Stage 1 实验报告

范逍宇 2019013273

一、工作概述。

Step 2

1.前端。

- 在 parser.y 添加两个一元运算对应的语法规则：

```
| LNOT Expr  %prec LNOT
       { $$ = new ast::NotExpr($2, POS(@1)); }
| BNOT Expr  %prec BNOT
       { $$ = new ast::BitNotExpr($2, POS(@1)); }
```

- 在 scanner.l 添加两个一元运算对应的词法规则：

```
"!"          { return yy::parser::make_LNOT  (loc);       }
"~"          { return yy::parser::make_BNOT  (loc);       }
```

2.中端。

- 在类型检查 (type_check.cpp) 中添加两个一元运算对应的递归遍历 AST 节点，对 AST 节点进行类型检查的函数。

```
void SemPass2::visit(ast::NotExpr *e) {
    e->e->accept(this);
    expect(e->e, BaseType::Int);
    e->ATTR(type) = BaseType::Int;
}
void SemPass2::visit(ast::BitNotExpr *e) {
    e->e->accept(this);
    expect(e->e, BaseType::Int);
    e->ATTR(type) = BaseType::Int;
}
```

- 同理，在翻译阶段 (translation.cpp/hpp) 中添加对应的递归遍历 AST 节点，根据节点生成对应的 TAC。

```
void Translation::visit(ast::NotExpr *e) {
    e->e->accept(this);
    e->ATTR(val) = tr->genLNot(e->e->ATTR(val));
}
void Translation::visit(ast::BitNotExpr *e) {
    e->e->accept(this);
    e->ATTR(val) = tr->genBNot(e->e->ATTR(val));
}
```

3.后端。

- 在 riscv_md.hpp 中添加需要用到的 risc-v 指令。

```
SEQZ
NOT
```

- 在生成汇编代码阶段 (riscv_md.cpp) 的 emitTac 函数中添加两个一元运算操作对应的 case ， 来将 TAC 指令翻译成 risc-v 指令。

```
case Tac::LNOT:
        emitUnaryTac(RiscvInstr::SEQZ, t);
        break;
case Tac::BNOT:
        emitUnaryTac(RiscvInstr::NOT, t);
        break;
```

- 在 emitInstr 函数中，添加对应操作的 case，用以将对应的 risc-v 指令输出出来。（这里逻辑取反对应的 risc-v 指令是 SEQZ ）

```
case RiscvInstr::SNEZ:
        oss << "snez" << i->r0->name << ", " << i->r1->name;
        break;

case RiscvInstr::NOT:
        oss << "not" << i->r0->name << ", " << i->r1->name;
        break;
```

Step 3

1.前端。

- 同 step 2， 添加 "+ - * / %" 对应的的语法规则和词法规则。（括号对应的词法规则和语法规则已经被写好）

```
| Expr PLUS Expr
                { $$ = new ast::AddExpr($1, $3, POS(@2)); }
| Expr MINUS Expr
                { $$ = new ast::SubExpr($1, $3, POS(@2)); }
| Expr TIMES Expr
                { $$ = new ast::MulExpr($1, $3, POS(@2)); }
| Expr SLASH Expr
                { $$ = new ast::DivExpr($1, $3, POS(@2)); }
| Expr MOD Expr
                { $$ = new ast::ModExpr($1, $3, POS(@2)); }
```

```
"+"          { return yy::parser::make_PLUS  (loc);       }
"-"          { return yy::parser::make_MINUS  (loc);       }
"*"          { return yy::parser::make_TIMES  (loc);       }
"/"          { return yy::parser::make_SLASH  (loc);       }
"%"          { return yy::parser::make_MOD  (loc);       }
```

2.中端。

- 括号操作在前端已经完成解析，实现了它对优先级的控制，它没有对应的 TAC 指令。
- 在类型检查 (type_check.cpp) 中添加五个运算对应的递归遍历 AST 节点，对 AST 节点进行类型检查的函数。

```
void SemPass2::visit(ast::AddExpr *e) {
    e->e1->accept(this);
    expect(e->e1, BaseType::Int);

    e->e2->accept(this);
    expect(e->e2, BaseType::Int);

    e->ATTR(type) = BaseType::Int;
}

void SemPass2::visit(ast::SubExpr *e) {
    e->e1->accept(this);
    expect(e->e1, BaseType::Int);

    e->e2->accept(this);
    expect(e->e2, BaseType::Int);

    e->ATTR(type) = BaseType::Int;
}

void SemPass2::visit(ast::MulExpr *e) {
    e->e1->accept(this);
    expect(e->e1, BaseType::Int);

    e->e2->accept(this);
    expect(e->e2, BaseType::Int);

    e->ATTR(type) = BaseType::Int;
}

void SemPass2::visit(ast::DivExpr *e) {
    e->e1->accept(this);
    expect(e->e1, BaseType::Int);

    e->e2->accept(this);
    expect(e->e2, BaseType::Int);

    e->ATTR(type) = BaseType::Int;
}

void SemPass2::visit(ast::ModExpr *e) {
    e->e1->accept(this);
    expect(e->e1, BaseType::Int);

    e->e2->accept(this);
    expect(e->e2, BaseType::Int);

    e->ATTR(type) = BaseType::Int;
}
```

- 同理，在翻译阶段 (translation.cpp/hpp) 中添加对应的递归遍历 AST 节点，根据节点生成对应的 TAC。

```
void Translation::visit(ast::AddExpr *e) {
```

```
    e->e1->accept(this);
    e->e2->accept(this);

    e->ATTR(val) = tr->genAdd(e->e1->ATTR(val), e->e2->ATTR(val));
}

void Translation::visit(ast::SubExpr *e) {
    e->e1->accept(this);
    e->e2->accept(this);

    e->ATTR(val) = tr->genSub(e->e1->ATTR(val), e->e2->ATTR(val));
}

void Translation::visit(ast::MulExpr *e) {
    e->e1->accept(this);
    e->e2->accept(this);

    e->ATTR(val) = tr->genMul(e->e1->ATTR(val), e->e2->ATTR(val));
}

void Translation::visit(ast::DivExpr *e) {
    e->e1->accept(this);
    e->e2->accept(this);

    e->ATTR(val) = tr->genDiv(e->e1->ATTR(val), e->e2->ATTR(val));
}

void Translation::visit(ast::ModExpr *e) {
    e->e1->accept(this);
    e->e2->accept(this);

    e->ATTR(val) = tr->genMod(e->e1->ATTR(val), e->e2->ATTR(val));
}
```

3.后端。

- 同 step 2， 在 riscv_md.hpp 中添加需要用到的 risc-v 指令。

```
ADD,
SUB,
MUL,
DIV,
REM,
```

- 同 step 2， 在生成汇编代码阶段 (riscv_md.cpp) 的 emitTac 函数中添加五个二元运算操作对应的 case， 来将 TAC 指令翻译成 risc-v 指令。

```
case Tac::ADD:
        emitBinaryTac(RiscvInstr::ADD, t);
        break;

case Tac::SUB:
        emitBinaryTac(RiscvInstr::SUB, t);
        break;

case Tac::MUL:
        emitBinaryTac(RiscvInstr::MUL, t);
        break;

case Tac::DIV:
        emitBinaryTac(RiscvInstr::DIV, t);
        break;

case Tac::MOD:
        emitBinaryTac(RiscvInstr::REM, t);
        break;
```

- 同 step 2， 在 emitInstr 函数中，添加对应操作的 case，用以将对应的 risc-v 指令输出出来。

```
case RiscvInstr::ADD:
        oss << "add" << i->r0->name << ", " << i->r1->name << ", " << i->r2->name;
        break;

case RiscvInstr::SUB:
        oss << "sub" << i->r0->name << ", " << i->r1->name << ", " << i->r2->name;
        break;

case RiscvInstr::MUL:
        oss << "mul" << i->r0->name << ", " << i->r1->name << ", " << i->r2->name;
        break;

case RiscvInstr::DIV:
```

```
        oss << "div" << i->r0->name << ", " << i->r1->name << ", " << i->r2->name;
        break;

case RiscvInstr::REM:
        oss << "rem" << i->r0->name << ", " << i->r1->name << ", " << i->r2->name;
        break;
```

Step 4

1.前端。

- 同 step 2， 添加 "<, >, <=, >=, ==, !=, &&, ||" 对应的的语法规则和词法规则。

```
| Expr LT Expr
                { $$ = new ast::LesExpr($1, $3, POS(@2)); }
| Expr GT Expr
                { $$ = new ast::GrtExpr($1, $3, POS(@2)); }
| Expr LEQ Expr
                { $$ = new ast::LeqExpr($1, $3, POS(@2)); }
| Expr GEQ Expr
                { $$ = new ast::GeqExpr($1, $3, POS(@2)); }
| Expr EQU Expr
                { $$ = new ast::EquExpr($1, $3, POS(@2)); }
| Expr NEQ Expr
                { $$ = new ast::NeqExpr($1, $3, POS(@2)); }
| Expr AND Expr
                { $$ = new ast::AndExpr($1, $3, POS(@2)); }
| Expr OR Expr
                { $$ = new ast::OrExpr($1, $3, POS(@2)); }
```

```
"<"          { return yy::parser::make_LT  (loc);       }
">"          { return yy::parser::make_GT  (loc);       }
"<="          { return yy::parser::make_LEQ  (loc);       }
">="          { return yy::parser::make_GEQ  (loc);       }
"=="          { return yy::parser::make_EQU  (loc);       }
"!="          { return yy::parser::make_NEQ  (loc);       }
"&&"          { return yy::parser::make_AND  (loc);       }
"||"          { return yy::parser::make_OR  (loc);       }
```

2.中端。

- 在类型检查 (type_check.cpp) 中添加五个运算对应的递归遍历 AST 节点，对 AST 节点进行类型检查的函数。

```
void SemPass2::visit(ast::LesExpr *e) {
    e->e1->accept(this);
    expect(e->e1, BaseType::Int);

    e->e2->accept(this);
    expect(e->e2, BaseType::Int);

    e->ATTR(type) = BaseType::Int;
}

void SemPass2::visit(ast::GrtExpr *e) {
    e->e1->accept(this);
    expect(e->e1, BaseType::Int);

    e->e2->accept(this);
    expect(e->e2, BaseType::Int);

    e->ATTR(type) = BaseType::Int;
}

void SemPass2::visit(ast::LeqExpr *e) {
    e->e1->accept(this);
    expect(e->e1, BaseType::Int);

    e->e2->accept(this);
    expect(e->e2, BaseType::Int);

    e->ATTR(type) = BaseType::Int;
}

void SemPass2::visit(ast::GeqExpr *e) {
    e->e1->accept(this);
    expect(e->e1, BaseType::Int);

    e->e2->accept(this);
    expect(e->e2, BaseType::Int);

    e->ATTR(type) = BaseType::Int;
}
```

```
void SemPass2::visit(ast::EquExpr *e) {
    e->e1->accept(this);
    expect(e->e1, BaseType::Int);

    e->e2->accept(this);
    expect(e->e2, BaseType::Int);

    e->ATTR(type) = BaseType::Int;
}

void SemPass2::visit(ast::NeqExpr *e) {
    e->e1->accept(this);
    expect(e->e1, BaseType::Int);

    e->e2->accept(this);
    expect(e->e2, BaseType::Int);

    e->ATTR(type) = BaseType::Int;
}

void SemPass2::visit(ast::AndExpr *e) {
    e->e1->accept(this);
    expect(e->e1, BaseType::Int);

    e->e2->accept(this);
    expect(e->e2, BaseType::Int);

    e->ATTR(type) = BaseType::Int;
}

void SemPass2::visit(ast::OrExpr *e) {
    e->e1->accept(this);
    expect(e->e1, BaseType::Int);

    e->e2->accept(this);
    expect(e->e2, BaseType::Int);

    e->ATTR(type) = BaseType::Int;
}
```

- 同理，在翻译阶段 (translation.cpp/hpp) 中添加对应的递归遍历 AST 节点，根据节点生成对应的 TAC。

```
void Translation::visit(ast::LesExpr *e) {
    e->e1->accept(this);
    e->e2->accept(this);

    e->ATTR(val) = tr->genLes(e->e1->ATTR(val), e->e2->ATTR(val));
}

void Translation::visit(ast::GrtExpr *e) {
    e->e1->accept(this);
    e->e2->accept(this);

    e->ATTR(val) = tr->genGtr(e->e1->ATTR(val), e->e2->ATTR(val));
}

void Translation::visit(ast::LeqExpr *e) {
    e->e1->accept(this);
    e->e2->accept(this);

    e->ATTR(val) = tr->genLeq(e->e1->ATTR(val), e->e2->ATTR(val));
}

void Translation::visit(ast::GeqExpr *e) {
    e->e1->accept(this);
    e->e2->accept(this);

    e->ATTR(val) = tr->genGeq(e->e1->ATTR(val), e->e2->ATTR(val));
}

void Translation::visit(ast::EquExpr *e) {
    e->e1->accept(this);
    e->e2->accept(this);

    e->ATTR(val) = tr->genEqu(e->e1->ATTR(val), e->e2->ATTR(val));
}

void Translation::visit(ast::NeqExpr *e) {
    e->e1->accept(this);
    e->e2->accept(this);

    e->ATTR(val) = tr->genNeq(e->e1->ATTR(val), e->e2->ATTR(val));
```

```
}

void Translation::visit(ast::AndExpr *e) {
    e->e1->accept(this);
    e->e2->accept(this);

    e->ATTR(val) = tr->genLAnd(e->e1->ATTR(val), e->e2->ATTR(val));
}

void Translation::visit(ast::OrExpr *e) {
    e->e1->accept(this);
    e->e2->accept(this);

    e->ATTR(val) = tr->genLOr(e->e1->ATTR(val), e->e2->ATTR(val));
}
```

3.后端。

- 同 step 2，在 riscv_md.hpp 中添加需要用到的 risc-v 指令。需要注意的是，这里添加的部分指令 (SLEQ, SGEQ, SEQ, SNE, LAND, LOR) 是伪指令，risc-v 中没有对应的指令，我们将通过其他的基本指令来实现这几条伪指令。

```
SLT,
SGT,
SLEQ,
SGEQ,
SEQ,
SNE,
LAND,
LOR,
SNEZ,
AND,
OR,
```

- 同 step 2，在生成汇编代码阶段 (riscv_md.cpp) 的 emitTac 函数中添加对应的 case，来将 TAC 指令翻译成 risc-v 指令。

```
case Tac::LES:
        emitBinaryTac(RiscvInstr::SLT, t);
        break;

case Tac::GTR:
        emitBinaryTac(RiscvInstr::SGT, t);
        break;

case Tac::LEQ:
        emitBinaryTac(RiscvInstr::SLEQ, t);
        break;

case Tac::GEQ:
        emitBinaryTac(RiscvInstr::SGEQ, t);
        break;

case Tac::EQU:
        emitBinaryTac(RiscvInstr::SEQ, t);
        break;

case Tac::NEQ:
        emitBinaryTac(RiscvInstr::SNE, t);
        break;

case Tac::LAND:
        emitBinaryTac(RiscvInstr::LAND, t);
        break;

case Tac::LOR:
        emitBinaryTac(RiscvInstr::LOR, t);
        break;
```

- 在 emitBinaryTac 函数中，我们需要特殊处理翻译定义的伪指令，用基本指令实现这些伪指令，例如，用 sgt r0, r1, r2 ; seqz r0, r0 ; 来实现 <= 的操作。

```
void RiscvDesc::emitBinaryTac(RiscvInstr::OpCode op, Tac *t) {
    // eliminates useless assignments
    if (!t->LiveOut->contains(t->op0.var))
        return;
    Set<Temp> *liveness = t->LiveOut->clone();
    liveness->add(t->op1.var);
    liveness->add(t->op2.var);
    int r1 = getRegForRead(t->op1.var, 0, liveness);
    int r2 = getRegForRead(t->op2.var, r1, liveness);
    int r0 = getRegForWrite(t->op0.var, r1, r2, liveness);
    // in risc-v, some instructions need to be implemented with several
    // basic ones
```

```
    switch (op) {
    case RiscvInstr::SLEQ:
        addInstr(RiscvInstr::SGT, _reg[r0], _reg[r1], _reg[r2], 0, EMPTY_STR,
                 NULL);
        addInstr(RiscvInstr::SEQZ, _reg[r0], _reg[r0], NULL, 0, EMPTY_STR, NULL);
        break;

    case RiscvInstr::SGEQ:
        addInstr(RiscvInstr::SLT, _reg[r0], _reg[r1], _reg[r2], 0, EMPTY_STR,
                 NULL);
        addInstr(RiscvInstr::SEQZ, _reg[r0], _reg[r0], NULL, 0, EMPTY_STR,
                 NULL);
        break;

    case RiscvInstr::SEQ:
        addInstr(RiscvInstr::SUB, _reg[r0], _reg[r1], _reg[r2], 0, EMPTY_STR,
                 NULL);
        addInstr(RiscvInstr::SEQZ, _reg[r0], _reg[r0], NULL, 0, EMPTY_STR,
                 NULL);
        break;

    case RiscvInstr::SNE:
        addInstr(RiscvInstr::SUB, _reg[r0], _reg[r1], _reg[r2], 0, EMPTY_STR,
                 NULL);
        addInstr(RiscvInstr::SNEZ, _reg[r0], _reg[r0], NULL, 0, EMPTY_STR,
                 NULL);
        break;
```

```
    case RiscvInstr::LAND:
        // snez d, s1; sub d, zero, d; and d, d, s2; snez d, d;
        addInstr(RiscvInstr::SNEZ, _reg[r0], _reg[r1], NULL, 0, EMPTY_STR,
                 NULL);
        addInstr(RiscvInstr::NEG, _reg[r0], _reg[r0], NULL, 0, EMPTY_STR,
                 NULL);
        addInstr(RiscvInstr::AND, _reg[r0], _reg[r0], _reg[r2], 0, EMPTY_STR,
                 NULL);
        addInstr(RiscvInstr::SNEZ, _reg[r0], _reg[r0], NULL, 0, EMPTY_STR,
                 NULL);
        break;

    case RiscvInstr::LOR:
        // or t3,t1,t2 ; snez t3,t3
        addInstr(RiscvInstr::OR, _reg[r0], _reg[r1], _reg[r2], 0, EMPTY_STR,
                 NULL);
        addInstr(RiscvInstr::SNEZ, _reg[r0], _reg[r0], NULL, 0, EMPTY_STR,
                 NULL);
        break;

    default:
        addInstr(op, _reg[r0], _reg[r1], _reg[r2], 0, EMPTY_STR, NULL);
        break;
    }
}
```

- 同 step 2 ，在 emitInstr 函数中，添加对应操作的 case，用以将对应的 risc-v 指令输出出来。注意，伪指令已经通过多条基本指令输出，不需要写伪指令对应的 case 。

```
    case RiscvInstr::SLT:
        oss << "slt" << i->r0->name << ", " << i->r1->name << ", " << i->r2->name;
        break;

    case RiscvInstr::SGT:
        oss << "sgt" << i->r0->name << ", " << i->r1->name << ", " << i->r2->name;
        break;

    case RiscvInstr::AND:
        oss << "and" << i->r0->name << ", " << i->r1->name << ", "
            << i->r2->name;
        break;

    case RiscvInstr::OR:
        oss << "or" << i->r0->name << ", " << i->r1->name << ", "
            << i->r2->name;
        break;

    case RiscvInstr::BEQZ:
        oss << "beqz" << i->r0->name << ", " << i->l;
        break;
```

二、思考题。

Step 2

Question:

我们在语义规范中规定整数运算越界是未定义行为，运算越界可以简单理解成理论上的运算结果没有办法保存在32位整数的空间中，必须截断高于32位的内容。请设计一个 minidecaf 表达式，只使用 `-~!` 这三个单目运算符和从 0 到 2147483647 范围内的非负整数，使得运算过程中发生越界。

提示：发生越界的一步计算是-

My answer:

`-~` 2147483647

Step 3

Question:

我们知道“除数为零的除法是未定义行为”，但是即使除法的右操作数不是 0，仍然可能存在未定义行为。请问这时除法的左操作数和右操作数分别是什么？请将这时除法的左操作数和右操作数填入下面的代码中，分别在你的电脑（请标明你的电脑的架构，比如 x86-64 或 ARM）中和 RISCV-32 的 qemu 模拟器中编译运行下面的代码，并给出运行结果。（编译时请不要开启任何编译优化）

My answer:

```c
#include <stdio.h>

int main() {
  int a = ~2147483647;
  int b = -1;
  printf("%d\n", a / b);
  return 0;
}
```

运行结果：

我的电脑 (x86-64): 无输出，Visual Studio 报错 0xC0000095: Integer overflow。

RISCV-32 的 qemu 模拟器：输出 -2147483648 。

Step 4

Question:

在 MiniDecaf 中，我们对于短路求值未做要求，但在包括 C 语言的大多数流行的语言中，短路求值都是被支持的。为何这一特性广受欢迎？你认为短路求值这一特性会给程序员带来怎样的好处？

My answer:

因为使用短路求值这一特性有如下优点：

- 会使一些条件判断的代码变得简洁
- 由于一旦短路，后面的表达式不会被解析
    - 提高了代码的运行效率
    - 放宽了对后面表达式的合法性的要求，在短路的条件下，后面的表达式可以在这种情况下不合法。

给程序员带来的方便：

条件判断的代码只需要一行即可完成：

```c
// 不使用短路求值
if(statement){
    function();
}
// 使用短路求值
statement && function();
// 在 statement 成立时，function() 操作不合法也没关系，因为它根本不会被执行。

// 不使用短路求值
if(satement){
    value = statement;
}
else {
    value = function();
}
// 使用短路求值
value = statement || function();
```