c，logic.c，nnim.c，qsort.c，recur.c，revstr.c，scope.c，sort.c，while.c，wseq.c．测试数据为 data/ 下的数据．
2. 个性功能的测试．
3. 网测上的二十六个测试文件．

测试效果

对于上边的（1）部分，除了 logic.c 要求实现短路未通过，其余都通过了正确性测试．其中 wseq.c 和 funny.c 在 `diff` 命令下显示结果不同，但编写脚本逐行比对结果相同，猜测是空白符号的问题，至少输出的所有有效信息都是相同的．

上面部分使用以下脚本检测：

```
my $test = @*ARGS[0];
for ^100 -> $i {
  shell("/opt/riscv-qemu/bin/qemu-riscv32 -L /opt/riscv/sysroot/ main < data/$test/data$i.in > out");
  $ = shell("diff out data/$test/data$i.out");
  say $i;
}
```

对于（3）部分，全部通过测试．

针对 wseq.c 得到的动态指令数效果如右．wseq.c(data/wseq/data0.in): 10296.

```
gz translate.c into gen intermediate code
gz translate.c init max_insns512
gz translate.c tb insns num is 1
```

```
gz translate.c total insns num is 10294
gz translate.c PC =  4134796902
gz tranclate.c : exec one 64C inst
gz translate.c tb insns num is 2
gz translate.c total insns num is 10295
gz translate.c PC =  4134796904
gz tranclate.c : exec one 64G instc : ARITH_IMM/_W
gz translate.c tb insns num is 3
gz translate.c total insns num is 10296
gz translate.c PC =  4134796908
gz tranclate.c : exec one 64G instc : SYSTEM
gz translate.c ------------------------------------------------
gz translate.c : done gen tb,size(tpc-hpc) is 10
```

对于错误报告测试，效果如下：

```
int main() {
  int a;
  int b[10];
  a = 42;
  b[0] = 0;
  a = a + b;
  return 0;
}

OUTPUT:
Type check fails!
  Get Array,
  expecting Scalar Number.
```

```
int main() {
  int a;
  int a
  return 0;
}

OUTPUT:
Cannot defined variable a!
  a has already been defined in this scope.
```

```
int f();
int f();

OK
```

```
int f();
int f(int x);

Parameters unfit!
  Call with 1 parameters,
  expecting 0 parameters.
```

```
int f();
int f() { return 42; }

OK
```

```
int f() { return 42; }
int f() { return 42; }

OUTPUT:
Cannot defined function f!
  f has already been defined in this scope.
```

```
int f(int x) { return x; }
int main() {
  int a[10];
  f(a);
  return 0;
}

OUTPUT:
Parameters unfit!
  Parameter 0 has type Array,
  expecting Scalar.
```

主要遇到的错误和解决

1. func.c 中的

```
int g(int x)
{
    if (x % 2 == 0)
        return f(x);
    else
        return f(x + 1);
}
```

这个函数在生成的 Eeyore 长这样：

```
f_g [1]
```

```
var t8
t8 = p0 % 2
var t9
t9 = t8 == 0
if t9 == 0 goto l2
param p0
var t10
t10 = call f_f
return t10
goto l3
l2:
var t11
t11 = p0 + 1
param t11
var t12
t12 = call f_f
return t12
l3:
end f_g
```

注意 `l3:` 后什么也没有，`goto l3` 也是死代码，在年代久远的一次调试中，生成的 Tigger 代码因为这个玩意一直段错误，随后加入了死代码消除逻辑.

死代码消除时，一开始是把所有没有前驱的去掉，迭代. 但是发现像循环这种东西自己连接自己，哪怕整个循环在 `reurn` 后面也都有前驱. 因此采用类似于标记清扫算法的办法，激进的去除所有语法上不可达的代码.

2. aarg.c 中，全局变量在多个函数内都有使用，因此发现了全局标量在函数跳转时必须写回.
3. funny.c 的逻辑比较混乱，出现了很多疑难杂症，但都不是什么大问题，比如说一元运算符忘记还有 `!` 之类的，还有数组操作的一些实现错，发现后马上知道错误原因并且修正.
4. qsort.c 和 wseq.c 的速度很慢，因此考虑了增加分析 `usedOnCall`，合理分配 Caller Save 和 Callee Save. 原本是打算直接全部 Caller Save 了事.
5. 常量传播和折叠时，一开始是并发的处理，后来发现由于不是静态单赋值，这样会出错，典型的场景是:

```
TURN 1:
a = 3
a = a + 1
a = a + 2
a = a + 3

TRUN 2:
a = 3
a = 4
a = 5
a = 6
```

这显然不对，因此改为每发生一次折叠就重新传播一次.

6. 错误检查一开始比较 naive，具体情况记不清了，在陶淼学长的帮助下找到了大多数遗漏的情况.

工具存在的问题

没有使用 Lex/Yacc，在后期甚至跳过 Tigger，生成的 Eeyore 也和 writeup 的 Eeyore 不兼容.

RISC-V 的模拟器没发现什么问题.

Eeyore 模拟器不支持 `param num` 的形式.

此外，课程改革时的代码包中的 Tigger 生成器和 RISC-V 生成器产生的代码好像并不能在对应的模拟器上执行.

Perl 6 的问题. 太多了[捂脸]，因为是一门新语言，虽然主要功能还行，但是一些边角的地方有各种奇怪的 BUG. 在做这个实习的过程中，和 Perl 6 社区的人特别是针对 Regex 和 Grammar 做了很多交流，完善了它的官方文档.

实习总结

收获和体会

1. 最关键的一点就是终于知道数据流分析是干嘛的了，上学期选编译原理的时候这一部分没有实践根本不知道是啥. 强烈推荐先选实习再选原理，实习的时候我几乎是把编译原理重新学了一遍.
2. 理解了线性扫遍算法，浏览了二次装箱算法，图着色算法和其他的寄存器分配算法，虽然最终还是实现了线性扫描算法，但是也看到了活跃区间和活跃范围的不同，第一次实现中使用活跃范围和跳出基本块时写回，实际效果也并没有那么糟糕，这是因为活跃范围大多很短，与连成一条的的活跃区间不同，活跃范围可以区分两次不同的定义，因此需要写回的实际上没有那么多.
3. 实践了 Perl 6，用它完成了一个工程. 参与到它的开发中，包括 BUG 修复，功能实现和文档撰写，也让我看到了现代语言实现的方式，编译实习课讲的就是实现一个语言到机器代码的过程，Perl 6 的工具链，类似于 Pypy，让我看到了最前沿的一些编程语言实现方法.
4. 理解了交流讨论的重要性. 虽然是一门独立完成的实习课，但是和肖博文，蔡佳晋，特古斯等同学的讨论交流启发了我很多的想法，帮助我克服了很多错误理解.
5. 信心. 每次面对新的任务，几乎都不知道要怎么弄，但是一旦决定去做，从查资料和思考到尝试到实现一般不超过两天.
6. 最麻烦的就是活性分析和寄存器分配了. 我一开始理解错了，只生成活跃范围导致分配出现问题，最恶心的莫过于控制流(分支，循环)和全局变量. 对于活跃范围，一个分支加载了，另一个没有加载，但是在线性 IR 上呈现出前后关系，因而卸载后面的分支会以为已经加载了而出错. 如果在控制流图上做，一方面分支合并时需要修复变量和寄存器的对应关系，另一方面面对循环时又乱了. 一开始就按照标准的活性分析做恐怕就没这么多问题了，不过在这个过程中也更深入的了解了静态单赋值形式和变量生存周期在实际中可能出现的各种极端情况.

课程建议

1. MiniC 那个 `main` 函数单独列出来的语法可以改一改了[捂脸].
2. 支持 `func(expr)` 好像不太难，可以作为 Base 集，包括不用强制接收返回值和参数可以

是表达式.

3. 讲解内容上，寄存器分配和线性扫描算法可以再讲详细一点，因为原理课上应该没讲，一旦跟我一样出现理解偏差会花很多时间在错误的方向.
4. 实习过程上，因为这一次给了很多灵活的空间，所以感觉已经很好了.