编绎原理 · stage-1 · 实验报告

计01　容逸朗　2020010869

实验内容

实验目标

本阶段需要实现一个基于 MiniDecaf 语言的整数常量计算器，可以完成基本的数学运算和逻辑比较运算，具体支持的功能符号如下所示：

- 算术运算符号： `-`、`~`、`!`、`+`、`-`、`*`、`/`、`%`、`(`、`)`
- 比较大小：`<`、`⩽`、`⩾`, `>`, `＝`, `≠`
- 逻辑符号：`&&`、`||`

具体实现

实验内容可以分为前、中、后端三部分：

- 前端
  - `frontend/parser.y`：在 `SECTION III` 的 `Expr` 部分加入了对应语法；
  - `frontend/scanner.l`：在 `SECTION III` 加入字符匹配规则；
- 中端
  - `translation/translation.cpp (.hpp)`：加入对应操作符节点的 `visit` 函数，用以生成对应的 TAC；
  - `translation/type_check.cpp`：加入对应操作符节点的 `visit` 函数，进行节点类型检查；
- 后端
  - `asm/riscv_md.cpp`：

    主要修改下列函数的内容：
    - `emitTac`：翻译中间代码，将不同的中间码指配到不同的函数进一步翻译；
    - `emitBirnaryTac`：增加二元指令/伪指令的翻译，对于用户自定义的指令需要用多条 RiscV 指令翻译；
    - `emitUnaryTac`：增加一元指令翻译；
    - `emitInstr`：增加指令的对应规则，正确发射翻译好的指令；
  - `asm/riscv_md.hpp`：
    - 在 `RiscvInstr:Instr` 中加入使用过的指令/伪指令。

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。

- 符合条件的 minidecaf 表达式如下：

```
-~2147483647
```

首先，32 位整数所能表达的最大正整数为 2147483647，利用 ~ 按位取反后得到可表达的最小负整数 -2147483648，最后通过取负（-）操作后发生越界。(按定义结果应为 2147483648，超过了 32 位整数能表达的最大正整数)

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

- 当左操作数 a = -2147483648，右操作数 b = -1 时出现未定义行为。
- 在 MacOS, x86-64 的系统下利用 g++ 运行代码，得到：

```
Floating point exception: 8
```

- 使用 spike 运行对应的代码，得到如下结果：

```
bbl loader
-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？

- 短路求值在第一个语句为 false 的情况下不会执行之后的语句。由于这一特性，在程序有大量判定性的情况下，短路求值可以节省判断后续语句所需的时间。除此之外，也可以通过增加判别语句，避免后续判断式在运行时出错（例如数组越界等）。由此可知，合理利用短路求值的性质可以使程序变得更简洁、更容易维护，故这一特性广受程序员欢迎。

参考

实现代码的过程中参考了实验思路指导与问答墙。