编译原理试点班报告

牛远卓

2019302973

2022/07/04

本实验使依据北大教程进行的。

https://pku-minic.github.io/online-doc/#/

Lv0.1. 配置 Docker

Docker 是容器技术的一种实现, 而容器技术又是一种轻量级的虚拟化技术. 你可以简单地把容器理解成虚拟机: 容器中可以运行另一个操作系统, 它和你的宿主系统是隔离的.

当然, 容器和虚拟机实际上并不相同, 你若感兴趣可自行 STFW, 此处不做过多介绍.

基于 Docker, 我们可以很方便地完成各类 “配环境” 的操作:

- 负责配置环境的人只需要写好 `Dockerfile`, 然后使用 Docker 构建镜像即可. 和环境相关的所有内容, 包括系统里的某些配置, 或者安装的工具链, 都被封装在了镜像里.
- 需要使用环境的人只要拿到镜像, 就可以基于此创建一个容器, 然后在里面完成自己的工作. 开箱即用, 不需要任何多余的操作, 十分省时省力.
- 如果某天不再需要环境, 直接把容器和镜像删除就完事了, 没残留也不拖泥带水, 干净又卫生.

安装 Docker

你可以访问 Docker 的官方网站 来安装 Docker. 安装完毕后, 你可能需要重启你的系统.

鉴于许多其他课程都要求使用 Linux 操作系统完成各类操作, 而很多同学的电脑都安装了 Windows 系统, 所以大家的电脑中可能都配置了装有 Linux 系统的虚拟机. 考虑到这种情况, 此处需要说明: Docker 是支持 Windows, macOS 和 Linux 三大平台的, 所以**你可以直接在你的宿主系统 (而不是虚拟机中) 安装 Docker**.

安装完毕后, 打开系统的命令行:

- 如果你使用的是 macOS 或 Linux, 你可以使用系统的默认终端.
- 如果你使用的是 Windows, 你可以打开 PowerShell.

执行:

```
docker
```

你将会看到 Docker 的帮助信息.

获取编译实践的镜像

在系统的命令行中执行:

```
docker pull maxxing/compiler-dev
```

如果你使用的是 Linux 系统, 则上述命令可能需要 `sudo` 才可正常执行.

编译实践的镜像较大, 但拉取镜像的速度可能并不快. 为了加快从 Docker Hub 拉取镜像的速度, 你可以自行 STFW, 为你系统中的 Docker 配置 Docker Hub Mirror.

Docker 的基本用法

你可以使用如下命令在编译实践的 Docker 镜像中执行命令:

```
docker run maxxing/compiler-dev ls -l /
```

你会看到屏幕上出现了 `ls -l /` 命令的输出, 内容是 `compiler-dev` 镜像根目录里所有文件的列表.

这个命令实际上会完成以下几件事:

- 使用 `compiler-dev` 这个镜像创建一个临时的容器.
- 启动这个临时容器.
- 在这个临时容器中执行命令 `ls -l /`.
- 关闭容器.

这里其实出现了两个概念: "镜像" 和 "容器". 你可以把它们理解为: 前者是一个硬盘, 里面装好了操作系统, 但它是静态的, 你不能直接拿它来运行. 后者是一台电脑, 里面安装了硬盘, 就能运行对应的操作系统.

当然实际上, 你可以在容器里修改文件系统的内容, 比如创建或者删除文件, 而容器对应的镜像完全不受影响. 比如你在容器里删文件把系统搞挂了, 这时候你只需要删掉这个容器, 然后从镜像创建一个新的容器, 一切就会还原到最初的样子.

刚刚的命令会根据 `compiler-dev` 创建一个容器, 但 Docker 并不会删除这个容器. 我们可以查看目前 Docker 中所有的容器:

```
$ docker ps -a
CONTAINER ID  IMAGE                          COMMAND     CREATED         STATUS
                PORTS       NAMES
696cbe1128ca  maxxing/compiler-dev:latest  ls -l /     19 seconds ago  Exited (0)
19 seconds ago              vibrant_tharp
```

命令会列出刚刚我们执行 `docker run` 时创建的临时容器. 很多情况下, 我们只是想用镜像里的环境做一些一次性的工作, 比如用里面的测试脚本测试自己的编译器, 然后查看测试结果. 在此之后这个临时容器就没有任何作用了. 我们可以执行如下命令来删除这个容器:

```
docker rm 696cbe1128ca
```

其中, `696cbe1128ca` 是 `docker ps -a` 命令输出的容器 ID.

当然我们可以简化上述操作:

```
docker run --rm maxxing/compiler-dev ls -l /
```

这条命令会使用 `compiler-dev` 镜像创建一个临时容器, 并在其中运行 `ls -l /` 命令, 然后删除刚刚创建的临时容器. 再次执行 `docker ps -a`, 你可以看到, 刚刚创建的容器并没有留下来.

我们还可以使用另一种方式运行容器:

```
docker run -it --rm maxxing/compiler-dev bash
```

这条命令会使用 `compiler-dev` 创建容器, 并在其中执行 `bash` ——这是许多 Linux 发行版的默认 Shell, 也就是大家启动终端后看到的命令行界面. 为了能在 Shell 中操作, 我们使用了 `-it` 参数, 这个参数会开启容器的 `stdin` 以便我们输入 (`-i`), 同时 Docker 会为容器分配一个终端 (`-t`).

执行完这条命令之后, 你会发现你进入了容器的 Shell, 你可以在其中执行任何命令:

```
root@e677c2d348fe:~# ls /
bin  boot  dev  etc  home  lib  lib32  lib64  libx32  media  mnt  opt  proc  root
run  sbin  srv  sys  tmp  usr  var
```

如需退出, 你可以执行 `exit`, 或者按下 `Ctrl + D`. 因为我们添加了 `--rm` 选项, Docker 会在退出后删除刚刚的容器, 所以在这种情况下请一定不要在容器里保存重要的内容.

在许多情况下, 我们需要让 Docker 容器访问宿主系统中的文件. 比如你的编译器存放在宿主机的 `/home/max/compiler` 目录下, 你希望 Docker 容器也能访问到这个目录里的内容, 这样你就可以使用容器中的测试脚本测试你的编译器了. 你可以执行:

```
docker run -it --rm -v /home/max/compiler:/root/compiler maxxing/compiler-dev
bash
```

这条命令和之前的命令相比多了一个 `-v /home/max/compiler:/root/compiler` 选项, 这个选项代表: 我希望把宿主机的 `/home/max/compiler` 目录, 挂载 (mount) 到容器的 `/root/compiler` 目录. 这样, 在进入容器之后, 我们就可以通过访问 `/root/compiler` 来访问宿主机的 `/home/max/compiler` 目录了.

关于 Docker 的其他用法, 请参考 Docker 的官方文档, 或根据情况自行 STFW.

免责声明

!> 请务必注意如下内容!

MaxXing 在设计 `compiler-dev` 的镜像时, 只考虑了直接使用 `docker run` 命令启动容器并在其中运行程序的操作, 并且**只对这种情况进行了测试**.

如果你使用其他方式连接了 Docker 容器, 例如在容器内安装了一个 SSH 然后远程连接, 则我们**不保证**这种使用方式不会出问题!

为避免遇到更多的问题, 我们建议你按照文档的指示来使用 Docker.

Lv0.2. Koopa IR 简介

?> 本节将带你大致了解什么是 Koopa IR, 后续章节中将结合实践内容, 详细介绍 Koopa IR 对应部分的特性.

关于 Koopa IR 的具体定义, 请参考 Koopa IR 规范.

什么是 Koopa IR

Koopa IR 是一种专为北京大学编译原理课程实践设计的教学用的中间表示 (IR), 它在设计上类似 LLVM IR, 但简化了很多内容, 方便大家上手和理解.

同时, 我们为 Koopa IR 开发了对应的框架 (koopa 和 libkoopa), 大家在使用 C/C++/Rust 编程时, 可以直接调用框架的接口, 实现 Koopa IR 的生成/解析/转换.

Koopa IR 是一种强类型的 IR, IR 中的所有值 (`Value`) 和函数 (`Function`) 都具备类型 (`Type`). 这种设计避免了一些 IR 定义上的模糊之处, 例如之前的教学用 IR 完全不区分整数变量和数组变量, 很容易出现混淆; 同时可以在生成 IR 之前就确定 IR 中存在的部分问题, 例如将任意整数作为内存地址并向其中存储数据.

Koopa IR 中, 基本块 (basic block) 必须是显式定义的. 即, 在描述函数内的指令时, 你必须把指令按照基本块分组, 每个基本块结尾的指令只能是分支/跳转/函数返回指令之一. 在 IR 的数据结构表示上, 指令也会被按照基本块分类. 这很大程度上方便了 IR 的优化, 因为许多优化算法都是在基本块的基础上对程序进行分析/变换的.

Koopa IR 还是一种 SSA 形式的 IR. 虽然这部分内容在课程实践中并非必须掌握, 但考虑到有些同学可能希望在课程实践的要求上, 做出一个更完备, 更强大的编译器, 我们将 Koopa IR 设计成了同时兼容非 SSA 形式和 SSA 形式的样子. 基于 SSA 形式下的 Koopa IR, 你可以开展更多复杂且有效的编译优化.

一个用 Koopa IR 编写的 “Hello, world!” 程序如下:

```
// SysY 中的 `putch` 函数的声明.
decl @putch(i32)

// 一个用来输出字符串 (其实是整数数组) 的函数.
// 函数会扫描输入的数组, 将数组中的整数视作 ASCII 码, 并作为字符输出到屏幕上,
// 遇到 0 时停止扫描.
fun @putstr(@arr: *i32) {
%entry:
  jump %loop_entry(@arr)

// Koopa IR 采用基本块参数代替 SSA 形式中的 Phi 函数.
// 当然这部分内容并不在实践要求的必选内容之中, 你无需过分关注.
%loop_entry(%ptr: *i32):
  %cur = load %ptr
  br %cur, %loop_body, %end

%loop_body:
  call @putch(%cur)
  %next = getptr %ptr, 1
  jump %loop_entry(%next)

%end:
  ret
}

// 字符串 "Hello, world!\n\0".
global @str = alloc [i32, 15], {
  72, 101, 108, 108, 111, 44, 32, 119, 111, 114, 108, 100, 33, 10, 0
}

// `main` 函数, 程序的入口.
fun @main(): i32 {
%entry:
  %str = getelemptr @str, 0
  call @putstr(%str)
  ret 0
}
```

?> **注意:** 上述代码只是一个示例, **你暂时不需要理解它的含义.** 在之后的章节中, 我们会逐步介绍 Koopa IR 的相关内容.

在线体验 Koopa IR

?> **TODO:** 待补充

本地运行 Koopa IR

假设你已经把一个 Koopa IR 程序保存在了文件 `hello.koopa` 中, 你可以在实验环境中运行这个 Koopa IR 程序:

```
koopac hello.koopa | llc --filetype=obj -o hello.o
clang hello.o -L$CDE_LIBRARY_PATH/native -lsysy -o hello
./hello
```

Lv0.4 选择你的编程语言

自 2021 年秋季学期开始, 在编译实践中, 你可以使用 C, C++ 或者 Rust 来开发你的编译器.

获取项目模板

在编译实践中, 无论是在线评测系统还是本地的自动测试脚本, 都要求你的编译器仓库里具备 Make/CMake/Cargo 三者之一的配置文件. 这样, 评测工具才能利用 Make/CMake/Cargo 来编译你的编译器, 并进行后续评测.

考虑到大部分同学对这些工具并不了解, 我们制作了三个对应的项目模板, 模板里已经包含了可直接使用的 `Makefile`, `CMakeLists.txt` 或 `Cargo.toml`. 你可以在模板的 `src` 目录中新建源代码文件, 然后开始开发你的编译器.

- **Make 模板:** 参考 `sysy-make-template`, 使用 C/C++ 开发编译器的同学可以参考.
- **CMake 模板:** 参考 `sysy-cmake-template`, 使用 C/C++ 开发编译器的同学可以参考.
- **Cargo 模板:** 你并不需要任何模板, 你只需要在安装好 Rust 工具链的情况下, 执行 `cargo new 项目名称`, 然后当前目录下就会多出一个名为 `项目名称` 的目录, 在这个目录中开发即可. 当然, 为了和 Make/CMake 模板呼应, 我们还是新建了一个叫做 `sysy-cargo-template` 的项目.

请务必仔细阅读模板中的 `README`. 如果你使用 Make/CMake 模板, 你应该根据你选择的语言 (C/C++), 更新 `Makefile` / `CMakeLists.txt` 中 `CPP_MODE` 参数的值.

建议的开发方式

我们建议大家:

1. 在你的宿主机 (Windows/macOS/Linux) 中完成 Docker 的配置.
2. 在你的宿主机中安装并配置合适的编辑器/IDE, 例如 VS Code 或 IDEA.
3. 选择对应的项目模板, 并在宿主机中开发你的编译器.
4. 配置在线评测平台的 GitLab, 并上传你的项目, 时刻使用 Git 管理代码.
5. 当需要本地测试时, 使用 Docker 环境中的自动测试脚本进行测试 (之后章节会介绍).
6. 当需要在线评测时, 向在线评测平台提交你的 GitLab 仓库 (之后章节会介绍).

课程结束后, 如果你什么都不想带走, 只需要:

1. 删除你的项目.
2. 删除 Docker 里的实验环境镜像, 必要时也可以删除 Docker.

建议的编程语言

通常情况下, 我们建议你使用 C/C++ 开发你的编译器. 这是最稳妥的选择, 因为你一定在很多其他课程中学习/使用过这两门编程语言. 在编译实践中, 除了使用 C/C++ 编程, 你还会遇到很多其他问题, 例如:

- 如何将你的项目分成多个部分, 拆分到多个文件中实现?
- 如何使用 C/C++ 构造复杂的数据结构?
- 如何优雅地管理内存?
- 如何调试, 定位并解决一些看起来很没有头绪的问题?

等等. 如果你之前仅限于 "会使用 C/C++ 编程", 相信在完成编译实践之后, 你会对 "如何使用 C/C++ 解决一个工程问题" 这件事有一个更为全面的认识.

我们建议之前接触过 Rust, 且对 Rust 感兴趣的同学使用 Rust 开发你的编译器:

- 一方面, 课程中的 Koopa IR 框架本身就是使用 Rust 开发的, 它在 Rust 层面提供了更丰富的接口和更合理的抽象.
- 另一方面, 如果你之前并未使用 Rust 完成过较为复杂的项目, 那么使用 Rust 开发编译器会是一个很不错的开始, 并且我们相信在此之后, 你对 Rust 的理解也会更进一步.

我们不建议之前从未接触过 Rust, 对内存管理/所有权等概念不敏感, 且对自己编码水平没什么信心的同学使用 Rust 开发编译器. 对这些同学来说, 在编译实践中上手 Rust 可能会比较痛苦.

Lv0.3. RISC-V 简介

本节将带你大致认识 RISC-V 指令系统, 后续章节中将结合实践内容, 详细介绍 RISC-V 指令系统中对应部分的特性.

关于 RISC-V 的更多介绍, 请参考 RISC-V 官网. 关于 RISC-V 中指令的相关定义, 请参考 RISC-V 指令速查.

什么是 RISC-V

在编译实践中, 你将开发一个生成 RISC-V 汇编的编译器. 那么首先, 什么是 RISC-V?

RISC-V, 读作 "risk-five", 是由加州大学伯克利分校设计并推广的第五代 RISC 指令系统体系结构 (ISA). RISC-V 没有任何历史包袱, 设计简洁, 高效低能耗, 且高度模块化——最主要的, 它还是一款完全开源的 ISA.

RISC-V 的指令系统由基础指令系统 (base instruction set) 和指令系统扩展 (extension) 构成. 每个 RISC-V 处理器必须实现基础指令系统, 同时可以支持若干扩展. 常用的基础指令系统有两种:

- `RV32I`: 32 位整数指令系统.
- `RV64I`: 64 位整数指令系统. 兼容 `RV32I`.

常用的标准指令系统扩展包括:

- `M` 扩展: 包括乘法和除法相关的指令.
- `A` 扩展: 包括原子内存操作相关的指令.
- `F` 扩展: 包括单精度浮点操作相关的指令.
- `D` 扩展: 包括双精度浮点操作相关的指令.
- `C` 扩展: 包括常用指令的 16 位宽度的压缩版本.

我们通常使用 `RV32/64I` + 扩展名称的方式来描述某个处理器/平台支持的 RISC-V 指令系统类型, 例如 `RV32IMA` 代表这个处理器是一个 32 位的, 支持 `M` 和 `A` 扩展的 RISC-V 处理器.

在课程实践中, 你的编译器将生成 `RV32IM` 范围内的 RISC-V 汇编.

一个使用 RISC-V 汇编编写的程序如下:

```
  # 代码段.
  .text
  # `main` 函数, 程序的入口.
  .globl main
main:
  addi  sp, sp, -16
  sw    ra, 12(sp)
  sw    s0, 8(sp)
  sw    s1, 4(sp)
  la    s0, hello_str
  li    s1, 0
1:
  add   a0, s0, s1
  lbu   a0, 0(a0)
  beqz  a0, 1f
  call  putch
  addi  s1, s1, 1
  j     1b
1:
  li    a0, 0
  lw    s1, 4(sp)
  lw    s0, 8(sp)
  lw    ra, 12(sp)
  addi  sp, sp, 16
  ret

  # 数据段.
  .data
  # 字符串 "Hello, world!\n\0".
hello_str:
  .asciz "Hello, world!\n"
```

编译/运行 RISC-V 程序

假设你已经把一个 RISC-V 汇编程序保存在了文件 `hello.S` 中, 你可以在实验环境中将这个 RISC-V 程序汇编并链接成可执行文件, 然后运行这个可执行文件:

```
clang hello.S -c -o hello.o -target riscv32-unknown-linux-elf -march=rv32im -
mabi=ilp32
ld.lld hello.o -L$CDE_LIBRARY_PATH/riscv32 -lsysy -o hello
qemu-riscv32-static hello
```

Lv1.1. 编译器的结构

编译器是如何工作的?

你之前可能完全没有思考过这个问题, 编译器对你来说只是个理所应当的工具, 只要在命令行里输入:

```
gcc hello.c -o hello
```

编译器就会把你写的文本形式的源代码, 变成二进制形式的可执行文件. 如同魔法一般, 如同梦境一般, ~充满幻想的故事, 在世界上传染, 在愉快中蔓延.~

当然, 你可能已经在其他课程中了解到, 编译器把源代码变成可执行文件的过程 (通常) 又分为:

1. **编译:** 将源代码编译为汇编代码 (assembly).
2. **汇编:** 将汇编代码汇编为目标文件 (object file).
3. **链接:** 将目标文件链接为可执行文件 (execuatble).

我们在课程中实现的编译器, 只涉及上述的第一点内容. 也就是, 我们只需要设计一个程序, 将输入的 SysY 源代码, 编译到 RISC-V 汇编即可. 在这种意义之下, 编译器通常由以下几个部分组成:

- **前端:** 通过词法分析和语法分析, 将源代码解析成抽象语法树 (abstract syntax tree, AST). 通过语义分析, 扫描抽象语法树, 检查其是否存在语义错误.
- **中端:** 将抽象语法树转换为中间表示 (intermediate representation, IR), 并在此基础上完成一些机器无关优化.
- **后端:** 将中间表示转换为目标平台的汇编代码, 并在此基础上完成一些机器相关优化.

词法/语法分析

在前端中, 我们的目的是把文本形式的源代码, 转换为内存中的一种树形的数据结构. 因为相比于处理字符串, 在树形结构上进行处理显然要更方便, 且效率更高. 把文本形式的源代码变成数据结构形式的 AST 有很多种方法, 但相对科学的方法是对源代码进行词法分析和语法分析.

对于这样一段保存在文件里的源程序:

```
int main() {
  // 我是注释诶嘿嘿
  return 0;
}
```

按照常规的思路, 在程序中, 我们会打开文件, 然后逐字符读入文件的内容. 此时相当于我们在操作一个字节流 (byte stream).

但这样做并不利于我们对输入程序作进一步处理, 因为在编程语言中, 单个的字节/字符通常没什么意义, 真正有意义的是字符组成的 “单词” (token). 就像你正在读的这篇文档, 文档里的每个字单拎出来都没什么意义, 连在一起, 组成单词, 组成句子, 组成段落和文章之后, 才会有意义.

词法分析的作用, 是把字节流转换为单词流 (token stream). 词法分析器 (lexer) 会按照某种规则读取文件, 并将文件的内容拆分成一个个 token 作为输出, 传递给语法分析器 (parser). 同时, lexer 还会忽略文件里的一些无意义的内容, 比如空格, 换行符和注释.

Lexer 生成的 token 会包含一些信息, 用来让 parser 区分 token 的种类, 以及在必要时获取 token 的内容. 例如上述程序可能能被转换成如下的 token 流:

1. **种类:** 关键字, **内容:** `int`.
2. **种类:** 标识符, **内容:** `main`.
3. **种类:** 其他字符, **内容:** `(`.
4. **种类:** 其他字符, **内容:** `)`.
5. **种类:** 其他字符, **内容:** `{`.
6. **种类:** 关键字, **内容:** `return`.
7. **种类:** 整数字面量, **内容:** `0`.
8. **种类:** 其他字符, **内容:** `;`.
9. **种类:** 其他字符, **内容:** `}`.

而语法分析的目的, 按照程序的语法规则, 将输入的 token 流变成程序的 AST. 例如, 对于 SysY 程序, 关键字 `int` 后跟的一定是一个标识符, 而不可能是一个整数字面量, 这便是语法规则. Parser 会通过某些语法分析算法, 例如 LL 分析法或 LR 分析法, 对 token 流做一系列的分析, 并最终得到 AST.

上述程序经分析后, 可能能得到如下的 AST:

```
CompUnit {
  items: [
    FuncDef {
      type: "int",
      name: "main",
      params: [],
      body: Block {
        stmts: [
          Return {
            value: 0
          }
        ]
      }
    }
  ]
}
```

?> 在这里和之前解释 token 流的部分, 我们都用了 "可能", 是因为 token 和 AST 这类数据结构仅在编译器内部出现, 并没有固定的规范. 它们的设计可以有很多种形式, 只要能够方便程序处理即可.

语义分析

在语法分析的基础上, 编译器会对 AST 做进一步分析, 以期 "理解" 输入程序的语义, 为之后的 IR 生成做准备. 一个符合语法定义的程序未必符合语义定义, 例如对于如下的 SysY 程序:

```
int main() {
  int a = 1;
  int a = 2;
  return 0;
}
```

它在语法上是正确的 (符合 SysY 语法定义), 能被 parser 构建得到 AST. 但我们可以看到, 程序在 `main` 函数里定义了两个名为 `a` 的变量, 这在 SysY 的语义约束上是不被允许的.

语义分析阶段, 编译器通常会:

- **建立符号表**, 跟踪程序里变量的声明和使用, 确定程序在某处用到了哪一个变量, 同时也可发现变量重复定义/引用未定义变量之类的错误.
- **进行类型检查**, 确定程序中是否存在诸如 "对整数变量进行数组访问" 这种类型问题. 同时标注程序中表达式的类型, 以便进行后续的生成工作. 对于某些编程语言 (例如 C++11 之后的 C++, Rust 等等), 编译器还会进行类型推断.
- **进行必要的编译期计算**. SysY 中支持使用常量表达式作为数组定义时的长度, 而我们在生成 IR 之前, 必须知道数组的长度 (SysY 不支持 VLA), 这就要求编译器必须能在编译的时候算出常量表达式的值, 同时对那些无法计算的常量表达式报错. 对于某些支持元编程的语言, 这一步可能会非常复杂.

至此, 我们就能得到一个语法正确, 语义清晰的 AST 表示了.

IR 生成

编译器通常不会直接通过扫描 AST 来生成目标代码 (汇编)——当然这么做也不是不可以, 因为从定义上讲, AST 也是一种 "中间表示". 只不过, AST 在形式上更接近源语言, 而且其中可能会包含一些更为高级的语义, 例如分支/循环, 甚至结构体/类等等, 这些内容要一步到位变成汇编还是比较复杂的.

所以, 编译器通常会将 AST 转换为另一种形式的数据结构, 我们把它称作 IR. IR 的抽象层次比 AST 更低, 但又不至于低到汇编代码的程度. 在此基础上, 无论是直接把 IR 进一步转换为汇编代码, 还是在 IR 之上做出一些优化, 都相对更容易.

有了 IR 的存在, 我们也可以大幅降低编译器的开发成本: 假设我们想开发 M 种语言的编译器, 要求它们能把输入编译成 N 种指令系统的目标代码, 在没有统一的 IR 的情况下, 我们需要开发 $M \times N$ 个相关模块. 如果我们先把所有源语言都转换到同一种 IR, 然后再将这种 IR 翻译为不同的目标代码, 我们就只需要开发 $M + N$ 个相关模块.

现实世界的确存在这样的操作, 例如 LLVM IR 就是一种被广泛使用的 IR. 有很多语言的编译器实现, 例如 Rust, Swift, Julia, 都会将源语言翻译到 LLVM IR. 同时, LLVM IR 可被生成为 x86, ARM, RISC-V 等一系列指令系统的目标代码. 此时, 编译器的前后端是完全解耦的, 两部分可以各自维护, 十分方便.

此外, IR 也可以极大地方便开发者调试自己的编译器. 在编译实践中, 你的编译器对于同一个 SysY 文件的输入, 既可以输出 Koopa IR, 也可以输出 RISC-V. 你可以借助相关测试工具来测试这两部分的正确性, 进而定位你的编译器到底是在前端/中端部分出了问题, 还是在后端的部分出了问题.

当然, 和 token, AST 等数据结构一样, IR 作为编译器内部的一种表示, 其形式也并不是唯一的. 在编译实践中, 我们指定了 IR 的形式为 Koopa IR, 大概长这样:

```
decl @getint(): i32

fun @main(): i32 {
%entry:
  @x = call @getint()
  %cond = lt @x, 10
  br %cond, %then, %else

%then:
  %0 = add %x, 1
  jump %end(%0)

%else:
  %1 = mul %x, 4
  jump %end(%1)

%end(%result: i32):
  ret %result
}
```

?> 这只是 Koopa IR 的文本形式, 在编译器运行时, Koopa IR 是一种可操作的数据结构. 我们提供的 Koopa IR 框架支持这两种形式的互相转换.

但你完全可以自行设计一种其他形式的 IR. 在业界, 编译器所使用的 IR 形式可谓百花齐放, 有的编译器 (例如 Open64) 甚至会同时使用多种形式的 IR, 以便于进行不同层次的优化.

目标代码生成

编译器进行的最后一步操作, 就是将 IR 转换为目标代码, 也就是目标指令系统的汇编代码. 通常情况下, 这一步通常要做以下几件事:

1. **指令选择:** 决定 IR 中的指令应该被翻译为哪些目标指令系统的指令. 例如前文的 Koopa IR 程序中出现的 `lt` 指令可以被翻译为 RISC-V 中的 `slt`/`slti` 指令.
2. **寄存器分配:** 决定 IR 中的值和指令系统中寄存器的对应关系. 例如前文的 Koopa IR 程序中的 `@x`, `%cond`, `%0` 等等, 它们最终可能会被放在 RISC-V 的某些寄存器中. 由于指令系统中寄存器的数量通常是有限的 (RISC-V 中只有 32 个整数通用寄存器, 且它们并不都能用来存放数据), 某些值还可能会被分配在内存中.
3. **指令调度:** 决定 IR 生成的指令序列最终的顺序如何. 我们通常希望编译器能生成一个最优化的指令序列, 它可以最大程度地利用目标平台的微结构特性, 这样生成的程序的性能就会很高. 例如编译器可能会穿插调度访存指令和其他指令, 以求减少访存导致的停顿.

当然, 课程实践中实现的编译器并不会涉及这么多内容, 你只需要重点关注第一部分.

Lv1.2. 词法/语法分析初见

上一节介绍了编译器的基本结构, 其中第一部分是词法/语法分析器. 既然我们要做一个编译器, 那么首先就必须完成这一部分.

一个好消息是, 目前已经有很多成熟的词法/语法分析器生成器, 你可以直接使用这些工具来帮助你根据正则表达式和 EBNF 生成词法/语法分析器.

当然, 你也可以使用手写递归下降分析器的方式实现词法/语法分析部分, 但本文档中不会对此方式进行讲解. 如果感兴趣, 你也许可以看看 Kaleidoscope.

一个例子

我们将使用词法/语法分析器生成器生成一个能解析下列程序的词法/语法分析器:

```
int main() {
  // 忽略我的存在
  return 0;
}
```

这个程序的语法用 EBNF 表示为 (开始符号为 `CompUnit`):

```
CompUnit  ::= FuncDef;

FuncDef   ::= FuncType IDENT "(" ")" Block;
FuncType  ::= "int";

Block     ::= "{" Stmt "}";
Stmt      ::= "return" Number ";";
Number    ::= INT_CONST;
```

其他规范见本章开头.

看懂 EBNF

EBNF, 即 Extended Backus–Naur Form, 扩展巴科斯范式, 可以用来描述编程语言的语法. 基于 SysY 的 EBNF, 我们可以从开始符号出发, 推导出任何一个符合 SysY 语法定义的 SysY 程序. 那么, 如何从上述的 EBNF 推导出示例的 SysY 程序呢?

我们不难注意到, EBNF 由若干条形如 `A ::= B;` 的规则构成. 这种规则告诉我们, 当我们遇到一个 `A` 时, 我们可以把 `A` 代换成 `B`, 这就完成了一次推导. 这其中, `A` 被称为非终结符, 因为它可以推导出其他的符号.

我们可以先从开始符, 即 `CompUnit` 开始推导:

```
CompUnit
```

利用规则 `CompUnit ::= FuncDef;`, 我们可以把 `CompUnit` 替换成:

```
FuncDef
```

进一步应用规则 `FuncDef ::= FuncType IDENT "(" ")" Block;`, 我们得到:

```
FuncType IDENT "(" ")" Block
```

这里我们遇到了一些看起来和其他符号画风不太一样的符号, 比如 `IDENT`, `"("` 和 `")"`. 在我们使用的 EBNF 记法中, 这种使用大写蛇形命名法的符号, 或者被双引号引起的字符串, 被称为终结符, 它们不能进一步被其他符号所替换, 你也不会在 EBNF 中看到它们出现在规则的左侧. 我们的目标是, 利用 EBNF 中的规则, 把开始符号推导成一系列终结符.

需要注意的是, 在谈论词法/语法分析器的时候, 词法分析器返回的一个 token 通常就代表终结符. 比如这里的 `IDENT`, 对应到词法分析器中, 实际上是词法分析器返回的, 表示标识符 (identifier) 的 token. 示例程序里的 `main` 就是一个 `IDENT`.

持续利用 `FuncType`, `Block`, `Stmt`, `Number` 规则, 我们最终可以得出这样的一串终结符:

```
"int" IDENT "(" ")" "{" "return" INT_CONST ";" "}"
```

把 `IDENT` 对应为 `main`, `INT_CONST` 对应为 `0`, 这串终结符表示的就是示例程序. 你没在这串终结符中看到空格, 换行符和注释, 是因为我们之前提到, 词法分析器会自动忽略空白符和注释.

除了非终结符, 在我们使用的 EBNF 中还会出现一些别的记法:

- `A | B` 表示可以推导出 `A`, 或者 `B`.
- `[...]` 表示方括号内包含的项可被重复 0 次或 1 次.
- `{...}` 表示花括号内包含的项可被重复 0 次或多次.

例如:

```
Params ::= Param {"," Param};
Param ::= Type IDENT;
Type ::= "int" | "long";
```

可以表示类似 `int param`, `int x, long y, int z` 这样的参数列表.

C/C++ 实现

在 C/C++ 中, 你可以使用 Flex 和 Bison 来分别生成词法分析器和语法分析器. 其中:

- Flex 用来描述 EBNF 中的终结符部分, 也就是描述 token 的形式和种类. 你可以使用正则表达式来描述 token.
- Bison 用来描述 EBNF 本身, 其依赖于 Flex 中的终结符描述. 它会生成一个 LALR parser.

关于如何入门 Flex/Bison, 你可以参考 Calc++, 或者自行 STFW/RTFM. 此处我们只介绍与开篇的示例程序相关的基本用法.

首先选择一个项目模板, 此处我们以 `Makefile` 模板为例. 在模板的 `src` 目录中新建两个文件: `sysy.l` 和 `sysy.y`, 前者将会描述词法规则并被 Flex 读取, 后者将会描述语法规则并被 Bison 读取. 由于 Flex 和 Bison 生成的 lexer 和 parser 会互相调用, 所以这两个文件里的内容也相互依赖.

`.l`/`.y` 文件有一些共同点, 比如它们的结构都是:

```
// 这里写一些选项, 可以控制 Flex/Bison 的某些行为

%{

// 这里写一些全局的代码
// 因为最后要生成 C/C++ 文件, 实现主要逻辑的部分都是用 C/C++ 写的
// 难免会用到头文件, 所以通常头文件和一些全局声明/定义写在这里
```

```
%}

// 这里写一些 Flex/Bison 相关的定义
// 对于 Flex，这里可以定义某个符号对应的正则表达式
// 对于 Bison，这里可以定义终结符/非终结符的类型

%%

// 这里写 Flex/Bison 的规则描述
// 对于 Flex，这里写的是 lexer 扫描到某个 token 后做的操作
// 对于 Bison，这里写的是 parser 遇到某种语法规则后做的操作

%%

// 这里写一些用户自定义的代码
// 比如你希望在生成的 C/C++ 文件里定义一个函数，做一些辅助工作
// 你同时希望在之前的规则描述里调用你定义的函数
// 那么，你可以把 C/C++ 的函数定义写在这里，声明写在文件开头
```

其中, 如果某些部分没被使用, 你可以把它们留空.

我们提供的 Make/CMake 模板会采用如下方式处理你的 Flex/Bison 文件:

```
# C++ 模式
flex -o 文件名.lex.cpp 文件名.l
bison -d -o 文件名.tab.cpp 文件名.y   # 此时 bison 还会生成 `文件名.tab.hpp`
# C 模式
flex -o 文件名.lex.c 文件名.l
bison -d -o 文件名.tab.c 文件名.y     # 此时 bison 还会生成 `文件名.tab.h`
```

于是, 假设我们使用 C++ 开发, 我们可以在 `sysy.l` 里描述 SysY 里所需的所有 token:

```
%option noyywrap
%option nounput
%option noinput

%{

#include <cstdlib>
#include <string>

// 因为 Flex 会用到 Bison 中关于 token 的定义
// 所以需要 include Bison 生成的头文件
#include "sysy.tab.hpp"

using namespace std;

%}

/* 空白符和注释 */
WhiteSpace    [ \t\n\r]*
LineComment   "//".*

/* 标识符 */
Identifier    [a-zA-Z_][a-zA-Z0-9_]*
```

```
/* 整数字面量 */
Decimal       [1-9][0-9]*
Octal         0[0-7]*
Hexadecimal   0[xX][0-9a-fA-F]+

%%

{WhiteSpace}    { /* 忽略, 不做任何操作 */ }
{LineComment}   { /* 忽略, 不做任何操作 */ }

"int"           { return INT; }
"return"        { return RETURN; }

{Identifier}    { yylval.str_val = new string(yytext); return IDENT; }

{Decimal}       { yylval.int_val = strtol(yytext, nullptr, 0); return INT_CONST;
}
{Octal}         { yylval.int_val = strtol(yytext, nullptr, 0); return INT_CONST;
}
{Hexadecimal}   { yylval.int_val = strtol(yytext, nullptr, 0); return INT_CONST;
}

.               { return yytext[0]; }

%%
```

以上内容应该不难理解:

- 文件最开头我们设置了一些选项, 你可以 RTFM 这些选项的含义. 如果不设置这些选项, Flex 就会要求我们在代码里定义一些额外的东西, 但我们实际上用不到这些东西.
- 第二部分声明了必要的头文件, 然后是经典的 `using namespace std`.
- 第三部分定义了一些正则表达式的规则, 比如空白符, 行注释等等. 这些规则其实直接写在第四部分也可以, 但那样太乱了, 还是给它们起个名字比较好理解一些.
- 第四部分定义了 lexer 的行为:
  - 遇到空白符和注释就跳过.
  - 遇到 `int`, `return` 这种关键字就返回对应的 token.
  - 遇到标识符就把标识符存起来, 然后返回对应的 token. `yytext` 代表 lexer 当前匹配到的字符串的内容, 它的类型是 `char *`, 在此处对应读取到的一个标识符. `yylval` 用来向 parser 传递 lexer 读取到的内容, `str_val` 和之后的 `int_val` 是我们在 Bison 文件中定义的字段, 之后再解释.
  - 遇到整数字面量, 先把读取到的字符串转换成整数, 然后存起来, 并返回对应的 token.
  - 遇到单个字符, 就直接返回单个字符作为 token. `.` 这个正则表达式会匹配任意单个字符, 而 `yytext[0]` 实际上读取了目前匹配到的这个字符. 在 Flex/Bison 中, token 实际就是整数, 之前出现的 `INT`, `RETURN`, `IDENT` 等, 其实是 Bison 根据我们的定义生成的枚举 (`enum`). 所以此处相当于, 我们取了当前匹配到的字符, 然后把它转成了整数, 交给 Bison 生成的 parser 处理. 在 Bison 里, 我们可以直接通过写字符 (比如 `'('`), 来表示我们希望匹配 lexer 返回的一个字符转换得到的 token.

!> 此处我们只处理了形如 `// ...` 的行注释, 你需要自行处理形如 `/* ... */` 的块注释. 块注释也可以用正则表达式表达, 但会稍微复杂一些.

之后, 我们可以在 `sysy.y` 中描述语法定义.

你应该还记得之前我们说过, parser 在解析完成后会生成 AST. 但我们现在还没有教大家怎么设计和定义 AST, 所以我们可以让 parser 把它扫描到的东西再保存成文本形式——也就是用字符串作为 AST. 我们的编译器如果读取到一个 `int main()` 程序, 那它就会原样输出一个相同的程序. (听起来是不是太无聊了点?)

```
%code requires {
  #include <memory>
  #include <string>
}

%{

#include <iostream>
#include <memory>
#include <string>

// 声明 lexer 函数和错误处理函数
int yylex();
void yyerror(std::unique_ptr<std::string> &ast, const char *s);

using namespace std;

%}

// 定义 parser 函数和错误处理函数的附加参数
// 我们需要返回一个字符串作为 AST, 所以我们把附加参数定义成字符串的智能指针
// 解析完成后, 我们要手动修改这个参数, 把它设置成解析得到的字符串
%parse-param { std::unique_ptr<std::string> &ast }

// yylval 的定义, 我们把它定义成了一个联合体 (union)
// 因为 token 的值有的是字符串指针, 有的是整数
// 之前我们在 lexer 中用到的 str_val 和 int_val 就是在这里被定义的
// 至于为什么要用字符串指针而不直接用 string 或者 unique_ptr<string>?
// 请自行 STFW 在 union 里写一个带析构函数的类会出现什么情况
%union {
  std::string *str_val;
  int int_val;
}

// lexer 返回的所有 token 种类的声明
// 注意 IDENT 和 INT_CONST 会返回 token 的值, 分别对应 str_val 和 int_val
%token INT RETURN
%token <str_val> IDENT
%token <int_val> INT_CONST

// 非终结符的类型定义
%type <str_val> FuncDef FuncType Block Stmt Number

%%

// 开始符, CompUnit ::= FuncDef, 大括号后声明了解析完成后 parser 要做的事情
// 之前我们定义了 FuncDef 会返回一个 str_val, 也就是字符串指针
// 而 parser 一旦解析完 CompUnit, 就说明所有的 token 都被解析了, 即解析结束了
// 此时我们应该把 FuncDef 返回的结果收集起来, 作为 AST 传给调用 parser 的函数
// $1 指代规则里第一个符号的返回值, 也就是 FuncDef 的返回值
CompUnit
  : FuncDef {
```

```
    ast = unique_ptr<string>($1);
  }
  ;

// FuncDef ::= FuncType IDENT '(' ')' Block;
// 我们这里可以直接写 '(' 和 ')', 因为之前在 lexer 里已经处理了单个字符的情况
// 解析完成后, 把这些符号的结果收集起来, 然后拼成一个新的字符串, 作为结果返回
// $$ 表示非终结符的返回值, 我们可以通过给这个符号赋值的方法来返回结果
// 你可能会问, FuncType, IDENT 之类的结果已经是字符串指针了
// 为什么还要用 unique_ptr 接住它们, 然后再解引用, 把它们拼成另一个字符串指针呢
// 因为所有的字符串指针都是我们 new 出来的, new 出来的内存一定要 delete
// 否则会发生内存泄漏, 而 unique_ptr 这种智能指针可以自动帮我们 delete
// 虽然此处你看不出用 unique_ptr 和手动 delete 的区别, 但当我们定义了 AST 之后
// 这种写法会省下很多内存管理的负担
FuncDef
  : FuncType IDENT '(' ')' Block {
    auto type = unique_ptr<string>($1);
    auto ident = unique_ptr<string>($2);
    auto block = unique_ptr<string>($5);
    $$ = new string(*type + " " + *ident + "() " + *block);
  }
  ;

// 同上, 不再解释
FuncType
  : INT {
    $$ = new string("int");
  }
  ;

Block
  : '{' Stmt '}' {
    auto stmt = unique_ptr<string>($2);
    $$ = new string("{ " + *stmt + " }");
  }
  ;

Stmt
  : RETURN Number ';' {
    auto number = unique_ptr<string>($2);
    $$ = new string("return " + *number + ";");
  }
  ;

Number
  : INT_CONST {
    $$ = new string(to_string($1));
  }
  ;

%%

// 定义错误处理函数, 其中第二个参数是错误信息
// parser 如果发生错误 (例如输入的程序出现了语法错误), 就会调用这个函数
void yyerror(unique_ptr<string> &ast, const char *s) {
  cerr << "error: " << s << endl;
}
```

在文件里我们做了几件事:

- 设置一些必要的选项, 比如 `%code requires`.
- 引用头文件, 声明 lexer 函数和错误处理函数. 如果不声明这些函数的话, parser 会找不到 Flex 中定义的 lexer, 也没办法正常报错.
- 定义 parser 函数的参数. Bison 生成的 parser 函数返回类型一定是 `int`, 所以我们没办法通过返回值返回 AST, 所以只能通过参数来返回 AST 了. 当然你也可以通过全局变量来返回 AST, 但, 那样做很 dirty (如果你接触过函数式编程或了解软件工程技术的话).
- 定义了 `yylval`, token 和非终结符的类型.
- 定义了语法规则, 同时定义了 parser 解析完语法规则之后执行的操作.
- 定义了错误处理函数.

接下来我们解释一下为什么要在开头写 `%code requires`: 这个玩意做的事情和 `%{ ... %}` 是类似的, 前者会把大括号里的内容塞到 Bison 生成的头文件里, 后者会把 `...` 对应的内容塞到 Bison 生成的源文件里.

生成的头文件会包括什么内容呢? 主要是 parser 函数的定义, 和 `yylval` 的定义. 前者是给用户用的, 比如我们想在编译器里调用 Bison 生成的 parser 帮我们解析 SysY 文件, 就需要引用这个头文件. 后者前文已经介绍过, 用来在 lexer 和 parser 之间传递信息, 我们已经在 Flex 文件中引用了这个头文件.

那么, 你一定注意到, 在 Bison 文件中, 我们指定了 parser 函数的参数类型是 `unique_ptr<string> &`, `yylval.str_val` 的类型是 `string *`, 他们都依赖于标准库里对应类的定义. 如果不在头文件里引用对应的头文件, 那么我们的编译器在引用 parser 函数的时候就可能会报错, Flex 生成的 lexer 在编译的时候也一定会报错.

以上就是 Flex 和 Bison 的基本用法了, 我们只需要写不太复杂的内容 ~(真的吗?)~, 就可以得到一个 lexer 和一个 parser. 最后的最后, 我们需要新建一个 `.cpp` 文件, 比如叫做 `main.cpp`, 来写一下程序的主函数:

```
#include <cassert>
#include <cstdio>
#include <iostream>
#include <memory>
#include <string>

using namespace std;

// 声明 lexer 的输入, 以及 parser 函数
// 为什么不引用 sysy.tab.hpp 呢? 因为首先里面没有 yyin 的定义
// 其次, 因为这个文件不是我们自己写的, 而是被 Bison 生成出来的
// 你的代码编辑器/IDE 很可能找不到这个文件, 然后会给你报错 (虽然编译不会出错)
// 看起来会很烦人, 于是干脆采用这种看起来 dirty 但实际很有效的手段
extern FILE *yyin;
extern int yyparse(unique_ptr<string> &ast);

int main(int argc, const char *argv[]) {
  // 解析命令行参数. 测试脚本/评测平台要求你的编译器能接收如下参数:
  // compiler 模式 输入文件 -o 输出文件
  assert(argc == 5);
  auto mode = argv[1];
  auto input = argv[2];
  auto output = argv[4];

  // 打开输入文件, 并且指定 lexer 在解析的时候读取这个文件
  yyin = fopen(input, "r");
  assert(yyin);
```

```
  // 调用 parser 函数, parser 函数会进一步调用 lexer 解析输入文件的
  unique_ptr<string> ast;
  auto ret = yyparse(ast);
  assert(!ret);

  // 输出解析得到的 AST, 其实就是个字符串
  cout << *ast << endl;
  return 0;
}
```

完成上述内容后, 项目的目录/文件结构是这样的:

- 项目目录.
  - `src` 目录.
    - `sysy.l` 文件, 用来描述 lexer.
    - `sysy.y` 文件, 用来描述 parser.
    - `main.cpp` 文件, 定义 `main` 函数, 调用 lexer 和 parser, 并输出结果.
  - `Makefile` 文件.
  - 其他文件.

在 Docker 的实验环境中, 我们可以在项目的目录里执行:

```
make
build/compiler -koopa hello.c -o hello.koopa
```

当然, 你需要把本节开头的示例程序先放到 `hello.c` 中. 然后你会看到输出:

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

成功了! 你可以尝试修改 `main` 函数的返回值, 或者把 `main` 改成其他什么内容, 观察输出的变化.

!> **请不要在 `src` 目录中放置其他文件!** Make/CMake 模板会处理 `src` 目录中所有的 C/C++/Flex/Bison 源文件, 并试图将它们编译和链接成一个最终的可执行文件.

如果你在 `src` 目录中放置了其他的相关文件, 比如你把你要喂给你的编译器的, 用来测试的 SysY 源程序保存在了 `src/hello.c` 中, 此时链接就会出错——因为 `hello.c` 和 `main.cpp` 中都会出现 `main` 函数, 链接器会报告出现了多个同名符号.

我们建议你在项目目录中新建一个用来存放临时文件的目录, 比如一个名字叫 `debug` 的目录. 你可以把所有和你的编译器的实现无关的文件都放在这个目录中. 记得在 `.gitignore` 文件中添加这个目录, 以防你不小心把这些临时文件提交到 Git 中, 详见如何使用 Git.

!> 如果你遇到了其他更奇怪的问题, 请仔细检查你是否按照文档中描述的方式使用了实验环境的 Docker 镜像. 详见免责声明.

Rust 实现

在 Rust 中, 你可以使用 lalrpop 来帮你生成词法/语法分析器. 和前文提到的 Flex/Bison 类似, lalrpop 也是一个 LR/LALR 分析器生成器 (从名字就能看出来), 不过它同时接管了词法部分和语法部分, 用起来更简单.

关于如何入门 lalrpop, 你可以参考 lalrpop 的文档, 里面有详细的教程可供学习.

如需在你的 Rust 项目中使用 lalrpop, 你首先需要在 `Cargo.toml` 中添加对应依赖:

```
[build-dependencies]
lalrpop = "0.19.7"

[dependencies]
lalrpop-util = { version = "0.19.7", features = ["lexer"] }
```

?> 写这篇文档时 lalrpop 的最新版本是 `0.19.7`, 你也许需要检查一下目前的最新版本.

然后在项目根目录 (即 `Cargo.toml` 所在目录) 新建 `build.rs`:

```
fn main() {
  lalrpop::process_root().unwrap();
}
```

之后就完成了. Cargo 在编译你的项目时, 会自动根据 `build.rs` 的配置, 扫描项目中的 `.lalrpop` 文件, 然后使用 lalrpop 生成 lexer 和 parser.

lalrpop 的具体用法和 Flex/Bison 大同小异, 建议选用 Rust 开发编译器的同学先简单看一下前文对 Flex/Bison 的描述. 之后, 我们可以在 `src` 目录中新建 `sysy.lalrpop`, 并写入如下内容:

```
// lalrpop 里的约定
grammar;

// 约束 lexer 的行为
match {
  // 跳过空白符和注释
  r"\s*" => {},
  r"//[^\n\r]*[\n\r]*" => {},
  // 剩下的情况采用默认方式处理
  _
}

// 定义 CompUnit, 其返回值类型为 String
// parser 在解析完成后的行为是返回 FuncDef 的值
pub CompUnit: String = <func_def: FuncDef> => func_def;

// 同上, 不解释
FuncDef: String = {
  <func_type: FuncType> <id: Ident> "(" ")" <block: Block> => {
    format!("{} {}() {}", func_type, id, block)
  }
}

FuncType: String = "int" => "int".to_string();

Block: String = "{" <stmt: Stmt> "}" => format!("{{ {} }}", stmt);

Stmt: String = "return" <num: Number> ";" => format!("return {};", num);
```

```
Number: String = <num: IntConst> => num.to_string();

// 如果匹配到标识符，就返回这个字符串
// 一对尖括号在此处指代的是正则表达式匹配到的字符串 (&str)
// 关于尖括号到底代表什么，请 RTFM
Ident: String = r"[_a-zA-Z][_a-zA-Z0-9]*" => <>.to_string();

// 对整数字面量的处理方式：把匹配到的字符串按对应进制转换成数字
IntConst: i32 = {
  r"[1-9][0-9]*" => i32::from_str_radix(<>, 10).unwrap(),
  r"0[0-7]*" => i32::from_str_radix(<>, 8).unwrap(),
  r"0[xX][0-9a-fA-F]+" => i32::from_str_radix(&<>[2..], 16).unwrap(),
}
```

!> 此处我们只处理了形如 `// ...` 的行注释，你需要自行处理形如 `/* ... */` 的块注释. 块注释也可以用正则表达式表达，但会稍微复杂一些.

在 `main.rs` 里我们可以这么写:

```
use lalrpop_util::lalrpop_mod;
use std::env::args;
use std::fs::read_to_string;
use std::io::Result;

// 引用 lalrpop 生成的解析器
// 因为我们刚刚创建了 sysy.lalrpop，所以模块名是 sysy
lalrpop_mod!(sysy);

fn main() -> Result<()> {
  // 解析命令行参数
  let mut args = args();
  args.next();
  let mode = args.next().unwrap();
  let input = args.next().unwrap();
  args.next();
  let output = args.next().unwrap();

  // 读取输入文件
  let input = read_to_string(input)?;

  // 调用 lalrpop 生成的 parser 解析输入文件
  let ast = sysy::CompUnitParser::new().parse(&input).unwrap();

  // 输出解析得到的 AST
  println!("{}", ast);
  Ok(())
}
```

完成上述内容后，项目的目录/文件结构是这样的:

- 项目目录.
  - `src` 目录.
    - `sysy.lalrpop` 文件，描述了 lexer 和 parser.
    - `main.rs` 文件，定义 `main` 函数，调用 lexer 和 parser，并输出结果.
  - `build.rs` 文件，描述 lalrpop 生成 lexer 和 parser 的操作.

- `Cargo.toml` 文件.
- 其他文件.

如果需要运行这个简单的编译器, 你只需要在项目目录执行:

```
cargo run -- -koopa hello.c -o hello.koopa
```

然后你可以在命令行中看到输出:

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

很简单吧!

Lv1.3. 解析 `main` 函数

上一节我们借助词法/语法分析器生成器实现了一个可以解析 `main` 函数的简单程序, 但它离真正的编译器还有一些差距: 这个程序只能~如蜜传如蜜~——把输入的源代码转换成, 呃, 源代码, 而不是 AST.

本节将带大家快速理解 AST 要怎么设计, 以及如何让你的编译器生成 AST.

设计 AST

设计 AST 这件事其实很简单, 你首先要知道:

1. AST 保留了程序语法的结构.
2. AST 是为了方便程序的处理而存在的, 不存在什么设计规范.

所以其实你自己用着怎么舒服, 就怎么设计, 这样就好了. 你学会了吗? 现在来写一个编译器吧! (bushi

好吧, 说正经的. 你确实只需要以上这两点, 更重要的是第一点: AST 需要保留一些必要的语法结构. 或者换句话说, EBNF 长什么样, AST 就可以长什么样. 比如本章需要大家处理的 EBNF 如下:

```
CompUnit  ::= FuncDef;

FuncDef   ::= FuncType IDENT "(" ")" Block;
FuncType  ::= "int";

Block     ::= "{" Stmt "}";
Stmt      ::= "return" Number ";";
Number    ::= INT_CONST;
```

`CompUnit` 由一个 `FuncDef` 组成, `FuncDef` 由 `FuncType`, `IDENT` 和 `Block` 组成 (中间的那对括号暂时没什么实际意义). 所以在 C++ 中, 我们可以这么写:

```
struct CompUnit {
  FuncDef func_def;
};

struct FuncDef {
  FuncType func_type;
  std::string ident;
  Block block;
};
```

!> **注意:** 之后我们不会再提示你对应的代码应该写在什么文件中, 你需要结合实际情况, 来设计自己项目的结构. 比如: 另开一个文件存放 AST 的定义, 等等.

当然, 考虑到 Flex/Bison 中返回指针比较方便, 我们可以用一点点 OOP 和智能指针来解决问题:

```
// 所有 AST 的基类
class BaseAST {
 public:
  virtual ~BaseAST() = default;
};

// CompUnit 是 BaseAST
class CompUnitAST : public BaseAST {
 public:
  // 用智能指针管理对象
  std::unique_ptr<BaseAST> func_def;
};

// FuncDef 也是 BaseAST
class FuncDefAST : public BaseAST {
 public:
  std::unique_ptr<BaseAST> func_type;
  std::string ident;
  std::unique_ptr<BaseAST> block;
};

// ...
```

其他 EBNF 对应的 AST 的定义方式与之类似, 不再赘述.

?> **建议:** 考虑到很多同学在此之前并没有使用 C/C++ 编写大型项目的经验, 对于使用 C/C++ 的同学, 我们建议你将 AST 放在一个单独的头文件中.

头文件 (header file) 通常是一种用来存放变量/函数/类声明的文件. 简而言之, 如果你需要在一个 C/C++ 文件中使用另一个文件内定义的变量或函数, 你可以使用 `#include "头文件名"` 的形式, 来引入头文件中的相关声明.

不过, 头文件在写法上和普通的 C/C++ 源文件还是存在一些区别的. 在实际的工程项目中, 头文件需要考虑被多次 `#include` 的问题. 比如你在头文件 `A` 里 `#include` 了头文件 `B`, 而在源文件 `C` 里又同时 `#include` 了 `A` 和 `B`, 此时头文件 `B` 会被 `#include` 两次. 如果你没有对 `B` 做任何处理, 那么 `B` 中的声明也会出现两次, 然后编译就会出错.

这个问题可以使用在头文件中加入 include guard 的方式解决. 更简单的方法是, 你可以在你写的所有头文件的第一行添加 `#pragma once`.

Rust 实现的编译器也可以采用这种方式定义 AST, 不过一方面, lalrpop 可以很方便地给不同的语法规则定义不同的返回类型; 另一方面, 在 Rust 里用指针, 引用或者多态 (trait object) 总会有些别扭, 所以我们不如直接把不同的 AST 定义成不同的类型.

```
pub struct CompUnit {
  pub func_def: FuncDef,
}

pub struct FuncDef {
  pub func_type: FuncType,
  pub ident: String,
  pub block: Block,
}

// ...
```

生成 AST

C++ 实现中, 我们可以在 `.y` 文件里添加一个新的类型声明:

```
%union {
  std::string *str_val;
  int int_val;
  BaseAST *ast_val;
}
```

当然, 在此之前, 所有的 AST 定义应该被放入一个头文件, 同时你应该在 `.y` 文件中正确处理头文件的引用.

此外, 我们还需要修改参数类型的声明 (其他相关声明也应该被一并修改):

```
%parse-param { std::unique_ptr<BaseAST> &ast }
```

然后适当调整非终结符和语法规则的定义即可:

```
%type <ast_val> FuncDef FuncType Block Stmt
%type <int_val> Number

%%

CompUnit
  : FuncDef {
    auto comp_unit = make_unique<CompUnitAST>();
    comp_unit->func_def = unique_ptr<BaseAST>($1);
    ast = move(comp_unit);
  }
  ;

FuncDef
  : FuncType IDENT '(' ')' Block {
    auto ast = new FuncDefAST();
    ast->func_type = unique_ptr<BaseAST>($1);
    ast->ident = *unique_ptr<string>($2);
    ast->block = unique_ptr<BaseAST>($5);
    $$ = ast;
  }
  ;

// ...
```

这样, 我们就在 Bison 生成的 parser 中完成了 AST 的构建, 并将生成的 AST 返回给了 parser 函数的调用者.

Rust 中, lalrpop 的操作也与之类似 (注意尖括号的用法):

```
pub CompUnit: CompUnit = <func_def: FuncDef> => CompUnit { <> };

FuncDef: FuncDef = {
  <func_type: FuncType> <ident: Ident> "(" ")" <block: Block> => {
    FuncDef { <> }
  }
}

FuncType: FuncType = "int" => FuncType::Int;

Block: Block = "{" <stmt: Stmt> "}" => Block { <> };

Stmt: Stmt = "return" <num: Number> ";" => Stmt { <> };

Number: i32 = <num: IntConst> => <>;
```

检查生成结果

目前的编译器已经能够正确生成 AST 了, 不过生成得到的 AST 暂时只能保存在内存里. 如果我们能把 AST 的内容也输出到命令行, 我们就能检查编译器是否按照我们的意愿生成 AST 了.

在 C++ 定义的 AST 中, 我们可以借助虚函数的特性, 给 `BaseAST` 添加一个虚函数 `Dump`, 来输出 AST 的内容:

```
class BaseAST {
 public:
  virtual ~BaseAST() = default;

  virtual void Dump() const = 0;
};
```

?> 当然这里你也可以给 AST 重载流输出运算符 (`operator<<`). C++ 的玩法实在是太多了, 这里只挑相对大众且便于理解的方法介绍.

然后分别为所有其他 AST 实现 `Dump`:

```
class CompUnitAST : public BaseAST {
 public:
  std::unique_ptr<BaseAST> func_def;

  void Dump() const override {
    std::cout << "CompUnitAST { ";
    func_def->Dump();
    std::cout << " }";
  }
};

class FuncDefAST : public BaseAST {
 public:
  std::unique_ptr<BaseAST> func_type;
  std::string ident;
```

```
  std::unique_ptr<BaseAST> block;

  void Dump() const override {
    std::cout << "FuncDefAST { ";
    func_type->Dump();
    std::cout << ", " << ident << ", ";
    block->Dump();
    std::cout << " }";
  }
};

// ...
```

在 `main` 函数中, 我们就可以使用 `Dump` 方法来输出 AST 的内容了:

```
// parse input file
unique_ptr<BaseAST> ast;
auto ret = yyparse(ast);
assert(!ret);

// dump AST
ast->Dump();
cout << endl;
```

运行后编译器会输出:

```
CompUnitAST { FuncDefAST { FuncTypeAST { int }, main, BlockAST { StmtAST { 0 } }
} }
```

对于 Rust, 事情变得更简单了: 我们只需要给每个 AST 的结构体/枚举 derive `Debug` trait 即可:

```
#[derive(Debug)]
pub struct CompUnit {
  pub func_def: FuncDef,
}

#[derive(Debug)]
pub struct FuncDef {
  pub func_type: FuncType,
  pub ident: String,
  pub block: Block,
}

// ...
```

然后稍稍在 `main` 函数的 `println!` 部分加三个字符:

```
// parse input file
let ast = sysy::CompUnitParser::new().parse(&input).unwrap();
println!("{:#?}", ast);
```

运行后编译器会输出:

```
CompUnit {
    func_def: FuncDef {
        func_type: Int,
        ident: "main",
        block: Block {
            stmt: Stmt {
                num: 0,
            },
        },
    },
}
```

Lv1.4. IR 生成

上一节, 我们的编译器已经可以将只包含 `main` 函数的简单 SysY 程序解析成 AST 了——这是好的, 而且没有任何坏处, 因为我们可以在此基础上做很多事情, 比如: 生成 IR.

一旦完成了 IR 的生成, 你的编译器就已经初步成型了. 因为有一些基础设施的存在, 比如 LLVM IR, 现代的很多编译器都只会进行到 IR 生成这一步, 然后把后续的工作交给这些基础设施完成. 对于编译实践来说, 完成 IR 生成部分之后, 你就可以进行在线评测, 然后拿到编译原理课程实践部分的第一桶分 (?).

本节将讲述如何将你设计的 AST 变成编译实践中使用的 IR: Koopa IR.

语义分析

在生成 IR 之前, 你也许还记得我们在本章的第一节讲过编译器的结构:

- 编译器首先会对源代码做词法/语法分析, 生成 AST.
- 然后在 AST 上进行语义分析, 建立符号表, 做类型检查, 报告语义错误, 等等.
- 接着, 遍历语义分析后的 AST, 生成 IR.

我们会在每章的开头给出本章涉及内容的语法规范和语义规范. 一个功能完备的编译器应该能够检查输入程序是否符合语义规范, 并在发生语义错误时报错.

不过, 在课程实践中, 所有的测试用例均为符合语法/语义规范的 SysY 程序. **我们不要求你编写的编译器具备处理语法/语义错误的能力, 也不会考察这些内容.** 但我们希望学有余力的同学, 能够在自己的编译器中检查这些问题, 并对其作出合适的处理, 比如像 `clang` 或 `rustc` 一样, 给出精确到行列的错误信息, 甚至具备忽略错误继续扫描, 以及对错误给出修改建议的高级功能.

比如在本章中, 你可以在生成 IR 之前, 或生成 IR 的同时, 检查你扫描到的函数的名称是否为 `main`, 如果不是, 就向 `stderr` 输出错误信息, 并退出, 同时返回一个非零的 exit code.

Koopa IR 基础

?> 我们建议你在完成本节的内容之前, 先阅读 Lv0.2. Koopa IR 简介部分的内容.

Koopa IR 中, 最大的单位是 `Program`, 它代表一个 Koopa IR 程序. `Program` 由若干全局变量 (`Value`) 和函数 (`Function`) 构成. `Function` 又由若干基本块 (`BasicBlock`) 构成, 基本块中是一系列指令, 指令也是 `Value`. 所以 Koopa IR 程序的结构如下所示:

- `Program`
  - 全局变量列表:
    - `Value` 1.
    - `Value` 2.
    - ...

- 函数列表:
  - `Function` 1.
    - 基本块列表:
      - `BasicBlock` 1.
        - 指令列表:
          - `Value` 1.
          - `Value` 2.
          - ...
      - `BasicBlock` 2.
      - ...
  - `Function` 2.
  - ...

上述 “全局变量”, “函数”, “指令” 的概念, 大家可能都理解, Koopa IR 中的这些概念也和编程语言中的同类概念一致. 那么, “基本块” 是什么?

基本块 (basic block) 是编译领域的一个很常见的概念, 它指的是一系列指令的集合, 基本块满足:

- **只有一个入口点:** 所有基本块中的指令如果要执行跳转, 只能跳到某个基本块的开头, 而不能跳到中间.
- **只有一个出口点:** 基本块中, 只有最后一条指令能进行控制流的转移, 也就是跳到其他基本块, 或者从函数中返回 (执行 `return` 操作).

基本块的存在可以简化很多编译过程中需要进行的分析, 所以 Koopa IR 要求函数中的指令必须预先按照基本块分类. 同时, Koopa IR 约定, 函数的第一个基本块为函数的入口基本块, 也就是执行函数时, 首先会执行第一个基本块中的指令.

现阶段, 我们可以暂时忽略全局变量, 同时我们也可以暂时认为, `Program` 的函数列表里只有一个 `Function`, `Function` 的基本块列表里只有一个 `BasicBlock` (也就是入口基本块).

接下来, 基本块中必须存在指令, 也就是 `Value`. Koopa IR 中主要有以下几种 `Value` (详见 `ValueKind` 的文档):

- **各类常量:** 整数常量 (`Integer`), 零初始化器 (`ZeroInit`), 等等.
- **参数引用:** 函数参数引用 (`FuncArgRef`) 等, 用来指代传入的参数.
- **内存分配:** 全局内存分配 (`GlobalAlloc`, 所有的全局变量都是这个玩意) 和局部内存分配 (`Alloc`).
- **访存指令:** 加载 (`Load`) 和存储 (`Store`).
- **指针运算:** `GetPtr` 和 `GetElemPtr`.
- **二元运算:** `Binary`, 比如加减乘除模/比较之类的运算都属于此类.
- **控制转移:** 条件分支 (`Branch`) 和无条件跳转 (`Jump`).
- **函数相关:** 函数调用 (`Call`) 和函数返回 (`Return`).

看起来有很多很多, 但其实在本章中我们只会用到函数返回指令和整数常量, 也就是 `Return` 和 `Integer`.

所以, 现在的目标很明确了:

1. 我们应该生成一个 Koopa IR 程序.
2. 程序中有一个名字叫 `main` 的函数.
3. 函数里有一个入口基本块.
4. 基本块里有一条返回指令.
5. 返回指令的返回值就是 SysY 里 `return` 语句后跟的值, 也就是一个整数常量.

这个程序写出来长这样:

```
fun @main(): i32 {  // main 函数的定义
%entry:             // 入口基本块
  ret 0             // return 0
}
```

是不是很简单? 当然, 你可能还是会有一些疑问:

- **为什么这个函数看起来叫 `@main` 而不叫 `main`?** 这是 Koopa IR 里的规定, `Function`, `BasicBlock`, `Value` 的名字必须以 `@` 或者 `%` 开头. 前者表示这是一个 "具名符号", 后者表示这是一个 "临时符号".
  - 这两者其实没有任何区别, 但我们通常用前者表示 SysY 里出现的符号, 用后者表示你的编译器在生成 IR 的时候生成的符号. 因为 `main` 是 SysY 里定义的, 所以这个函数叫 `@main`. 关于符号名称的细节见 Koopa IR 规范.
- **`i32` 是什么?** Koopa IR 是一种强类型 IR, 也就是说, 诸如函数参数, 返回值, 所有指令, 它们都是有类型的.
  - `i32` 指的是 32 位有符号整数 (**32**-bit signed **i**nteger), 对应 SysY 里的 `int`. 你在编译实践前期相当长的一段时间内只会见到这一种类型.
  - Koopa IR 的分析器可以进行类型推导, 所以我们可以省略一部分类型标注. 比如你看到程序最后出现了 `ret 0`, 这个 `0` 也是有类型的. 但分析器知道 `0` 就是个整数, 它的类型是 `i32`, 所以我们不需要写 `ret i32 0`. 这点在目前看来没啥用, 但之后能帮我们简化很多复杂的表述.
- **`%entry` 是什么?** 之前解释过了, `%entry` 是这个基本块的名字. 因为基本块是你的编译器定义的, 所以它以 `%` 开头.
  - Koopa IR 不会约束名称的定义, 这个基本块叫什么名字都行. 你懒得起名的话, 可以设置个计数器, 把基本块和 `Value` 的名字定义成 `%0`, `%1`, ... 这样的. 只不过, 起一个有意义一些的名字能方便你 debug 编译器输出的 IR.

生成 Koopa IR

生成 Koopa IR 非常简单: 之前我们已经介绍过怎么输出你的 AST 了, 比如给所有 AST 都实现一个 `Dump` 方法, 然后去调用 `Dump` 方法即可. 输出字符串形式的 Koopa IR 与之类似, 此处不做过多赘述.

你可能会注意到, 我们刚刚提到了 "字符串形式的 Koopa IR", 难道 Koopa IR 除了字符串形式还有其他形式吗?

你好, 有的. Koopa IR 目前有两种形式:

1. **文本形式:** 就是你在前文见到的字符串形式, 方便人类阅读.
2. **内存形式:** 即数据结构的形式, 你可以把它理解为另一种 "AST", 方便程序处理.

你的编译器输出的文本形式 IR, 最终会被 Koopa IR 的相关工具 (比如 `koopac`) 读取, 变成内存形式的 IR, 然后作进一步处理. Koopa IR 的框架也提供了在两种形式的 IR 之间互相转换的接口.

所以, 考虑到上述情况, 你有以下几种生成 IR 的思路:

- 遍历 AST, 输出文本形式的 IR. 这样最简单, 适用于任何语言实现的编译器.
- 调用 Koopa IR 框架提供的接口. 使用 Rust 的同学可以尝试, 详见 Koopa IR 框架的 crates.io 以及文档.
- 像定义 AST 一样定义表示 Koopa IR 的数据结构 (比如指令/基本块/函数等等), 然后遍历 AST 输出这种结构, 再遍历这种结构输出字符串.
- 对于使用 C/C++ 的同学, 在上一条的基础上, 你可以考虑把这种结构转换成 raw program, 然后使用 `libkoopa` 中的相关接口, 将 raw program 转换成其他形式的 Koopa IR 程序.

!> 关于最后一种思路, 你可以参考 `libkoopa` 的头文件, 其中定义了 raw program 的相关结构, 以及 Koopa IR 框架对 C/C++ 暴露的相关接口. 对熟悉 C/C++ 的同学来说, 这些内容应该不难理解.

由于文档作者 MaxXing 单枪匹马持续输出, 关于 `libkoopa` 提供的基础设施, 以及如何借助这些设施生成内存形式的 Koopa IR 程序的相关内容暂未补全. 希望他有时间可以写一下.

?> 如果你基于 Make/CMake 模板, 使用 C/C++ 开发你的编译器, 那么你的编译器会和实验环境中的 `libkoopa` 库自动链接, 你无需进行任何修改.

Lv1.5. 测试

如果你已经完成了前四节的阅读, 你就可以顺利地得到一个最初级的编译器了, 接下来要做的事情是测试.

本地测试

!> 本地测试很重要! 请务必认真完成.

编译实践不同于其他同样使用 OJ 的课程: 你向 OJ 提交的并不是单个的代码文件, 而是一个完整的编译器项目. 如果不在本地的实验环境内预先测试/调试, 提交在线评测后, 你可能会遇到很多无从下手的问题.

假设你已经完成了 Docker 的配置, 你可以执行:

```
docker run -it --rm -v 项目目录:/root/compiler maxxing/compiler-dev \
  autotest -koopa -s lv1 /root/compiler
```

你需要将 `项目目录` 替换为你的编译器项目在宿主机上的路径. 同时, 在运行测试前, 你需要确保你的编译器 (假设名称为 `compiler`) 能处理如下的命令行参数:

```
compiler -koopa 输入文件 -o 输出文件
```

其中, `-koopa` 代表你的编译器要输出 Koopa IR 文件, `输入文件` 代表输入的 SysY 源文件的路径, `输出文件` 代表 Koopa IR 的输出文件路径. 你的编译器应该解析 `输入文件`, 并把生成的 Koopa IR 输出到 `输出文件` 中.

测试程序会使用你的编译器将输入编译为 Koopa IR, 然后借助 LLVM 将 Koopa IR 进一步编译成可执行文件. 最后, 测试程序执行可执行文件, 检查程序的返回值 (也就是 `main` 的返回值) 是否符合预期. 测试程序**不会**检查你输出的 Koopa IR 的形式, 你输出的 IR **只要功能正确, 即可通过测试.**

关于实验环境/测试脚本的详细使用方法, 请参考实验环境使用说明. 关于调试编译器的相关思路, 请参考调试你的编译器. 关于测试脚本的工作原理, 请 RTFSC.

上传代码到评测平台

学期初, 我们会向所有选修编译原理课的同学的 PKU 邮箱中发送在线评测平台的账号, 详情请关注课上的说明, 或课程群通知.

你可以使用发放的账号登录评测平台的代码托管平台 (eduxiji.gitlab.net), 然后新建 repo. 之后你就可以按照使用 Git 的一般流程来向代码托管平台提交代码了.

!> **注意:** 请务必将你创建的 repo 的可见性设为 “Private”, 否则所有人都将在平台上看到你提交的代码!

此外, 平台的 GitLab **不支持 SSH 登录**, 在从平台 clone 仓库或向平台提交代码时, 请注意使用 HTTPS.

在线评测

?> **TODO:** 待补充.

关于在线评测系统的详细使用方法, 请参考在线评测使用说明.

Lv2.1. 处理 Koopa IR

本节将引导你在上一章的基础上, 建立内存形式的 Koopa IR, 并在程序中访问这些数据结构.

建立内存形式的 Koopa IR

上一章中, 你的编译器已经可以输出 Koopa IR 程序了. 你可能会采用两种思路完成这一操作:

1. 遍历 AST, 输出文本形式的 Koopa IR 程序.
2. 遍历 AST, 直接建立 (某种) 内存形式的 Koopa IR, 再将其转换为文本形式输出.

对于第二种思路, 无论你是通过阅读 Koopa IR 的文档, 直接建立了内存形式 IR, 还是根据Koopa IR 规范, 自行设计了一套数据结构来表示 Koopa IR 程序, 你其实都已经得到了一个可被你程序处理的内存形式的 Koopa IR. 在目标代码生成阶段, 你可以直接让你的编译器遍历这些数据结构, 并生成代码. **此时, 你可以跳过本节.**

第一种思路可能是大部分同学会采用的思路, 因为它相当简单且直观, 实现难度很低. 但其缺点是, 你在生成目标代码之前, 不得不再次将文本形式的 Koopa IR 转换成某种数据结构——这相当于再写一个编译器. 否则, 你的程序几乎无法直接基于文本形式 IR 生成汇编.

不过好在, 我们为大家提供了能够处理 Koopa IR 的库, 你可以使用其中的实现, 来将文本形式的 IR 转换为内存形式.

C/C++ 实现

你可以使用 `libkoopa` 中的接口将文本形式 Koopa IR 转换为 raw program, 后者是 C/C++ 可以直接操作的, 由各种 `struct`, `union` 和指针组成的, 表示 Koopa IR 的数据结构.

首先你需要在代码中引用 `libkoopa` 的头文件:

```
#include "koopa.h"
```

?> **注意:** 你**只需要**在代码中引用这个头文件, 而不需要去 GitHub 上找到这个头文件, 然后把它下载下来, 放在编译器的代码目录中.

我们提供的 Make/CMake 模板会自动处理 `koopa.h` 的引用, 你不需要关心任何其他的细节. 这件事情实现的原理, 和你的编译器可以直接引用 `stdio.h`, 而不需要把这个文件放在代码目录里, 是完全一致的. 如果你对此感兴趣, 可以 RTFSC 模板的代码.

然后, 假设你生成的 Koopa IR 程序保存在了字符串 (类型为 `const char *`) `str` 中, 你可以执行:

```
```

```c
// 解析字符串 str, 得到 Koopa IR 程序
koopa_program_t program;
koopa_error_code_t ret = koopa_parse_from_string(str, &program);
assert(ret == KOOPA_EC_SUCCESS);  // 确保解析时没有出错
// 创建一个 raw program builder, 用来构建 raw program
koopa_raw_program_builder_t builder = koopa_new_raw_program_builder();
// 将 Koopa IR 程序转换为 raw program
koopa_raw_program_t raw = koopa_build_raw_program(builder, program);
// 释放 Koopa IR 程序占用的内存
koopa_delete_program(program);

// 处理 raw program
// ...

// 处理完成, 释放 raw program builder 占用的内存
// 注意, raw program 中所有的指针指向的内存均为 raw program builder 的内存
// 所以不要在 raw program 处理完毕之前释放 builder
koopa_delete_raw_program_builder(builder);
```

其中, raw program 的结构和我们在 Lv1 中提到的 Koopa IR 程序的结构完全一致:

- 最上层是 `koopa_raw_program_t`, 也就是 `Program`.
- 之下是全局变量定义列表和函数定义列表.
  - 在 raw program 中, 列表的类型是 `koopa_raw_slice_t`.
  - 本质上这是一个指针数组, 其中的 `buffer` 字段记录了指针数组的地址 (类型是 `const void **`), `len` 字段记录了指针数组的长度, `kind` 字段记录了数组元素是何种类型的指针
  - 在访问时, 你可以通过 `slice.buffer[i]` 拿到列表元素的指针, 然后通过判断 `kind` 来决定把这个指针转换成什么类型.
- `koopa_raw_function_t` 代表函数, 其中是基本块列表.
- `koopa_raw_basic_block_t` 代表基本块, 其中是指令列表.
- `koopa_raw_value_t` 代表全局变量, 或者基本块中的指令.

如果你的项目基于我们提供的 Make/CMake 模板, 则测试脚本/评测平台编译你的项目时, 会自动链接 `libkoopa`, 你无需为此操心.

如果你在编码时需要让编辑器/IDE 识别 `koopa.h` 文件中的声明, 你可以在 `libkoopa` 的仓库中获取到这个头文件. 同时, 头文件中包含了所有 raw program 相关的数据结构的定义 (含详细注释), 你可以通过 RTFSC 来进一步了解 raw program 的结构.

Rust 实现

你可以使用 `koopa` 这个 crate 来处理 Koopa IR. 请根据 crates.io 上的说明, 在你的项目中添加最新版本的 `koopa` 的依赖.

假设你生成的 Koopa IR 程序保存在了字符串 `s` 中, 你可以执行:

```rust
let driver = koopa::front::Driver::from(s);
let program = driver.generate_program().unwrap();
```

来得到一个内存形式的 Koopa IR 程序. 这个程序的结构和 Lv1 中的描述完全一致, 详情请参考文档.

Lv2.2. 目标代码生成

之前, 你的编译器已经可以把:

```
int main() {
  // 阿卡林
  return 0;
}
```

编译成如下的 Koopa IR 程序:

```
fun @main(): i32 {
%entry:
  ret 0
}
```

我们的目标是, 进一步把它编译为:

```
  .text
  .globl main
main:
  li a0, 0
  ret
```

遍历内存形式的 IR

C/C++ 实现

C/C++ 中, 得到 raw program 过后, 你可以遍历它的函数列表:

```
koopa_raw_program_t raw = ...;
// 使用 for 循环遍历函数列表
for (size_t i = 0; i < raw.funcs.len; ++i) {
  // 正常情况下, 列表中的元素就是函数, 我们只不过是在确认这个事实
  // 当然, 你也可以基于 raw slice 的 kind, 实现一个通用的处理函数
  assert(raw.funcs.kind == KOOPA_RSIK_FUNCTION);
  // 获取当前函数
  koopa_raw_function_t func = (koopa_raw_function_t) raw.funcs.buffer[i];
  // 进一步处理当前函数
  // ...
}
```

对于示例程序, `raw.funcs.len` 一定是 `1`, 因为程序里显然只有一个函数. 进一步查看 `koopa.h` 中 `koopa_raw_function_t` 的定义, 我们据此可以遍历函数中所有的基本块:

```
for (size_t j = 0; j < func->bbs.len; ++j) {
  assert(func->bbs.kind == KOOPA_RSIK_BASIC_BLOCK);
  koopa_raw_basic_block_t bb = (koopa_raw_basic_block_t) func->bbs.buffer[j];
  // 进一步处理当前基本块
  // ...
}
```

同样, 对于示例程序, `func->bbs.len` 也一定是 `1`, 因为 `@main` 函数内只有一个名为 `%entry` 的基本块. 遍历指令的方法与之类似, 此处不再赘述.

最后你应该可以得到一个 `koopa_raw_value_t`, 我们需要对其进行进一步处理:

```
koopa_raw_value_t value = ...;
// 示例程序中, 你得到的 value 一定是一条 return 指令
assert(value->kind.tag == KOOPA_RVT_RETURN);
// 于是我们可以按照处理 return 指令的方式处理这个 value
// return 指令中, value 代表返回值
koopa_raw_value_t ret_value = value->kind.data.ret.value;
// 示例程序中, ret_value 一定是一个 integer
assert(ret_value->kind.tag == KOOPA_RVT_INTEGER);
// 于是我们可以按照处理 integer 的方式处理 ret_value
// integer 中, value 代表整数的数值
int32_t int_val = ret_value->kind.data.integer.value;
// 示例程序中, 这个数值一定是 0
assert(int_val == 0);
```

上述代码展示了在 C/C++ 中如何读取 raw program 中函数, 基本块和 return/integer 指令的数据, 相信你从中不难举一反三, 推导出访问其他指令的方法. 当然, 要想实现 "访问 Koopa IR" 的目标, 我们最好还是定义一系列对应的访问函数, 用 DFS 的思路来访问 raw program. 以 C++ 为例:

```
// 函数声明略
// ...

// 访问 raw program
void Visit(const koopa_raw_program_t &program) {
  // 执行一些其他的必要操作
  // ...
  // 访问所有全局变量
  Visit(program.values);
  // 访问所有函数
  Visit(program.funcs);
}

// 访问 raw slice
void Visit(const koopa_raw_slice_t &slice) {
  for (size_t i = 0; i < slice.len; ++i) {
    auto ptr = slice.buffer[i];
    // 根据 slice 的 kind 决定将 ptr 视作何种元素
    switch (slice.kind) {
      case KOOPA_RSIK_FUNCTION:
        // 访问函数
        Visit(reinterpret_cast<koopa_raw_function_t>(ptr));
        break;
      case KOOPA_RSIK_BASIC_BLOCK:
        // 访问基本块
        Visit(reinterpret_cast<koopa_raw_basic_block_t>(ptr));
        break;
      case KOOPA_RSIK_VALUE:
        // 访问指令
        Visit(reinterpret_cast<koopa_raw_value_t>(ptr));
        break;
      default:
        // 我们暂时不会遇到其他内容, 于是不对其做任何处理
        assert(false);
    }
  }
}
```

```
// 访问函数
void Visit(const koopa_raw_function_t &func) {
  // 执行一些其他的必要操作
  // ...
  // 访问所有基本块
  Visit(func->bbs);
}

// 访问基本块
void Visit(const koopa_raw_basic_block_t &bb) {
  // 执行一些其他的必要操作
  // ...
  // 访问所有指令
  Visit(bb->insts);
}

// 访问指令
void Visit(const koopa_raw_value_t &value) {
  // 根据指令类型判断后续需要如何访问
  const auto &kind = value->kind;
  switch (kind.tag) {
    case KOOPA_RVT_RETURN:
      // 访问 return 指令
      Visit(kind.data.ret);
      break;
    case KOOPA_RVT_INTEGER:
      // 访问 integer 指令
      Visit(kind.data.integer);
      break;
    default:
      // 其他类型暂时遇不到
      assert(false);
  }
}

// 访问对应类型指令的函数定义略
// 视需求自行实现
// ...
```

相信从上述代码中, 你已经基本掌握了在 C/C++ 中遍历 raw program 的方法.

Rust 实现

Rust 中对内存形式 Koopa IR 的处理方式和 C/C++ 大同小异: 都是遍历列表, 然后根据类型处理其中的元素. 比如得到 `Program` 后, 你同样可以遍历其中的函数列表:

```
let program = ...;
for &func in program.func_layout() {
  // 进一步访问函数
  // ...
}
```

但需要注意的是, 在 Koopa IR 的内存形式中, “IR 的数据” 和 “IR 的 layout” 是彼此分离表示的.

?> “Layout” 直译的话是 “布局”. 这个词不太好用中文解释, 虽然 Koopa IR 的相关代码确实是我写的, 我也確實是個平時講中文的中國大陸北方網友.

比如对于基本块的指令列表: 指令的数据并没有直接按照指令出现的顺序存储在列表中. 指令的数据被统一存放在函数内的一个叫做 `DataFlowGraph` 的结构中, 同时每个指令具有一个指令 ID (或者也可以叫 handle), 你可以通过 ID 在这个结构中获取对应的指令. 指令的列表中存放的其实是指令的 ID.

这么做看起来多套了一层, 但实际上 “指令 ID” 和 “指令数据” 的对应关系, 就像 C/C++ 中 “指针” 和 “指针所指向的内存” 的对应关系, 理解起来并不复杂. 至于为什么不直接把数据放在列表里? 为什么不用指针或者引用来代替 “指令 ID”? 如果对 Rust 有一定的了解, 你应该会知道这么做的后果...

所以, 此处你可以通过遍历 `func_layout`, 来按照程序中函数出现的顺序来获取函数 ID, 然后据此从程序中拿到函数的数据, 进行后续访问:

```
for &func in program.func_layout() {
  let func_data = program.func(func);
  // 访问函数
  // ...
}
```

访问基本块和指令也与之类似, 但需要注意: 基本块的数据里没有指令列表, 只有基本块的名称之类的信息. 基本块的指令列表在函数的 layout 里.

```
// 遍历基本块列表
for (&bb, node) in func_data.layout().bbs() {
  // 一些必要的处理
  // ...
  // 遍历指令列表
  for &inst in node.insts().keys() {
    let value_data = func_data.dfg().value(inst);
    // 访问指令
    // ...
  }
}
```

指令的数据里记录了指令的具体种类, 你可以通过模式匹配来处理你感兴趣的指令:

```
use koopa::ir::ValueKind;
match value_data.kind() {
  ValueKind::Integer(int) => {
    // 处理 integer 指令
    // ...
  }
  ValueKind::Return(ret) => {
    // 处理 ret 指令
    // ...
  }
  // 其他种类暂时遇不到
  _ => unreachable!(),
}
```

如需遍历访问 Koopa IR, 你同样需要将程序实现成 DFS 的模式. 此处推荐通过为内存形式 IR 扩展 trait 来实现这一功能:

```
// 根据内存形式 Koopa IR 生成汇编
trait GenerateAsm {
  fn generate(&self, /* 其他必要的参数 */);
}

impl GenerateAsm for koopa::ir::Program {
  fn generate(&self) {
    for &func in program.func_layout() {
      program.func(func).generate();
    }
  }
}

impl GenerateAsm for koopa::ir::FunctionData {
  fn generate(&self) {
    // ...
  }
}
```

生成汇编

生成汇编的思路和生成 Koopa IR 的思路类似, 都是遍历数据结构, 输出字符串. 此处不做过多赘述. 不过我们依然需要解释一下, 你生成的 RISC-V 汇编到底做了哪些事情.

在 SysY 程序中, 我们定义了一个 `main` 函数, 这个函数什么也没做, 只是返回了一个整数, 之后就退出了. RISC-V 程序所做的事情与之一致:

1. 定义了 `main` 函数.
2. 将作为返回值的整数加载到了存放返回值的寄存器中.
3. 执行返回指令.

所以你需要知道几件事:

- **如何定义函数?**
  - 所谓函数, 从处理器的角度看只不过是一段指令序列. 调用函数时处理器跳转到序列的入口执行, 执行到序列中含义是 "函数返回" 的指令时, 处理器退出函数, 回到调用函数前的指令序列继续执行.
  - 在汇编层面 "定义" 函数, 其实只需要标注这个序列的入口在什么位置即可, 其余函数返回之类的操作都属于函数内的指令要完成的事情.
- **RISC-V 中如何设置返回值?**
  - RISC-V 指令系统的 ABI 规定, 返回值应当被存入 `a0` 和 `a1` 寄存器中. RV32I 下, 寄存器宽度为 32 位, 所以用寄存器可以传递两个 32 位的返回值.
  - 在编译实践涉及的所有情况下, 函数的返回值只有 32 位. 所以我们在传递返回值时, 只需要把数据放入 `a0` 寄存器即可.
- **如何将整数加载到寄存器中?**
  - RISC-V 的汇编器支持 `li` 伪指令. 这条伪指令的作用是加载立即数 (**l**oad **i**mmediate) 到指定的寄存器中.

所以你输出的汇编的含义其实是:

```
  .text           # 声明之后的数据需要被放入代码段中
  .globl main     # 声明全局符号 main, 以便链接器处理
main:             # 标记 main 的入口点
  li a0, 0        # 将整数 0 加载到存放返回值的 a0 寄存器中
  ret             # 返回
```

关于 RISC-V 指令的官方定义, 请参考 RISC-V 的规范. 当然, 我们整理了编译实践中需要用到的 RISC-V 指令的相关定义, 你可以参考 RISC-V 指令速查.

最后的最后, 有时你可能实在不清楚, 对于某段特定的 C/SysY 程序, 编译器到底应该输出什么样的 RISC-V 汇编. 此时你可以去 Compiler Explorer 这个网站, 该网站可以很方便地查看某种编译器编译某段 C 程序后究竟会输出何种汇编.

你可以在网站右侧的汇编输出窗口选择使用 “RISC-V rv32gc clang (trunk)” 编译器, 然后将编译选项设置为 `-O3 -g0`, 并查看窗口内的汇编输出.

Lv2.3. 测试

你已经写出了一个功能简单但初具形态的编译器了, 恭喜! 之后的章节中, 我们会进一步给这个编译器添加新的特性, 直至它能处理所有符合 SysY 语言规范的程序. 但在此之前, 你应该完成编译器的测试工作.

本地测试

假设你已经完成了 Docker 的配置, 你可以执行:

```
docker run -it --rm -v 项目目录:/root/compiler maxxing/compiler-dev \
  autotest -riscv -s lv1 /root/compiler
```

你需要将 `项目目录` 替换为你的编译器项目在宿主机上的路径. 同时, 在运行测试前, 你需要确保你的编译器 (假设名称为 `compiler`) 能处理如下的命令行参数:

```
compiler -riscv 输入文件 -o 输出文件
```

其中, `-riscv` 代表你的编译器要输出 RISC-V 汇编文件, `输入文件` 代表输入的 SysY 源文件的路径, `输出文件` 代表 RISC-V 汇编的输出文件路径. 你的编译器应该解析 `输入文件`, 并把生成的 RISC-V 汇编输出到 `输出文件` 中.

?> 为了同时兼容 Koopa IR 和 RISC-V 的测试, 你的编译器应该能够根据命令行参数的值, 判断当前正在执行何种测试, 然后决定只需要进行 IR 生成, 还是同时需要进行目标代码生成, 并向输出文件中输出 Koopa IR 或 RISC-V 汇编.

关于实验环境/测试脚本的详细使用方法, 请参考实验环境使用说明.

在线评测

?> **TODO:** 待补充.

关于在线评测系统的详细使用方法, 请参考在线评测使用说明.

Lv3.1. 一元表达式

因为本章开头的语法规范里突然多出一大堆产生式, 所以你可能会觉得有些手足无措. 那我们不如把这堆新加的内容再做一些拆分, 先来实现一元表达式的部分.

本节新增/变更的语法规范如下:

```
Stmt        ::= "return" Exp ";";

Exp         ::= UnaryExp;
PrimaryExp  ::= "(" Exp ")" | Number;
Number      ::= INT_CONST;
UnaryExp    ::= PrimaryExp | UnaryOp UnaryExp;
UnaryOp     ::= "+" | "-" | "!";
```

你需要让你的编译器支持 `+`, `-` 和 `!` 运算, 同时支持括号表达式.

一个例子

```
int main() {
  return +(- -!6);  // 看起来像个颜文字
}
```

词法/语法分析

本节新增了三个运算符: `+`, `-` 和 `!`. 对于 C/C++ 实现, 你可以对 Flex 部分稍作修改, 使其对这些运算符进行特殊处理. 但你应该还记得, Lv1 中, 我们添加了可以匹配任意单个字符的规则. 因为新增的三个运算符刚好各自只有一个字符, 所以这个规则也可以用来匹配这些新增内容, 所以你不对 Flex 部分作修改也是可以的.

语法分析部分, 你需要根据语法规则, 设计一些新的 AST. 也许你需要回顾一下 Lv1 中关于如何设计 AST 的相关部分. 然后修改你的语法分析器, 使其支持根据新增的语法规则生成对应的 AST.

需要注意的是, 目前的 EBNF 中出现了一些包含 `|` 的规则, 比如:

```
PrimaryExp  ::= "(" Exp ")" | Number;
UnaryExp    ::= PrimaryExp | UnaryOp UnaryExp;
```

对于这种情况, 设计 AST 时, 你可以采取很多种处理方式, 比如:

- 对 `::=` 右侧的每个规则都设计一种 AST, 在 parse 到对应规则时, 构造对应的 AST.
- 或者, 只为 `::=` 左侧的符号设计一种 AST, 使其涵盖 `::=` 右侧的所有规则. 比如在 Rust 中, 你可以用 `enum` 来表达这种行为.

实现语法分析器时, 你需要使用语法分析器支持的语法实现 `|`, 详情请自行 RTFM 或 STFW.

语义分析

暂无需要添加的内容.

IR 生成

查询 Koopa IR 规范, 你会发现, Koopa IR 并不支持一元运算, 而只支持如下的二元运算:

- **比较运算:** `ne` (比较不等), `eq` (比较相等), `gt` (比较大于), `lt` (比较小于), `ge` (比较大于等于), `le` (比较小于等于). 返回整数形式的真 (非 0) 或假 (0).
- **算术运算:** `add` (加法), `sub` (减法), `mul` (乘法), `div` (除法), `mod` (模运算).
- **位运算:** `and` (按位与), `or` (按位或), `xor` (按位异或).
- **移位:** `shl` (左移), `shr` (逻辑右移), `sar` (算术右移).

这是因为, 目前已知有意义的一元操作均可用二元操作表示:

- **变补 (取负数)**: 0 减去操作数.
- **按位取反**: 操作数异或全 1 (即 `-1`).
- **逻辑取反**: 操作数和 0 比较相等.

所以, 示例代码可以生成如下的 Koopa IR:

```
fun @main(): i32 {
%entry:
  %0 = eq 6, 0
  %1 = sub 0, %0
  %2 = sub 0, %1
  ret %2
}
```

你需要注意的是:

- `+` 运算实际上不会生成任何 IR.
- Koopa IR 中, `%` 后可以跟任意正整数, 同样表示临时符号.
- Koopa IR 是 "单赋值" 的, 即: 所有符号都只能在定义的时候被赋值一次, 所以你必须保证函数内所有的符号 (包括指令和基本块) 都具备不同的名称. 如下的 IR 程序是不合法的:

```
fun @main(): i32 {
%entry:
  %0 = eq 6, 0
  // 不能重复定义符号
  %0 = sub 0, %0
  ret %0
}
```

目标代码生成

对于由示例代码生成的 Koopa IR:

```
fun @main(): i32 {
%entry:
  %0 = eq 6, 0
  %1 = sub 0, %0
  %2 = sub 0, %1
  ret %2
}
```

可以进一步生成如下的 RISC-V 汇编:

```
  .text
  .globl main
main:
  # 实现 eq 6, 0 的操作, 并把结果存入 t0
  li    t0, 6
  xor   t0, t0, x0
  seqz  t0, t0
  # 减法
  sub   t1, x0, t0
  # 减法
```

```
  sub   t2, x0, t1
  # 设置返回值并返回
  mv    a0, t2
  ret
```

你需要注意的是:

- 在文本形式的 Koopa IR 中, 你经常会看到两条先后出现的指令, 例如上述示例中的 `%0 = eq 6, 0` 和 `%1 = sub 0, %0`. 而在处理 Koopa IR 的内存形式时, 你需要注意:
  - 在 C/C++ 中, 并没有 `%0 = ...` 和 `%1 = ...` 这样的结构. 前一条 `eq` 指令和后一条 `sub` 指令的指针会被按照顺序, 先后存放在 `%entry` 基本块中的指令列表中. `sub` 指令中出现的 `%0`, 表示在内存形式中实际上就是一个指向前一条 `eq` 指令的指针.
  - 在 Rust 中, 同样不存在 `%0 = ...` 之类的结构. 指令的 ID 会按照顺序存放在基本块的指令 layout 中, 同时, 你可以在 `dfg` 中根据 ID 访问指令的数据. `sub` 指令中出现的 `%0`, 表示在内存形式中, 实际上就是前一条 `eq` 指令的 ID.
- 将上述 Koopa IR 翻译到 RISC-V 汇编的方式有很多种, 你不难找到一些更简洁的翻译方式, 虽然它们实现起来可能并不那么简单.
- `x0` 是一个特殊的寄存器, 它的值恒为 0, 且向它写入的任何数据都会被丢弃.
- `t0` 到 `t6` 寄存器, 以及 `a0` 到 `a7` 寄存器可以用来存放临时值.

?> 你也许会注意到, 如果按照一条指令的结果占用一个临时寄存器的目标代码生成思路, 在表达式足够复杂的情况下, 所有的临时寄存器很快就会被用完. 本章出现的测试用例中会避免出现这种情况, 同时, 你可以自行思考: 用何种方式可以缓解这个问题. 在 Lv4 中, 我们会给出一种一劳永逸的思路来解决这个问题.

Lv3.2. 算数表达式

本节新增/变更的语法规范如下:

```
Exp         ::= AddExp;
PrimaryExp  ::= ...;
Number      ::= ...;
UnaryExp    ::= ...;
UnaryOp     ::= ...;
MulExp      ::= UnaryExp | MulExp ("*" | "/" | "%") UnaryExp;
AddExp      ::= MulExp | AddExp ("+" | "-") MulExp;
```

一个例子

```
int main() {
  return 1 + 2 * 3;
}
```

词法/语法分析

词法/语法分析部分同上一节, 你需要处理新增的运算符, 根据新增的语法规范设计新的 AST, 或者修改现有的 AST, 然后让你的 parser 支持新增的语法规范.

语义分析

暂无需要添加的内容.

IR 生成

示例代码可以生成如下的 Koopa IR:

```
fun @main(): i32 {
%entry:
  %0 = mul 2, 3
  %1 = add 1, %0
  ret %1
}
```

按照常识, `*` / / / `%` 运算符的优先级应该高于 `+` / `-` 运算符, 所以你生成的代码应该先计算乘法, 后计算加法. SysY 的语法规范中已经体现了运算符的优先级, 如果你正确建立了 AST, 那么你在后序遍历 AST 时, 生成的代码自然会是上述形式.

目标代码生成

关键部分的 RISC-V 汇编如下:

```
  li  t0, 2
  li  t1, 3
  mul t1, t0, t1
  li  t2, 1
  add t2, t1, t2
```

Lv3.3. 比较和逻辑表达式

本节新增/变更的语法规范如下:

```
Exp         ::= LOrExp;
PrimaryExp  ::= ...;
Number      ::= ...;
UnaryExp    ::= ...;
UnaryOp     ::= ...;
MulExp      ::= ...;
AddExp      ::= ...;
RelExp      ::= AddExp | RelExp ("<" | ">" | "<=" | ">=") AddExp;
EqExp       ::= RelExp | EqExp ("==" | "!=") RelExp;
LAndExp     ::= EqExp | LAndExp "&&" EqExp;
LOrExp      ::= LAndExp | LOrExp "||" LAndExp;
```

一个例子

```
int main() {
  return 1 <= 2;
}
```

词法/语法分析

同上一节. 但需要注意的是, 本节出现了一些两个字符的运算符, 比如例子中的 `<=`. 你需要修改 lexer 来适配这一更改.

语义分析

暂无需要添加的内容.

IR 生成

示例代码可以生成如下的 Koopa IR:

```
fun @main(): i32 {
%entry:
  %0 = le 1, 2
  ret %0
}
```

!> **注意:** Koopa IR 只支持按位与或, 而不支持逻辑与或, 但你可以用其他运算拼凑出这些运算.

详见本节下一部分的描述.

目标代码生成

关键部分的 RISC-V 汇编如下:

```
  li    t0, 1
  li    t1, 2
  # 执行小于等于操作
  sgt   t1, t0, t1
  seqz  t1, t1
```

如果你查阅 RISC-V 规范第 24 章 (Instruction Set Listings, 130 页), 你会发现 RISC-V 只支持小于指令 (`slt` 等). 而上述汇编中出现的 `sgt` 是一个伪指令, 也就是说, 这条指令并不真实存在, 而是用其他指令实现的.

已知, `slt t0, t1, t2` 指令的含义是, 判断寄存器 `t1` 的值是否小于 `t2` 的值, 并将结果 (0 或 1) 写入 `t0` 寄存器. 思考:

- `sgt t0, t1, t2` (判断 `t1` 的值是否大于 `t2` 的值) 是怎么实现的?
- 上述汇编中判断小于等于的原理是什么?
- 如何使用 RISC-V 汇编判断大于等于?

你可以使用 Lv2 提到的方法, 看看 Clang 是如何将这些运算翻译成 RISC-V 汇编的, 比如这个例子.

Lv3.4. 测试

目前你的编译器已经可以处理一些简单的表达式计算了, 就像计算器一样, 可喜可贺!

在完成本章之前, 先进行一些测试吧.

本地测试

测试 Koopa IR:

```
docker run -it --rm -v 项目目录:/root/compiler maxxing/compiler-dev \
  autotest -koopa -s lv3 /root/compiler
```

测试 RISC-V 汇编:

```
docker run -it --rm -v 项目目录:/root/compiler maxxing/compiler-dev \
  autotest -riscv -s lv3 /root/compiler
```

测试程序对编译器的要求和之前章节一致, 此处及之后章节将不再赘述.

在线评测

?> **TODO:** 待补充.

Lv4.1. 常量

本节新增/变更的语法规范如下:

```
Decl          ::= ConstDecl;
ConstDecl     ::= "const" BType ConstDef {"," ConstDef} ";";
BType         ::= "int";
ConstDef      ::= IDENT "=" ConstInitVal;
ConstInitVal  ::= ConstExp;

Block         ::= "{" {BlockItem} "}";
BlockItem     ::= Decl | Stmt;

LVal          ::= IDENT;
PrimaryExp    ::= "(" Exp ")" | LVal | Number;

ConstExp      ::= Exp;
```

一个例子

```
int main() {
  const int x = 1 + 1;
  return x;
}
```

词法/语法分析

本节增加了一些新的关键字, 你需要修改 lexer 来支持它们. 同样, 你需要根据新增的语法规则, 来设计新的 AST, 以及更新你的 parser 实现.

本节的 EBNF 中出现了一种新的表示: `{ ... }`, 这代表花括号内包含的项可被重复 0 次或多次. 在 AST 中, 你可以使用 `std::vector` / `Vec` 来表示这种结构.

语义分析

本章的语义规范较前几章来说复杂了许多, 你需要在编译器中引入一些额外的结构, 以便进行必要的语义分析. 这种结构叫做**符号表**.

符号表可以记录作用域内所有被定义过的符号的信息. 在本节中, 符号表负责记录 `main` 函数中, 常量符号和其值之间的关系. 具体来说, 符号表需要支持如下操作:

- **插入符号定义:** 向符号表中添加一个常量符号, 同时记录这个符号的常量值, 也就是一个 32 位整数.
- **确认符号定义是否存在:** 给定一个符号, 查询符号表中是否存在这个符号的定义.
- **查询符号定义:** 给定一个符号表中已经存在的符号, 返回这个符号对应的常量值.

你可以选用合适的数据结构来实现符号表.

在遇到常量声明语句时, 你应该遍历 AST, 直接算出语句右侧的 `ConstExp` 的值, 得到一个 32 位整数, 然后把这个常量定义插入到符号表中.

在遇到 `LVal` 时, 你应该从符号表中查询这个符号的值, 然后用查到的结果作为常量求值/IR 生成的结果. 如果没查到, 说明 SysY 程序出现了语义错误, 也就是程序里使用了未定义的常量.

你可能需要给你的 AST 扩展一些必要的方法, 来实现编译期常量求值.

!> SysY 中 “常量” 的定义和 C 语言中的定义有所区别: SysY 中, 所有的常量必须能在编译时被计算出来; 而 C 语言中的常量仅代表这个量不能被修改.

SysY 中的常量有些类似于 C++ 中的 `consteval`, 或 Rust 中的 `const`.

IR 生成

所有的常量定义均已在编译期被求值, 所以:

- `Exp` 里, 所有出现 `LVal` 的地方均可直接替换为整数常量.
- 因上一条, 常量声明本身不需要生成任何 IR.

综上所述, 本节的 IR 生成部分不需要做任何修改.

示例程序生成的 Koopa IR 为:

```
fun @main(): i32 {
%entry:
  ret 2
}
```

目标代码生成

由于 IR 生成部分未作修改, 目标代码生成部分也无需变更.

示例程序生成的 RISC-V 汇编为:

```
  .text
  .globl main
main:
  li a0, 2
  ret
```

Lv4.2. 变量和赋值

本节新增/变更的语法规范如下:

```
Decl          ::= ConstDecl | VarDecl;
ConstDecl     ::= ...;
BType         ::= ...;
ConstDef      ::= ...;
ConstInitVal  ::= ...;
VarDecl       ::= BType VarDef {"," VarDef} ";";
VarDef        ::= IDENT | IDENT "=" InitVal;
```

```
InitVal       ::= Exp;

...

Block         ::= ...;
BlockItem     ::= ...;
Stmt          ::= LVal "=" Exp ";"
                | "return" Exp ";";
```

一个例子

```
int main() {
  int x = 10;
  x = x + 1;
  return x;
}
```

词法/语法分析

同上一节, 你需要设计新的 AST, 同时修改 parser 的实现.

语义分析

与上一节类似, 你依然需要一个符号表来管理所有的变量定义. 与常量定义不同的是, 变量定义存储的是变量的符号, 及其对应的 `alloc`, 即变量的内存分配. 本节的 IR 生成部分将解释 `alloc` 在 Koopa IR 中的含义.

所以, 你需要修改你的符号表, 使其支持保存一个符号所对应的常量信息或者变量信息. 也就是说, 符号表里使用符号可以查询到一个数据结构, 这个数据结构既可以用来存储变量信息, 又可以用来存储常量信息. 在 C/C++ 中, 你可以使用一个 `struct`, tagged union 或者 `std::variant` 来实现这一性质. 在 Rust 中, 使用 `enum` (本身就是个 tagged union) 来实现这一性质再合适不过了.

在遇到 `LVal` 时, 你需要从符号表中查询这个符号的信息, 然后用查到的结果作为常量求值/IR 生成的结果. 注意, 如下情况属于语义错误:

- 在进行常量求值时, 从符号表里查询到了变量而不是常量.
- 在处理赋值语句时, 赋值语句左侧的 `LVal` 对应一个常量, 而不是变量.
- 其他情况, 如符号重复定义, 或者符号未定义.

IR 生成

要想实现变量和赋值语句, 只使用我们之前介绍到的 Koopa IR 指令是做不到的. 比如你**不能**把本节的示例程序翻译成如下形式:

```
// 错误的
fun @main(): i32 {
%entry:
  %0 = 10
  %0 = add %0, 1
  ret %0
}
```

虽然它 "看起来" 很符合常识. 但你也许还记得, 在 Lv3.1 中我们介绍过, Koopa IR 是 **"单赋值"** 的, 所有符号都只能在定义的时候被赋值一次. 而在上面的程序中, `%0` 被赋值了两次, 这是不合法的.

如果要表示变量的定义, 使用和赋值, 我们必须引入三种新的指令: `alloc`, `load` 和 `store`:

```
// 正确的
fun @main(): i32 {
%entry:
  // int x = 10;
  @x = alloc i32
  store 10, @x

  // x = x + 1;
  %0 = load @x
  %1 = add %0, 1
  store %1, @x

  // return x;
  %2 = load @x
  ret %2
}
```

三种新指令的含义如下:

- `%x = alloc T`: 申请一块类型为 `T` 的内存. 在本节中, `T` 只能是 `i32`. 返回申请到的内存的指针, 也就是说, `%x` 的类型是 `T` 的指针, 记作 `*T`.
  - 这个操作和 C 语言中 `malloc` 函数的惯用方式十分相似:

```
// 申请一块可以存放 int 型数据的内存, 这块内存本身的类型是 int*
int *x = (int *)malloc(sizeof(int));
```

- `%x = load %y`: 从指针 `%y` 对应的内存中读取数据, 返回读取到的数据. 如果 `%y` 的类型是 `*T`, 则 `%x` 的类型是 `T`.
- `store %x, %y`: 向指针 `%y` 对应的内存写入数据, 不返回任何内容. 如果 `%y` 的类型是 `*T`, 则 `%x` 的类型必须是 `T`.

当然, 以防你忘记 Koopa IR 的规则, 再次提醒: 示例中的 `alloc` 叫做 `@x`, 是因为这个 `alloc` 对应 SysY 中的变量 `x`. 实际上, 这个 `alloc` 叫什么名字都行, 比如 `@AvavaAvA`, `%x` 或者 `%0`, 叫这个名字只是为了调试方便.

目标代码生成

本节中出现了新的 Koopa IR 指令, 这些指令要求我们进行内存分配. 实际上, 此处提到的 "内存分配" 可能和你脑海里的 "内存分配" 不太一样——后者在其他编程语言中通常指分配堆内存 (heap memory), 而此处指的**通常**是分配栈内存 (stack memory).

为什么要说 "通常"? 我们在之前的章节提到过, RISC-V 指令系统中定义了一些寄存器, 可供我们存放一些运算的中间结果. 我们都知道 (应该吧?), 计算机的存储系统分很多层次, 每一层的访问速度都存在数量级上的差距, 处理器访问寄存器的速度要远快于访问各级缓存和内存的速度. 如果我们能找到一种方法, 把 SysY/Koopa IR 程序中的变量映射到寄存器上, 那程序的速度肯定会得到大幅度提升.

事实上, 这种办法是存在的, 我们把这种方法叫做寄存器分配. 此时, `alloc` 对应的可能就不再是一块栈内存了, 而是一个 (或多个) 寄存器. 但要实现一种真正高效的寄存器分配算法是极为困难的: 寄存器分配问题本身是一个 NPC 问题, 编译器必须消耗大量的时间才能算出最优的分配策略. 考虑到执行效率, 业界的编译器在进行寄存器分配时, 通常会采取一些启发式算法, 但这些算法的实现依旧不那么简单.

另一方面, 指令系统中定义的寄存器的数量往往是有限的, 比如 RISC-V 中有 32 个 ISA 层面的整数寄存器, 但其中只有不多于 28 个寄存器可以用来存放变量. 如果输入程序里的某个函数相对复杂, 编译器就无法把其中所有的变量都映射到寄存器上. 此时, 这些变量就不得不被 "spill" 到栈内存中.

你当然可以选择实现一些复杂的寄存器分配算法, 详见 Lv9+.2. 寄存器分配的相关内容. 但此时, 我建议你先实现一种最简单的寄存器分配方式: **把所有变量都放在栈上.** 什么? 这也算是寄存器分配吗? 寄存器都没用到啊喂! 正所谓大道至简——你可以认为, 寄存器分配算法要做的事情是: 决定哪些变量应该被放在寄存器中, 哪些变量应该被放在栈上. 我们的算法只不过是固执地选择了后者……而已, 你很难说它不是一种寄存器分配算法.

在此之前, 你应该了解, RISC-V 程序中的栈内存是如何分配的.

栈帧

程序在操作系统中运行时, 操作系统会为其分配堆内存和栈内存. 其中, 栈内存的使用方式是连续的. 程序 (进程) 执行前, 操作系统会为进程设置栈指针 (stack pointer)——通常是一个寄存器. 栈指针会指向栈内存的起点.

进程内, 在执行函数调用时, 函数开头的指令会通过移动栈指针, 来在栈上开辟出一块仅供这个函数自己使用的内存区域, 这个区域就叫做栈帧 (stack frame). 函数在执行的过程中, 会通过 "栈指针 + 偏移量" 的手段访问栈内存, 所以在函数退出前, 会有相关指令负责复原栈指针指向的位置, 以防调用这个函数的函数访问不到正确的栈内存.

至于栈内存为什么叫栈内存, 想必是很容易理解的: 函数在调用和返回的过程中, 栈内存中栈帧的变化, 就符合栈 LIFO 的特性.

那函数通常会在栈帧中存放什么内容呢? 最容易想到的是函数的返回地址. 所谓 "函数调用" 和 "函数返回" 的操作, 其实可以分解成以下几个步骤:

- **函数调用:** 把函数调用指令的后一条指令的地址存起来, 以便函数调用结束后返回. 然后跳转到函数的入口开始执行.
- **函数返回:** 找到之前保存的返回地址, 然后跳转到这个地址, 执行函数调用之后的指令.

由于函数的调用链可能非常之长, 函数的返回地址通常会被放在内存中, 而不是寄存器中. 因为相比于内存, 寄存器能存放的返回地址个数实在是少得可怜, 更何况寄存器还要用来存储变量和其他数据. 所以, 类似 "返回地址" 这种数据就可以被放在栈帧里.

扯点别的: 说到函数调用和返回, 你肯定会好奇 RISC-V 是怎么做的. 在 RISC-V 中, 函数的调用和返回可以通过 `call` 和 `ret` 这两条伪指令来完成. 我们之前提到过, "伪指令" 并不是 ISA 中定义的指令, 这些指令通常是用其他指令实现出来的. 比如 `call func` 这条伪指令代表调用函数 `func`, 它的实际上会被汇编器替换成:

```
auipc ra, func地址的高位偏移量
jalr  ra, func地址的低位偏移量(ra)
```

`auipc` 指令负责加载 `func` 的一部分地址到 `ra` 寄存器中. `jalr` 指令负责执行跳转操作: 把 `func` 剩下的一部分地址和 `ra` 里刚刚加载的地址相加得到完整地址, 然后把返回地址 (`jalr` 后一条指令的地址) 存入 `ra`, 最后跳转到刚刚计算得出的地址, 来执行函数.

首先为什么地址会被拆成两半? 因为 RV23I 里的地址和指令的长度都是 32 位, 要想在指令里编码其他内容 (比如指令的操作), 就必然不可能把整个地址全部塞进指令中.

然后你会发现, `call` 指令实际上会把返回地址放到 `ra` 寄存器中, 而不是直接放到栈上. 实际上, `ra` 寄存器的全名正是 "return address register". 这是因为 RISC-V 是一种 RISC 指令系统, RISC 中为了精简指令的实现, 通常只有加载/存储类的指令能够访存, 其他指令只会操作寄存器.

`ret` 伪指令会被汇编器替换成:

```
jalr x0, 0(ra)
```

这条指令会读出 `ra` 寄存器的值, 加一个偏移量 0 (相当于没加) 得到跳转的目标地址, 然后把后一条指令的地址放到寄存器 `x0` 里, 最后跳转到刚刚计算出来的地址处执行.

首先我们之前曾提到 `x0` 是一个特殊的寄存器, 它的值恒为 0, 任何试图向写入 `x0` 写入数据的操作都相当于什么都没干, 所以这里的 `jalr` 相当于没写任何寄存器. 其次, `ra` 中存放的是函数的返回地址, 所以这条指令实际上相当于执行了函数返回的操作.

RISC-V 仅用一种指令就同时实现了函数调用和函数返回的操作 (`auipc` 只是加载了地址), 我们很难不因此被 RISC-V 的设计之精妙所折服. RISC-V 中还有很多其它这样的例子, 如你对此感兴趣, 可以查看 RISC-V 规范第 139 页, 其中列举了很多伪指令和对应的实现方法.

但感慨之余, 你可能会意识到, RISC-V 的返回地址不直接保存在栈帧里, 而是被放在一个叫做 `ra` 的寄存器里——毕竟 RISC-V 有 32 个寄存器, 用掉一个也还有很多富余. 这么做其实有一个好处: 函数可以自由决定自己要不要把返回地址保存到栈帧里. 如果当前函数里没有再调用其他函数, 那 `ra` 的值就不会被覆盖, 我们就可以放心大胆地使用 `ret` 来进行函数返回, 同时节省一次内存写入的开销——事实上, 你的编译器到目前为止生成的 RISC-V 汇编都是这么处理的.

那么最后, 总结一下, 栈帧里通常会放这些东西:

- **函数的返回地址:** 上文已经解释过.
- **某些需要保存的寄存器:** 函数执行时可能会用到某些寄存器, 为了避免完成函数调用后, "调用者" 函数的寄存器被 "被调用者" 函数写乱了, 在执行函数调用时, 调用者/被调用者可能会把某些寄存器保存在栈帧里.
- **被 spill 到栈上的局部变量:** 因为寄存器不够用了, 这些变量只能放在栈上.
- **函数参数:** 在调用约定中, 函数的一部分参数会使用寄存器传递. 但因为寄存器数量是有限的, 函数的其余参数会被放到栈帧里.

RISC-V 的栈帧

在不同的指令系统中, 栈帧的布局可能都是不同的. 指令系统的调用约定 (calling convension) 负责规定程序的栈帧应该长什么样. 你的程序要想在 RISC-V 的机器上运行, 尤其是和其他标准 RISC-V 的程序交互, 就必须遵守 RISC-V 的调用约定.

关于栈帧的约定大致如下:

- `sp` 寄存器用来保存栈指针, 它的值必须是 16 字节对齐的 (RISC-V 规范第 107 页).
- 函数中栈的生长方向是从高地址到低地址, 也就是说, 进入函数的时候, `sp` 的值应该减小.
- `sp` 中保存的地址是当前栈帧最顶部元素的地址.
- 栈帧的布局如下图所示. 其中, 栈帧分为三个区域, 这三个区域并不是必须存在的. 例如, 如果函数中没有局部变量, 那局部变量区域的大小就为 0.


栈帧的布局

生成代码

Koopa IR 程序中需要保存到栈上的内容包括:

- `alloc` 指令分配的内存. 目前你的编译器只会生成 `alloc i32`, 所以应为其分配的内存大小为 4 字节.
- 除 `alloc` 外, 其他任何存在返回值的指令的返回值, 比如 `%0 = load @x`, `%1 = add %0, 1` 中的 `%0` 和 `%1`. 当然, 诸如 `store` 等不存在返回值的指令不需要处理. 目前你的编译器中只会使用到返回值类型为 `i32` 的指令, 所以应为其分配的内存大小为 4 字节.

生成代码的步骤如下:

1. 扫描函数中的所有指令, 算出需要分配的栈空间总量 S (单位为字节).
2. 计算 S 对齐到 16 后的数值, 记作 $S^\prime$.
3. 在函数入口处, 生成更新栈指针的指令, 将栈指针减去 $S^\prime$. 这个过程叫做函数的 prologue.
    - 你可以用 `addi sp, sp, 立即数` 指令来实现这个操作, 这条指令会为 `sp` 加上立即数.
    - 需要注意的是, `addi` 指令中立即数的范围是 $[-2048, 2047]$, 即 12 位有符号整数的范围. 立即数一旦超过这个范围, 你就只能用 `li` 加载立即数到一个临时寄存器 (比如 `t0`), 然后用 `add` 指令来更新 `sp` 了.
4. 使用 RISC-V 中的 `lw` 和 `sw` 指令来实现 `load` 和 `store`.
    - `lw 寄存器1, 偏移量(寄存器2)` 的含义是将 `寄存器2` 的值和 `偏移量` 相加作为内存地址, 然后从内存中读取一个 32 位的数据放到 `寄存器1` 中.
    - `sw 寄存器1, 偏移量(寄存器2)` 的含义是将 `寄存器2` 的值和 `偏移量` 相加作为内存地址, 将 `寄存器1` 中的值存入到内存地址对应的 32 位内存空间中.
    - `lw`/`sw` 中偏移量的范围和 `addi` 一致.
5. 对于在指令中用到的其他指令的返回值, 比如 `add %1, %2` 中的 `%1` 和 `%2`, 用 `lw` 指令从栈帧中读数据到临时寄存器中, 然后再计算结果.
6. 对于所有存在返回值的指令, 比如 `load` 和 `add`, 计算出指令的返回值后, 用 `sw` 指令把返回值存入栈帧.
7. 函数返回前, 即 `ret` 指令之前, 你需要生成复原栈指针的指令, 将栈指针加上 $S^\prime$. 这个过程叫做函数的 epilogue.

如何判断一个指令存在返回值呢? 你也许还记得 Koopa IR 是强类型 IR, 所有指令都是有类型的. 如果指令的类型为 `unit` (类似 C/C++ 中的 `void`), 则这条指令不存在返回值.

在 C/C++ 中, 每个 `koopa_raw_value_t` 都有一个名叫 `ty` 的字段, 它的类型是 `koopa_raw_type_t`. `koopa_raw_type_t` 中有一个字段叫做 `tag`, 存储了这个类型具体是何种类型. 如果它的值为 `KOOPA_RTT_UNIT`, 说明这个类型是 `unit` 类型.

在 Rust 中, 每个 `ValueData` 都有一个名叫 `ty()` 的方法, 这个方法会返回一个 `&Type`. 而 `Type` 又有一个方法叫做 `is_unit()`, 如果这个方法返回 `true`, 说明这个类型是 `unit` 类型.

或者你实在懒得判断的话, 给所有指令都分配栈空间也不是不行, 只不过这样会浪费一些栈空间.

示例程序生成的 RISC-V 汇编为:

```
  .text
  .globl main
main:
  # 函数的 prologue
  addi sp, sp, -16

  # store 10, @x
  li t0, 10
  sw t0, 0(sp)

  # %0 = load @x
  lw t0, 0(sp)
  sw t0, 4(sp)

  # %1 = add %0, 1
  lw t0, 4(sp)
  li t1, 1
  add t0, t0, t1
  sw t0, 8(sp)
```

```
  # store %1, @x
  lw t0, 8(sp)
  sw t0, 0(sp)

  # %2 = load @x
  lw t0, 0(sp)
  sw t0, 12(sp)

  # ret %2, 以及函数的 epilogue
  lw a0, 12(sp)
  addi sp, sp, 16
  ret
```

栈帧的分配情况如下图所示:

在这个示例中, S 恰好对齐了 16 字节, 所以 $S = S^\prime$. 此外, RISC-V 的调用约定中没有规定栈帧内数据的排列方式, 比如顺序, 或者对齐到栈顶还是栈底 (除了函数参数必须对齐到栈顶之外), 所以你想怎么安排就可以怎么安排, 只要你的编译器内采用统一标准即可.

Lv4.3. 测试

本章的涉及的内容相对较多, 理解难度相较前几章也更难. 你能进行到这一步实属不易, 给你比一个~大母猪~大拇指! (´∀`)b

目前你的编译器已经可以处理常量和变量了, 能处理的程序看起来也已经有模有样了, 十分不错!

在完成本章之前, 先进行一些测试吧.

本地测试

测试 Koopa IR:

```
docker run -it --rm -v 项目目录:/root/compiler maxxing/compiler-dev \
  autotest -koopa -s lv4 /root/compiler
```

测试 RISC-V 汇编:

```
docker run -it --rm -v 项目目录:/root/compiler maxxing/compiler-dev \
  autotest -riscv -s lv4 /root/compiler
```

在线评测

?> **TODO:** 待补充.

Lv5.1. 实现

本节新增/变更的语法规范如下:

```
Stmt ::= LVal "=" Exp ";"
       | [Exp] ";"
       | Block
       | "return" [Exp] ";";
```

一个例子

```
int main() {
  int a = 1;
  {
    a = 2;
    int a = 3;
  }
  return a;
}
```

词法/语法分析

本节新增了语法规则 `[Exp] ";"`, 这代表一条仅由 `Exp` 组成的语句, 比如 `1 + 2;`. 你可能需要设计新的 AST, 同时更新你的 parser 实现.

本节的 EBNF 中出现了一种新的表示: `[ ... ]`, 这代表方括号内包含的项可被重复 0 次或 1 次. 也就是说, 单个分号 (`;`) 在 SysY 程序中也是一个合法的语句. 在 AST 中, 你可以使用空指针或 `Option` 来表示这种结构.

语义分析

Lv4 中, 你的编译器已经支持了一种简单的符号表: 这种符号表只支持单个作用域 (不支持作用域嵌套), 但可以检测在当前作用域内的符号重定义情况.

本节, 你只需对这个符号表稍加改动, 使其:

- **支持作用域嵌套:** 你可以把作用域的嵌套理解为, 原先只有一个符号表, 现在可以有多个, 并且它们之间存在层次关系.
- **在进入和退出代码块时更新符号表的层次结构:** 进入代码块时, 在这个结构里新建一个符号表, 这个符号表就代表当前的符号表; 退出代码块时, 删除刚刚创建的符号表, 进入代码块之前的那个符号表就代表当前的符号表.
- **只在当前作用域添加符号:** 也就是说, 只在当前层次的符号表中插入符号定义.
- **能够跨作用域查询符号定义:** 在查询符号定义时, 先在当前符号表中查询, 如果找不到就去上一层中查询. 如果在所有符号表中都没有找到这个符号的定义, 说明输入的 SysY 程序存在语义错误.

你可以选用合适的数据结构来实现这种符号表.

IR 生成

语句块和作用域只影响了语义分析, IR 生成部分无需做任何修改.

示例程序生成的 Koopa IR 为:

```
fun @main(): i32 {
%entry:
  @a_1 = alloc i32
  store 1, @a_1
  store 2, @a_1
  @a_2 = alloc i32
  store 3, @a_2
  %0 = load @a_1
  ret %0
}
```

注意:

- Koopa IR 的函数内不能定义相同的符号.
- 虽然示例程序中没有出现单个 `Exp` 表示的语句, 但在遇到这种情况时, 你必须生成 `Exp` 对应的 IR, 而不能将其跳过.

目标代码生成

由于 IR 生成部分未作修改, 目标代码生成部分也无需变更.

示例程序生成的 RISC-V 汇编为:

```
  .text
  .globl main
main:
  addi sp, sp, -16
  li t0, 1
  sw t0, 0(sp)
  li t0, 2
  sw t0, 0(sp)
  li t0, 3
  sw t0, 4(sp)
  lw t0, 0(sp)
  sw t0, 8(sp)
  lw a0, 8(sp)
  addi sp, sp, 16
  ret
```

Lv5.2. 测试

目前你的编译器已经可以处理块语句了. 同时, 你的编译器在语义分析阶段还可以处理作用域. 非常棒!

在完成本章之前, 先进行一些测试吧.

本地测试

测试 Koopa IR:

```
docker run -it --rm -v 项目目录:/root/compiler maxxing/compiler-dev \
  autotest -koopa -s lv5 /root/compiler
```

测试 RISC-V 汇编:

```
docker run -it --rm -v 项目目录:/root/compiler maxxing/compiler-dev \
  autotest -riscv -s lv5 /root/compiler
```

在线评测

?> **TODO:** 待补充.

Lv6.1. 处理 `if/else`

本节新增/变更的语法规范如下:

```
Stmt ::= ...
       | ...
       | ...
       | "if" "(" Exp ")" Stmt ["else" Stmt]
       | ...;
```

一个例子

```c
int main() {
  int a = 2;
  if (a) {
    a = a + 1;
  } else a = 0;  // 在实际写 C/C++ 程序的时候别这样, 建议 if 的分支全部带大括号
  return a;
}
```

词法/语法分析

本节新增了关键字 `if` 和 `else`, 你需要修改你的 lexer 来支持它们. 同时, 你需要针对 `if/else` 语句设计 AST, 并更新你的 parser 实现.

如果你完全按照本文档之前的内容, 例如选用了 C/C++ + Flex/Bison, 或 Rust + lalrpop 来实现你的编译器, 那么你在为你的 parser 添加 `if/else` 的语法时, 应该会遇到一些语法二义性导致的问题. 例如, Bison 会提示你发生了移进/规约冲突, lalrpop 会检测到二义性文法并拒绝生成 parser.

这个问题产生的原因是, 在 SysY (C 语言) 中, `if/else` 语句的 `else` 部分可有可无, 一旦出现了若干个 `if` 和一个 `else` 的组合, 在符合 EBNF 语法定义的前提下, 我们可以找到不止一种语法的推导 (或规约) 方法. 例如对于如下 SysY 程序:

```c
if (a) if (b) x; else y;
```

我们可以这样推导:

```
Stmt
-> "if" "(" Exp ")" Stmt
-> "if" "(" "a" ")" "if" "(" Exp ")" Stmt "else" Stmt
-> "if" "(" "a" ")" "if" "(" "b" ")" "x" ";" "else" "y" ";"
```

也可以这样:

```
Stmt
-> "if" "(" Exp ")" Stmt "else" Stmt
-> "if" "(" "a" ")" "if" "(" Exp ")" Stmt "else" "y" ";"
-> "if" "(" "a" ")" "if" "(" "b" ")" "x" ";" "else" "y" ";"
```

虽然都能抵达相同的目的地, 但我们走的路线却是不同的.

这会导致一些问题, 比如你使用 Bison 或 lalrpop 生成的 parser 会尝试根据 lexer 返回的 token 来规约得到 EBNF 中的非终结符. 你可以把规约理解成推导的逆过程, 所以对于上述 SysY 程序, parser 也能通过两种完全不同的方式进行语法的规约. 也就是说, 在这个过程中, 你可能会得到两棵完全不同的 AST——这就导致了 “二义性”, 你肯定不愿意看到这种情况的发生.

以上这个关于解析 `if/else` 的问题可以说相当之经典了, 甚至它还有一个单独的名字: 空悬 `else` 问题 (dangling else problem). 为了避免这样的问题, SysY 的语义规定了 `else` 必须和最近的 `if` 进行匹配.

但是你可能会说: 这个问题都导致 parser 没法 "正常工作" 了, 编译器根本进行不到语义分析阶段, 在语法分析阶段就直接歇菜了, 那还怎么搞嘛. 其实这个问题是可以直接在语法层面解决的, 你只需对 `if/else` 的语法略加修改 (提示: 拆分), 就可以完全规避这个问题.

语义分析

无需新增内容. 记得对 `if/else` 的各部分 (条件和各分支) 进行语义分析即可.

IR 生成

从本章开始, 你生成的程序的结构就不再是线性的了, 而是带有分支的. 在 Lv1 中我们提到过, Koopa IR 程序的结构按层次可以分为程序, 函数, 基本块和指令. 而你可以通过基本块和控制转移指令, 来在 Koopa IR 中表达分支的语义.

Koopa IR 中, 控制转移指令有两种:

1. `br 条件, 目标1, 目标2` **指令:** 进行条件分支, 其中 `条件` 为整数, 两个目标为基本块. 如果 `条件` 非 0, 则跳转到 `目标1` 基本块的开头执行, 否则跳转到 `目标2`.
2. `jump 目标` **指令:** 进行无条件跳转, 其中 `目标` 为基本块. 直接跳转到 `目标` 基本块的开头执行.

在之前的 Koopa IR 程序中, 只有一个入口基本块 `%entry`. 现在, 你可以通过划分新的基本块, 来标记控制流转移的目标.

示例程序生成的 Koopa IR 为:

```
fun @main(): i32 {
%entry:
  @a = alloc i32
  store 2, @a
  // if 的条件判断部分
  %0 = load @a
  br %0, %then, %else

// if 语句的 if 分支
%then:
  %1 = load @a
  %2 = add %1, 1
  store %2, @a
  jump %end

// if 语句的 else 分支
%else:
  store 0, @a
  jump %end

// if 语句之后的内容, if/else 分支的交汇处
%end:
  %3 = load @a
  ret %3
}
```

需要注意的是, 基本块的结尾必须是 `br`, `jump` 或 `ret` 指令其中之一 (并且, 这些指令只能出现在基本块的结尾). 也就是说, 即使两个基本块是相邻的, 例如上述程序的 `%else` 基本块和 `%end` 基本块, 如果你想表达执行完前者之后执行后者的语义, 你也必须在前者基本块的结尾添加一条目标为后者的 `jump` 指令. 这点和汇编语言中 label 的概念有所不同.

?> 上述文本形式的 Koopa IR 程序中, 四个基本块看起来都是相邻的, 但实际转换到内存形式后, 这些基本块并不存在所谓的 "相邻" 关系. 它们之间通过控制转移指令, 建立了图状的拓扑关系, 即, 这些基本块构成了一个控制流图 (control-flow graph).

编译器在大部分情况下都在处理诸如控制流图的图结构, 但一旦生成目标代码, 编译器就不得不把图结构压扁, 变成线性的结构——因为处理器从内存里加载程序到执行程序的过程中, 根本不存在 "图" 的概念. 这个压缩过程必然会损失很多信息, 这也是编译优化和体系结构之间存在的 gap.

目标代码生成

RISC-V 中也存在若干能表示分支和跳转的指令/伪指令, 你可以使用其中两条来翻译 Koopa IR 中的 `br` 和 `jump` 指令:

1. `bnez 寄存器, 目标`: 判断 `寄存器` 的值, 如果不为 0, 则跳转到目标, 否则继续执行下一条指令.
2. `j 目标`: 无条件跳转到 `目标`.

同时, 在 RISC-V 汇编中, 你可以使用 `名称:` 的形式来定义一个 label, 标记控制转移指令的目标.

示例程序生成的 RISC-V 汇编为:

```
  .text
  .globl main
main:
  addi sp, sp, -32
  li t0, 2
  sw t0, 0(sp)
  lw t0, 0(sp)
  sw t0, 4(sp)

  # if 的条件判断部分
  lw t0, 4(sp)
  bnez t0, then
  j else

  # if 语句的 if 分支
then:
  lw t0, 0(sp)
  sw t0, 8(sp)
  lw t0, 8(sp)
  li t1, 1
  add t0, t0, t1
  sw t0, 12(sp)
  lw t0, 12(sp)
  sw t0, 0(sp)
  j end

  # if 语句的 else 分支
else:
  li t0, 0
```

```
  sw t0, 0(sp)
  j end

  # if 语句之后的内容, if/else 分支的交汇处
end:
  lw t0, 0(sp)
  sw t0, 16(sp)
  lw a0, 16(sp)
  addi sp, sp, 32
  ret
```

Lv6.2. 短路求值

本节没有任何语法规范上的变化.

一个例子

```
int main() {
  int a = 0, b = 1;
  if (a || b) {
    a = a + b;
  }
  return a;
}
```

词法/语法分析

因为语法规范不变, 所以这部分没有需要改动的内容.

语义分析

同上, 暂无需要改动的内容.

IR 生成

SysY 程序中的逻辑运算符, 即 `||` 和 `&&`, 在求值时遵循短路求值的语义. 所谓短路求值, 指的是, 求值逻辑表达式时先计算表达式的左边 (left-hand side, LHS), 如果表达式左左边的结果已经可以确定整个表达式的计算结果, 就不再计算表达式的右边 (right-hand side, RHS).

比如对于一个 `||` 表达式, 如果 LHS 的值是 1, 根据或运算的性质, 无论 RHS 求出何值, 整个表达式的求值结果一定是 1, 所以此时就不再计算 RHS 了. `&&` 表达式同理.

编译器实现短路求值的思路, 其实和上述思路没什么区别. 例如, 短路求值 `lhs || rhs` 本质上做了这个操作:

```
int result = 1;
if (lhs == 0) {
  result = rhs != 0;
}
// 表达式的结果即是 result
```

你的编译器可以按照上述思路, 在生成 IR 时, 把逻辑表达式翻译成若干分支, 跳转和赋值.

当然, 目前对逻辑表达式进行短路求值和进行非短路求值是没有任何区别的, 要想体现这一区别, RHS 必须是一个带有副作用 (side effect) 的表达式. 而 SysY 中, 仅有包含函数调用的表达式才可能产生副作用, 例如调用了一个可能修改全局变量的函数, 或可能进行 I/O 操作的函数, 等等. 但为了你的编译器顺利通过之后的测试, 你必须正确实现这一功能.

短路求值逻辑表达式有什么用呢? 首先它能提出很多不必要的计算, 例如表达式的 RHS 进行了一个非常耗时的计算, 如果编程语言支持短路求值, 在求出 LHS 就能确定逻辑表达式结果的情况下, 计算机就不必劳神再把 RHS 算一遍了. 此外, 利用短路求值的性质, 你可以简化某些程序的写法, 例如在 C/C++ 中可以这么写:

```
void *ptr = ...;
if (ptr != nullptr && check(ptr)) {
  // 执行一些操作
  // ...
}
```

编译器会保证函数 `check` 被调用时, 指针 `ptr` 一定非空, 此时 `check` 函数可以放心地解引用指针而不必担心段错误.

目标代码生成

本节并未用到新的 Koopa IR 指令, 也不涉及 Koopa IR 中的新概念, 所以这部分没有需要改动的内容.

Lv6.3. 测试

你的编译器已经可以处理 `if/else` 语句了, 它能处理的程序又变得复杂了很多, 看起来也不再像个简单的计算器了, 事情变得有趣了起来!

在完成本章之前, 先进行一些测试吧.

本地测试

测试 Koopa IR:

```
docker run -it --rm -v 项目目录:/root/compiler maxxing/compiler-dev \
  autotest -koopa -s lv6 /root/compiler
```

测试 RISC-V 汇编:

```
docker run -it --rm -v 项目目录:/root/compiler maxxing/compiler-dev \
  autotest -riscv -s lv6 /root/compiler
```

在线评测

?> **TODO:** 待补充.

Lv7.1. 处理 `while`

本节新增/变更的语法规范如下:

```
Stmt ::= ...
       | ...
       | ...
       | ...
       | "while" "(" Exp ")" Stmt
       | ...;
```

一个例子

```
int main() {
  int i = 0;
  while (i < 10) i = i + 1;
  return i;
}
```

词法/语法分析

本节新增了关键字 `while`, 你需要修改你的 lexer 来支持它们. 同时, 你需要针对 `while` 语句设计 AST, 并更新你的 parser 实现.

语义分析

无需新增内容. 记得对 `while` 的各部分 (条件和循环体) 进行语义分析即可.

IR 生成

根据 `while` 的语义, 生成所需的基本块, 条件判断和分支/跳转指令即可. 相信在理解了 `if/else` 语句 IR 生成的原理之后, 这部分对你来说并不困难.

示例程序生成的 Koopa IR 为:

```
fun @main(): i32 {
%entry:
  @i = alloc i32
  store 0, @i
  jump %while_entry

%while_entry:
  %0 = load @i
  %cond = lt %0, 10
  br %cond, %while_body, %end

%while_body:
  %1 = load @i
  %2 = add %1, 1
  store %2, @i
  jump %while_entry

%end:
  %3 = load @i
  ret %3
}
```

当然, 可能还存在其他生成 `while` 的方式.

目标代码生成

本节并未用到新的 Koopa IR 指令, 也不涉及 Koopa IR 中的新概念, 所以这部分没有需要改动的内容.

Lv7.2. `break` 和 `continue`

本节新增/变更的语法规范如下:

```
Stmt ::= ...
       | ...
       | ...
       | ...
       | ...
       | "break" ";"
       | "continue" ";"
       | ...;
```

一个例子

```
int main() {
  while (1) break;
  return 0;
}
```

词法/语法分析

本节新增了关键字 `break` 和 `continue`, 你需要修改你的 lexer 来支持它们. 同时, 你需要针对这两种语句设计 AST, 并更新你的 parser 实现.

语义分析

注意 `break` 和 `continue` 只能出现在循环内. 例如, 以下的程序存在语义错误:

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

?> 其实在写编译器的时候你会发现, 在进行 IR 生成时, 你很容易判断 `break` / `continue` 是否出现在了循环内.

IR 生成

`break` 和 `continue` 本质上执行的都是跳转操作, 只不过一个会跳转到循环结尾, 一个会跳转到循环开头. 所以, 为了正确获取跳转的目标, 你的编译器在生成循环时必须记录循环开头和结尾的相关信息.

此外需要注意的是, `while` 循环是可以嵌套的, 所以, 你应该选择合适的数据结构来存储 `break` / `continue` 所需的信息.

示例程序生成的 Koopa IR **可能**为:

```
fun @main(): i32 {
%entry:
```

```
  jump %while_entry

%while_entry:
  br 1, %while_body, %end

%while_body:
  jump %end

%while_body1:
  jump %while_entry

%end:
  ret 0
}
```

!> 上面的 Koopa IR 程序是文档作者根据经验, 模仿一个编译器生成出来的 (事实上文档里所有的 Koopa IR 示例都是这么写的) (人形编译器 MaxXing 实锤), 仅代表一种可能的 IR 生成方式.

你会看到, 程序中出现了一个不可达的基本块 `%while_body1`. 这件事情在人类看来比较费解: 为什么会这样呢? ~怎么会事呢?~ 但对于编译器的 IR 生成部分而言, 这么做是最省事的. 你也许可以思考一下背后的原因.

目标代码生成

本节并未用到新的 Koopa IR 指令, 也不涉及 Koopa IR 中的新概念, 所以这部分没有需要改动的内容.

Lv7.3. 测试

到目前为止, 你的编译器已经可以处理分支和循环这两种复杂的程序结构了, 越来越像那么回事了!

在完成本章之前, 先进行一些测试吧.

本地测试

测试 Koopa IR:

```
docker run -it --rm -v 项目目录:/root/compiler maxxing/compiler-dev \
  autotest -koopa -s lv7 /root/compiler
```

测试 RISC-V 汇编:

```
docker run -it --rm -v 项目目录:/root/compiler maxxing/compiler-dev \
  autotest -riscv -s lv7 /root/compiler
```

在线评测

?> **TODO:** 待补充.

Lv8.1. 函数定义和调用

本节新增/变更的语法规范如下:

```
CompUnit    ::= [CompUnit] FuncDef;

FuncDef     ::= FuncType IDENT "(" [FuncFParams] ")" Block;
FuncType    ::= "void" | "int";
FuncFParams ::= FuncFParam {"," FuncFParam};
FuncFParam  ::= BType IDENT;

UnaryExp    ::= ...
              | IDENT "(" [FuncRParams] ")"
              | ...;
FuncRParams ::= Exp {"," Exp};
```

一个例子

```
int half(int x) {
  return x / 2;
}

void f() {}

int main() {
  f();
  return half(10);
}
```

词法/语法分析

本节新增了关键字 `void`, 你需要修改你的 lexer 来支持它们. 同时, 你需要针对发生变化的语法规则, 例如 `CompUnit`, `FuncDef` 等, 设计新的 AST, 并更新你的 parser 实现.

语义分析

本节 `CompUnit` 的定义发生了变化, 其允许多个不同的函数同时存在于全局范围内. 为了避免函数之间的重名, 你可以把全局范围内所有的函数 (包括之后章节中会出现的全局变量) 都放在同一个作用域内, 即全局作用域. 全局作用域应该位于所有局部作用域的外层, 你可以修改你的符号表使其支持这一特性.

IR 生成

在 Koopa IR 中, 你可以像定义 `@main` 函数一样, 定义其他的函数. 如果函数有参数, 可以直接在函数名之后的括号内写明参数名称和类型. 使用上, 函数的形式参数变量和函数内的其他变量并无区别.

Koopa IR 中, 使用 `call 函数名(参数, ...)` 指令可以完成一次函数调用. `call` 指令是否具备返回值, 以及具备什么类型的返回值, 取决于指令所调用的函数的具体类型. 如果 `call` 指令不具备返回值, 则不能写为 `%v = call ...` 的形式.

对于返回值类型为 `void` 的函数, 定义函数时省略类型标注, 函数内使用 `ret` 指令时不需要附带返回值. 即, 函数需要返回时, 直接写 `ret` 即可.

示例程序生成的 Koopa IR 为:

```
fun @half(@x: i32): i32 {
%entry:
  %x = alloc i32
  store @x, %x
```

```
  %0 = load %x
  %1 = div %0, 2
  ret %1
}

fun @f() {
%entry:
  ret
}

fun @main(): i32 {
%entry:
  call @f()
  %0 = call @half(10)
  ret %0
}
```

!> 本章引入了全局符号定义. 在 Koopa IR 中, 全局的符号不能和其他全局符号同名, 局部的符号 (位于函数内部的符号) 不能和其他全局符号以及局部符号同名. 上述规则对具名符号和临时符号都适用.

你可能会注意到, 在 `@half` 函数中, 我们为参数 `@x` 又单独分配了一块名为 `%x` 的内存空间. 直接使用 `@x` 不行吗? 比如像这样:

```
fun @half(@x: i32): i32 {
%entry:
  %0 = div @x, 2
  ret %0
}
```

可以的, 完全正确! 但此处采取了看起来更繁琐的做法, 是为了方便目标代码生成部分的处理. 否则, 目标代码生成必须做复杂处理, 或者直接生成出错误的代码.

目标代码生成

目标代码生成部分无非涉及以下几个问题, 即, 在 RISC-V 汇编中:

- 如何定义函数?
- 如何调用函数?
- 如何传递/接收函数参数?
- 如何传递/接收返回值, 以及从函数中返回?

这些问题的基础, 都和 RISC-V 的调用约定, 以及栈帧的构造 (这其实也是调用约定的一部分) 相关. 建议你先回顾一下 Lv4.2 中的相关内容.

函数的调用和返回

RISC-V 中, `call` 和 `ret` 伪指令可以实现函数的调用和返回——确切的说, 其中的 `jalr` 指令实现了函数调用和返回中的关键操作: 跳转和保存返回地址. 当然, 从这两条指令的含义中我们得知:

1. 在汇编层面, "函数调用和返回" 并不包括参数和返回值的传递.
2. 函数的返回地址保存在寄存器 `ra` 中.

第二点告诉我们, 一旦一个函数中还会调用其他函数, 这个函数 (或另外的函数) 就必须保存/恢复自己的 `ra` 寄存器. 比如本节示例中的 `main` 函数调用了 `half` 函数和 `f` 函数, `main` 必须保存自己的 `ra`. 否则在调用其他函数时, `call` 指令会修改 `ra` 的值, 而 `main` 在执行 `ret` 时, `ra` 的值就不再是进入 `main` 时的值了, `main` 就无法返回到正确的位置, 只能在自己的函数体里无限循环.

把函数之间的调用关系想象成一个图 (即调用图, call graph), 那么一个永远不会调用其他函数的函数就位于图中的叶子结点, 我们把这种函数称为叶子函数 (leaf function). 与之相对的, 还有非叶子函数.

在 RISC-V 中, 非叶子函数通常需要在 prologue 中将自己的 `ra` 寄存器保存到栈帧中. 在 epilogue 中, 非叶子函数需要先从栈帧中恢复 `ra` 寄存器, 之后才能执行 `ret` 指令. 叶子函数可以不必进行上述操作——当然进行了也不会出问题, 只是会做一些无用功导致性能变差.

传递/接收参数

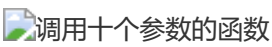
RISC-V 有 8 个寄存器: `a0`-`a7`, 它们专门用来在函数调用时传递函数的非浮点参数, 前提是函数参数可以被塞进这些寄存器 (在本课程中不会出现塞不进去的情况, 指所有函数参数数据的位宽均小于等于 32 位). 函数的前 8 个参数必须按照从前到后的顺序依次放入 `a0` 到 `a7` 寄存器.

那你可能会问:

> "如果函数参数超过 8 个怎么办?"

注意: 你在课程中实现的编译器必须考虑函数参数超过 8 个的情况, 同时我们提供的测试用例中也会涉及针对该情况的测试.

对于这种情况, RISC-V 的调用约定规定, 超出部分的参数必须放在 "调用者" 函数 (caller, 也就是调用超 8 个参数函数的那个函数) 的栈帧中. 具体来说, 第 9 个参数放在 `sp + 0` 的位置, 后续参数按照从前到后的顺序依次放在 `sp + 4` 及之后——当然, 这是建立在你传的参数的长度都是 32 位的基础上来讨论的. 具体如图所示:


调用十个参数的函数

所以, 函数在建立自己栈帧的时候, 必须给函数内所有超 8 个参数的函数调用留出足够大的空间, 来传递函数参数.

> "为什么这么麻烦啊?"

没办法, 写编译器就是一个入乡随俗的过程. 编译器必须得依着目标指令系统的性子, 生成格式正确的指令, 同时按照约定的规矩处理栈帧啊, 传参啊, 这啊那啊的一大堆破事. 所以, 下次编译器给你写的代码报错的时候, 记得好好哄哄编译器, 人家也不容易. 报警告的时候也得哄, 因为警告和错误一样重要!

> "写编译器的程序员也不容易啊, 那你能好好哄哄我吗?"

——哄……哄也不是不可以啦……

写文档的助教也不容易, 你能哄……哎, 人呢? 怎么跑了?

关于寄存器

目前对于大部分的 Koopa IR 变量, 包括 `alloc` 分配的内存, 以及其他指令的返回值, 你的编译器会把它们全部保存在栈帧里, 而不是寄存器上. 即便用到寄存器, 也只是诸如 `t0`-`t6` 的临时寄存器. 你有没有想过这些寄存器为什么叫 "临时寄存器"? 临时寄存器的含义又是什么?

RISC-V 的 ABI 中, 32 个整数寄存器的名称和含义如下表所示:

寄存器	ABI 名称	描述	保存者
`x0`	`zero`	恒为 0	N/A
`x1`	`ra`	返回地址	调用者
`x2`	`sp`	栈指针	被调用者
`x3`	`gp`	全局指针	N/A
`x4`	`tp`	线程指针	N/A
`x5`	`t0`	临时/备用链接寄存器	调用者
`x6-7`	`t1-2`	临时寄存器	调用者
`x8`	`s0 / fp`	保存寄存器/帧指针	被调用者
`x9`	`s1`	保存寄存器	被调用者
`x10-11`	`a0-1`	函数参数/返回值	调用者
`x12-17`	`a2-7`	函数参数	调用者
`x18-27`	`s2-11`	保存寄存器	被调用者
`x28-31`	`t3-6`	临时寄存器	调用者

ABI 中明确了 32 个整数寄存器的作用. 这些寄存器中, `x0` 硬件上无法被修改, `x2`-`x4` 软件上一旦被修改程序就会出问题, 剩下所有的寄存器可以用来随便存数据. 其中, `t0` 到 `t6` 这七个寄存器为临时寄存器, 这些寄存器是由 “调用者” 保存的, 即 “caller-saved”, 所以它们很适合用来存储临时数据.

在 ABI (或者说调用约定) 中, 保存寄存器的操作可以由两方来进行: 一方是负责调用函数的那个函数, 另一方是被调用的那个函数. 前者负责保存的寄存器叫 “调用者保存的寄存器” (caller-saved registers), 后者负责的叫 “被调用者保存的寄存器” (callee-saved register).

你可能听得一头雾水, 我换种说法: “调用者保存” 的寄存器, 在 “被调用” 的函数中可以随便乱写. 因为 ABI 规定这些寄存器已经被 “调用这个函数的函数” 保存在它自己的栈帧里了, “被调用” 的函数不需要操心怎么保存他们. 反之, 如果一个函数需要使用某些 “被调用者保存” 的寄存器, 在这个函数进入的时候 (即 prologue 中), 这个函数就必须把它要使用的这些寄存器保存一遍. 而对于这个函数的调用者, 在它看来, 调用函数前后, 所有 “被调用者保存” 的寄存器的内容都是不会发生变化的.

比如, 对于如下这个程序:

```
li t0, 42
call func
addi t0, t0, 1
```

我们不能保证, `addi` 指令执行完成后, `t0` 一定是 43. 因为根据 ABI, `func` 不负责保存 `t0` 寄存器的值, 它可以往里随便写数据. 于是, `call` 指令执行完成后, `t0` 的值就未必是 42 了.

以上这个故事告诉我们, 你在实现编译器时, 必须按照 ABI 的要求使用 RISC-V 中的寄存器. 比如, 如果你的编译器会在生成某个函数的时候, 生成用到 `s0` 的指令, 编译器就必须同时在生成函数的 prologue/epilogue 时, 生成保存/恢复这个寄存器的指令. `t0`, `a0` 之类的寄存器可以随便用, 但你的编译器必须保证, 在需要读寄存器前不能进行函数调用, 否则寄存器的值有一定概率被破坏.

生成代码

由于本节引入了函数的定义和调用, 在生成代码时, 你需要考虑一些 ABI 相关的内容. 具体步骤如下:

1. 扫描 Koopa IR 程序中的每一个函数, 对所有的函数, 都像生成 `main` 一样生成代码.
2. 生成 prologue 之前, 扫描函数内的每条指令, 统计需要为局部变量分配的栈空间 S, 需要为 `ra` 分配的栈空间 R, 以及需要为传参预留的栈空间 A, 上述三个量的单位均为字节.
   - R 的计算方法: 如果函数中出现了 `call`, 则为 4, 否则为 0.
   - A 的计算方法: 设函数内有 n 条 `call` 指令, $call_i$ 用到的参数个数为 len_i, 则 $A = \max { \max_{i=0}^{n-1} len_i - 8, 0 } \times 4$
3. 计算 $S + R + A$, 向上取整到 16, 得到 $S^\prime$.
4. 生成 prologue: 首先, 根据 $S^\prime$ 生成 `addi` 指令来更新 `sp`. 然后, 如果 R 不为 0, 在 $sp + S^\prime - 4$ 对应的栈内存中保存 `ra` 寄存器的值.
5. 生成函数体中的指令.
6. 如果遇到 Koopa IR 中的 `call` 指令, 你需要先将其所有参数变量的值读出, 存放到参数寄存器或栈帧中, 然后再生成 RISC-V 的 `call`.
7. 生成 epilogue: 如必要, 从栈帧中恢复 `ra` 寄存器. 然后, 复原 `sp` 寄存器的值. 最后生成 `ret`.

示例程序生成的 RISC-V 汇编为:

```
  .text
  .globl half
half:
  addi sp, sp, -16
  sw a0, 0(sp)
  lw t0, 0(sp)
  sw t0, 4(sp)
  lw t0, 4(sp)
  li t1, 2
  div t0, t0, t1
  sw t0, 8(sp)
  lw a0, 8(sp)
  addi sp, sp, 16
  ret

  .text
  .globl f
f:
  ret

  .text
  .globl main
main:
  addi sp, sp, -16
  sw ra, 12(sp)
  call f
  li a0, 10
  call half
  sw a0, 0(sp)
  lw a0, 0(sp)
  lw ra, 12(sp)
  addi sp, sp, 16
  ret
```

Lv8.2. SysY 库函数

本节没有任何语法规范上的变化.

一个例子

```
int main() {
  return getint();
}
```

词法/语法分析

因为语法规范不变, 所以这部分没有需要改动的内容.

语义分析

根据 SysY 的规定, SysY 库函数可以不加声明就直接使用, 所以你可能需要预先在全局符号表中添加和库函数相关的符号定义, 以防无法正确处理相关内容.

参考 SysY 运行时库.

IR 生成

虽然 SysY 中可以不加声明就使用所有库函数, 但在 Koopa IR 中, 所有被 `call` 指令引用的函数必须提前声明, 否则会出现错误. 你可以使用 `decl` 语句来预先声明所有的库函数.

示例程序生成的 Koopa IR 为:

```
decl @getint(): i32
decl @getch(): i32
decl @getarray(*i32): i32
decl @putint(i32)
decl @putch(i32)
decl @putarray(i32, *i32)
decl @starttime()
decl @stoptime()

fun @main(): i32 {
%entry:
  %0 = call @getint()
  ret %0
}
```

?> 注: `decl` 要求在括号内写明参数的类型, 某些库函数会接收数组参数, 你可以认为这种参数的类型是 `*i32`, 即 `i32` 的指针. Lv9 将讲解其中的具体原因.

目标代码生成

RISC-V 汇编中, 函数符号无需声明即可直接使用. 关于到底去哪里找这些外部符号, 这件事情由链接器负责. 除此之外, 调用库函数和调用 SysY 内定义的函数并无区别.

Koopa IR 中, 函数声明是一种特殊的函数, 它们和函数定义是放在一起的. 也就是说, 在上一节的基础上, 你需要在扫描函数时跳过 Koopa IR 中的所有函数声明.

Koopa IR 的函数声明和普通函数的区别是: 函数声明的基本块列表是空的. 在 C/C++ 中, 你可以通过判断 `koopa_raw_function_t` 中 `bbs` 字段对应的 slice 的长度, 来判断函数的基本块列表是否为空. 在 Rust 中, `FunctionData` 提供的 `layout()` 方法会返回函数内基本块/指令的布局, 返回类型为 `&Layout`. 而 `Layout` 中的 `entry_bb()` 方法可以返回函数入口基本块的 ID, 如果函数为声明, 这个方法会返回 `None`.

示例程序生成的 RISC-V 汇编为:

```
  .text
  .globl main
main:
  addi sp, sp, -16
  sw ra, 12(sp)
  call getint
  sw a0, 0(sp)
  lw a0, 0(sp)
  lw ra, 12(sp)
  addi sp, sp, 16
  ret
```

Lv8.3. 全局变量和常量

本节新增/变更的语法规范如下:

```
CompUnit ::= [CompUnit] (Decl | FuncDef);
```

一个例子

```
int var;

const int one = 1;

int main() {
  return var + one;
}
```

词法/语法分析

本节中, 语法规则 `CompUnit` 发生了变化, 你可能需要为其设计新的 AST, 并更新你的 parser 实现.

语义分析

本节 `CompUnit` 的定义较上一节又发生了变化, 不仅允许多个函数存在于全局范围内, 还允许变量/常量声明存在于全局范围内. 你需要把所有全局范围内的声明, 都放在全局作用域中.

此外, 全局常量和局部常量一样, 都需要在编译期求值. 你的编译器在处理全局常量时, 需要扫描它的初始值, 并直接算出结果, 存入符号表.

IR 生成

对于所有全局变量, 你的编译器应该生成全局内存分配指令 (`global alloc`). 这种指令的用法和 `alloc` 类似, 区别是全局分配必须带初始值.

示例程序生成的 Koopa IR 为:

```
global @var = alloc i32, zeroinit

fun @main(): i32 {
%entry:
  %0 = load @var
  %1 = add %0, 1
  ret %1
}
```

未初始化的全局变量的值为 0, 所以我们使用 `zeroinit` 作为初始值, 初始化了全局内存分配 `@var`.

此处的 `zeroinit` 代表零初始化器 (zero initializer). `zeroinit` 是一个通用的 0 值, 它可以是多种类型的. 不管是向 `i32` 类型的 `alloc` 中写入 `zeroinit`, 还是向你将在 Lv9 中遇到的数组类型的 `alloc` 中写入 `zeroinit`, 这些 `alloc` 分配的内存都会被填充 0.

当然, 对于这个示例, 你写 `global @var = alloc i32, 0` 也完全没问题.

目标代码生成

在操作系统层面, 局部变量和全局变量的内存分配是不同的. 前者的内存空间在程序运行时, 由函数在栈上动态开辟出来, 或者直接放在寄存器里. 后者在程序被操作系统加载时, 由操作系统根据可执行文件 (比如 PE/COFF, ELF 或 Mach-O) 中定义的 layout, 静态映射到虚拟地址空间, 进而映射到物理页上.

体现在 RISC-V 汇编中, 局部变量和全局变量的描述方式也有所区别. 前者的描述方式你已经见识过了, 基本属于润物细无声级别的: 你需要在函数中操作一下 `sp`, 然后搞几个偏移量, 再 `lw`/`sw`, 总体来说比较抽象. 后者就很直接了, 我们先看示例程序生成的 RISC-V 汇编:

```
  .data
  .globl var
var:
  .zero 4

  .text
  .globl main
main:
  addi sp, sp, -16
  la t0, var
  lw t0, 0(t0)
  sw t0, 0(sp)
  lw t0, 0(sp)
  li t1, 1
  add t0, t0, t1
  sw t0, 4(sp)
  lw a0, 4(sp)
  addi sp, sp, 16
  ret
```

这段汇编代码中, `.data` 是汇编器定义的一个 “directive”——你可以理解成一种特殊的语句. `.data` 指定汇编器把之后的所有内容都放到数据段 (data segment). 对操作系统来说, 数据段对应的内存里放着的所有东西都会被视作数据, 程序一般会把全局变量之类的内容都塞进数据段.

`.data` 之后的 `.zero` 也同样是一个 directive——事实上, 汇编程序中所有 `.` 开头的语句基本都是 directive. `.zero 4` 代表往当前地址处 (这里代表 `var` 对应的地址) 填充 4 字节的 0, 这对应了 Koopa IR 中的 `zeroinit`. 如果 Koopa IR 是以下这种写法:

```
global @var = alloc i32, 233
```

那我们就应该使用 `.word 233` 这个 directive, 这代表往当前地址处填充机器字长宽度 (4 字节) 的整数 `233`.

于是现在, 你可能已经明白 `main` 之前的 `.text` 代表什么含义了: 它表示之后的内容应该被汇编器放到代码段 (code segment). 代码段和数据段的区别是, 代码段里的 "数据" 是只读且可执行的, 数据段里的数据可读可写但不可执行. 顺便一提, 这种 "可执行/不可执行" 的特性, 是操作系统加载可执行文件到虚拟地址空间时, 通过设置页表中虚拟页的权限位来实现的, 处理器将负责保证权限的正确性.

除了代码段和数据段, 可执行文件里通常还会有很多其他的段: 比如 `bss` 段存放需要零初始化的数据, 操作系统加载 `bss` 时会自动将其清零, 所以可执行文件中只保存 `bss` 的长度而不保存数据, 可以节省一些体积; `rodata` 段存放只读的数据 (**r**ead-**o**nly **data**). 其实这么看, 示例程序里的 `var` 放在 `bss` 段是最合适的, 但为了简化编译器的实现, 你可以把它放在 `data` 段.

全局变量的内存分配完毕之后, 我们要怎么访问到这块内存呢? 你可以注意到在 `main` 里出现了一条 `la t0, var` 指令 (其实是伪指令), 这条指令会把符号 `var` 对应的地址加载到 `t0` 寄存器中. 之后的 `lw` 指令以 `t0` 为地址, 读取了 4 字节数据到 `t0`. 这两条指令共同完成了加载全局变量的操作.

?> `la` 伪指令之后的符号并不只局限在数据段, 其他段中符号的地址也是可以被加载的. 你可以尝试使用 RISC-V 汇编实现一个简单的程序, 读取 `main` 函数的第一条指令的值并输出, 然后对照 RISC-V 规范, 查看这条指令是否对应了你在汇编中所写的那条指令.

Lv8.4. 测试

到目前为止, 你的编译器已经可以处理包括函数定义和调用, SysY 库函数和全局变量的程序了, 这又是一次巨大的飞跃!

有了 SysY 库函数, 你的编译器生成的程序就能进行输入/输出字符之类的 I/O 操作了. 测试程序会通过指定标准输入以及检查标准输出的方式, 来进一步确认你的程序是否执行正确.

已经可以看到胜利的曙光了, 加油!

在完成本章之前, 先进行一些测试吧.

本地测试

测试 Koopa IR:

```
docker run -it --rm -v 项目目录:/root/compiler maxxing/compiler-dev \
  autotest -koopa -s lv8 /root/compiler
```

测试 RISC-V 汇编:

```
docker run -it --rm -v 项目目录:/root/compiler maxxing/compiler-dev \
  autotest -riscv -s lv8 /root/compiler
```

在线评测

?> **TODO:** 待补充.

Lv9.1. 一维数组

本节新增/变更的语法规范如下:

```
ConstDef      ::= IDENT ["[" ConstExp "]"] "=" ConstInitVal;
ConstInitVal  ::= ConstExp | "{" [ConstExp {"," ConstExp}] "}";
VarDef        ::= IDENT ["[" ConstExp "]"]
                | IDENT ["[" ConstExp "]"] "=" InitVal;
InitVal       ::= Exp | "{" [Exp {"," Exp}] "}";

LVal          ::= IDENT ["[" Exp "]"];
```

一个例子

```
int x[2] = {10, 20};

int main() {
  int arr[5] = {1, 2, 3};
  return arr[2];
}
```

词法/语法分析

针对本节发生变化的语法规则, 设计新的 AST, 并更新你的 parser 实现即可.

语义分析

数组定义时, 数组长度使用 `ConstExp` 表示, 所以你在编译时必须求出代表数组长度的常量表达式. 同时, 对于全局数组变量, 语义规定它的初始值也必须是常量表达式, 你也需要对其求值.

需要注意的是, 常量求值时, 你应该只考虑整数类型的常量定义, **而不要求值常量数组**. 例如:

```
const int a = 10;
const int arr[2] = {1, 2};
```

对于常量 `a`, 你的编译器应该在语义分析阶段把它求出来, 存入符号表. 对于常量 `arr`, 你的编译器只需要将其初始化表达式中的常量算出来, 但不需要再设计一种代表数组常量的数据结构并将其存入符号表. 在生成的 IR 中, `a` 是不存在的 (全部被替换成了常量); 而 `arr` 是存在的, 体现在 IR 中就是一个数组.

综上所述, 如果你在常量表达式中扫描到了对常量数组的解引用, 你可以将其视为发生了语义错误. 例如如下 SysY 程序是存在语义错误的:

```
// 允许定义常量数组, 此时数组 arr 不能被修改
const int arr[2] = {1, 2};
// arr 本身不会参与编译期求值, 所以编译器无法在编译时算出 arr[1] 的值
// 所以此处出现了语义错误
const int a = 1 + arr[1];
```

此外, 由于引入了数组定义, 变量的类型将不再是单一的整数. 所以你需要考虑在语义分析阶段加入类型检查机制, 来避免某些语义错误的情况发生, 例如:

```
int arr[10];
// 类型不匹配
int a = arr;
```

当然, 再次重申, 为了降低难度, 测试/评测时我们提供的输入都是合法的 SysY 程序, 并不会出现语义错误. 你的编译器不进行类型检查也是可以的.

IR 生成

数组的定义和变量的定义类似, 同样使用 `alloc` 指令完成, 不过之后的类型需要换成数组类型. Koopa IR 中, `[T, len]` 可以表示一个元素类型为 `T`, 长度为 `len` 的数组类型. 例如 `[i32, 3]` 对应了 SysY 中的 `int[3]`, `[[i32, 3], 2]` 对应了 SysY 中的 `int[2][3]` (注意这里的 2 和 3 是反着的).

SysY 中数组相关的操作通常包含两步: 通过 `[i]` 定位数组的元素, 然后读写这个元素. 根据写 C/C++ 时积累的经验, 你不难发现, 这种操作本质上其实是指针计算. 在 Koopa IR 中, 针对数组的指针计算可以使用 `getelemptr` 指令完成.

`getelemptr` 指令的用法类似于 `getelemptr 指针, 偏移量`. 其中, `指针` 必须是一个数组类型的指针, 比如 `*[i32, 3]`, 或者 `*[[i32, 3], 2]`. 我们在 Lv4 中提到, `alloc T` 指令返回的类型是 `T` 的指针, 所以 `alloc` 数组时, 你刚好就可以得到一个数组的指针.

`getelemptr ptr, index` 指令执行了如下操作: 假设指针 `ptr` 的类型是 `*[T, N]`, 指令会算出一个新的指针, 这个指针的值是 `ptr + index * sizeof(T)`, 类型是 `*T`. 在逻辑上, 这种操作和 C 语言中的数组访问操作是完全一致的. 比如:

```
int arr[2];
arr[1];
```

翻译到 Koopa IR 就是:

```
@arr = alloc [i32, 2]        // @arr 的类型是 *[i32, 2]
%ptr = getelemptr @arr, 1    // %ptr 的类型是 *i32
%value = load %ptr           // %value 的类型是 i32
// 这是一段类型和功能都正确的 Koopa IR 代码
```

本质上相当于:

```
int arr[2];
// 在 C 语言的指针运算中, int 指针加 1
// 就相当于对指针指向的地址的数值加了 1 * sizeof(int)
int *ptr = arr + 1;
*ptr;
```

对于多维数组也是一样, 虽然本节你暂时不需要实现多维数组:

```
int arr[2][3];
arr[1][2];
```

翻译到 Koopa IR 就是:

```
@arr = alloc [[i32, 3], 2]    // @arr 的类型是 *[[i32, 3], 2]
%ptr1 = getelemptr @arr, 1    // %ptr1 的类型是 *[i32, 3]
%ptr2 = getelemptr %ptr1, 2   // %ptr2 的类型是 *i32
%value = load %ptr2           // %value 的类型是 i32
// 这是一段类型和功能都正确的 Koopa IR 代码
```

`getelemptr` 得到的是一个指针, 指针既可以被 `load`, 也可以被 `store`. 所以, 你的编译器可以通过生成 `getelemptr` 和 `store` 的方法, 来处理 SysY 中写入数组元素的操作. 以此类推, 对于局部数组变量的初始化列表, 你也可以把它编译成若干指针计算和 `store` 的形式. 但是, 你可能会问: 全局数组变量的初始化要怎么处理呢?

确实, Koopa IR 的全局作用域内是不能出现 `store` 指令的. 因为对应到汇编层面, 并不存在一段可以在程序被操作系统加载的时候执行的代码, 并让它帮你执行一系列的 `store` 操作. 操作系统加载程序的时候, 会把可执行文件中各段 (segment) 对应的数据, 逐步复制到内存中. 所以, 如果我们能把数组初始化列表里的元素表示成数据的形式, 我们就可以实现全局数组初始化.

这在 SysY 中是完全可行的, 因为语义规定, 全局数组变量的初始化列表中只能出现常量表达式, 所以你的编译器一定能在编译时把初始化列表里的每个元素都确定下来. 既然你已经预先知道了所有的元素, 你的编译器就可以把它们写死成数据, 然后在生成汇编的时候用 `.word` 之类的 directive 告诉汇编器, 把这些数据塞进数据段.

Koopa IR 中, 你可以使用 "aggregate" 常量来表示一个常量的数组初始化列表, 比如:

```
// 这是个全局数组
int arr[3] = {1, 2, 3};
```

翻译到 Koopa IR 就是:

```
global @arr = alloc [i32, 3], {1, 2, 3}
```

Aggregate 中出现的元素必须为彼此之间类型相同的常量, 比如整数, `zeroinit`, 或者另一个 aggregate, 所以多维数组也可以用这种方式初始化. 此外, aggregate 中不能省略任何元素, 对于如下 SysY 程序:

```
// 这是个全局数组
int arr[3] = {5};
```

你的编译器必须将其翻译为:

```
global @arr = alloc [i32, 3], {5, 0, 0}
```

之前提到, Koopa IR 是强类型 IR, 且能够自动推导部分类型. 假设 aggregate 中各元素的类型为 `T`, 且总共有 `len` 个元素, 那么 aggregate 本身的类型就会被推导为 `[T, len]`. 所以:

```
// 如下 Koopa IR 指令不合法的原因是, alloc 的类型和初始化类型不符
// 右边的 aggregate 的类型为 [i32, 1]
global @arr = alloc [i32, 3], {5}
```

最后, 以防你忘了, 多提一句: `zeroinit` 也是可以用来零初始化数组的:

```
// 相当于 SysY 中的全局数组 int zeroed_array[2048];
global @zeroed_array = alloc [i32, 2048], zeroinit
```

最后的最后, 你可能会问: 既然我能在全局数组初始化的时候使用 aggregate/`zeroinit`, 那我能不能在局部变量初始化的时候也这么用呢? 答案是可以:

```
@arr = alloc [i32, 5]
store {1, 2, 3, 0, 0}, @arr
```

但这么搞的话, 你的编译器需要在目标代码生成的时候进行一些额外的处理.

综上所述, 示例程序生成的 Koopa IR 为:

```
global @x = alloc [i32, 2], {10, 20}
```

```
fun @main(): i32 {
%entry:
  @arr = alloc [i32, 5]
  // arr 的初始化列表, 别忘了补 0 这个事
  %0 = getelemptr @arr, 0
  store 1, %0
  %1 = getelemptr @arr, 1
  store 2, %1
  %2 = getelemptr @arr, 2
  store 3, %2
  %3 = getelemptr @arr, 3
  store 0, %3
  %4 = getelemptr @arr, 4
  store 0, %4

  %5 = getelemptr @arr, 2
  %6 = load %5
  ret %6
}
```

目标代码生成

本节生成的 Koopa IR 中出现了若干新概念, 下面我们来逐一过一下.

计算类型大小

要想进行内存分配, 你的编译器必须先算出应分配的内存的大小. `alloc` 指令中出现了不同于 `i32` 的数组类型, 以及 `getelemptr` 的返回值是个指针, 它们的大小自然也是需要计算出来的. 这其中, 指针的大小在 RV32I 上就是 4 字节, 毕竟 RV32I 是 32 位的指令系统. 而数组类型的大小, 我应该不用多提了, 大家不难通过几次乘法算出这个数值.

C/C++ 中, Koopa IR 类型在内存中表示为 `koopa_raw_type_t`, 你可以 DFS 遍历其内容, 并按照我们提到的规则计算得到类型的大小. Rust 中, `Type` 提供了 `size()` 方法, 这个方法会用 DFS 遍历的方式帮你求出类型的大小.

但需要注意的是, 考虑到 Koopa IR 的泛用性 (比如搞不好你哪天心血来潮想给 Koopa IR 写个 x86-64 后端), 默认情况下, Rust 的 `koopa` crate 中的 `Type::size()` 会按照当前平台的指针大小来计算指针类型的大小. 因为目前大家用的基本都是 64 位平台, 所以在遇到指针类型时, `size()` 会返回 8. 为了适配 riscv32 的指针宽度, 你需要在进行代码生成前 (比如在 `main` 里), 调用 `Type::set_ptr_size(4)`, 来设置指针类型的大小为 4 字节.

处理 `getelemptr`

前文已经描述过 `getelemptr` 的含义了, 无非是做了一次乘法和加法. 所以, 对于如下 Koopa IR 程序:

```
@arr = alloc [i32, 2]
%ptr = getelemptr @arr, 1
```

假设 `@arr` 位于 `sp + 4`, 则对应的 RISC-V 汇编可以是:

```
  # 计算 @arr 的地址
  addi t0, sp, 4
  # 计算 getelemptr 的偏移量
  li t1, 1
  li t2, 4
  mul t1, t1, t2
  # 计算 getelemptr 的结果
  add t0, t0, t1
  # 保存结果到栈帧
  # ...
```

注意:

1. 检查 `addi` 中立即数的范围.
2. 上述 RISC-V 汇编并非是最优的. 对于偏移量是 2 的整数次幂的情况, 你可以用移位指令来替换乘法指令.
3. 对于全局数组变量的指针运算, 代码生成方式和上述类似, 只不过你需要用 `la` 加载全局变量的地址.

处理全局初始化

之前章节解释过如何在全局生成 `zeroinit` 和整数常量: 对于前者, 生成一个 `.zero sizeof(T)`, 其中 `sizeof(T)` 代表 `zeroinit` 的类型的大小. 对于后者, 生成一个 `.word 整数`.

Aggregate 的生成方式就是把上述内容组合起来, 比如:

```
global @arr = alloc [i32, 3], {1, 2, 3}
```

生成的 RISC-V 汇编为:

```
  .data
  .globl arr
arr:
  .word 1
  .word 2
  .word 3
```

生成代码

在理解上述概念之后, 你不难自行完成目标代码的生成, 故此处不再赘述.

Lv9.2. 多维数组

本节新增/变更的语法规范如下:

```
ConstDef      ::= IDENT {"[" ConstExp "]"} "=" ConstInitVal;
ConstInitVal  ::= ConstExp | "{" [ConstInitVal {"," ConstInitVal}] "}";
VarDef        ::= IDENT {"[" ConstExp "]"}
                | IDENT {"[" ConstExp "]"} "=" InitVal;
InitVal       ::= Exp | "{" [InitVal {"," InitVal}] "}";

LVal          ::= IDENT {"[" Exp "]"};
```

一个例子

```
int main() {
  int arr[2][3] = {1, 2};
  return arr[0][2];
}
```

词法/语法分析

针对本节发生变化的语法规则, 设计新的 AST, 并更新你的 parser 实现即可.

语义分析

上一节的相关注意事项可推广至这一节, 唯一需要注意的是, **多维数组的初始化列表会更复杂.** 比如示例程序中, `int[2][3]` 的数组使用了一个 `{1, 2}` 形式的初始化列表. 我们可以把它写得完整一些:

```
int arr[2][3] = {{1, 2, 0}, {0, 0, 0}};
```

看起来似乎很好理解: 把这个 2 乘 3 的数组展平, 前两个元素已经给出了, 后续元素填充 0 即可, 对吧? 然而, 实际情况比你想象的还要复杂, 比如以下这个初始化列表:

```
int arr[2][3][4] = {1, 2, 3, 4, {5}, {6}, {7, 8}};
```

这个就不太好理解了. 如果你遇到这种不好理解的例子, 可以利用我们之前提到的网站——Compiler Explorer, 直接查看 C 语言代码对应的汇编. 比如以上这个例子, 如果我们把 `arr` 视作一个全局数组, 那么对应的汇编为:

```
arr:
  .word   1
  .word   2
  .word   3
  .word   4
  .word   5
  .word   0
  .word   0
  .word   0
  .word   6
  .word   0
  .word   0
  .word   0
  .word   7
  .word   8
  .word   0
  .word   0
  .zero   16
  .zero   16
```

之前我们已经介绍过 `.word` 和 `.zero` 的含义, 你不难把上面的汇编代码复原回初始化列表:

```
int arr[2][3][4] = {
  {{1, 2, 3, 4}, {5, 0, 0, 0}, {6, 0, 0, 0}},
  {{7, 8, 0, 0}, {0, 0, 0, 0}, {0, 0, 0, 0}}
};
```

所以我们建议, 在处理 SysY 的初始化列表前, 你应该先把初始化列表转换为已经填好 0 的形式, 这样处理难度会大幅度下降. 因为用户输入的程序里未必一定会写出一个合法的初始化列表, 比如:

```
int arr[2][2][2] = {{}, 1, {}};
```

所以这一步应该被放在语义分析阶段, 以便你在转换的同时报告语义错误. 但考虑到由于测试输入一定是合法的, 有的同学不会做语义分析, 所以这一步放到 IR 生成阶段也是可以的.

读到此处, 我觉得你最关心的问题应该是: 到底应该如何理解 SysY 的初始化列表, 然后把它转换成一个填好 0 的形式呢? 其实, 处理 SysY (或 C 语言) 中的初始化列表时, 可以遵循这几个原则:

1. 记录待处理的 n 维数组各维度的总长 $len_1, len_2, \cdots, len_n$. 比如 `int[2][3][4]` 各维度的长度分别为 2, 3 和 4.
2. 依次处理初始化列表内的元素, 元素的形式无非就两种可能: 整数, 或者另一个初始化列表.
3. 遇到整数时, 从当前待处理的维度中的最后一维 (第 n 维) 开始填充数据.
4. 遇到初始化列表时:
    - 当前已经填充完毕的元素的个数必须是 len_n 的整数倍, 否则这个初始化列表没有对齐数组维度的边界, 你可以认为这种情况属于语义错误.
    - 检查当前对齐到了哪一个边界, 然后将当前初始化列表视作这个边界所对应的最长维度的数组的初始化列表, 并递归处理. 比如:
        - 对于 `int[2][3][4]` 和初始化列表 `{1, 2, 3, 4, {5}}`, 内层的初始化列表 `{5}` 对应的数组是 `int[4]`.
        - 对于 `int[2][3][4]` 和初始化列表 `{1, 2, 3, 4, 1, 2, 3, 4, 1, 2, 3, 4, {5}}`, 内层的初始化列表 `{5}` 对应的数组是 `int[3][4]`.
        - 对于 `int[2][3][4]` 和初始化列表 `{{5}}`, 内层的初始化列表 `{5}` 之前没出现任何整数元素, 这种情况其对应的数组是 `int[3][4]`.

鉴于写文档的人的归纳能力比较捉急, 你可以在 Compiler Explorer 上多写几个初始化列表, 进一步体会上述内容的含义.

!> **注意:** 当你在 Compiler Explorer 查看数组初始化的情况时, 如果你注意到编译器 (比如 GCC) 对你写的初始化列表报告了相关警告 (warning), 那么你可以认为这个初始化列表是不合法的. 你可以通过在编译选项中 (网站右侧面板的文本框) 添加 `-Werror` 来将所有警告转换为编译错误.

例如, 对于代码 `int arr[2][3][4] = {1, 2, {3}};`, GCC 会报告 `{3}` 处出现警告 “braces around scalar initializer”. 也就是说, GCC 认为 `{3}` 实际上代表了标量 (scalar) `3`, 而非聚合类型 (aggregate) `{3}`.

在 SysY 中, 你的编译器不需要像这样聪明到足以纠正用户错误的程度. 在遇到上述 `{3}` 时, 你的编译器只需认为此处出现了一个新的初始化列表, 然后按照规则报错即可.

IR 生成

与上一节类似, 此处不再赘述.

目标代码生成

本节并未用到新的 Koopa IR 指令, 也不涉及 Koopa IR 中的新概念, 所以这部分没有需要改动的内容.

Lv9.3. 数组参数

本节新增/变更的语法规范如下:

```
FuncFParam ::= BType IDENT ["[" "]" {"[" ConstExp "]"}];
```

一个例子

```
int f(int arr[]) {
  return arr[1];
}

int main() {
  int arr[2] = {1, 2};
  return f(arr);
}
```

词法/语法分析

针对本节发生变化的语法规则, 设计新的 AST, 并更新你的 parser 实现即可.

语义分析

函数的数组参数中, 数组第一维的长度省略不写, 后序维度的长度是常量表达式, 你需要在编译时求出它们的值.

此外, 在本节中, 数组是可以被部分解引用的, 但得到的剩余部分的数组只能用来作为参数传入函数. 如果你进行了类型相关的检查, 你应该处理这种情况.

IR 生成

回忆一下 C 语言的相关内容: 函数形式参数中的 `int arr[]`, `int arr[][10]` 等, 实际上表示的是指针, 也就是 `int *arr` 和 `int (*arr)[10]`, 而不是数组. SysY 中的情况与之类似.

那么如何在 IR 中表示这种参数呢? 看了前几节的内容, 你不难得出结论: 在一个类型之前添加 `*` 就可以表示这个类型的指针类型. 所以 `int arr[]` 和 `int arr[][10]` 对应的类型分别为 `*i32` 和 `*[i32, 10]`.

那么现在问题来了: 如果我们想读取 `int arr[]` 的第二个元素, 即得到 `arr[1]` 的值, 对应的 Koopa IR 该怎么写? `getelemptr` 此时已经不好使了, 因为它要求指针必须是一个数组指针, 而 `arr` 是一个整数的指针. 为了应对这种情况, 我们引入了另一种指针运算指令: `getptr`.

`getptr ptr, index` 指令执行了如下操作: 假设指针 `ptr` 的类型是 `*T`, 指令会算出一个新的指针, 这个指针的值是 `ptr + index * sizeof(T)`, 但类型依然是 `*T`. 在逻辑上, 这种操作和 C 语言中指针运算的操作是完全一致的. 比如:

```
int *arr;   // 和 int arr[] 形式的参数等价
arr[1];
```

翻译到 Koopa IR 就是:

```
@arr = alloc *i32          // @arr 的类型是 **i32
%ptr1 = load @arr          // %ptr1 的类型是 *i32
%ptr2 = getptr %ptr1, 1    // %ptr2 的类型是 *i32
%value = load %ptr2        // %value 的类型是 i32
// 这是一段类型和功能都正确的 Koopa IR 代码
```

本质上相当于:

```
int *arr;
int *ptr = arr + 1;    // 注意这是 C 中的指针运算
*ptr;
```

对于数组的指针也同理:

```
int (*arr)[3];
arr[1][2];
```

翻译到 Koopa IR 就是:

```
@arr = alloc *[i32, 3]         // @arr 的类型是 **[i32, 3]
%ptr1 = load @arr              // %ptr1 的类型是 *[i32, 3]
%ptr2 = getptr %ptr1, 1        // %ptr2 的类型是 *[i32, 3]
%ptr3 = getelemptr %ptr2, 2    // %ptr3 的类型是 *i32
%value = load %ptr3            // %value 的类型是 i32
// 这是一段类型和功能都正确的 Koopa IR 代码
```

`getptr` 的规则就是如此, 你可以用它和 `getelemptr` 组合出和 SysY 数组相关的任意指针运算. 事实上, 如果你对 LLVM IR 有所了解, 你会发现 Koopa IR 中的 `getptr` 和 `getelemptr` 指令, 就是照着 LLVM IR 中的 `getelementptr` 指令设计的 (把这条指令拆成了两条指令), 但后者更为复杂, 对初学者而言很不友好.

综上所述, 示例程序生成的 Koopa IR 为:

```
fun @f(@arr: *i32): i32 {
%entry:
  %arr = alloc *i32
  store @arr, %arr
  %0 = load %arr
  %1 = getptr %0, 1
  %2 = load %1
  ret %2
}

fun @main(): i32 {
%entry:
  @arr = alloc [i32, 2]
  %0 = getelemptr @arr, 0
  store 1, %0
  %1 = getelemptr @arr, 1
  store 2, %1
  // 传递数组参数相当于传递其第一个元素的地址
  %2 = getelemptr @arr, 0
  %3 = call @f(%2)
  ret %3
```

```
}
```

目标代码生成

前文已经描述过 `getptr` 的含义了, 它所做的操作和 `getelemptr` 在汇编层面完全一致, 所以你不难自行得出生成目标代码的方法.

本节乃至本章的指针运算较多, 建议你在编码时时刻保持头脑清晰.

Lv9.4. 测试

"咔哒!"

你按下了 `Ctrl + S` (也可能是 `Cmd + S`), 动作干净利落, 仿佛武士收刀入鞘. 眼前那条曾在你眼中永远都无法击败的喷火龙, 现在已经奄奄一息.

脚踏一片焦土, 眼前是黎明的曙光, 心中则是百感交集.

至此, 你的编译器已经可以处理所有合法的 SysY 程序了! 祝贺, 你是最棒的!

在完成本章之前, 先进行一些测试吧.

本地测试

只测试本章

测试 Koopa IR:

```
docker run -it --rm -v 项目目录:/root/compiler maxxing/compiler-dev \
  autotest -koopa -s lv9 /root/compiler
```

测试 RISC-V 汇编:

```
docker run -it --rm -v 项目目录:/root/compiler maxxing/compiler-dev \
  autotest -riscv -s lv9 /root/compiler
```

测试所有章节

测试 Koopa IR:

```
docker run -it --rm -v 项目目录:/root/compiler maxxing/compiler-dev \
  autotest -koopa /root/compiler
```

测试 RISC-V 汇编:

```
docker run -it --rm -v 项目目录:/root/compiler maxxing/compiler-dev \
  autotest -riscv /root/compiler
```

在线评测

?> **TODO:** 待补充.