编译器开发工具

杨侯哲 李煦阳

September 2020

目录

1 工具简要介绍

在课程中我们要求每位同学独立实现一个简单的编译器，为此我们需要一些工具。根据预习内容，我们知道编译过程可粗略分为词法分析、语法分析、语法制导翻译三个过程，字符流被解析为 token 流、形成一棵语法树，最终生成某目标汇编文件。在实现我们的编译器过程中，我们将利用 Lex（词法分析器生成器）/Yacc（语法分析器生成器）辅助完成词法分析与语法分析，并最后用相应汇编器检验生成的汇编程序。本篇指导将这部分工具的安装与使用做简要说明。

此外，这次的编译课程作业或许是同学们第一个 C/C++ 工程作业，我们也希望同学能充分利用辅助工具，提高开发调试效率。[1]

1.1 Lex&Flex

在 Wiki 中对于 Lex 的描述如下：

> 在计算机科学里面，Lex 是一个产生词法分析器（Lexical analyzer，“扫描器”（scanners）或者 “Lexers”）的程序。Lex 常常与 Yacc 语法分析器产生程序（parser generator）一起使用。Lex（最早是埃里克 · 施密特和迈克 · 莱斯克制作）是许多 UNIX 系统的标准词法分析器（Lexical analyzer）产生程序，而且这个工具所作的行为被详列为 POSIX 标准的一部分。
>
> Lex 读进一个代表词法分析器规则的文件，然后输出以 C 语言为载体的词法分析器源代码。[2]

简单来说，Lex 是用来辅助生成词法分析程序的，它可以将大量重复性的工作自动计算，通过相对简单的代码可以生成词法分析程序。

而 Flex 则是 Lex 的继承者，在 Linux 中被更广泛的使用，也更容易获得。

1.2 Yacc&Bison

同样在 Wiki 中：

> Yacc（Yet Another Compiler Compiler），是 Unix/Linux 上一个用来生成编译器的编译器（编译器代码生成器）[3]。Yacc 生成的编译器主要是用 C 语言写成的语法解析器（Parser），（一般）需要与词法解析器 Lex 一起使用，再把两部分产生出来的 C 程序一并编译。Yacc 本来只在（类）Unix 系统上才有，但现时已普遍移植往 Windows 及其他平台。

而 Bison 是 Yacc 的 GNU 扩展/实现，同样作为自由软件而较容易获得与使用。

1.3 工程辅助工具

`GCC` GCC 全称为 GNU Compiler Collection，将是我们使用的主要编译工具。不仅用于编译我们的编译器，也将用于编译我们的编译器生成的汇编代码。

`git` 项目往往需要进行版本控制，比如进行代码回溯、分支管理，`git` 便是最常使用的工具。你可以在这里对它了解更多。同时为了避免实验环境的损坏等原因导致的意外，希望同学们将代码托管至云上（如github）；此外，commit 记录也是独立完成作业的一种证明。

[1] 我们不希望看见大家在 shell 中手动敲入好长好长的指令:-)

[2] 这个 C 程序可以将输入字符串流以其词法分析规则分析为 token 流，每个 token 有它的类型、值、和它的字符串内容本身。

[3] 由此可知，语法分析是编译过程中的很核心部分。

`make` 一个大型项目往往有着复杂的文件依赖关系，编译也是一个耗时的工作。`make` 便是用于辅助管理复杂的编译的依赖，并指定编译指令的程序。当然，为了程序员的方便，它引入了更多特性如伪文件，方便项目的管理与测试。你可以在这个或这个网站上了解更多。

`qemu` 为了能够提供一个标准的运行环境，在 Linux 系统下我们往往使用 qemu，这是一款开源的硬件模拟器，支持多种架构。qemu 支持多种模式，我们的课程和操作系统的课程中主要用到的是 User mode 和 System mode，在 User mode 即用户模式下 qemu 会运行针对不同指令编译的单个 Linux 程序，系统调用与 32/64 适应。在这种模式下，qemu 能够运行不同架构下的可执行文件，功能类似于一个中间层，接受程序的指令并翻译为当前系统能够识别的系统调用等；而在 Sy 位接口 stem mode 即系统模式下 qemu 会模拟一个完整的计算机系统，包括外围设备等，可以理解为抽象出了一整个计算机系统。

2 实验环境

今年的课程中我们较多的改动了实验要求。为使同学们将注意力更好地集中于学习“实现编译器”本身，我们更新、统一实验环境，并对目标汇编代码做出限制。

1. 我们将在 Linux 系统环境来中完成试验。如 Windows 平台的 WSL，docker，双系统，虚拟机等。对于机器不支持 WSL 的同学，也可使用 cygwin 完成实验。
2. 我们要求目标汇编代码为 32bit AT&T 格式，并可用 GCC assembler 生成可执行文件。目标架构默认选择 x86，在 Linux 上可额外挑战 arm。

在实验环境出现问题时，大家需要对几项信息保持敏感：操作系统、环境变量、程序版本、程序位数、目标平台，某些 linux 常用指令可以帮助你快速获取相关信息。[4]

2.1 为什么使用 Linux 与 GCC

Linux 会是我们在将来非常常用的环境，包括本学期的操作系统实验以及将来进行科研或工作环境，通过命令行操控计算机的形式是十分常用的方式，如果同学们以前没有过相关的经验，希望能够认真查阅一下相关资料，难度曲线可能在一开始会稍陡峭一些，但熟悉之后往往能够比鼠标操控的效率高得多。

在本课程中我们会要求同学们手写简单的汇编代码，以及最后的大作业还会要求通过自己编写的编译器来将高级程序语言翻译为汇编代码，这其中汇编代码的正确性就需要通过真正的编译运行来检测，而我们希望通过 GCC 编译器尽可能地统一检验代码。同时，GCC 也是工业级别工具，也希望借此课程增加同学们对它的了解。在实践中，同学们会发现不同版本的 GCC 自动生成的汇编代码在不同平台会带有不同的伪指令，相互间不一定能够通用，我们会提供常用环境的汇编代码模板供同学们参考使用，希望能够更方便地解决同学们遇到的问题。

2.2 架构的选择

arm 架构与 x86 架构分别是很典型的 RISC 指令集和 CISC 指令集。x86 相信同学们都有了解，因为大家的机器大多是 x86 架构（笑）；对于 arm，你可以在这里学到你需要的全部内容；你可以部分

[4] 本次实验所用工具均为常用工具，环境搭建不困难；配置、维护系统环境也是程序员的最基本技能。

地将它与同为 RISC 指令集的 MIPS 类比。就大作业而言，CISC 指令集将提供更丰富随意的指令用于代码生成，RISC 指令集则会为中间代码的分析提供方便。两者没有根本上的难度差异。

对于选择 arm 架构的同学们来说，Linux 系统环境是必须的，目前我们常用的笔记本、PC 等基本都是 x86 架构的硬件，想要运行 arm 架构的可执行文件往往需要通过额外的中间层，我们要求选择 arm 架构的同学们在 Linux 系统上安装 Qemu 来运行程序。`-static` 可能会成为你常用的编译 flag。

对于选择 x86 架构的同学们，我们希望你们尽可能地使用 Linux 系统环境，由于我们希望统一实现 32 位汇编代码而目前的硬件往往都已经是 64 位，最好也在 Linux 系统上安装 Qemu 来运行程序。对于 Windows 平台的同学们来说，WSL 是很方便的方式，但如果有部分同学由于条件限制实在无法在 Linux 系统环境下运行程序，可以通过 Cygwin，一个将 POSIX 系统（包括 Linux 系统）上的软件移植到 Windows 上的软件集合，来运行整个实验流程要求的所有软件，但要注意的是，这样会造成一些兼容性的问题，我们会同样提供 Cygwin 平台的样例代码，但是移植软件仍然可能由于版本不同、移植实现程度不同等等问题导致一些可能难以解决的问题，在配置环境时希望同学们小心注意。而对于 Mac 平台的同学，建议你借助 Docker 使用 linux 环境。[5]

2.3 系统环境的安装

2.3.1 Windows

WSL 的安装 WSL 是 Windows Subsystem for Linux 的简写，也是在 Windows 上最推荐的实验环境。它允许开发者在 Windows 上方便的使用 Linux 程序，避免了安装运行虚拟机的复杂性。

WSL 对于系统有一定需求，你可以在这里找到相应说明。你可以通过升级操作系统以满足需求；但若你的机器是 32 位的，则无法使用 WSL。WSL 的安装说明可以在这里找到。

你可能会依次遇到这个错误和这个错误。你可能需要升级 WSL kernel，并借助官方工具将 Windows 升级至最新版本。

我们使用 Ubuntu18.04 用于举例。实验工具的安装见后续工具安装部分。

cygwin 的安装 你可以在这里获取 cygwin，并在安装时点选好你需要的程序（GCC/Flex/Bison/-make/git/vim 等）。你可能想要了解 cygwin 与 wsl 的区别，和与 mingw 的区别，以减少实验中踩坑几率。cygwin 中的 GCC 与 Linux 中 GCC 参数会有差异（比如是否拥有`-m32` 参数），借此机会，你可以练习查阅手册与探索 `stackoverflow`。

为了在 64 位 cygwin 中编译与运行 32 位程序，我们需要安装一些额外的工具，具体的可以看这里的讨论，不用担心没有一次安装全部内容，cygwin 的安装程序可以非常便捷的增加或减少安装的程序包。

完成后你可以直接进入测试步骤。

2.3.2 Mac

Docker 实际上 Docker 是在 Windows 和 Mac 通用的，但是 Windows 上有更加方便的解决方式；而对于 Mac 的同学，Docker 要比虚拟机要方便一些的。

你可以看这一份教程启用 Ubuntu 容器。[6]了解 docker 与虚拟机的差异也会很有帮助。

Vscode 拥有 remote-container 插件，或许可以帮助你进行开发。

[5] qemu 在 macos 上并不支持 user mode 直接运行其他架构的二进制程序。同时，在 10.15 后，mac 上亦强制不能直接运行 32 位程序（即使你已编译出可执行文件）。

[6] 使用 lastest ubuntu 即可

3 词法分析以及语法分析辅助工具的安装

当程序员考虑要写一个文本解析程序时，他可能会想到 Flex+Bison 或 antlr。由于工具不是我们关注的重点，本次实验将要求使用 Flex+Bison，但你仍可以利用搜索引擎进一步了解它们的差异。实际上，LL/LR 之外，还有其他 parsing techniques，同学们也可拓展了解。

3.1 基本工具

由于你不得不决定在 Linux 环境完成作业，工具的安装就变得非常方便。你只需要使用你的 Linux 发行版的相应包管理器[7]安装即可。比如在 Ubuntu 上，你可以通过一下指令获得它们：

```
sudo apt update
sudo apt install build-essential
sudo apt install gcc-multilib
sudo apt install -y flex
sudo apt install -y bison
sudo apt install -y qemu
sudo apt install -y qemu-system
sudo apt install -y qemu-user
# whatever you need
# 如果接触过Linux的同学们可能会发现，我们有时候用到apt-get有时候用apt，那么两者具体有什么差异吗
# 其实这是非常简单的问题，同学们感兴趣的话自行查询可以很容易得到答案
```

若你使用其他 Linux 发行版，相信聪明的你可以找到相应的安装方法: -)

此外，WSL 与 docker 或许需要远程连接。vscode 有相应插件可以帮助你。

3.2 Arm 工具链

之前我们说过，目前我们使用的电脑往往是 x86 架构的，那么我们要是想要编译出 arm 架构的可执行文件该怎么办呢？这时候就会用到交叉编译了，在 Ubuntu 的 terminal 中，你可以使用以下指令安装 arm 交叉编译器：

```
    # 实际上我们可以看到有gcc-arm-linux-gnueabi和gcc-arm-linux-gnueabihf两种格式 具体的差异同样推荐同学们自行查阅了解
    sudo apt-get install gcc-arm-linux-gnueabi
```

4 环境测试

4.1 测试 GCC/make/Flex/Bison

在实验环境搭建成功后，你可以 `clone` 这个仓库进行测试。测试方法即根据 `README.md`。

若你不在 Linux 平台上测试（比如 cygwin），可能需要微调 `Makefile` 中的编译参数。

4.2 熟悉 x86 工具链

假设在某文件夹中，你写了一个测试用的 `main.c` 文件。你可以先将它编译为某要求的汇编文件，再将它编译为可执行程序，再运行它做测试。

如果你使用 cygwin，可能需要根据提示错误修改编译参数。也可能你需要的 GCC 是 `i686-pc-cygwin-gcc`。

```
    # 在实验中你可能遇到“不知道汇编代码应该是什么样子”的情况。那么就让GCC生成一个标准的给你看！
    # 这里的编译工作只是为了让你熟悉整个工作流而进行的，之后真正写作业时汇编代码会是你手写的或者通过你编写的编译器而非GCC生成的
    # 你可能会认为可以以这个方式逃避汇编编程作业，但我们会问你你写的每一条代码的含义，包括伪指令，以确保你的了解。
    # 编译参数是什么含义，就由你大展身手了。

    gcc -O0 -o test.s -S -masm=att -m32 -fno-exceptions -fno-asynchronous-unwind-tables -fno-builtin -fno-pie main.c
```

[7] 如果下载速度太慢的话记得换下载源哦，具体方法请自行搜索

```
gcc -m32 test.s -o test
# 在我们用的Ubuntu中可能不能运行32位的程序，我们推荐使用qemu-i386来进行测试，从而不必关注一些运行库的问题
qemu-i386 ./test
# 如果使用Cygwin的话是没有qemu可用的，所以需要同学们设置运行环境，具体遇到问题的话可以参考在之前提到的讨论
```

4.3 熟悉 arm 工具链

Linux 下的 `file` 指令会帮助你判断可执行文件的目标架构。

```
arm-linux-gnueabi-gcc -o arm.s -S -O0 main.c  -fno-asynchronous-unwind-tables
# -static 参数是在编译的哪个阶段起作用呢
    arm-linux-gnueabi-gcc arm.s -o arm -static
# 你可能会神奇地发现不加qemu-arm也可以运行，是进行隐式调用的原因
qemu-arm ./arm
```