



南开大学
Nankai University

南 开 大 学

计 算 机 学 院

编译系统原理实验报告

定义你的编译器、汇编编程 & 熟悉辅助工具

张刘明 2110049

艾明旭 2111033

年级：2021 级

专业：信息安全-法学 信息安全

指导教师：王刚 李忠伟

2023 年 10 月 10 日

目录

一、 概述	1
(一) 实验描述	1
1. 思考	1
2. 要求:	1
二、 定义你的编译器	1
(一) 上下文无关文法	1
(二) SysY 语言特性	2
(三) CFG 描述 SysY 语言特性	2
1. 关键字	2
2. 变量	3
3. 常量	4
4. 运算符和表达式	4
5. 语句	5
6. 函数	6
(四) 形式化定义	6
1. 变量声明	7
2. 常量声明	7
3. 表达式	7
4. 赋值表达式	8
5. 逻辑表达式	8
6. 关系表达式	8
7. 算数表达式	8
8. 函数	9
9. 系统操作	9
10. 循环语句	9
11. 分支语句	9
12. 跳转语句	9
(五) 思考题: 实现一个计算机程序(编译器)来将 sysy 程序转换为汇编程序	9
三、 汇编编程流程	15
1. 代码说明	16
2. 代码解析	16
(一) 验证具体操作	19
四、 总结	28

一、 概述

(一) 实验描述

确定你要实现的编译器支持哪些 SysY 语言特性，给出其形式化定义——学习教材第 2 章及第 2 章讲义中的 2.2 节、参考 SysY 中巴克斯瑙尔范式定义，用上下文无关文法描述你的 SysY 语言子集。

设计几个 SysY 程序（如“预备工作 1”给出的阶乘或斐波那契），编写等价的 ARM 汇编程序，用汇编器生成可执行程序，调试通过、能正常运行得到正确结果。这些程序应该尽可能全面地包含你支持的语言特性。

1. 思考

如果不是人“手工编译”，而是要实现一个计算机程序（编译器）来将 SysY 程序转换为汇编程序，应该如何做？这个编译器程序的数据结构和算法设计是怎样的？

注意：编译器不能只会翻译一个源程序，而是要有能力翻译所有合法的 SysY 程序。而穷举所有 SysY 程序（无穷无尽）是不可能的，怎么办？搞定每个语言特性如何翻译即可！

2. 要求：

可学习 GCC 生成的其他 C 程序的汇编程序，仿照着编写自己 SysY 程序的汇编程序。

二、 定义你的编译器

这一部分作业的主要要求是了解你的编译器所支持的 SysY 语言特性，如支持何种数据类型（int 等），支持变量声明，赋值语句，复合语句，if 分支语句，以及 while/for 循环，支持算术运算（加减乘除、按位与或等）、逻辑运算（逻辑与或等）、关系运算（不等、等于、大于、小于等），支持函数、数组指针等等。从中选取你要实现的部分定义为你编译器功能，使用上下文无关文法描述你所选取的 SysY 语言子集。

(一) 上下文无关文法

上下文无关文法是一种用于描述程序设计语言语法的表示方式。一般来说，一个上下文无关文法（context-free grammar）由四个元素组成：

(1) 一个终结符号集合 VT，它们有时也称为“词法单元”。终结符号是该文法所定义的语言的基本符号的集合。

(2) 一个非终结符号集合 VN，它们有时也称为“语法变量”。每个非终结符号表示一个终结符号串的集合。

(3) 一个产生式集合 P，其中每个产生式包括一个称为产生式头或左部的非终结符号，一个箭头，和一个称为产生式体或右部的由终结符号及非终结符号组成的序列。产生式主要用来表示某个语法构造的某种书写形式。如果产生式头非终结符号代表一个语法构造，那么该产生式体就代表了该构造的一种书写方式。

(4) 指定一个非终结符号为开始符号 S。因此，上下文无关文法可以通过 (VT, VN, P, S) 这个四元式定义。在描述文法时，我们将数位、符号和加粗字符串看作终结符号，将斜体字符串看作非终结符号，以同一个非终结符号为头部的多个产生式的右部可以放在一起表示，不同的右部之间用符号 | 分隔。

(二) SysY 语言特性

- 数据类型: int
- 变量声明、常量声明, 常量、变量的初始化
- 语句: 赋值 (=)、表达式语句、语句块、if、while、return
- 表达式: 算术运算 (+、-、*、/、%、其中 +、- 都可以是单目运算符)、关系运算 (==, >, <, >=, <=, !=) 和逻辑运算 (&& (与)、|| (或)、! (非))
- 注释
- 输入输出
- 数组
- 变量、常量作用域——在语句块中包含变量、常量声明, break、continue 语句
- 函数

(三) CFG 描述 SysY 语言特性

CFG 是一个四元组 $G = (N, T, P, S)$, 其中

- (1) N 是非终结符 (Nonterminals) 的有限集合;
- (2) T 是终结符 (Terminals) 的有限集合, 且 $N \cap T = \emptyset$;
- (3) P 是产生式 (Productions) 的有限集合, $A \rightarrow a$, 其中 $A \in N$ (左部), $a \in (N \cup T)^*$ (右部), 若 $a = \epsilon$, 则称 $A \rightarrow \epsilon$ 为空产生式 (也可以记为 $A \rightarrow$);
- (4) S 是文法的开始符号 (Start symbol), $S \in N$

CFG 产生语言的基本方法: 推导

推导的定义

推导的定义将产生式左部的非终结符替换为右部的文法符号序列 (展开产生式, 用标记 \Rightarrow 表示), 直到得到一个终结符序列。

推导的符号: \Rightarrow

推导的输入: 产生式左部

推导的输出: 一个终结符序列

1. 关键字

C++ 常用关键字如表1所示, 每一个关键字在上下文无关文法中都会看作一个终结符, 即语法树的叶结点。本实验将选取其中的一部分作为子集, 构造 SysY 语言。

类型	关键字
数据类型相关	<i>int, bool, true, false, char, wchar_t, int, double, float, short, long, signed, unsigned</i>
控制语句相关	<i>switch, case, default, do, for, while, if, else, break, continue, goto</i>
定义、初始化相关	<i>const, volatile, enum, export, extern, public, protected, private, template, static, struct, class, union, mutable, virtual</i>
系统操作相关	<i>catch, throw, try, new, delete, friend, inline, operator, reinterpret_cast, typename</i>
命名相关	<i>using, namespace, typeid</i>
函数和返回值相关	<i>void, return, sizeof, typedef</i>
其他	<i>this, asm, _cast</i>

表 1: C++ 关键字

2. 变量

C 语言中规定，将一些程序运行中可变的值称之为变量，与常量相对。在程序运行期间，随时可能产生一些临时数据，应用程序会将这些数据保存在一些内存单元中，每个内存单元都用一个标识符来标识。这些内存单元我们称之为变量，定义的标识符就是变量名，内存单元中存储的数据就是变量的值。变量可以作左值，常量则只能作为右值。变量除了与常量相同的整型类型、实型类型、字符类型这三个基本类型之外，还有构造类型、指针类型、空类型。

三种基本类型

整型变量 整型常量为整数类型 *int* 的数据。可分别如下表示为八进制、十进制、十六进制

- 十进制整型变量：0, 123, -1
- 八进制整型变量：0123, -01
- 十六进制整型变量：0x123, -0x88

实型变量 实型变量是实际中的小数，又称为浮点型变量。按照精度可以分为单精度浮点数（float）和双精度浮点数（double）。浮点数的表示有三种方式如下：

- 指明精度的表示：以 f 结尾为单精度浮点数，如：2.3f；以 d 结尾为双精度浮点数，如：3.6d
- 不加任何后缀的表示：11.1, 5.5
- 指数形式的表示：5.022e+23f, 0f

字符变量 *char* 用于表示一个字符，表示形式为 '需要表示的字符常量'。其中，所表示的内容可以是英文字母、数字、标点符号以及由转义序列来表示的特殊字符。如 'a' '3' ' ' '\n'。

变量的定义 用于为变量分配存储空间，还可为变量指定初始值。程序中，变量有且仅有一个定义。变量有三个基本要素：变量名，代表变量的符号；变量的数据类型，每一个变量都应具有一种数据类型且内存中占据一定的储存空间；变量的值，变量对应的存储空间中所存放的内容。变量的定义可以以如下的形式：

```
1 type variable_list
```

在我们定义的 SysY 语言中，将支持整型变量（十进制）、字符型变量以及行主存储的整型一维数组类型。

3. 常量

C 语言中规定，将一些不可变的值称之为常量。常量可以分为整型常量、实型常量、字符常量这三种常量，其形式与整型变量、实型变量、字符变量基本相同，只不过在声明时必须初始化且在程序中不可以改变其值。在我们定义的 SysY 语言中，将定义整型常量（十进制）和字符型常量。

4. 运算符和表达式

在 C 语言中，运算符分为算术运算符、关系运算符、逻辑运算符、位运算符、赋值运算符、杂项运算符六大类。其中，我们所设计的 SysY 语言将定义以下运算符：

运算符	描述
+	把两个操作数相加
-	从第一个操作数中减去第二个操作数
*	把两个操作数相乘
/	分子除以分母
%	取模运算符，整除后的余数

表 2: 算术运算符

运算符	描述
==	检查两个操作数的值是否相等，如果相等则条件为真。
!=	检查两个操作数的值是否相等，如果不相等则条件为真。
>	检查左操作数的值是否大于右操作数的值，如果是则条件为真。
<	检查左操作数的值是否小于右操作数的值，如果是则条件为真。
>=	检查左操作数的值是否大于或等于右操作数的值，如果是则条件为真。
<=	检查左操作数的值是否小于或等于右操作数的值，如果是则条件为真。

表 3: 关系运算符

表达式 由运算分量和运算符按一定规则组成。运算分量是运算符操作的对象，通常是各种类型的数据。运算符指明表达式的类型；表达式的运算结果是一个值——表达式的值。出现在赋值运算符左边的分量为左值，代表着一个可以存放数据的存储空间；左值只能是变量，不能是常量或表达式，因为只有变量才可以带表存放数据的存储空间。出现在赋值运算符右边的分量为右值，右值没有特殊要求。

运算符	描述
&&	称为逻辑与运算符。如果两个操作数都非零，则条件为真。
	称为逻辑或运算符。如果两个操作数中有任何一个非零，则条件为真。
!	称为逻辑非运算符。用来逆转操作数的逻辑状态。

表 4: 逻辑运算符

运算符	描述
=	简单的赋值运算符，把右边操作数的值赋给左边操作数。

表 5: 赋值运算符

运算符	描述
&	返回变量的地址。
*	指向一个变量。

表 6: 复杂运算符

运算符优先级 运算符中优先级确定了表达式中项的组合，这会极大地影响表达式的计算过程以及结果。运算的优先顺序为：括号优先运算 → 优先级高的运算符优先运算 → 优先级相同的运算参照运算符结合性依次进行。当表达式包含多个同级运算符时，运算的先后次序分为左结合规则和右结合规则。其中左结合规则是从左向右依次计算，包括的运算符有双目的算术运算符、关系运算符、逻辑运算符、位运算符、逗号运算符；右结合规则是从右向左依次计算，包括的运算符有可以连续运算的单目运算符、赋值运算符、条件运算符。运算符优先级由高到低排列：后缀 → 一元 → 乘除 → 加减 → 移位 → 关系 → 相等 → 位与 → 位异或 → 位或 → 逻辑与 → 逻辑或 → 条件 → 赋值 → 逗号

5. 语句

在 C 语言中，语句分为说明语句、表达式语句、控制语句、标签语句、复合语句和块语句。在我们所定义的 SysY 语言中，我们将定义表达式语句和控制语句，其中控制语句分为分支语句、循环语句和转向语句。

表达式语句 任意有效表达式都可以作为表达式语句，其形式为表达式后面加上“;”。

分支语句 if 语句和 if……else 语句，由关键字 if 和 else 组成。其基本形式如下

```

1  if(expr){
2      stmts
3  }
4  else{
5      stmts
6  }
```

循环语句 又称重复语句，用于重复执行某些语句。本实验的循环语句由 while 语句实现，基本形式如下

```
1 while{  
2     stmts  
3 }
```

转向语句 用于从循环体跳出的 break 语句；用于立即结束本次循环而去继续下一次循环的 continue 语句；用于立即从某个函数中返回到调用该函数位置的 return 语句。

6. 函数

函数的定义 函数是一组一起执行一个任务的语句。程序中功能相同，结构相似的代码段可以用函数进行描述。函数的功能相对独立，用来解决某个问题，具有明显的入口和出口。函数也可以称为方法、子例程或程序等等。

函数说明 C 语言中，函数必须先说明后调用。函数的说明方式有两种，一种是函数原型，相当于“说明语句”，必须出现在调用函数之前；一种是函数定义，相当于“说明语句 + 初始化”，可以出现在程序的任何合适的地方。在函数声明中，参数的名称并不重要，只有参数的类型是必需的。函数的声明形式如下所示：

```
1 return_type function_name(parameter list);
```

形式化定义 C 语言中，函数的形式化定义如下所示：

```
1 return_type function_name(parameter list)  
2 {  
3     body of the function  
4 }
```

本实验定义的 SysY 语言的函数定义完全与此相同。

函数的参数 函数可以分为有参函数和无参函数。如果函数要使用参数，则必须声明接受参数值的变量，这些变量称为函数的形式参数。形式参数和函数中的局部变量一样，在函数创建时被赋予地址，在函数退出是被销毁。函数参数的调用分为传值调用和引用调用两种，传值调用是将实际的变量的值复制给形式参数，形式参数在函数体中的改变不会影响实际变量；引用调用是将形式参数作为指针调用指向实际变量的地址，当对在函数体中对形式参数的指向操作时，就相当于对实际参数本身进行的操作。

除此之外，函数还可以分为内联函数、外部函数等等，并还可以进行重载等操作。**本次实验的 SysY 语言，只对函数最基本的功能进行实现。**

(四) 形式化定义

接下来，我们将采用 CFG 即上下文无关文法对 SysY 语言进行形式化定义。上下文无关文由一个终结符号集合 V_T 、一个非终结符号集合 V_N 、一个产生式集合 P 和一个开始符号 S 四个元素组成。在接下来的定义中，数位、符号和黑体字符串将被看作终结符号，斜体字符串将被看

作非终结符号。若多个产生式以一个非终结符号为头部，则这些产生式的右部可以放在一起，并用 $|$ 分割。

名称	符号	名称	符号
声明语句	<i>decl</i>	标识符	<i>id</i>
标识符列表	<i>idlist</i>	数据类型	<i>type</i>
表达式	<i>expr</i>	一元表达式	<i>unary_expr</i>
赋值表达式	<i>assign_expr</i>	逻辑表达式	<i>logical_expr</i>
算数表达式	<i>math_expr</i>	关系表达式	<i>relation_expr</i>
数字	<i>digit</i>	整数	<i>decimal</i>
符号和字母	<i>character</i>	常量定义	<i>const_init</i>
分配内存	<i>allocate</i>	回收内存	<i>recovery</i>
语句	<i>stmt</i>	循环语句	<i>loop_stmt</i>
分支语句	<i>selection_stmt</i>	跳转语句	<i>jmp_stmt</i>
函数定义	<i>funcdef</i>	函数参数	<i>para</i>
函数参数列表	<i>paralist</i>	函数名称	<i>funcname</i>
函数返回值	<i>re_type</i>		

表 7: 下文中各符号含义

1. 变量声明

变量可以声明分为仅声明变量和声明变量且赋初值。(整型变量只支持十进制) 数组由指针实现。

$$\begin{aligned}
 idlist &\rightarrow idlist, id \mid id \\
 type &\rightarrow \text{int} \mid \text{char} \\
 decl &\rightarrow type \ idlist \mid \\
 &\quad type \ id = logical_expr \mid \\
 &\quad type \ id = unary_expr \mid \\
 &\quad type \ *id = \&id
 \end{aligned}$$

2. 常量声明

常量包括整型常量（十进制）和字符型常量。

$$const_init \rightarrow \text{const } type \ id = unary_expr$$

3. 表达式

表达式可以分为一元表达式、赋值表达式、逻辑表达式、算数表达式、关系表达式。

$$expr \rightarrow unary_expr \mid assign_expr \mid logical_expr \mid math_expr \mid relation_expr$$

4. 赋值表达式

赋值表达式不能对常量进行赋值。

$$\begin{aligned} \text{digit} &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \\ \text{decimal} &\rightarrow \text{digit} \mid \text{decimal digit} \\ \text{character} &\rightarrow _ \mid \text{a} - \text{z} \mid \text{A} - \text{Z} \\ \text{unary_expr} &\rightarrow \text{decimal} \mid \text{'character'} \mid \text{id} \\ \text{assign_expr} &\rightarrow \text{id} = \text{unary_expr} \mid \\ &\quad \text{id} = \text{logical_expr} \mid \\ &\quad \text{id} = \text{funcname}(\text{paralist}) \end{aligned}$$

5. 逻辑表达式

逻辑表达式包括逻辑与、逻辑或、逻辑非运算。

$$\begin{aligned} \text{logical_expr} &\rightarrow \text{unary_expr} \mid \\ &\quad \text{!(logical_expr)} \mid \\ &\quad \text{logical_expr} \mid \text{logical_expr} \mid \\ &\quad \text{logical_expr} \& \& \text{logical_expr} \end{aligned}$$

6. 关系表达式

关系表达式包括判断两个值是否相等或比较两值的大小。

$$\begin{aligned} \text{relation_expr} &\rightarrow \text{unary_expr} == \text{unary_expr} \mid \\ &\quad \text{unary_expr} != \text{unary_expr} \mid \\ &\quad \text{unary_expr} > \text{unary_expr} \mid \\ &\quad \text{unary_expr} < \text{unary_expr} \mid \\ &\quad \text{unary_expr} >= \text{unary_expr} \mid \\ &\quad \text{unary_expr} <= \text{unary_expr} \end{aligned}$$

7. 算数表达式

算数表达式包括加、减、乘、除、取模、取负六种运算。

$$\begin{aligned} \text{math_expr} &\rightarrow \text{unary_expr} \mid \\ &\quad - \text{unary_expr} \mid \\ &\quad \text{math_expr} + \text{math_expr} \mid \\ &\quad \text{math_expr} - \text{math_expr} \mid \\ &\quad \text{math_expr} * \text{math_expr} \mid \\ &\quad \text{math_expr} / \text{math_expr} \mid \\ &\quad \text{math_expr} \% \text{math_expr} \end{aligned}$$

8. 函数

函数的返回值有整型、字符型、指针型三种，参数有整型、字符型、指针型、引用四种。（数组由指针实现）

$$\begin{aligned} funcdef &\rightarrow re_type\ funcname(paralist)\ stmt \\ paralist &\rightarrow para, paralist \mid para \\ para &\rightarrow type\ id \mid \\ &\quad type * id \mid \\ &\quad type\& id \\ re_type &\rightarrow type \mid type * \mid void \end{aligned}$$

9. 系统操作

系统操作包括分配内存和回收内存。

$$\begin{aligned} allocate &\rightarrow type *id = new\ type [id] \mid \\ &\quad type *id = new\ type [decimal] \\ recovery &\rightarrow delete [] id \end{aligned}$$

10. 循环语句

循环语句利用 while 实现。

$$loop_stmt \rightarrow while(expr) \{stmt\}$$

11. 分支语句

分支语句利用 if else 实现。

$$\begin{aligned} selection_stmt &\rightarrow if(expr) \{stmt\} \mid \\ &\quad if(expr) \{stmt\} else \{stmt\} \end{aligned}$$

12. 跳转语句

跳转语句包括继续执行循环、退出循环和返回值。

$$\begin{aligned} jmp_stmt &\rightarrow continue \mid \\ &\quad break \mid \\ &\quad return \mid \\ &\quad return\ expr \end{aligned}$$

(五) 思考题：实现一个计算机程序（编译器）来将 sysy 程序转换为汇编程序

要实现一个编译器将 sysy 程序转换为汇编程序，需要设计以下数据结构和算法：

词法分析器：将输入的 sysy 程序转换为一系列的 token，每个 token 代表一个语法单元，如变量名、关键字、运算符等。可以使用有限状态自动机（DFA）或正则表达式来实现词法分析器。

语法分析器：将 token 序列转换为抽象语法树（AST），并检查语法错误。可以使用递归下降分析器或 LR 分析器来实现语法分析器。

语义分析器：对 AST 进行类型检查、符号表管理和中间代码生成。可以使用类型推导、符号表和三地址码等技术来实现语义分析器。

代码生成器：将 AST 转换为目标汇编代码。可以使用基于栈的虚拟机或直接生成汇编代码的方法来实现代码生成器。

在实现编译器时，需要使用适当的数据结构来存储 AST、符号表和中间代码等信息。例如，可以使用树形结构来表示 AST，哈希表来表示符号表，三元组来表示中间代码等。

此外，还需要考虑如何优化生成的汇编代码，以提高程序的性能和效率。可以使用常量折叠、死代码消除、循环展开等技术来进行优化。

词法分析器

```
1  %{
2  #include<stdio.h>
3  #include<stdlib.h>
4  int line = 1;
5  int column = 1;
6  int charnum=0;
7  char path[100];
8
9  void clearcolumn();
10 void addcolumn(int);
11 %}
12
13 VOID void
14 INT int
15 CHAR char
16 IF if
17 WHILE while
18 FOR for
19 BREAK break
20 CONTINUE continue
21 RETURN return
22 SEMICOLON ;
23 COMMA ,
24 LPAREN \(
25 RPAREN\)
26 LBRACE \{
27 RBRACE \}
28 CMPEQUAL \==
29 EQUAL \=
30 ADD \+
31 SUB \-
32 MUL \*
33 DIV \/
34 SMALLER \<
35 BIGGER \>
```

```

36
37
38
39
40 digit [0-9]
41 octal_digit [0-7]
42 hexadecimal_digit [0-9a-fA-F]
43 identifier_nondigit [a-zA-Z_]
44
45 decimal_const [1-9]{digit}*
46 octal_const (0){octal_digit}*
47 hexadecimal_const (0x|0X){hexadecimal_digit}*
48
49 Identifier ({identifier_nondigit})({identifier_nondigit}|{digit})*
50
51
52
53
54 %%
55
56 {decimal_const} {
57     addcolumn(charnum);
58     printf("%-20s%-20s%-10d%-10d%-20s\n", "decimal_const", yytext, line, column,
59         yytext);
60     charnum = yyleng;
61 }
62 {octal_const} {
63     addcolumn(charnum);
64     printf("%-20s%-20s%-10d%-10d%-20s\n", "octal_const", yytext, line, column,
65         yytext);
66     charnum = yyleng;
67 }
68 {hexadecimal_const} {
69     addcolumn(charnum);
70     printf("%-20s%-20s%-10d%-10d%-20s\n", "hexadecimal_const", yytext, line,
71         column, yytext);
72     charnum = yyleng;
73 }
74 {VOID} {
75     addcolumn(charnum);
76     printf("%-20s%-20s%-10d%-10d\n", "VOID", yytext, line, column);
77     charnum = yyleng;
78 }
79 {INT} {
80     addcolumn(charnum);
81     printf("%-20s%-20s%-10d%-10d\n", "INT", yytext, line, column);
82     charnum = yyleng;
83 }

```

```
81 {CHAR} {
82     addcolumn(charnum);
83     printf("%-20s%-20s%-10d%-10d\n", "CHAR", yytext, line, column);
84     charnum = yyleng;
85 }
86 {IF} {
87     addcolumn(charnum);
88     printf("%-20s%-20s%-10d%-10d\n", "IF", yytext, line, column);
89     charnum = yyleng;
90 }
91 {WHILE} {
92     addcolumn(charnum);
93     printf("%-20s%-20s%-10d%-10d\n", "WHILE", yytext, line, column);
94     charnum = yyleng;
95 }
96 {FOR} {
97     addcolumn(charnum);
98     printf("%-20s%-20s%-10d%-10d\n", "FOR", yytext, line, column);
99     charnum = yyleng;
100 }
101 {BREAK} {
102     addcolumn(charnum);
103     printf("%-20s%-20s%-10d%-10d\n", "BREAK", yytext, line, column);
104     charnum = yyleng;
105 }
106 {CONTINUE} {
107     addcolumn(charnum);
108     printf("%-20s%-20s%-10d%-10d\n", "CONTINUE", yytext, line, column);
109     charnum = yyleng;
110 }
111 {RETURN} {
112     addcolumn(charnum);
113     printf("%-20s%-20s%-10d%-10d\n", "RETURN", yytext, line, column);
114     charnum = yyleng;
115 }
116 {Identifier} {
117     addcolumn(charnum);
118     printf("%-20s%-20s%-10d%-10d%-20p\n", "Identifier", yytext, line, column,
119           yytext);
119     charnum = yyleng;
120 }
121 {SEMICOLON} {
122     addcolumn(charnum);
123     printf("%-20s%-20s%-10d%-10d\n", "SEMICOLON", yytext, line, column);
124     charnum = yyleng;
125 }
126 {COMMA} {
127     addcolumn(charnum);
```

```
128     printf("%-20s%-20s%-10d%-10d\n", "COMMA", yytext, line, column);
129     charnum = yyleng;
130 }
131 {LPAREN} {
132     addcolumn(charnum);
133     printf("%-20s%-20s%-10d%-10d\n", "LPAREN", yytext, line, column);
134     charnum = yyleng;
135 }
136 {RPAREN} {
137     addcolumn(charnum);
138     printf("%-20s%-20s%-10d%-10d\n", "RPAREN", yytext, line, column);
139     charnum = yyleng;
140 }
141 {LBRACE} {
142     addcolumn(charnum);
143     printf("%-20s%-20s%-10d%-10d\n", "LBRACE", yytext, line, column);
144     charnum = yyleng;
145 }
146 {RBRACE} {
147     addcolumn(charnum);
148     printf("%-20s%-20s%-10d%-10d\n", "RBRACE", yytext, line, column);
149     charnum = yyleng;
150 }
151 {CMPEQUAL} {
152     addcolumn(charnum);
153     printf("%-20s%-20s%-10d%-10d\n", "CMPEQUAL", yytext, line, column);
154     charnum = yyleng;
155 }
156 {EQUAL} {
157     addcolumn(charnum);
158     printf("%-20s%-20s%-10d%-10d\n", "EQUAL", yytext, line, column);
159     charnum = yyleng;
160 }
161 {ADD} {
162     addcolumn(charnum);
163     printf("%-20s%-20s%-10d%-10d\n", "ADD", yytext, line, column);
164     charnum = yyleng;
165 }
166 {SUB} {
167     addcolumn(charnum);
168     printf("%-20s%-20s%-10d%-10d\n", "SUB", yytext, line, column);
169     charnum = yyleng;
170 }
171 {MUL} {
172     addcolumn(charnum);
173     printf("%-20s%-20s%-10d%-10d\n", "MUL", yytext, line, column);
174     charnum = yyleng;
175 }
```

```

176 {DIV} {
177     addcolumn(charnum);
178     printf("%-20s%-20s%-10d%-10d\n", "DIV", yytext, line, column);
179     charnum = yyleng;
180 }
181 {SMALLER} {
182     addcolumn(charnum);
183     printf("%-20s%-20s%-10d%-10d\n", "SMALLER", yytext, line, column);
184     charnum = yyleng;
185 }
186 {BIGGER} {
187     addcolumn(charnum);
188     printf("%-20s%-20s%-10d%-10d\n", "BIGGER", yytext, line, column);
189     charnum = yyleng;
190 }
191
192
193 \n {
194     ++line;
195     clearcolumn();
196 }
197
198 . {
199     addcolumn(1);
200 }
201 [ \t]+ {}
202
203 %%
204 int main() {
205     scanf("%s", path);
206     yyin = fopen(path, "r");
207     yylex();
208     return 0;
209 }
210 int yywrap() {
211     return 1;
212 }
213 void clearcolumn() {
214     column = 1;
215     charnum = 0;
216 }
217 void addcolumn(int temp) {
218     column += temp;
219 }

```

Makefile

```

1 .PHONY: lex, clean
2 lex:

```



```

3      lex source_code/Lexical_Analyzer.l
4      gcc lex.yy.c -o Lexical_Analyzer -ll
5      ./Lexical_Analyzer
6 clean:
7      rm -fr lex.yy.c Lexical_Analyzer

```

Makefile 中定义了 lex：编译并运行词法分析器及 clean：清理编译文件

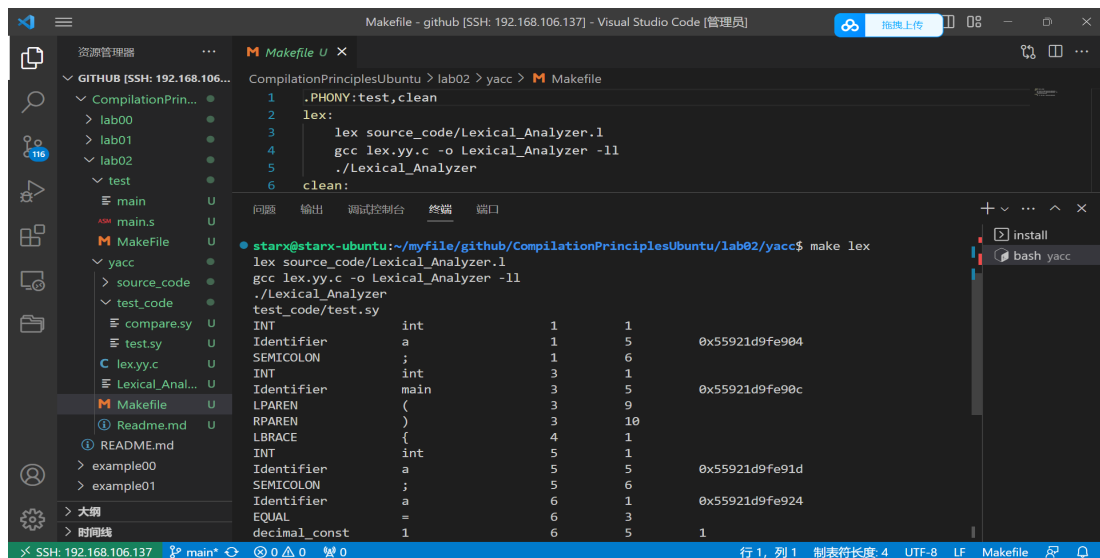


图 1: 测试词法分析器

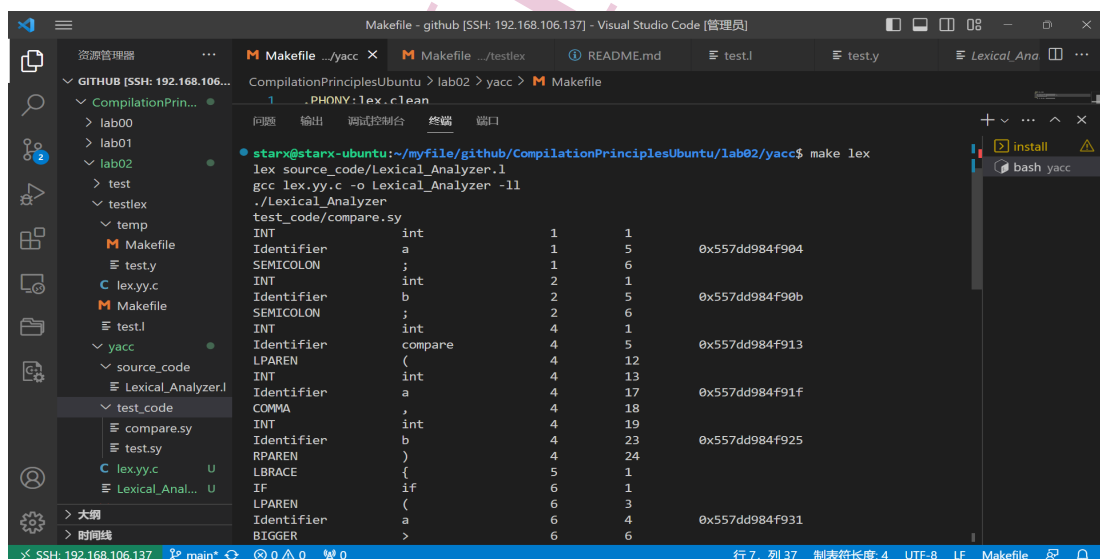


图 2: 测试词法分析器 2

三、 汇编编程流程

通过下面 C 代码为例来介绍 arm 汇编编程

逐列访问平凡算法

```
1  #include<stdio.h>
2  int a = 0;
3  int b = 0;
4  int max(int a, int b) {
5      if(a >= b) {
6          return a;
7      }
8      else {
9          return b;
10     }
11 }
12 int main() {
13     scanf("%d%d", &a, &b);
14     printf("max is: %d\n", max(a, b));
15     return 0;
16 }
```

1. 代码说明

这其中有一系列的指令是编译器指令，作用是告知编译器要如何编译，通常以 . 开始，其他指令则为汇编指令。

对每个函数的声明，观察可以发现一般首先为

```
1  .text
2  .global functionname
3  .type functionname, %function
```

即声明为代码段，将函数名添加到全局符号表中，声明类型为函数。

对全局变量与常量的声明，示例中已经给得比较详细，另外对于数组的使用，可以看到在声明时是毫无特殊的，而在使用时地址则为 `varname+offset`，其中偏移量即为数据类型大小乘个数。另外值得说明的是，在进行函数调用时，一般前四个函数参数使用 `r0-r3` 号寄存器进行传参，其余参数压入栈中进行传参，往往按照从右至左的顺序逐个压栈；在函数返回时，一般来讲默认将函数的返回值放到 `r0` 寄存器中。

我们可以发现汇编与 C 的不同：汇编的语言要素就是“标签”（指示地址）、寄存器移动/计算指令。尤其标签的灵活使用：上述汇编代码中利用 `_bridge` 标签，“桥接”了在 C 代码中隐性的全局变量的地址。

编程语言理论，建立在“组合”之上——组合意味着复用，意味着抽象。在理解汇编代码时，我们希望将“理解其抽象、其作为整体的语义”作为思考目标（操作系统课或许会接触一些乍一看难理解汇编代码）；在编写程序时，也常常是自顶向下的思维过程。

2. 代码解析

1. 定义目标架构：`.arch armv5t` 表示使用 ARMv5t 指令集。
2. 定义数据区：`.comm a, 4` 定义了一个全局变量 `a`，大小为 4 字节。`.comm b, 4` 定义了一个全局变量 `b`，大小为 4 字节。
3. 定义文本区：`.text` 表示接下来是代码区。

4. 对 rodata 数据区进行对齐: `.align 2` 表示下一个数据项的地址应该是对齐的, 且偏移量应为 2 字节。

5. 定义 rodata 数据区: `.section .rodata` 表示接下来是 rodata 数据区。

6. 定义常量字符串: `.ascii "%d %d 0"` 定义了一个包含两个整数格式化字符串的常量字符串, 用

于在后续的 `scanf` 和 `printf` 函数中使用。

7. 定义文本区: `.text` 表示接下来是代码区。

8. 定义全局变量 `max`: `.global max` 表示将变量 `max` 声明为全局变量。

9. `max` 函数实现:

- 保存寄存器: `strfp, [sp, # - 4]` 表示将寄存器 `fp` 压入栈中, 同时更新 `sp` 指针。
- 设置寄存器: `mov fp, sp` 表示将 `sp` 指针设置为当前栈指针。
- 分配栈空间: `sub sp, sp, #12` 表示为局部变量分配 12 字节的栈空间。
- 保存参数: `strr0, [fp, # - 8]` 表示将参数 `a` 压入栈中, 偏移量为-8。
- 保存参数: `strr1, [fp, # - 12]` 表示将参数 `b` 压入栈中, 偏移量为-12。
- 比较参数: `cmp r0, r1` 表示比较参数 `a` 和 `b` 的大小。
- 分支: `blt .L2` 表示如果 `a` 小于 `b`, 则跳转到标签 `.L2`。
- 返回最大值: `ldrr0, [fp, # - 8]` 表示将参数 `a` 从栈中弹出, 并赋值给寄存器 `r0`。
- 分支: `b .L3` 表示如果 `a` 大于等于 `b`, 则跳转到标签 `.L3`。
- 返回最大值: `ldrr0, [fp, # - 12]` 表示将参数 `b` 从栈中弹出, 并赋值给寄存器 `r0`。
- 恢复栈空间: `add sp, fp, #0` 表示将栈指针恢复到原始值。
- 恢复寄存器: `ldr fp, [sp], #4` 表示从栈中弹出 `fp` 寄存器, 并更新 `sp` 指针。
- 返回: `bx lr` 表示恢复 `pc` 指针, 并返回。

10. `main` 函数实现:

- 保存寄存器: `push fp, lr` 表示将寄存器 `fp` 和 `lr` 压入栈中, 并更新 `sp` 指针。
- 定义局部变量: `add fp, sp, #4` 表示为局部变量分配 4 字节的栈空间。
- 加载常量字符串: `ldr r2, __bridge` 表示将常量 `b` 的地址赋值给寄存器 `r2`。
- 加载常量字符串: `ldr r1, __bridge+4` 表示将常量 `a` 的地址赋值给寄存器 `r1`。
- 加载常量字符串: `ldr r0, __bridge+8` 表示将常量字符串 `__str0` 的地址赋值给寄存器 `r0`。
- 调用函数: `bl __isoc99_scanf` 表示调用 `scanf` 函数读取两个整数参数 `a` 和 `b`。
- 加载参数 `a`: `ldr r3, __bridge+4` 表示将常量 `a` 的地址赋值给寄存器 `r3`。
- 加载参数 `b`: `ldr r1, [r3]` 表示将 `r3` 地址处的值 (即 `a` 的值) 赋值给寄存器 `r1`。
- 调用函数: `bl max` 表示调用 `max` 函数计算 `a` 和 `b` (分别存在 `r0`, `r1`) 的最大值, 返回结果在 `r0` (默认返回 `r0`)。

• 保存结果: `mov r1, r0` 表示将结果赋值给寄存器 `r1`。

- 加载常量字符串: `ldr r0, __bridge+12` 表示将常量字符串地址 `__str1` 赋值给寄存器 `r0`。

• 调用函数: `bl printf` 表示调用 `printf` 函数输出, `r0` 为字符串参数, `r1` 为 `max` 结果, 两者作为函数调用的参数。

- 返回 0: `mov r0, #0` 表示将 0 赋值给寄存器 `r0`, 作为 `main` 函数的返回值。
- 恢复寄存器: `pop fp, pc` 表示将 `fp` 寄存器弹出并恢复, 并更新 `pc` 指针。

11. 定义 `__bridge` 的符号表项 • `.word b`: 这是一个引用, 表示符号 `b` 的 32 位 (`.word`) 地址。

- `.word a`: 表示符号 `a` 的地址。
- `.word __str0`: 表示符号 `__str0` 的地址。
- `.word __str1`: 表示符号 `__str1` 的地址。

• `.section .note.GNU-stack,"",progbits`: 这一行代码是用于 GNU Assembler (GAS) 的, 通常出现在汇编语言源文件中。它用于指定一个特殊的节 (section) `.note.GNU-stack`。这个节没有实际的代码或数据, 而是用于向链接器和操作系统传递有关程序堆栈的信息。具体来说, `.note.GNU-stack` 用于控制生成的可执行文件的堆栈是否是可执行的。在默认情况下, Linux 操作系统通常会将堆栈标记为不可执行, 这有助于阻止堆栈执行攻击。这是一个安全性措施, 旨在防止攻击者通过注入可执行代码到程序堆栈来攻击程序。

接下来是示例比较两个数大小的汇编编程代码, 经过实验可以成功运行

```

1      1 .arch armv5t
2      2 @ comm section save global variable without initialization
3      3 .comm a, 4 @global variables
4      4 .comm b, 4
5      5 .text
6      6 .align 2
7      7 @ rodata section save constant
8      8 .section .rodata
9      9 .align 2
10     10 __str0:
11     11 .ascii "%d %d\0" @\000 is also one representation for null character
12     12 .align 2
13     13 __str1:
14     14 .ascii "max is: %d\n"
15     15 @ text section code
16     16 .text
17     17 .align 2
18     18
19     19 .global max
20     20 max: @ function int max(int a, int b)
21     21 str fp, [sp, #-4]! @ pre-index mode, sp = sp - 4, push fp
22     22 mov fp, sp
23     23 sub sp, sp, #12 @ allocate space for local variable
24     24 str r0, [fp, #-8] @ r0 = [fp, #-8] = a
25     25 str r1, [fp, #-12] @ r1 = [fp, #-12] = b
26     26 cmp r0, r1
27     27 blt .L2
28     28 ldr r0, [fp, #-8]
29     29 b .L3
30     30 .L2:
31     31 ldr r0, [fp, #-12]
32     32 .L3:
33     33 add sp, fp, #0
34     34 ldr fp, [sp], #4
35     35 bx lr @ recover sp fp pc
36     36
37     37
38     38 .global main
39     39 main:
40     40 push {fp, lr}

```

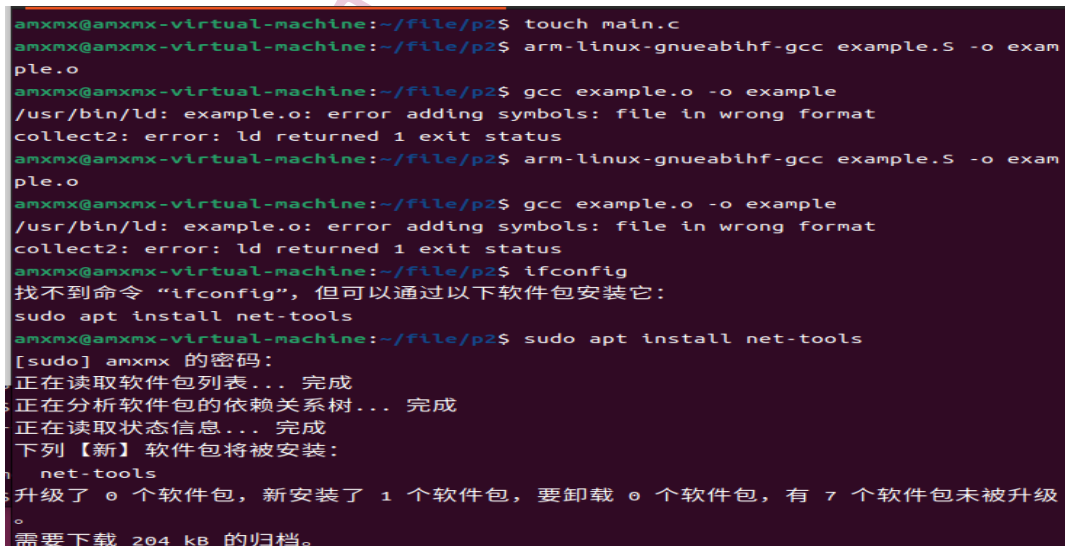
```

41 add fp, sp, #4
42 ldr r2, _bridge @ r2 = &b
43 ldr r1, _bridge+4 @ r1 = &a
44 ldr r0, _bridge+8 @ *r0 = "%d %d\000"
45 bl __isoc99_scanf @ scanf("%d %d", &a, &b)
46 ldr r3, _bridge+4 @ r3 = &a
47 ldr r0, [r3] @ r0 = a
48 ldr r3, _bridge @ r3 = &b
49 ldr r1, [r3] @ r1 = b
50 bl max
51 mov r1, r0 @ r1 = r0
52 ldr r0, _bridge+12 @ *r0 = "max is: %d\0"
53 bl printf @ printf("max is: %d", max(a, b));
54 mov r0, #0
55 pop {fp, pc} @ return 0
56
57 _bridge:
58 .word b
59 .word a
60 .word _str0
61 .word _str1
62
63 .section .note.GNU-stack,"",%progbits @ do you know what's the use of this
    :-)

```

(一) 验证具体操作

首先我们需要安装 nettools



```

amxxmx@amxxmx-virtual-machine:~/file/p2$ touch main.c
amxxmx@amxxmx-virtual-machine:~/file/p2$ arm-linux-gnueabi-gcc example.S -o example.o
amxxmx@amxxmx-virtual-machine:~/file/p2$ gcc example.o -o example
/usr/bin/ld: example.o: error adding symbols: file in wrong format
collect2: error: ld returned 1 exit status
amxxmx@amxxmx-virtual-machine:~/file/p2$ arm-linux-gnueabi-gcc example.S -o example.o
amxxmx@amxxmx-virtual-machine:~/file/p2$ gcc example.o -o example
/usr/bin/ld: example.o: error adding symbols: file in wrong format
collect2: error: ld returned 1 exit status
amxxmx@amxxmx-virtual-machine:~/file/p2$ ifconfig
找不到命令“ifconfig”，但可以通过以下软件包安装它：
sudo apt install net-tools
amxxmx@amxxmx-virtual-machine:~/file/p2$ sudo apt install net-tools
[sudo] amxxmx 的密码：
正在读取软件包列表... 完成
正在分析软件包的依赖关系树... 完成
正在读取状态信息... 完成
下列【新】软件包将被安装：
net-tools
升级了 0 个软件包，新安装了 1 个软件包，要卸载 0 个软件包，有 7 个软件包未被升级。
需要下载 204 kB 的归档。

```

图 3: nettools 的安装

接下来我们需要安装 openssh-server 这个插件

```

amxmx@amxmx-virtual-machine:~$ sudo apt-get install openssh-server
[sudo] amxmx 的密码:
正在读取软件包列表... 完成
正在分析软件包的依赖关系树... 完成
正在读取状态信息... 完成
将会同时安装下列软件:
  ncurses-term openssh-sftp-server ssh-import-id
建议安装:
  molly-guard monkeysphere ssh-askpass
下列【新】软件包将被安装:
  ncurses-term openssh-server openssh-sftp-server ssh-import-id
升级了 0 个软件包, 新安装了 4 个软件包, 要卸载 0 个软件包, 有 7 个软件包未被升级。
需要下载 751 kB 的归档。
解压缩后会消耗 6,046 kB 的额外空间。
您希望继续执行吗? [Y/n] Y
获取:1 http://mirrors.tuna.tsinghua.edu.cn/ubuntu jammy-updates/main amd64 opens
sh-sftp-server amd64 1:8.9p1-3ubuntu0.4 [38.7 kB]
获取:2 http://mirrors.tuna.tsinghua.edu.cn/ubuntu jammy-updates/main amd64 opens
sh-server amd64 1:8.9p1-3ubuntu0.4 [434 kB]
获取:3 http://mirrors.tuna.tsinghua.edu.cn/ubuntu jammy-updates/main amd64 ncurs
es-term all 6.3-2ubuntu0.1 [267 kB]
获取:4 http://mirrors.tuna.tsinghua.edu.cn/ubuntu jammy/main amd64 ssh-import-id
all 5.11-0ubuntu1 [10.1 kB]

```

图 4: 安装 openssh

之后我们输入下面的代码激活 ssh

```

1  sudo service ssh --full-restart
2  sudo systemctl enable ssh

```

```

3072 SHA256:9bRed/YwdiHcZTo6nInCowCgVhJEpxAx2mLPDnISa6s root@amxmx-virtual-machi
ne (RSA)
Creating SSH2 ECDSA key; this may take some time ...
256 SHA256:LWcLZBnnL0qvCqj0wWS8D+w1mz0qyT4N4BJ3F/WsFU4 root@amxmx-virtual-machin
e (ECDSA)
Creating SSH2 ED25519 key; this may take some time ...
256 SHA256:STQ2xT+VvTMJiWf8KdM7NCjy8baN15vW2YqQ4h7tf6g root@amxmx-virtual-machin
e (ED25519)
Created symlink /etc/systemd/system/ssh.service → /lib/systemd/system/ssh.servi
ce.
Created symlink /etc/systemd/system/multi-user.target.wants/ssh.service → /lib/s
ystemd/system/ssh.service.
rescue-ssh.target is a disabled or a static unit, not starting it.
ssh.socket is a disabled or a static unit, not starting it.
正在设置 ssh-import-id (5.11-0ubuntu1) ...
正在设置 ncurses-term (6.3-2ubuntu0.1) ...
正在处理用于 man-db (2.10.2-1) 的触发器 ...
正在处理用于 ufw (0.36.1-4ubuntu0.1) 的触发器 ...
amxmx@amxmx-virtual-machine:~$ sudo service ssh --full-restart
amxmx@amxmx-virtual-machine:~$ sudo systemctl enable ssh
Synchronizing state of ssh.service with SysV service script with /lib/systemd/sy
stemd-sysv-install.
Executing: /lib/systemd/systemd-sysv-install enable ssh
amxmx@amxmx-virtual-machine:~$

```

图 5: 激活 ssh

7

vscode 验证链接 我们在编辑实验所需代码的过程当中, 可以通过 vscode 远程连接的方法, 可以方便我们对实验当中代码编写以及格式的保存。

首先我们需要查询目标虚拟机的 ip 地址

```

amxmx@amxmx-virtual-machine:~/file/p2$ ifconfig
ens33: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.130.133 netmask 255.255.255.0 broadcast 192.168.130.255
    inet6 fe80::3101:aef4:51bb:48e6 prefixlen 64 scopeid 0x20<link>
    ether 00:0c:29:9e:95:62 txqueuelen 1000 (以太网)
    RX packets 1153 bytes 383462 (383.4 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 1066 bytes 99607 (99.6 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
    device interrupt 19 base 0x2000

lo: flags=73<UP,LOOPBACK,RUNNING> mtu 65536
    inet 127.0.0.1 netmask 255.0.0.0
    inet6 ::1 prefixlen 128 scopeid 0x10<host>
    loop txqueuelen 1000 (本地环回)
    RX packets 298 bytes 30462 (30.4 KB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 298 bytes 30462 (30.4 KB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

```

图 6: 虚拟机的 ip 地址

之后我们通过 vscode 输入目标的 ip 地址和主机名称之后进行远程连接

- 首先打开 vscode, 找到 Extensions, 搜索 Remote, 下载 Remote-Developoment 插件, 会自动安装其他的 Remote 插件, 其中会包含 Remote-SSH 插件
- 进入设置, 搜索 ssh, 找到并选中拓展中的 Remote-SSH 中的 Show Login Terminal 选项, 因为在连接的时候, 终端会让你输入 yes 或者密码等:
- 接着输入自己的地址信息:

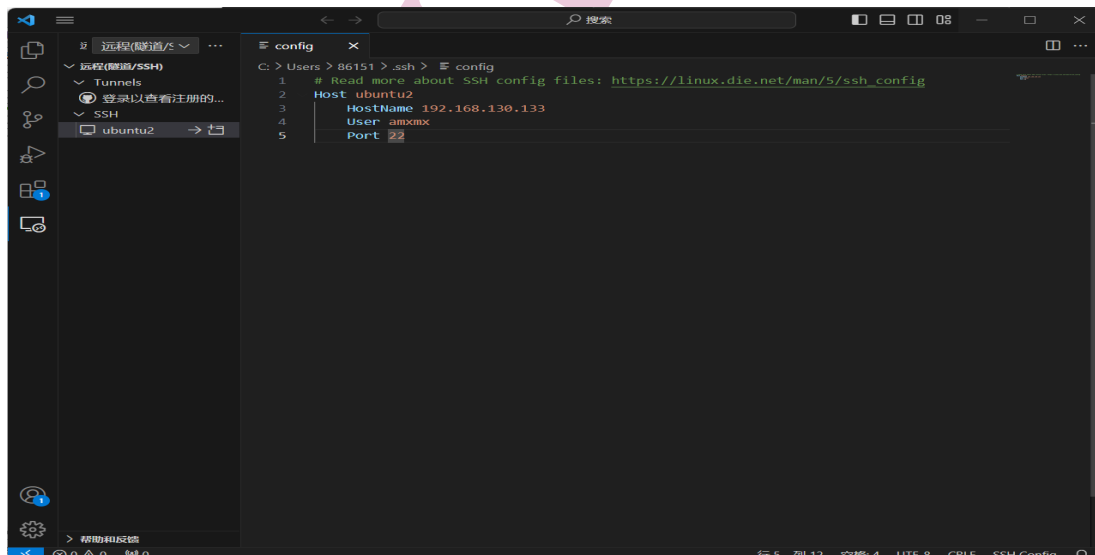


图 7: 输入地址即可连接

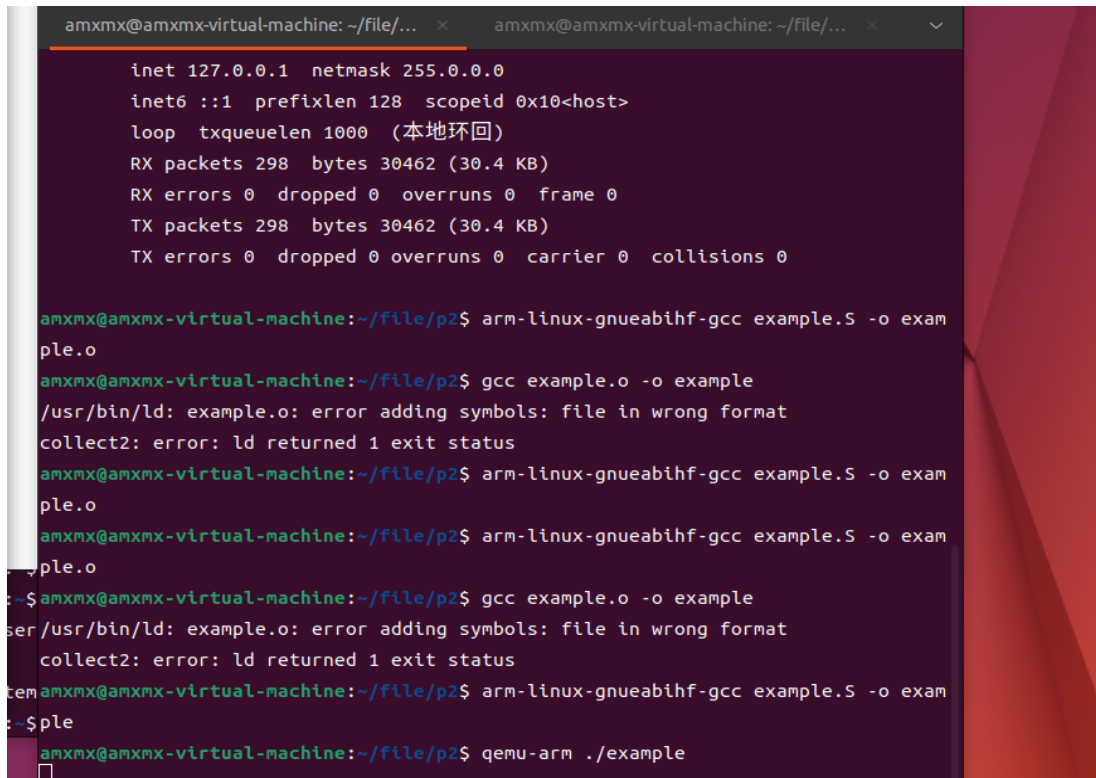
接下来我们创建相应的 main.S 文件, 并且输入测试代码

之后我们输入下列代码成功链接为可执行文件, 并且开始进行调试工作。

```

1 arm-linux-gnueabi-gcc main.S -o main
2 qemu-arm ./main

```

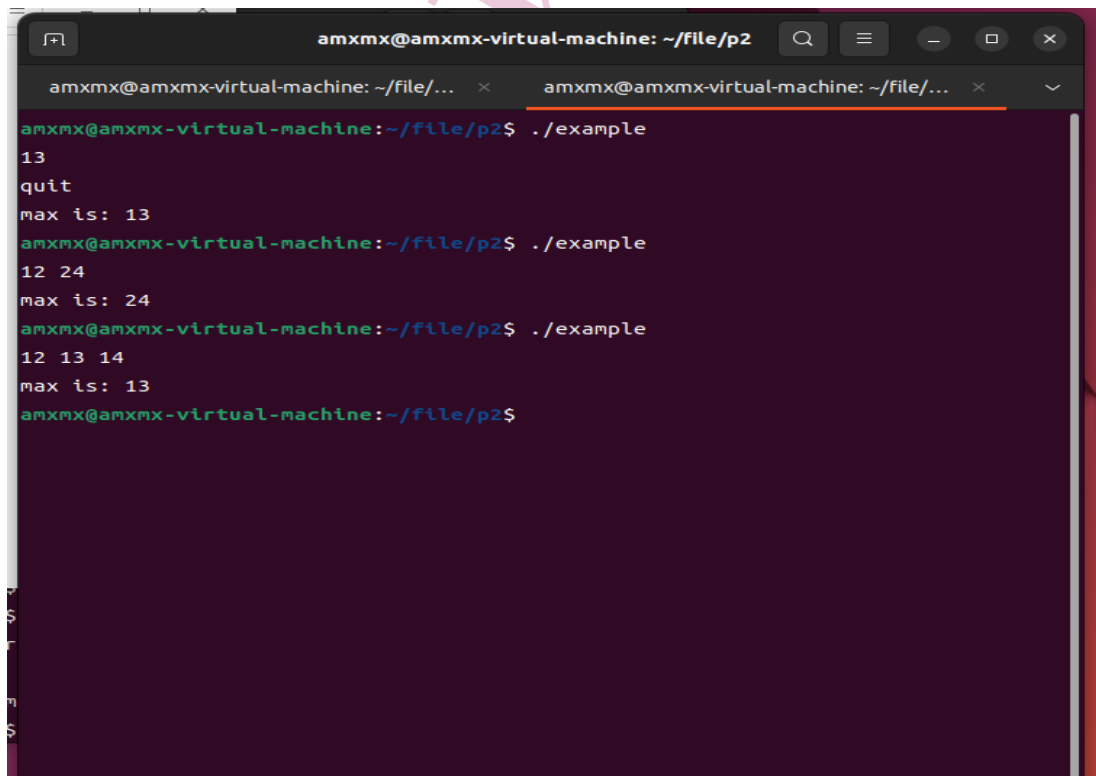



```
amxmx@amxmx-virtual-machine: ~/file/... x amxmx@amxmx-virtual-machine: ~/file/... x
inet 127.0.0.1 netmask 255.0.0.0
inet6 ::1 prefixlen 128 scopeid 0x10<host>
loop txqueuelen 1000 (本地环回)
RX packets 298 bytes 30462 (30.4 KB)
RX errors 0 dropped 0 overruns 0 frame 0
TX packets 298 bytes 30462 (30.4 KB)
TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0

amxmx@amxmx-virtual-machine:~/file/p2$ arm-linux-gnueabi-gcc example.S -o exam
ple.o
amxmx@amxmx-virtual-machine:~/file/p2$ gcc example.o -o example
/usr/bin/ld: example.o: error adding symbols: file in wrong format
collect2: error: ld returned 1 exit status
amxmx@amxmx-virtual-machine:~/file/p2$ arm-linux-gnueabi-gcc example.S -o exam
ple.o
amxmx@amxmx-virtual-machine:~/file/p2$ arm-linux-gnueabi-gcc example.S -o exam
ple.o
amxmx@amxmx-virtual-machine:~/file/p2$ gcc example.o -o example
/usr/bin/ld: example.o: error adding symbols: file in wrong format
collect2: error: ld returned 1 exit status
amxmx@amxmx-virtual-machine:~/file/p2$ arm-linux-gnueabi-gcc example.S -o exam
ple.o
amxmx@amxmx-virtual-machine:~/file/p2$ qemu-arm ./example
```

图 8: Caption

接下来我们打开测试样例及进行调试，可以得到一个准确的答案



```
amxmx@amxmx-virtual-machine: ~/file/p2
amxmx@amxmx-virtual-machine: ~/file/... x amxmx@amxmx-virtual-machine: ~/file/... x
amxmx@amxmx-virtual-machine:~/file/p2$ ./example
13
quit
max is: 13
amxmx@amxmx-virtual-machine:~/file/p2$ ./example
12 24
max is: 24
amxmx@amxmx-virtual-machine:~/file/p2$ ./example
12 13 14
max is: 13
amxmx@amxmx-virtual-machine:~/file/p2$
```

图 9: 对测试样例进行测试

我们还可以通过 vscode 远程对该程序的代码进行测试

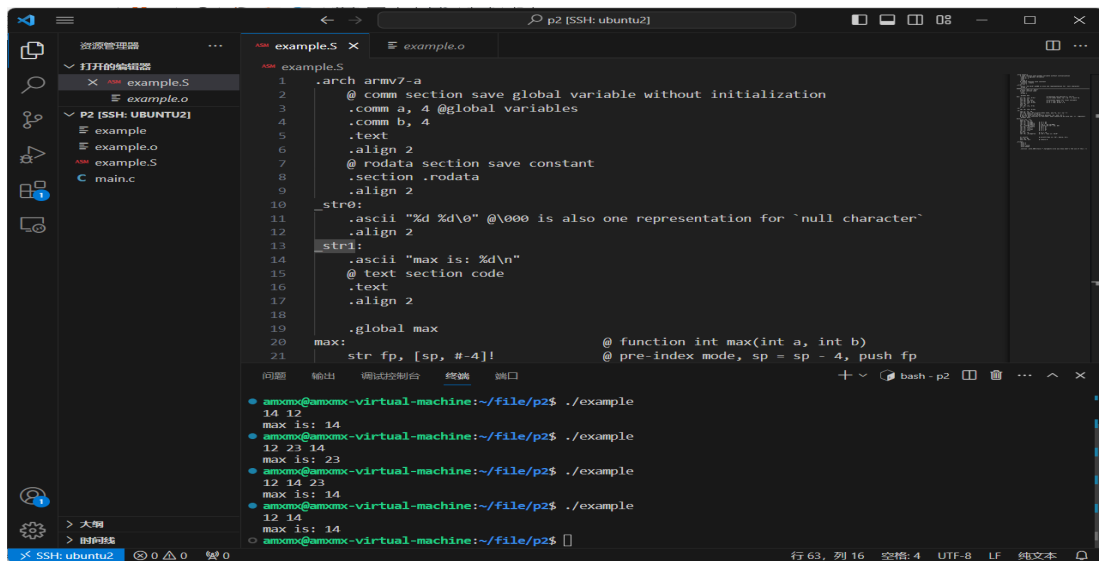


图 10: 在 vscode 远程连接验证该程序

之后我们将下列代码编写进 makefile 当中

```
1 .PHONY: test , clean
2 test:
3     arm-linux-gnueabi-gcc main.S -o main
4     qemu-arm ./main
5 clean:
6     rm main
```

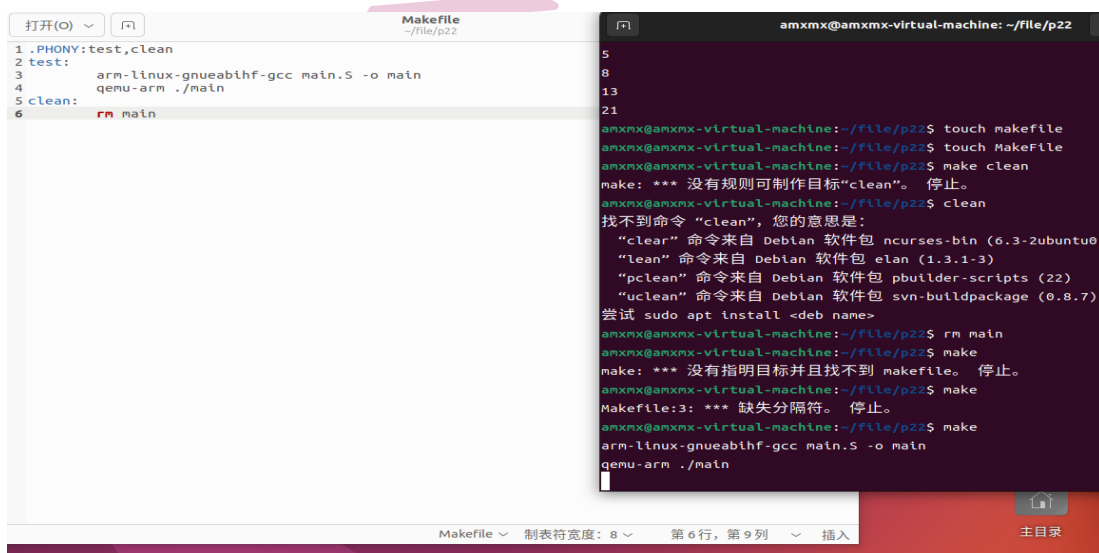


图 11: makefile

接下来我们还可以对斐波那契代码以及求平方的代码进行验证，代码如下：

```
1 .arch armv7-a @处理器架构
```

```
2      .arm
3      @r0是格式化字符串, r1是对应的printf对应的第二个参数
4      @代码段
5      @主函数
6      .text @代码段
7      .global main
8      .type main, %function
9      main:
10     push {fp, lr} @将fp的当前值保存在堆栈上, 然后将sp寄存器的值保存在fp中, lr中
        存储的是pc的保存在lr中
11     sub sp, sp, #4 @在栈中开辟一块大小为4的内存地址, 用于存储即将输入的数据
12     ldr r0, =_cin
13     mov r1, sp @将sp的值传输给r1寄存器, 使scanf传入的值存储在栈上, 即栈顶的值是
        n
14     bl scanf
15     ldr r6, [sp, #0] @取出sp指针指向的地址中的内容, 即栈顶中的内容 (输入的n的
        值)
16     add sp, sp, #4 @恢复栈顶, 释放内存空间
17
18     @测试是否写入
19     @ldr r0, =_bridge3
20     @mov r1, r2
21     @bl printf
22
23     mov r4, #0 @a = 0
24     mov r5, #1 @b = 1
25     mov r7, #1 @i = 1
26     @r4中存a的值, r5中存b的值, r7中存i的值, r6中存n的值
27     ldr r0, =_bridge
28     mov r1, r4 @将r4中的值即a的值赋予r1
29     bl printf @打印a的值
30     ldr r0, =_bridge2
31     mov r1, r5 @将r5中的值即b的值赋予r1
32     bl printf @打印b的值
33     ldr r0, =_bridge4
34     bl printf
35
36     @输出进行调试
37     @ldr r0, =_bridge3
38     @mov r1, r6
39     @bl printf
40     @ldr r0, =_bridge3
41     @mov r1, r7
42     @bl printf
43
44     Loop:
45     @输出进行调试
46     @ldr r0, =_bridge4
```

```
47  @bl printf
48  @ldr r0, =_bridge3
49  @mov r1, r6
50      @bl printf
51  @ldr r0, =_bridge3
52      @mov r1, r7
53      @bl printf
54
55  cmp r6, r7
56  ble RETURN @比较r7和r6（即i和n）的大小用于跳转
57  mov r8, r5 @t = b @r8为临时变量的寄存器
58  add r5, r5, r4 @b = a + b
59  ldr r0, =_bridge3
60  mov r1, r5 @将r5中的值即b的值赋予r1
61  bl printf @cout << b << endl;
62  mov r4, r8 @a = t
63  add r7, r7, #1 @i = i + 1
64  b Loop
65
66 RETURN:
67     pop {fp, lr} @上下文切换
68     bx lr @return 0
69 .data @数据段
70 _cin:
71     .asciz "%d"
72
73 _bridge:
74     .asciz "a:%d\n"
75
76 _bridge2:
77     .asciz "b:%d\n"
78
79 _bridge3:
80     .asciz "%d\n"
81
82 _bridge4:
83     .asciz "We are going to loop now! \n"
84
85 .section .note.GNU-stack,"",%progbits
```

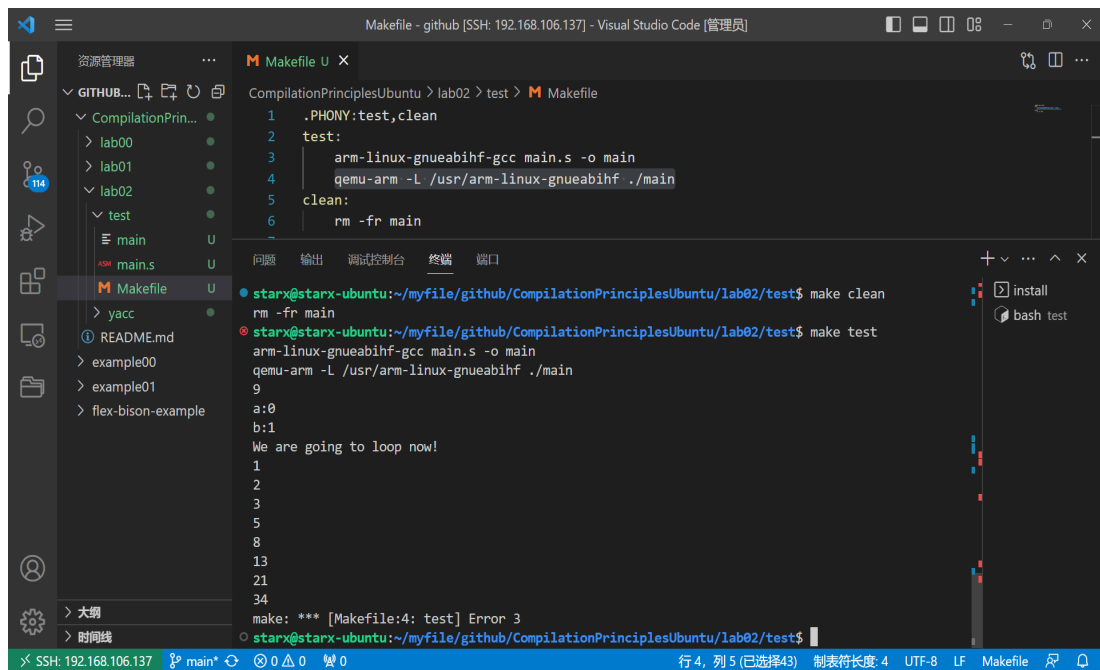


图 12: 在 vscode 远程连接验证该程序

另外我们还编写了求平方的代码，代码以及验证结果如下图所示：

```

1  .arch armv7-a
2  .arm
3
4  .text @代码段
5  .global square
6  square: @function int square(int a)
7      str fp, [sp, #-4]! @pre-index mode, sp = sp -4, push fp
8      mov fp, sp
9      sub sp, sp, #8 @为本地变量开辟空间
10     str r0, [fp, #-8] @r0 = [fp, #-8] = a
11     mul r1, r0, r0
12     mov r0, r1
13     add sp, fp, #0
14     ldr fp, [sp], #4
15     bx lr
16
17     .text @代码段
18     .global main
19     .type main, %function
20 main:
21     push {fp, lr}
22     sub sp, sp, #4
23     ldr r0, =_cin
24     mov r1, sp
25     bl scanf
26     ldr r0, [sp, #0] @取出输入的内容放入r0中

```

```
27 add sp, sp, #4
28 bl square
29 mov r1, r0
30 ldr r0, =_cout
31 bl printf
32 mov r0, #0
33 pop {fp, lr}
34 bx lr
35
36 .data @数据段
37 _cin:
38 .asciz "%d"
39
40 _cout:
41 .asciz "%d\n"
42
43 .section .note.GNU-stack,"",%progbits
```

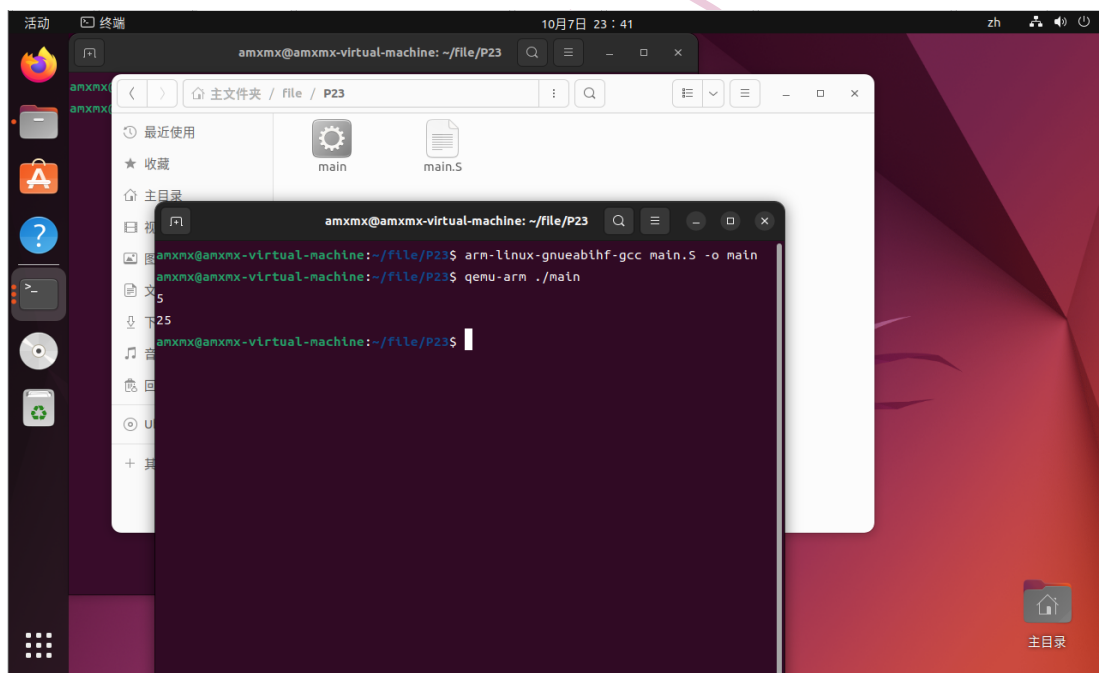


图 13: 在 vscode 远程连接验证该程序

四、 总结

本次实验初步了解了编译器的工作原理, 了解了 LLVM 的工作原理, 并可以自主使用 LLVM IR 进行自己的编程, 为后续的编译器实验和操作打下了坚实基础。

实验分工:

艾明旭, 求平方以及斐波那契数列的编程实现, sys 框架的研究

张刘明: 斐波那契数列的测试, sys 编译器的编写

github 代码库地址:

<https://github.com/newstarming/CompilationPrinciplesUbuntu/tree/main/lab02>

参考文献

- [1] 杨侯哲, 李煦阳, 杨科迪, 费迪, 周辰霏, 谢子涵, and 杨科迪. 编译器开发环境部署. 2023.
- [2] 杨侯哲, 李煦阳, 孙一丁, 李世阳, 杨科迪, 周辰霏, 尧泽斌, 时浩铭, 贺祎昕, 张书睿. 预备工作 1——了解编译器及 llvm ir 编程. 2023.
- [3] 杨侯哲李煦阳费迪张书睿贺祎昕预备工作 2——定义你的编译器 & 汇编编程