Stage-1

2019011008

无92 刘雪枫

目录

Step1

在 step1 中无代码更改。运行：

```
$ cd src
$ make
$ cd ../minidecaf-tests/
$ STEP_UNTIL=1 ./check.sh
```

运行结果：

```
The test is not in CI.
All testcases are taken into account.
11 cases in total
gcc found
qemu-riscv32 found
running tests serially
OK testcases/step1/multi_digit.c
```

```
OK testcases/step1/newlines.c
OK testcases/step1/no_newlines.c
OK testcases/step1/return_0.c
OK testcases/step1/return_2.c
OK testcases/step1/spaces.c
OK failcases/step1/badint_2.c
OK failcases/step1/badlex.c
OK failcases/step1/badparse_2.c
OK failcases/step1/no_main.c
OK failcases/step1/tailing_trash.c
Pass 11 / 11 cases in total.
```

可以成功通过测试

Step1 无思考题

Step2

代码修改

由于实验框架已经给出单目运算符 `-` 的实现，因此只需补充 `!` 和 `~` 的实现即可

修改 Makefile

由于本实验所用编译器较新，默认标准为 `-std=c++17`，因此修改 `Makefile`，指定语言标准为 `std=c++11`：

```
-CFLAGS = $(INCLUDES) $(DEFINES)  -g -Wall -pipe -DUSING_GCC
-CXXFLAGS = $(INCLUDES) $(DEFINES)  -g -Wall -pipe -DUSING_GCC
+CFLAGS = $(INCLUDES) $(DEFINES)  -g -Wall -pipe -DUSING_GCC -std=c++11
+CXXFLAGS = $(INCLUDES) $(DEFINES)  -g -Wall -pipe -DUSING_GCC -std=c++11
```

修改词法分析文件

首先修改词法分析文件 `scanner.l`，使其识别 `!` 和 `~`，并分别生成 token `LNOT` 和 `BNOT`：

```
  "+"          { return yy::parser::make_PLUS  (loc);       }
  "-"          { return yy::parser::make_MINUS  (loc);       }
+ "!"          { return yy::parser::make_LNOT  (loc);       }
+ "~"          { return yy::parser::make_BNOT  (loc);       }
  "if"         { return yy::parser::make_IF  (loc);       }
```

修改语法分析文件

然后修改语法分析文件 `parser.y`，增加非终结符 `UnaryExpr` 表示单目运算表达式，指定类型为 `mind::ast::Expr*`（与表达式 `Expr` 相同），并修改语法树，书写单目运算的语法规则：

```
  %nterm<mind::ast::Expr*> Expr
+ %nterm<mind::ast::Expr*> UnaryExpr
```

```
Expr        : ICONST
                { $$ = new ast::IntConst($1, POS(@1)); }
            | LPAREN Expr RPAREN
                { $$ = $2; }
            | Expr PLUS Expr
```

```
                { $$ = new ast::AddExpr($1, $3, POS(@2)); }
            | Expr QUESTION Expr COLON Expr
                { $$ = new ast::IfExpr($1,$3,$5,POS(@2)); }
-           | MINUS Expr  %prec NEG
+           | UnaryExpr
+           ;
+UnaryExpr  : MINUS Expr  %prec NEG
                { $$ = new ast::NegExpr($2, POS(@1)); }
+           | LNOT Expr
+               { $$ = new ast::NotExpr($2, POS(@1)); }
+           | BNOT Expr
+               { $$ = new ast::BitNotExpr($2, POS(@1)); }
            ;
```

增加类型检查

增加对 `!` 和 `~` 运算符操作数的类型检查，保证操作数类型为 `int`。故为 `SemPass2` 增加成员函数：

```
    virtual void visit(ast::NotExpr *);
    virtual void visit(ast::BitNotExpr *);
```

```
/* 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;
}
```

增加三地址码生成

增加将 `!` 和 `~` 生成三地址码的代码，即为 `Translation` 类增加成员函数：

```
     virtual void visit(ast::NegExpr *);
+    virtual void visit(ast::NotExpr *);
+    virtual void visit(ast::BitNotExpr *);
```

```
/* 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));
}
```

将三地址码生成汇编码

首先增加表示 `!` 和 `~` 所对应的三地址码所要生成的汇编的枚举成员 `LNOT` 和 `BNOT`：

```
     struct RiscvInstr : public Instr {
         /* ... */
         LI,
         SW,
         MOVE,
+        LNOT,
+        BNOT
         // You could add other instructions/pseudo instructions here
     } op_code; // operation code
```

在接收到三地址码时，先判断三地址码的类型，再调用生成对应的汇编码函数：

```
     void RiscvDesc::emitTac(Tac *t) {
         /* ... */
     case Tac::NEG:
         emitUnaryTac(RiscvInstr::NEG, t);
         break;

+    case Tac::LNOT:
+        emitUnaryTac(RiscvInstr::LNOT, t);
+        break;
+
+    case Tac::BNOT:
+        emitUnaryTac(RiscvInstr::BNOT, t);
+        break;
+
     case Tac::ADD:
         emitBinaryTac(RiscvInstr::ADD, t);
         break;
```

最后，在生成汇编处，判断要生成的汇编指令的类型（逻辑非 `LNOT` 和按位非 `BNOT`），并生成对应的汇编指令。其中逻辑非需要寄存器的值与 `0` 作比较，为零则结果为 `1`，否则结果为零，因此使用 `seqz` 指令；按位非则直接使用 `not` 指令即可：

```
     void RiscvDesc::emitInstr(RiscvInstr *i) {
         /* ... */
+    case RiscvInstr::LNOT:
+        oss << "seqz" << i->r0->name << ", " << i->r1->name;
+        break;
+
+    case RiscvInstr::BNOT:
+        oss << "not" << i->r0->name << ", " << i->r1->name;
+        break;
```

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。

   > 提示：发生越界的一步计算是`-`。

   答：`~2147483647`

Step3

代码修改

代码框架已经给出 `+` 的实现，只需实现 `-`、`*`、`/`、`%` 即可

修改词法分析文件

更改 `scanner.l`，增加对以上运算符的词法分析，生成 token：

```
  "}"          { return yy::parser::make_RBRACE  (loc);       }
  "+"          { 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);       }
  "!"          { return yy::parser::make_LNOT  (loc);       }
```

修改语法分析文件

更改 `parser.y`，修改语法分析规则，增加对加减法表达式和乘除法表达式的语法支持：

```
  %nterm<mind::ast::Type*> Type
  %nterm<mind::ast::Statement*> Stmt  ReturnStmt ExprStmt IfStmt  CompStmt
WhileStmt
  %nterm<mind::ast::Expr*> Expr
+ %nterm<mind::ast::Expr*> AdditiveExpr
+ %nterm<mind::ast::Expr*> MultiplicativeExpr
  %nterm<mind::ast::Expr*> UnaryExpr
+ %nterm<mind::ast::Expr*> PrimaryExpr
```

```
- Expr        : ICONST
-                { $$ = new ast::IntConst($1, POS(@1)); }
-              | LPAREN Expr RPAREN
```

```
-                 { $$ = $2; }
-             | Expr PLUS Expr
-                 { $$ = new ast::AddExpr($1, $3, POS(@2)); }
-             | Expr QUESTION Expr COLON Expr
+ Expr        : Expr QUESTION Expr COLON Expr
                  { $$ = new ast::IfExpr($1,$3,$5,POS(@2)); }
-             | UnaryExpr
+             | AdditiveExpr
              ;
- UnaryExpr   : MINUS Expr  %prec NEG
+ AdditiveExpr        : MultiplicativeExpr
+                     | AdditiveExpr PLUS MultiplicativeExpr
+                         { $$ = new ast::AddExpr($1, $3, POS(@2)); }
+                     | AdditiveExpr MINUS MultiplicativeExpr
+                         { $$ = new ast::SubExpr($1, $3, POS(@2)); }
+                     ;
+ MultiplicativeExpr  : UnaryExpr
+                     | MultiplicativeExpr TIMES UnaryExpr
+                         { $$ = 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
+             | MINUS UnaryExpr  %prec NEG
                  { $$ = new ast::NegExpr($2, POS(@1)); }
-             | LNOT Expr
+             | LNOT UnaryExpr
                  { $$ = new ast::NotExpr($2, POS(@1)); }
-             | BNOT Expr
+             | BNOT UnaryExpr
                  { $$ = new ast::BitNotExpr($2, POS(@1)); }
              ;
+ PrimaryExpr : ICONST
+                 { $$ = new ast::IntConst($1, POS(@1)); }
+             | LPAREN Expr RPAREN
+                 { $$ = $2; }
```

增加类型检查

```
      class SemPass2 : public ast::Visitor {
          /* ... */
      virtual void visit(ast::AddExpr *);
+     virtual void visit(ast::SubExpr *);
+     virtual void visit(ast::MulExpr *);
+     virtual void visit(ast::DivExpr *);
+     virtual void visit(ast::ModExpr *);
```

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

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

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

/* 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;
}
```

增加三地址码生成

修改 `Translation` 类，生成相应的三地址码：

```
      class Translation : public ast::Visitor {
          /* ... */
      virtual void visit(ast::AddExpr *);
+     virtual void visit(ast::SubExpr *);
+     virtual void visit(ast::MulExpr *);
+     virtual void visit(ast::DivExpr *);
+     virtual void visit(ast::ModExpr *);
```

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

/* Translating an ast::MulExpr node.
 */
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));
}

/* Translating an ast::DivExpr node.
 */
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));
}

/* 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));
}
```

将三地址码生成汇编码

所做修改与 Step2 中几乎完全相同：

```
      struct RiscvInstr : public Instr {
          /* ... */
          LNOT,
-         BNOT
+         BNOT,
+         SUB,
+         MUL,
+         DIV,
+         MOD
```

```
      void RiscvDesc::emitTac(Tac *t) {
          /* ... */
+     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::MOD, t);
+         break;
```

使用 `sub`、`mul`、`div`、`rem` 指令分别实现 `-`、`*`、`/`、`%`：

```
      void RiscvDesc::emitInstr(RiscvInstr *i) {
          /* ... */
+     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::MOD:
+         oss << "rem" << i->r0->name << ", " << i->r1->name << ", "
+             << i->r2->name;
+         break;
```

思考题

1. 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 = 左操作数;
  int b = 右操作数;
  printf("%d\n", a / b);
  return 0;
}
```

答：左操作数为 `int`（设此处为 4 个字节）的最小值，即 `-2147483647 - 1`，右操作数为 `-1`，即完整程序如下：

```
#include <stdio.h>

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

在我的电脑（x86 架构，WSL2 上的 `Fedora Linux 35 (Workstation Edition)` 操作系统）上，程序运行会崩溃，显示出 `Floating point exception` 异常；在 RISC-V 32 架构上，使用 QEMU 运行，得到结果 `-2147483648`。

Step4

代码修改

本节需实现比较运算符（`==`、`!=`、`<`、`<`、`<=`、`>=`）以及逻辑运算符（`&&`、`||`）

首先修改 `scanner.l`，将这些运算符纳入词法分析，更改方法与前两个 `Step` 几乎相同，故此处不再赘述。

对 `parser.y`，需要更改语法树，支持这些表达式：

```
+ %nterm<mind::ast::Expr*> LogicalOrExpr
+ %nterm<mind::ast::Expr*> LogicalAndExpr
+ %nterm<mind::ast::Expr*> EqualityExpr
+ %nterm<mind::ast::Expr*> RationalExpr
```

```
  Expr        : Expr QUESTION Expr COLON Expr
-                 { $$ = new ast::IfExpr($1,$3,$5,POS(@2)); }
-             | AdditiveExpr
+                 { $$ = new ast::IfExpr($1, $3, $5, POS(@2)); }
+             | LogicalOrExpr
```

```
                ;
+ LogicalOrExpr       : LogicalAndExpr
+                     | LogicalOrExpr OR LogicalAndExpr
+                         { $$ = new ast::OrExpr($1, $3, POS(@2)); }
+                     ;
+ LogicalAndExpr      : EqualityExpr
+                     | LogicalAndExpr AND EqualityExpr
+                         { $$ = new ast::AndExpr($1, $3, POS(@2)); }
+                     ;
+ EqualityExpr        : RationalExpr
+                     | EqualityExpr EQU RationalExpr
+                         { $$ = new ast::EquExpr($1, $3, POS(@2)); }
+                     | EqualityExpr NEQ RationalExpr
+                         { $$ = new ast::NeqExpr($1, $3, POS(@2)); }
+                     ;
+ RationalExpr        : AdditiveExpr
+                     | RationalExpr LT AdditiveExpr
+                         { $$ = new ast::LesExpr($1, $3, POS(@2)); }
+                     | RationalExpr GT AdditiveExpr
+                         { $$ = new ast::GrtExpr($1, $3, POS(@2)); }
+                     | RationalExpr LEQ AdditiveExpr
+                         { $$ = new ast::LeqExpr($1, $3, POS(@2)); }
+                     | RationalExpr GEQ AdditiveExpr
+                         { $$ = new ast::GeqExpr($1, $3, POS(@2)); }
+                     ;
  AdditiveExpr        : MultiplicativeExpr
                      | AdditiveExpr PLUS MultiplicativeExpr
                         { $$ = new ast::AddExpr($1, $3, POS(@2)); }
```

然后更改类型检查代码，以 `||` 为例，其余的运算符都是类似的：

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

生成三地址码的代码，以及根据三地址码调用生成汇编的函数的过程也均与 Step3 中相似，故不赘述

在生成 RISC-V 汇编时，本次很多运算符难以使用一条指令完成，需要拆分为多条指令：

1. `||` 运算符，可以先使用 `or` 指令进行按位或，此时若两个数中有一个不为零，则结果不为零。因此使用 `snez` 指令即可得到结果
2. `&&` 运算符，两个数均不为零，即两个数至少有一个为零的非。而两个数至少有一个为零，只需要对两个操作数分别使用 `seqz` 指令到目标寄存器上，则两个数只要有一个为零，则结果为 `1`。然后对结果使用 `seqz` 来取非即可。因此使用三条指令，即三条 `seqz` 即可得到结果
3. `==` 和 `!=` 均先使用 `sub` 指令相减，然后对其使用 `seqz`（或 `snez`）判断其是否为零即可

4. `<` 和 `>` 均可以使用内建的 `slt` 和 `sgt` 指令以一条指令完成
5. `>=` 和 `<=` 只需要将 `sgt` 和 `slt` 的结果使用 `seqz` 取非即可，两条指令可以完成

在代码中，如果需要多条指令，则除了最后一条之外，每次完成一条指令都调用 `emit` 函数，向文件中写入指令，并且清空 `oss` 的 `buffer`。将该操作封装为一个闭包函数：

```
auto emitInstImpl = [&] {
    emit(EMPTY_STR, oss.str().c_str(), NULL);
    oss.str("");
    oss << std::left << std::setw(6);
};
```

则代码如下：

```
case RiscvInstr::LOR:
    oss << "or" << i->r0->name << ", " << i->r1->name << ", "
        << i->r2->name;
    emitInstImpl();
    oss << "snez" << i->r0->name << ", " << i->r0->name;
    break;
case RiscvInstr::LAND:
    oss << "seqz" << i->r0->name << ", " << i->r1->name;
    emitInstImpl();
    oss << "seqz" << i->r0->name << ", " << i->r2->name;
    emitInstImpl();
    oss << "seqz" << i->r0->name << ", " << i->r0->name;
    break;
case RiscvInstr::EQU:
    oss << "sub" << i->r0->name << ", " << i->r1->name << ", "
        << i->r2->name;
    emitInstImpl();
    oss << "seqz" << i->r0->name << ", " << i->r0->name;
    break;
case RiscvInstr::NEQ:
    oss << "sub" << i->r0->name << ", " << i->r1->name << ", "
        << i->r2->name;
    emitInstImpl();
    oss << "snez" << i->r0->name << ", " << i->r0->name;
    break;
case RiscvInstr::LES:
    oss << "slt" << i->r0->name << ", " << i->r1->name << ", "
        << i->r2->name;
    break;
case RiscvInstr::GTR:
    oss << "sgt" << i->r0->name << ", " << i->r1->name << ", "
        << i->r2->name;
    break;
case RiscvInstr::LEQ:
    oss << "sgt" << i->r0->name << ", " << i->r1->name << ", "
        << i->r2->name;
    emitInstImpl();
    oss << "seqz" << i->r0->name << ", " << i->r0->name;
    break;
case RiscvInstr::GEQ:
    oss << "slt" << i->r0->name << ", " << i->r1->name << ", "
        << i->r2->name;
```

```
    emitInstImpl();
    oss << "seqz" << i->r0->name << ", " << i->r0->name;
    break;
```

然后运行测例，即可全部通过

思考题

1. > 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？

   答：首先，我认为短路求值语义可以带来性能上的提升。因为诸如 `&&`、`||` 这样的运算符，若左操作数已知是假（真）的了，则该运算符所在的表达式的值就已经是确定的了，省略右操作数的求值过程可以避免带来不必要的计算开销。第二，短路求值的特性可以作为一些不含 `else` 的 `if` 语句的简要书写形式，使代码更短。如果是在存储器非常有限的机器上，这可以节省一些存储空间