MiniDecaf Stage 1: 常量表达式

> My-laniaKeA

实验内容

Step 2: 一元操作

前端

词法分析和语法分析

在 `scanner.l` 中添加上述字符的词法规则：

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

在 `parser.y` 中添加 `!` 和 `~` 的语法规则：

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

中端

AST 节点类型检查

在 `SemPass2` 类中，添加对应逻辑非与按位非表达式 AST 节点的 `visit` 成员函数，递归访问子节点。

```
/* Visits an ast::NotExpr node.
 * PARAMETERS:
 *   e     - the ast::NotExpr node
 */
void SemPass2::visit(ast::NotExpr *e) {
    e->e->accept(this);
    expect(e->e, BaseType::Int);
    e->ATTR(type) = BaseType::Int;
}

/* Visits an ast::BitNotExpr node.
 * PARAMETERS:
 *   e     - the ast::BitNotExpr node
 */
void SemPass2::visit(ast::BitNotExpr *e) {
    e->e->accept(this);
    expect(e->e, BaseType::Int);
    e->ATTR(type) = BaseType::Int;
}
```

中间代码生成

在 `Translation` 类中，添加对应逻辑非与按位取反表达式 AST 节点的 `visit` 成员函数，递归访问子节点。

```
/* Translating an ast::NotExpr node.
 */
void Translation::visit(ast::NotExpr *e) {
    e->e->accept(this);
    e->ATTR(val) = tr->genLNot(e->e->ATTR(val));
}

/* Translating an ast::BitNotExpr node.
 */
void Translation::visit(ast::BitNotExpr *e) {
    e->e->accept(this);
    e->ATTR(val) = tr->genBNot(e->e->ATTR(val));
}
```

后端

RISC-V汇编代码生成

经过分析，逻辑非的中间代码与汇编代码分别如下：

```
_T1 = LNot _T0
```

```
seqz t1, t0
```

按位取反的中间代码与汇编代码分别如下：

```
_T1 = BNot _T0
```

```
not t1, t0
```

在 `RiscvInstr` 类中声明 `SEQZ` 和 `NOT` 两个指令类型，在 `RiscvDesc::emitTac` 函数中，增加 TAC 为 `LNOT` 和 `BNOT` 的情况。

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

case Tac::LNOT:
emitUnaryTac(RiscvInstr::SEQZ, t);
break;
```

在 `emitInst` 函数中，添加对应指令的发射操作：

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

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

Step 3: 加减乘除模

前端

词法分析和语法分析

在 `parser.y` 中增加非终结符的声明，以实现加减乘除模的运算规则：

```
%nterm<mind::ast::Expr*> Expr MultiplicativeExpr UnaryExpr PrimaryExpr
```

按照实验指导书，更改 `Expr` 的语法规则如下：

```
Expr           : MultiplicativeExpr     { $$ = $1; }
               | Expr PLUS MultiplicativeExpr
                   { $$ = new ast::AddExpr($1, $3, POS(@2)); }
               | Expr MINUS MultiplicativeExpr
                   { $$ = new ast::SubExpr($1, $3, POS(@2)); }
               ;
MultiplicativeExpr  : UnaryExpr      { $$ = $1; }
               |   MultiplicativeExpr TIMES UnaryExpr
                   { $$ = new ast::MulExpr($1, $3, POS(@2)); }
               |   MultiplicativeExpr SLASH UnaryExpr
                   { $$ = new ast::DivExpr($1, $3, POS(@2)); }
               |   MultiplicativeExpr MOD UnaryExpr
                   { $$ = new ast::ModExpr($1, $3, POS(@2)); }
               ;
UnaryExpr     : PrimaryExpr           { $$ = $1; }
               | MINUS UnaryExpr  %prec NEG
                   { $$ = new ast::NegExpr($2, POS(@1)); }
               | BNOT UnaryExpr
                   { $$ = new ast::BitNotExpr($2, POS(@1)); }
               | LNOT UnaryExpr
                   { $$ = new ast::NotExpr($2, POS(@1)); }
               ;
PrimaryExpr : ICONST
                   { $$ = new ast::IntConst($1, POS(@1)); }
               | LPAREN Expr RPAREN     { $$ = $2; }
               ;
```

与 **Step 2** 类似，在 `scanner.l` 中增加相应的词法规则。

中端

与 **Step 2** 类似，在 `SemPass2` 类和 `Translation` 类中，添加加减乘除模对应的 `visit` 成员函数。以模运算为例：

```
/* Visits an ast::ModExpr node.
 * PARAMETERS:
 *   e     - the ast::ModExpr node
 */
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;
}
```

```
/* Translating an ast::ModExpr node.
 */
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));
}
```

后端

RISC-V汇编代码生成

各运算符的中间代码与汇编代码分别如下：

```
_T2 = Add _T0 _T1       # +
_T2 = Sub _T0 _T1       # -
_T2 = Mul _T0 _T1       # *
_T2 = Div _T0 _T1       # /
_T2 = Mod _T0 _T1       # %
```

```
add t2, t0, t1      # +
sub t2, t0, t1      # -
mul t2, t0, t1      # *
div t2, t0, t1      # /
rem t2, t0, t1      # %
```

与 **Step 2** 类似，在 `RiscvInstr` 类中声明 `REM` 等指令类型；在 `RiscvDesc::emitTac` 函数中，增加 TAC 的情况；在 `emitInst` 函数中，添加发射操作，不再赘述。

Step 4: 比较和逻辑表达式

前端

词法分析和语法分析

在 `parser.y` 中增加非终结符的声明，以实现比较和逻辑表达式：

```
%nterm<mind::ast::Expr*> Expr
%nterm<mind::ast::Expr*> Additive Multiplicative Unary Primary
%nterm<mind::ast::Expr*> Logical_And Logical_Or
%nterm<mind::ast::Expr*> Equality Relational
```

按照实验指导书，更改 `Expr` 的语法规则如下：

```
Expr        : Logical_Or    { $$ = $1; }
            ;
Logical_Or  : Logical_And   { $$ = $1; }
            | Logical_Or OR Logical_And
                { $$ = new ast::OrExpr($1, $3, POS(@2)); }
            ;
Logical_And : Equality      { $$ = $1; }
            | Logical_And AND Equality
                { $$ = new ast::AndExpr($1, $3, POS(@2)); }
            ;
Equality    : Relational    { $$ = $1; }
            | Equality EQU Relational
                { $$ = new ast::EquExpr($1, $3, POS(@2)); }
            | Equality NEQ Relational
                { $$ = new ast::NeqExpr($1, $3, POS(@2)); }
            ;
Relational  : Additive      { $$ = $1; }
            | Relational GT Additive
                { $$ = new ast::GrtExpr($1, $3, POS(@2)); }
            | Relational LT Additive
                { $$ = new ast::LesExpr($1, $3, POS(@2)); }
            | Relational GEQ Additive
                { $$ = new ast::GeqExpr($1, $3, POS(@2)); }
            | Relational LEQ Additive
                { $$ = new ast::LeqExpr($1, $3, POS(@2)); }
            ;
Additive    : Multiplicative    { $$ = $1; }
            | Additive PLUS Multiplicative
                { $$ = new ast::AddExpr($1, $3, POS(@2)); }
            | Additive MINUS Multiplicative
                { $$ = new ast::SubExpr($1, $3, POS(@2)); }
            ;
Multiplicative  : Unary     { $$ = $1; }
            |   Multiplicative TIMES Unary
                { $$ = new ast::MulExpr($1, $3, POS(@2)); }
            |   Multiplicative SLASH Unary
                { $$ = new ast::DivExpr($1, $3, POS(@2)); }
            |   Multiplicative MOD Unary
                { $$ = new ast::ModExpr($1, $3, POS(@2)); }
            ;
```

```
Unary       : Primary       { $$ = $1; }
            | MINUS Unary  %prec NEG
                { $$ = new ast::NegExpr($2, POS(@1)); }
            | BNOT Unary
                { $$ = new ast::BitNotExpr($2, POS(@1)); }
            | LNOT Unary
                { $$ = new ast::NotExpr($2, POS(@1)); }
            ;
Primary     : ICONST
                { $$ = new ast::IntConst($1, POS(@1)); }
            | LPAREN Expr RPAREN    { $$ = $2; }
            ;
```

与 **Step 2** 类似，在 `scanner.l` 中增加相应的词法规则。

中端

与 **Step 3** 类似，在 `SemPass2` 类和 `Translation` 类中，添加加减乘除模对应的 `visit` 成员函数。

后端

RISC-V汇编代码生成

各运算符的汇编代码分别如下：

```
sgt     t0, t1, t2  # >

slt     t0, t1, t2  # <

slt     t0, t1, t2  # >=
xori    t0, t0

sgt     t0, t1, t2  # <=
xori    t0, t0

sub     r0, r1, r2  # ==
seqz    r0, r0

sub     r0, r1, r2  # !=
snez    r0, r0

or      r0, r1, r2  # ||
snez    r0, r0

snez    r0, r1      # &&
neg     r0, r0
and     r0, r0, r2
snez    r0, r0
```

与 **Step 3** 类似，在 `RiscvInstr` 类中声明 `SNEZ` 等指令与 `SNE` 等伪指令；在 `RiscvDesc::emitTac` 函数中，增加 TAC 的情况；在 `emitInst` 函数中，添加发射操作。特别地，在 `emitBinaryTac` 函数中，将伪指令拆解为无分支跳转的 RISC-V 指令，通过 `addInstr` 添加。

思考题

1. 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。

   `-~ 2147483647`。带符号数表示方法下，32位整数空间所能表示的正负数数量不等，于是可以通过`--2147483648`来制造越界。无法直接使用 2147483648，于是通过 2147483647 按位取反得到。

2. 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）

   ```
   #include <stdio.h>
   int main() {
     int a = -2147483648;
     int b = -1;
     printf("%d\n", a / b);
     return 0;
   }
   ```

   x86-64：出现`Integer overflow`异常，无输出。

   qemu：-2147483648。

3. 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？

   可以避免对第二个及之后的子判别表达式进行运算，节约计算资源；可以通过第一个子判别式来避免后续子判别式的运行错误，如空指针、非法访问内存等边界情况。

代码借鉴

无。