前言

本指导书为 C0 指导书，阅读本指导书前，请确保你已经顺利完成了 mini 实验并且对编译器的结构和编译过程有了基本认识。

这次实验要实现的编译器，根据你选择的编程语言、模型不同，代码量可能会在 2000--5000 行之间。作为参考，使用 Rust 编写的参考实现的代码量约为 3500 行。

本指导书仍然还是 Beta 版本。如果你在书中发现了（包括但不限于）以下问题，欢迎积极联系助教，或者提 Issue/PR 修正，可能会有加分哦 ouo：

- 难以理解的表述
- 逻辑/知识错误
- 代码错误
- 前后矛盾
- 代码不对应/过时
- 任何可以优化的部分

以上，祝各位同学编译愉快！

—— 你们的魔鬼助教（笑）

C0 编译实验安排及要求

安排

时间: 2020-11-13（第 10 周） -- 2021-01-03（第 17 周）（作业提交截止到考期结束）

提交方式: 于 https://oj.karenia.cc 在线评测

要求

1. 以个人为单位进行开发，不得多人合作完成；
2. 利用实验课和课余时间完成；
3. 编程语言、实现方式自定；
4. 输入语言为 C0，输出语言为 虚拟机代码；
5. 存在选做部分（扩展 C0），实现选做部分可以获得更高分数；
6. 程序具体要求见本指导书其余内容。

成绩评定

学生进行在线评测，取提交截止前最后一次测试的结果，所有通过的测试点权重之和为学生最终成绩。

参考书

龙书、虎书、狼书等

编译过程概述

这次我们使用的编译目标是 C0，一种简化了 C 语言语义和编译过程、同时魔改了语法的玩具编程语言。

编译过程

整个 C0 工具链分为两个部分，分别是你接下来需要写的 **编译器** 和我们提供的 **虚拟机**。整个编译流程如下：

- 编译器（你要写的）
  - 读入 c0 源代码
  - 输出 o0 二进制代码
- 虚拟机
  - 读入 o0 二进制代码
  - 解释执行代码
  - 输出运行结果

你的编译器至少能通过某种**命令行接口**读入一个 c0 文件，并将编译出的 o0 文件输出到另一个文件中。接口的实际特征我们不做规定，需要你自己定义并写在评测配置中，具体见 评测要求。

o0 二进制代码的定义见 这里。

我们不限制你编写编译器使用的语言，且允许复用 miniplc0 实验中的代码。

如果你决定编译到其他目标（自己设计的指令集、LLVM IR、Java Bitcode、物理 CPU 指令集等），请联系助教进行单独手动评测。由于需要单独评测，你不太可能因此获得额外的加分。

编译目标

我们编译的目标是本课程自行设计的 `navm` 虚拟机使用的 `o0` 代码。有关 `navm` 虚拟机的设计、结构参见对 navm 虚拟机的介绍。

比如如果你编译这个文件：

```
fn foo(i: int) -> int {
    return -i;
}

fn main() -> void {
    putint(foo(-123456));
}
```

你得到的结果应该类似于这个（`o0` 格式没有规定相应的文字形式，以下是官方参考实现 `natrium` 和 `navm` 使用的输出格式）：

```
static: 66 6F 6F (`foo`)

static: 70 75 74 69 6E 74 (`putint`)

static: 6D 61 69 6E (`main`)

static: 5F 73 74 61 72 74 (`_start`)


fn [3] 0 0 -> 0 {
    0: StackAlloc(0)
    1: Call(2)
}

fn [0] 0 1 -> 1 {
    0: ArgA(0)
    1: ArgA(1)
    2: Load64
    3: NegI
    4: Store64
    5: Ret
}

fn [2] 0 0 -> 0 {
    0: StackAlloc(0)
    1: StackAlloc(1)
    2: Push(123456)
    3: NegI
    4: Call(1)
    5: CallName(1)
    6: Ret
}
```

语法表示说明

本文介绍的是一种 EBNF 的变体，用于描述字符写成的编程语言语法。这个变体将每一个非终结符的产生式都表示成了一个正则表达式，以此来使得语言定义写起来更加精炼。

本变体的格式大量参考了 Rust 的语法描述语言。

非终结符

非终结符由一段由字母、数字或下划线组成的字符串表示，例如 `expr`，`if_stmt`。其中，字母全大写的字符串，如 `IDENT`，表示这个非终结符是一个单词（token）。

终结符

终结符包括字符串、字符范围和正则表达式。其中：

- 字符串由双引号 `"` 或单引号 `'` 包括，表示等同于内容的字符序列，如 `"while"`、`"y"`、`"+="`。
- 字符范围由方括号包括，内部填写包括的字符或字符范围，表示符合要求的任一字符。其中，用短横线 `-` 连接的两个字符表示字符编码中介于两个字符值之间（含端点）的字符。如 `[abcde]`（等同于 `[a-e]`）、`[_0-9a-zA-Z]`。

  以 `^` 开头的字符范围表示不在范围内的任一字符，如 `[^abc]`、`[^A-Z]`。
- 正则表达式由 `regex( )` 包括，内部是正则表达式，如 `regex(\w+)`、`regex(\w+://(\w+\.)+\.com)`

字符范围和字符串遵循 C 风格的转义序列，即使用反斜线 `\\` 后跟随字符组成。如果某个转义序列没有含义，则表示反斜线后的字符本身。

产生式

产生式左侧是非终结符，右侧是一个由终结符和非终结符组成的正则表达式，中间以箭头 `->` 连接。一个产生式占一行和之后缩进的所有行，如：

```
sign -> [+-]
fractional_part -> "." dec_number
my_expression ->
    this_is_a_very_long_keyword this_is_a_very_long_expression ";"
```

当一个非终结符有多个产生式时，右侧的不同产生式用竖线 `|` 分隔，表示“或”的关系。当产生式过长时，也可另起一行缩进书写。如：

```
my_keyword -> "fn" | "class"
binary_operator ->
      "+" | "-" | "*" | "/"
    | "=" | ">" | "<" | ">=" | "<=" | "==" | "!="
```

和正则表达式相同，可以省略的表达式用问号 `?` 修饰，如 `"public"? "class" identifier`；

可以重复一次或多次的表达式用加号 `+` 修饰，如 `[0-9a-f]+`；

可以重复零次或多次的表达式用星号 `*` 修饰，如 `[1-9] [0-9]*`；

可以重复指定次数次的表达式后面用大括号括起来数字修饰，其中：

- `{m}` 表示指定重复 `m` 次；

- `{m,n}` 表示重复 `m` 到 `n` 次；
- `{m,}` 表示重复 `m` 次及以上；
- `{,n}` 表示重复 0 到 `n` 次；

将一系列符号用小括号 `(` `)` 包括起来表示分组，分组内的符号作为一个整体看待，如 `(item ",")+` 。

c0 语法说明

c0 是一个用于编译原理课程的微型语言。c0 提供的功能类似于 C，但是为了减少编译器实现的压力（减少前瞻和/或回溯），在语言风格上大量参考了 Rust 的设计。请注意，这个语言**并不是**对 Rust 语言的简化。

语法定义

以下是 c0 语言的全部语法定义，包括所有扩展语法。

```
// # 单词

// ## 关键字
FN_KW     -> 'fn'
LET_KW    -> 'let'
CONST_KW  -> 'const'
AS_KW     -> 'as'
WHILE_KW  -> 'while'
IF_KW     -> 'if'
ELSE_KW   -> 'else'
RETURN_KW -> 'return'
BREAK_KW  -> 'break'
CONTINUE_KW -> 'continue'

// ## 字面量
digit -> [0-9]
UINT_LITERAL -> digit+
FLOAT_LITERAL -> digit+ '.' digit+ ([eE] digit+)?

escape_sequence -> '\' [\\"'nrt]
string_regular_char -> [^"\\]
STRING_LITERAL -> '"' (string_regular_char | escape_sequence)* '"'

char_regular_char -> [^'\\]
CHAR_LITERAL -> '\'' (char_regular_char | escape_sequence)* '\''

// ## 标识符
IDENT -> [_a-zA-Z] [_a-zA-Z0-9]*

// ## 符号
PLUS      -> '+'
MINUS     -> '-'
MUL       -> '*'
DIV       -> '/'
ASSIGN    -> '='
EQ        -> '=='
NEQ       -> '!='
LT        -> '<'
GT        -> '>'
LE        -> '<='
GE        -> '>='
L_PAREN   -> '('
R_PAREN   -> ')'
L_BRACE   -> '{'
R_BRACE   -> '}'
ARROW     -> '->'
COMMA     -> ','
COLON     -> ':'
SEMICOLON -> ';'

// ## 注释
COMMENT -> '//' regex(.*) '\n'

// # 表达式
expr ->
      operator_expr
    | negate_expr
    | assign_expr
    | as_expr
    | call_expr
    | literal_expr
```

```
    | ident_expr

binary_operator -> '+' | '-' | '*' | '/' | '==' | '!=' | '<' | '>' | '<=' | '>='
operator_expr -> expr binary_operator expr

negate_expr -> '-' expr

assign_expr -> l_expr '=' expr

as_expr -> expr 'as' ty

call_param_list -> expr (',' expr)*
call_expr -> IDENT '(' call_param_list? ')'

literal_expr -> UINT_LITERAL | FLOAT_LITERAL | STRING_LITERAL | CHAR_LITERAL

ident_expr -> IDENT

// ## 左值表达式
l_expr -> IDENT

// ## 类型
ty -> IDENT

// # 语句
stmt ->
      expr_stmt
    | decl_stmt
    | if_stmt
    | while_stmt
    | break_stmt
    | continue_stmt
    | return_stmt
    | block_stmt
    | empty_stmt

expr_stmt -> expr ';'

let_decl_stmt -> 'let' IDENT ':' ty ('=' expr)? ';'
const_decl_stmt -> 'const' IDENT ':' ty '=' expr ';'
decl_stmt -> let_decl_stmt | const_decl_stmt

if_stmt -> 'if' expr block_stmt ('else' 'if' expr block_stmt)* ('else' block_stmt)?

while_stmt -> 'while' expr block_stmt

break_stmt -> 'break' ';'

continue_stmt -> 'continue' ';'

return_stmt -> 'return' expr? ';'

block_stmt -> '{' stmt* '}'

empty_stmt -> ';'

// # 函数
function_param -> IDENT ':' ty
function_param_list -> function_param (',' function_param)*
function -> 'fn' IDENT '(' function_param_list? ')' '->' ty block_stmt
```

```
// # 程序
item -> function | decl_stmt
program -> item*
```

其中，表达式中运算符的优先级从高到低为：

运算符	结合性
函数调用	-
前置 `-`	-
`as`	-
`*` `/`	左到右
`+` `-`	左到右
`>` `<` `>=` `<=` `==` `!=`	左到右
`=`	右到左

语法参考

以下是一些符合语法规范的程序。

```
fn fib(x: int) -> int {
    if x<=1 {
        return 1;
    }
    let result: int = fib(x - 1);
    result = result + fib(x - 2);
    return result;
}

fn main() -> int {
    let i: int = 0;
    let j: int;
    j = getint();
    while i < j {
        putint(i);
        putchar(32);
        putint(fib(i));
        putln();
        i = i + 1;
    }
    return 0;
}
```

单词 (Token)

单词是词法分析的结果。

关键字

```
FN_KW     -> 'fn'
LET_KW    -> 'let'
CONST_KW  -> 'const'
AS_KW     -> 'as'
WHILE_KW  -> 'while'
IF_KW     -> 'if'
ELSE_KW   -> 'else'
RETURN_KW -> 'return'

// 这两个是扩展 c0 的
BREAK_KW  -> 'break'
CONTINUE_KW -> 'continue'
```

c0 有 8 个关键字。扩展 c0 增加了 2 个关键字。

字面量

```
digit -> [0-9]
UINT_LITERAL -> digit+

escape_sequence -> '\' [\\"'nrt]
string_regular_char -> [^"\\]
STRING_LITERAL -> '"' (string_regular_char | escape_sequence)* '"'

// 扩展 c0
FLOAT_LITERAL -> digit+ '.' digit+ ([eE] digit+)?

char_regular_char -> [^'\\]
CHAR_LITERAL -> '\'' (char_regular_char | escape_sequence)* '\''
```

基础 c0 有两种字面量，分别是 *无符号整数* 和 *字符串常量*。扩展 c0 增加了 *浮点数常量* 和 *字符常量*。

语义约束：

- 字符串字面量中的字符可以是 ASCII 中除了双引号 `"`、反斜线 `\\`、空白符 `\r` `\n` `\t` 以外的任何字符。转义序列可以是 `\'`、`\"`、`\\`、`\n`、`\t`、`\r`，含义与 C 中的对应序列相同。

> UB: 对于无符号整数和浮点数常量超出相应数据类型表示范围的情况我们不做规定。你可以选择报错也可以选择无视。

标识符

```
IDENT -> [_a-zA-Z] [_a-zA-Z0-9]*
```

c0 的标识符由下划线或字母开头，后面可以接零或多个下划线、字母或数字。标识符不能和关键字重复。

运算符

```
PLUS      -> '+'
MINUS     -> '-'
MUL       -> '*'
DIV       -> '/'
ASSIGN    -> '='
EQ        -> '=='
NEQ       -> '!='
LT        -> '<'
GT        -> '>'
LE        -> '<='
GE        -> '>='
L_PAREN   -> '('
R_PAREN   -> ')'
L_BRACE   -> '{'
R_BRACE   -> '}'
ARROW     -> '->'
COMMA     -> ','
COLON     -> ':'
SEMICOLON -> ';'
```

注释

注释是扩展 c0 内容，见 扩展 c0

```
COMMENT -> '//' regex(.*) '\n'
```

类型系统

基础类型

c0 有一个十分简单的类型系统。在基础 C0 中你会用到的类型有两种：

- 64 位有符号整数 `int`
- 空类型 `void`

扩展 C0 增加了一种类型：

- 64 位 IEEE-754 浮点数 `double`

类型表示

```
ty -> IDENT
```

在 C0 中，用到类型的地方使用一个标识符表示。这个标识符的所有可能值就是上面列出的基础类型。填入其他值的情况应被视为编译错误。

关于布尔类型

比较运算符的运行结果是布尔类型。在 c0 中，我们并没有规定布尔类型的实际表示方式。在 navm 虚拟机中，所有非 0 的布尔值都被视为 `true`，而 0 被视为 `false`。

表达式

```
expr ->
      operator_expr
    | negate_expr
    | assign_expr
    | as_expr
    | call_expr
    | literal_expr
    | ident_expr
```

表达式是代码中运算的最小单位。在语法解析的时候，一个表达式会被展开成一棵树，称作表达式树。

运算符表达式

```
binary_operator -> '+' | '-' | '*' | '/' | '==' | '!=' | '<' | '>' | '<=' | '>='
operator_expr -> expr binary_operator expr
```

运算符表达式是中间由一个运算符分隔、两边是子表达式的表达式。r0 一共有 10 种双目运算符。它们分别是：

- 算数运算符 `+` `-` `*` `/`
- 比较运算符 `>` `<` `>=` `<=` `==` `!=`

每个运算符的两侧必须是相同类型的数据。各运算符含义如下：

运算符	含义	参数类型	结果类型	结合性
`+`	将左右两侧相加	数值	与参数相同	左到右
`-`	左侧减去右侧	数值	与参数相同	左到右
`*`	将左右两侧相乘	数值	与参数相同	左到右
`/`	左侧除以右侧	数值	与参数相同	左到右
`>`	如果左侧大于右侧则为真	数值	布尔*	左到右
`<`	如果左侧小于右侧则为真	数值	布尔*	左到右
`>=`	如果左侧大于等于右侧则为真	数值	布尔*	左到右
`<=`	如果左侧小于等于右侧则为真	数值	布尔*	左到右
`==`	如果左侧等于右侧则为真	数值	布尔*	左到右
`!=`	如果左侧不等于右侧则为真	数值	布尔*	左到右

*** 关于布尔类型**

由于布尔类型的表达式只会出现在 `if` 和 `while` 语句的条件表达式中，我们不规定布尔类型的表现形式。在 `navm` 中，所有非 0 值均表示 `true`，0 表示 `false`。

取反表达式

```
negate_expr -> '-' expr
```

取反表达式是在表达式前添加负号组成的表达式。取反表达式的语义是将表达式转换成它的相反数。

赋值表达式

```
l_expr -> IDENT
assign_expr -> l_expr '=' expr
```

赋值表达式是由 *左值表达式*、*等号* `=`、*表达式* 组成的表达式。赋值表达式的值类型永远是 `void`（即不能被使用）。

左值表达式是一个局部或全局的变量名。

赋值表达式的语义是将右侧表达式的计算结果赋给左侧表示的值。

类型转换表达式

```
as_expr -> expr 'as' ty
```

类型转换表达式是由 *表达式*、*关键字* `as`、*类型* 组成的表达式。类型转换表达式的语义是将左侧表达式表示的值转换成右侧类型表示的值。

在 C0 实验中只会涉及到整数 `int` 和浮点数 `double` 之间的互相转换。

函数调用表达式

```
call_param_list -> expr (',' expr)*
call_expr -> IDENT '(' call_param_list? ')'
```

函数调用表达式是由 *函数名* 和 *调用参数列表* 组成的表达式。函数调用表达式的语义是使用给出的参数调用函数名代表的函数。函数必须在调用前声明过（也就是说不存在先出现调用后出现声明的函数）。

特殊情况

标准库中的函数在调用前不需要声明，见 标准库文档。

字面量表达式

```
literal_expr -> UINT_LITERAL | FLOAT_LITERAL | STRING_LITERAL

digit -> [0-9]
UINT_LITERAL -> digit+
FLOAT_LITERAL -> digit* '.' digit+ ([eE] digit+)?

escape_sequence -> '\' [\\"'nrt]
string_regular_char -> [^"\\]
STRING_LITERAL -> '"' (string_regular_char | escape_sequence)* '"'
```

字面量表达式可以是一个无符号整数、浮点数或者字符串的字面量。*整数* 和 *浮点数字面量* 的语义就是用对应类型表示的字面量的值（64 位）；*字符串字面量* 只会在 `putstr` 调用中出现，语义是对应的全局常量的编号。

标识符表达式

```
ident_expr -> IDENT
```

标识符表达式是由标识符组成的表达式。其语义是标识符对应的局部或全局变量。标识符表达式的类型与标识符的类型相同。

语句

```
stmt ->
      expr_stmt
    | decl_stmt
    | if_stmt
    | while_stmt
    | return_stmt
    | block_stmt
    | empty_stmt
```

语句是函数的最小组成部分。

表达式语句

```
expr_stmt -> expr ';'
```

表达式语句由 *表达式* 后接分号组成。表达式如果有值，值将会被丢弃。

声明语句

```
let_decl_stmt -> 'let' IDENT ':' ty ('=' expr)? ';'
const_decl_stmt -> 'const' IDENT ':' ty '=' expr ';'
decl_stmt -> let_decl_stmt | const_decl_stmt
```

声明语句由 `let`（声明变量）或 `const`（声明常量）接 *标识符*、*类型* 和可选的 *初始化表达式* 组成。其中，常量声明语句必须有初始化表达式，而变量声明语句可以没有。

一个声明语句会在当前作用域中创建一个给定类型和标识符的变量或常量。声明语句有以下语义约束：

- 在同一作用域内，一个标识符只能由一个变量或常量使用。
- 变量或常量的类型不能为 `void`。
- 如果存在初始化表达式，其类型应当与变量声明时的类型相同。
- 常量只能被读取，不能被修改。

出现违反约束的声明语句是编译期错误。

> UB: 没有初始化的变量的值未定义。我们不规定对于使用未初始化变量的行为的处理方式，你可以选择忽略、提供默认值或者报错。

以下是一些可以通过编译的变量声明的例子：

```
let i: int;
let j: int = 1;
const k: double = 1.20;
```

以下是一些不能通过编译的变量声明的例子：

```
// 没有类型
let l = 1;
// 没有初始化
const m: int;
// 类型不匹配
let n: double = 3;
// 常量不能被修改
let p: double = 3.0;
p = 3.1415;
```

控制流语句

基础 C0 中有三种控制流语句，分别是 `if`、`while` 和 `return` 语句。

`if` 语句

```
if_stmt -> 'if' expr block_stmt ('else' (block_stmt | if_stmt))?
//              ^~~~ ^~~~~~~~~~          ^~~~~~~~~~~~~~~~~~~~~~
//              |     if_block            else_block
//             condition
```

`if` 语句代表一组可选执行的语句。

`if` 语句的执行流程是：

- 求 `condition` 的值
  - 如果值为 `true`，则执行 `if_block`
  - 否则，如果存在 `else_block`，执行 `else_block`
  - 否则，执行下一条语句

请注意，**if 语句的条件表达式可以没有括号，且 条件执行的语句都必须是代码块。**

以下是一些合法的 if 语句：

```
if x > 0 {
  x = x + 1;
}

if y < 0 {
  z = -1;
} else if y > 0 {
  z = 1;
} else {
  z = 0
}
```

以下是一些不合法的 if 语句：

```
// 必须是代码块
if x > 0
  x = x + 1;
```

`while` 语句

```
while_stmt -> 'while' expr block_stmt
//                    ^~~~ ^~~~~~~~~~while_block
//                     condition
```

while 语句代表一组可以重复执行的语句。

while 语句的执行流程是：

- 求值 `condition`
  - 如果为 `true`
    - 执行 `while_block`
    - 回到开头重新求值

- 如果为 `false` 则执行之后的代码

`return` 语句

```
return_stmt -> 'return' expr? ';'
```

使用 `return` 语句从一个函数中返回。return 语句可以携带一个表达式作为返回值。

return 语句有以下的语义约束：

- 如果函数声明的返回值是 `void`，return 语句不能携带返回值；否则，return 语句必须携带返回值
- 返回值表达式的类型必须与函数声明的返回值类型相同
- 当执行到返回值类型是 `void` 的函数的末尾时，应视作存在一个 return 语句进行返回

> UB: 在基础 C0 中不会出现部分分支没有返回值的情况，所以没有返回语句的分支的返回值是未定义的。在扩展 C0 中你必须检查每个分支都能够正常返回。

代码块

```
block_stmt -> '{' stmt* '}'
```

一个代码块可以包含一条或多条语句。执行代码块的效果是顺序执行这些语句。在扩展 c0 中，一个代码块是其所在作用域的子作用域。

空语句

```
empty_stmt -> ';'
```

空语句没有任何作用，只是一个分号而已。

函数和全局变量

函数

```
function_param -> 'const'? IDENT ':' ty
function_param_list -> function_param (',' function_param)*
function -> 'fn' IDENT '(' function_param_list? ')' '->' ty block_stmt
//              ^~~~      ^~~~~~~~~~~~~~~~~~~~~          ^~ ^~~~~~~~~~
//              |              |                          |  |
//              function_name  param_list     return_type  function_body
```

与 miniplc0 不同，c0 中存在函数。

c0 中一个函数的定义由 *函数名*、*参数列表*、*返回类型* 和 *函数体* 组成。

函数有以下语义约束：

- 函数的名称 `function_name` 不能重复。
- 函数的参数声明 `param_list` 与 含有初始化表达式的变量声明 有相同的语义约束。
- 函数体、函数的参数声明 在同一个作用域（函数作用域）中，是全局作用域的子作用域。

另外再提醒一下，返回值类型 `return_type` 即使为 `void` 也不能省略。

函数体的组成单位是语句，见 语句页面。

全局变量

全局变量的声明与局部变量相同，都是使用 声明语句 进行声明。全局变量的定义方式和约束与局部变量相同。全局变量所在作用域是全局，因此有可能被函数内定义的局部变量覆盖。

程序结构

```
program -> decl_stmt* function*
```

一个 c0 的程序中可以存在多个 *变量声明*，后接多个 *函数声明*。

语义约束：

- 一个合法的 c0 程序必须存在一个名为 `main` 的函数作为程序入口，否则应视为编译错误。

标准库

由于 c0 语言本身比较简单，为了实现输入输出的功能，我们规定了 8 个不需要声明就可以调用的函数，它们的分别是：

```
/// 读入一个整数
fn getint() -> int;

/// 读入一个浮点数
fn getdouble() -> double;

/// 读入一个字符
fn getchar() -> int;

/// 输出一个整数
fn putint(int) -> void;

/// 输出一个浮点数
fn putdouble(double) -> void;

/// 输出一个字符
fn putchar(int) -> void;

/// 输出整数代表的全局常量字符串
fn putstr(int) -> void;

/// 输出一个换行
fn putln() -> void;
```

在实现时，这些函数既可以编译成使用虚拟机中的 `callname` 指令调用，也可以编译成相应的虚拟机指令（`scan.i`, `print.i` 等），在虚拟机实现上两者是等价的。

扩展 C0

这里列出了实现之后可以获得加分的扩展 C0 特性。

加分的单位尚未确定，目前的加分数量都是相对值。

注释

加分：5pt

```
COMMENT -> '//' regex(.*) '\n'
```

C0 的注释是从 `//` 开始到这行结束（遇到第一个 `\n`）为止的字符序列。注释不应当被词法分析输出。

字符字面量

加分：5pt

```
char_regular_char -> [^'\\\n\r]
CHAR_LITERAL -> '\'' (char_regular_char | escape_sequence)* '\''
literal_expr -> UINT_LITERAL | FLOAT_LITERAL | STRING_LITERAL | CHAR_LITERAL
```

字符字面量是由单引号 `'` 包裹的单个字符或转义序列。其中单个字符可以是 ASCII 中除了单引号 `'`、反斜线 `\\`、空白符 `\r`(CR)、`\n`(LF)、`\t`(Tab) 以外的任何字符。转义序列可以是 `\'`、`\"`、`\\`、`\n`、`\t`、`\r`，含义与 C 中的对应序列相同。

字符字面量 的语义是被包裹的字符的 ASCII 编码无符号扩展到 64 位的整数值，类型是 `int`。

类型转换 & 浮点数类型

加分：25pt

类型转换

```
AS_KW -> 'as'
as_expr -> expr 'as' ty
expr -> .. | as_expr
```

显式类型转换通过 `as` 表达式实现。语言中没有隐式类型转换。

`表达式 as 类型` 表示将 `表达式` 的计算结果转换为 `类型` 所表示的类型的数据。`as` 表达式的左侧数据类型和右侧类型都不能是 `void`。

允许的类型转换包括：

- 类型 T 转换到它自己
- 浮点数 `double` 和整数 `int` 之间互相转换

浮点数类型

```
FLOAT_LITERAL -> digit+ '.' digit+ ([eE] digit+)?
```

浮点数类型 `double` 是遵循 IEEE 754 标准的 64 位浮点数（在其它语言中经常称作 `double`、`float64` 或 `f64`）。

浮点数和整数之间不能进行运算。浮点数之间进行四则运算的结果仍为浮点数。

作用域嵌套

加分：10pt

简而言之，在任何一个代码块中都可以声明变量。

要求：

- 每个代码块（`block_stmt`）都是一级作用域。
- 每级作用域内部的变量声明不能重复。

- 作用域内声明的变量可以覆盖上一级作用域中的变量。
- 每个作用域内定义的变量在作用域结束后即失效

比如，下面的函数中变量 `x` (1)、`counter` (2)、循环内的 `x` (3) 可以被访问的区域如竖线左侧所示：

```
    1  |  fn fib_iter(x: int) -> int {  // (1)
    |  |      let last_val: int = 1;
    |  |      let cur_val: int = 1;
  2 |  |      let counter: int = x - 2; // (2)
  | |  |      while counter > 0 {
3 |    |          let x: int = cur_val + last_val; // (3)
| |    |          last_val = cur_val;
| |    |          cur_val = x;
- |    |      }
  | |  |      return cur_val;
  - -  |  }
```

变量声明增强

加分：5pt

在每一级作用域中，你不仅可以在作用域顶端声明变量，也能在作用域中间声明变量。在作用域中间声明的变量同样遵循上一条的生命周期。

`break` 和 `continue`

加分：10pt

```
BREAK_KW  -> 'break'
CONTINUE_KW -> 'continue'

break_stmt -> 'break' ';'

continue_stmt -> 'continue' ';'
```

- `break` 和 `continue` 必须在循环体内使用，在其他地方使用是编译错误。
- `break` 代表跳出循环体，控制转移到循环外的下一条语句。
- `continue` 代表跳过本次循环体的代码，控制转移到循环体的最后一条语句。

> 提示：进入循环之前记录一下跳转的目标位置

函数返回路径检查

加分：10pt

你需要对每一个函数的所有控制流进行检查，保证如果函数有返回值，那么所有可能的控制流都能导向 `return` 语句。比如，以下的函数不能通过编译：

```
fn foo(i: int) -> int {
    if i == 0 {
        return 1;
    } else {
        putint(0);
    }
    // 这个分支没有返回
}
```

这个也不行：

```
fn bar() -> int {
    let i: int;
    i = getint();
    while i > 0 {
        i = i - 1;
        if i <= 0 {
            return i;
        }
    }
    // 这个分支没有返回
}
```

这个可以，因为在到达函数结尾之前两个分支都返回了：

```
fn baz(i: int) -> int {
    if i == 0 {
        return 1;
    } else {
        return 0;
    }
    // 没有分支可以到达这里
}
```

> 提示：用基本块表示函数就能看得一清二楚了。

> UB: 我们不会考察对于无限循环的控制流检查。你可以选择报错，也可以选择无视。

实现方法指导

实现一个 parser 有很多种方法，这里会提供一些关于代码实现（而不是理论）的方法指导。

对于没有讲到的内容，可以参考 去年的指导书

一些通用的部分

类型定义

对于词法、语法分析时用到的类型，因为类型确定且已知，可以使用继承实现。在支持和类型 (sum type) 的语言里也可以用和类型实现。这样做可以显著降低判断 token 或者语法树节点类型时的工作量，因为可以直接判断变量本身的类型，甚至直接进行模式匹配。比如：

```
/* 词法分析器 */

class Token {}

class NumberToken : Token {
    public double value;
}

// ...

/* 语法分析器 */

class Expr {}

class Literal : Expr {}

class IntegerLiteral : Literal {
    public long value;
}

class StringLiteral : Literal {
    public string value;
}

class BinaryExpr : Expr {
    public Operator op;
    public Expr lhs;
    public Expr rhs;
}

// ...
```

或者在支持的语言里使用带标签的联合类型：

```
enum Expr {
    Literal(LiteralExpr),
    Binary(BinaryExpr),
    // ...
}

enum LiteralExpr {
    Integer(i64),
    String(String),
    // ...
}

struct BinaryExpr {
    pub op: Operator,
    pub lhs: Ptr<Expr>,
    pub rhs: Ptr<Expr>,
}

// ...
```

迭代器

迭代器（Iterator）是对一系列值的抽象，比如说一列输入的字符或者解析完的 token。使用迭代器可以有效地将输入数据和对数据的获取操作解耦，方便在不同时候使用不同方式输入数据，以及进行测试。常见高级语言都有对于迭代器的抽象，包括：

- Java: `java.util.Iterator`
- C#: `System.Collections.Generic.IEnumerator`
- C++: `std::iterator::iterator_traits`
- C++20: concept `std::ranges::input_iterator`
- Python: 实现 `__next__` 的类型
- JavaScript: 实现 `Symbol.iterator` 的类型

由于在解析时常常要回溯，使用的迭代器可以提供一些额外的方法，比如 `peek()` 用于查看下一个值但不移动迭代器，或者 `unread(value)` 用于将已经读出的值放回迭代器。

词法分析

词法分析这个主题比较简单，基本上就是对于每个 token 使用自动机（或者退化成普通的逻辑分支）进行解析。token 的组成一般比较简单，可以在分析时参考正则表达式的状态来设计自动机或逻辑分支。

当然，也有一些库允许你直接用正则表达式定义 token 来进行自动分析。好耶。

> 不要学助教用逻辑分支模拟自动机（逃

语法分析

普通的递归下降分析法

递归下降是一个很简单、很直观的分析法，也是大多数人实现语法分析的首选方法。在实现递归下降分析器的时候，有一些可以降低编码难度的方法。

使用迭代器和辅助函数

看 miniplc0 java 版本基本上就够了（逃）

解析器组合子 (Parser Combinator)

助教没有试过这么写，如果你用 Haskell 来写的话或许可以试试 `parsec` 这个库。

使用 LL/LR 解析器生成器

自动生成解析器代码总感觉有点作弊的意思，不过用了就用了吧（笑）。如果你确定要用的话，记得选一个好用的，比如 ANTLR。

navm 虚拟机标准

本次大作业的编译目标是 Natrium 虚拟机 (navm) 的汇编 (s0)。其设计参考了 JVM、DotNet CLR 和上学期的 c0 虚拟机。

虚拟机简介

navm 是一个 栈式虚拟机 —— 简单来说就是，它的寄存器是一个栈。除了少数内存访问指令以外，navm 的大部分指令都只操作位于栈顶的数据。堆栈式计算机的指令与 逆波兰表示法（后缀表示法 表示的表达式（或者说后序遍历的表达式树）有简单的对应关系。

navm 有 64 位有符号整数、无符号整数、浮点数三种数据类型。详见 数据类型 节。

navm 使用 64 位无符号整数表示地址，具体实现不需要关心。

navm 使用自制的指令集，共有 50+ 个指令，详见 指令集说明。

内存空间

navm 的内存空间以 8 位（1 字节）为单位寻址。8、16、32、64 位的数据类型分别以其大小为单位在内存中对齐。当读取或写入操作未对齐时，会产生 `UnalignedAccess` 错误。

navm 的栈空间以 8 字节为一个 slot，压栈、弹栈以及各种运算操作均以 slot 为单位进行。默认情况下，栈的大小是 1 MiB (1048576 字节)，即 131072 个 slot。栈空时弹栈和栈满时压栈分别会产生 `StackUnderflow` 和 `StackOverflow` 错误。

数据类型

navm 在运算中支持三种基本数据类型，分别是 64 位无符号整数 `u64`、64 位有符号整数 `i64`、64 位浮点数 `f64`。长度更短的整数可以使用 `u64` 和 `i64` 模拟。

`u64` 和 `i64` 都是 64 位整数，使用二进制补码形式表示。两种类型在多数整数运算中不做区分，仅在 `cmp.T`（比较指令，见下）等两种运算结果有差别的地方有所区分。在运算溢出时，两种类型均采用环绕 (wrap-around) 方式处理结果。`u64` 同时也可以表示虚拟机中的内存地址。

`f64` 是符合 IEEE 754 规定的双精度浮点数。

二进制格式

o0 是 navm 所使用的二进制程序文件格式，其作用和内容类似 Java 的 `.class` 文件或者 DotNet 的 `.dll` 文件。

下面的结构体表示了 o0 的二进制文件结构。其中， `uXX` 表示 XX 位无符号整数。

```
/// 整个 o0 二进制文件
struct o0 {
    /// 魔数
    magic: u32 = 0x72303b3e,
    /// 版本号，定为 1
    version: u32 = 0x00000001,
    /// 标志位，留空
    flags: u32 = 0x00000000,
    /// 全局变量表
    globals: Array<GlobalDef>,
    /// 函数列表
    functions: Array<FunctionDef>,
}

/// 类型为 T 的通用数组的定义
struct Array<T> {
    /// 数组的长度
    count: u32,
    /// 数组所有元素的无间隔排列
    items: T[],
}

/// 单个全局变量
struct GlobalDef {
    /// 是否为常量？非零值视为真
    is_const: u8,
    /// 值
    value: Array<u8>,
}

/// 函数
struct FunctionDef {
    /// 函数名称在全局变量中的位置
    name: u16,
    /// 返回值占据的 slot 数
    return_slots: u16,
    /// 参数占据的 slot 数
    param_slots: u16,
    /// 局部变量占据的 slot 数
    loc_slots: u16,
    /// 函数体
    body: Array<Instruction>,
}

/// 指令，可以是以下三个选择之一
union Instruction {
    /// 无参数的指令，占 1 字节
    variant NoParam {
        opcode: u8
    },
    /// 有 4 字节参数的指令，占 5 字节
    variant u32Param {
        opcode: u8,
        param: u32,
    }
    /// 有 8 字节参数的指令，占 9 字节
    variant u64Param {
        opcode: u8,
        param: u64
```

```
    }
}
```

栈帧结构

> 这里描述的是 **这个** navm 实现中使用的栈帧结构。

```
| ...            |
|                | <- 栈顶 %sp
| 表达式栈 ...   |
| 表达式栈       |
| 局部变量 ...   |
| 局部变量       |
| 虚拟机参数...  |
| 虚拟机参数     | <- 被调用者栈底 %bp
|================|===
| 调用参数 ...   | v
| 调用参数       | |
| 返回值         | |
| 中间结果       | 调用者栈
| ...            | ^
|================|===
```

其中，调用参数和返回值由调用者压栈，调用参数在函数返回后由被调用者清理。

虚拟机参数

虚拟机会在调用参数和局部变量之间插入一系列的虚拟机参数以辅助虚拟机运行，目前本虚拟机存储的参数格式为（从栈顶到栈底）：

```
| ...              |
| 局部变量         |
|==================|
| 调用者函数 ID    |
| 调用者 %ip       |
| 调用者 %bp       |
|==================|
| 参数             |
| ...              |
```

函数调用时栈帧变化示例

假设现有一个函数 `test`，有 1 slot 的返回值、2 slot 的参数和 2 slot 的局部变量。

```
/// 可以看成是这样的一个函数
fn test(a: int, b: int) -> int {
    let c: int = ...;
    let d: int = ...;
    ...
    return ...;
}
```

现在，它被编号为 1 的函数 `main` 调用。在调用前，调用者应压入 1 slot 的返回值预留空间、2 slot 的参数（顺序压栈），再通过调用指令调用这个函数。调用前的栈应该长这样：

```
| -           |
|=============|<- 栈顶
| b           | ↑
| a           | 参数
| _ret        | 返回值
| ...         | ...表达式栈
```

在执行 `call` 指令后，栈中的变量以及对应的偏移量如下：

```
| -             | <- 栈顶（表达式栈）
| d             | ↑           loc.1
| c             | 局部变量    loc.0
|===============|
| 1             | ↑
| %ip           |
| %bp           | 虚拟机数据
|===============|
| b             | ↑           arg.2
| a             | 参数        arg.1
| _ret          | 返回值      arg.0
| ...           |
```

在函数调用返回后，栈如下：

```
| -           |
| // d        |
| // c        |
| // 1        |
| // %ip      |
| // %bp      |  ↑
| // b        |  |
| // a        | 以上内容被弹栈
|=============| <- 栈顶
| _ret        | 返回值
| ...         |
```

程序入口

navm 总是会最先运行函数列表里编号为 0 的（也就是整个列表中第一个）函数，按照惯例这个函数的名称为 `_start`。`_start` 函数没有任何参数，也不返回任何值，这两项的参数会被忽略。`_start` 函数不能有返回指令。

一般来说，程序会在 `_start` 中设置全局变量的值，以及进行其他的准备工作。在准备工作完成之后，`_start` 函数应当调用 `main` 函数开始正式的程序运行。如果需要，`_start` 函数也可以在 `main` 函数返回之后进行清理工作。`_start` 函数不需要返回。

一个示例的 `_start` 函数如下：

```
fn _start 0 0 -> 0 {
    // 设置全局变量 1 的值为 1 + 1;
    globa 1
    push 1
    push 1
    add.i
    store.64
    // 调用 main
    call 4
    // 没有返回语句
}
```

指令集

navm 的指令使用 8 位（1 字节）无符号整数标识，后面跟随可变长度的操作数。操作数类型为 `u64` `i64` 时，长度为 64 位（8 字节），类型为 `u32` `i32` 时，长度为 32 位（6 字节）。

下表展示了 navm 的所有指令。其中弹栈和压栈的格式为：`栈变化范围[:变量]`，数字按照栈底到栈顶编号。

指令	指令名	操作数	弹栈	压栈	介绍
0x00	`nop`	-	-	-	空指令
0x01	`push`	num:u64	-	1:num	将 num 压栈
0x02	`pop`	-	1		弹栈 1 个 slot
0x03	`popn`	num:u32	1-num	-	弹栈 num 个 slot
0x04	`dup`	-	1:num	1:num, 2:num	复制栈顶 slot
0x08	`loca`	off:u32	-	1:addr	加载 off 个 slot 处局部变量的地址
0x08	`arga`	off:u32	-	1:addr	加载 off 个 slot 处参数/返回值的地址
0x09	`globa`	n:u32	-	1:addr	加载第 n 个全局变量/常量的地址
0x10	`load.8`	-	1:addr	1:val	从 addr 加载 8 位 value 压栈
0x11	`load.16`	-	1:addr	1:val	从 addr 加载 16 位 value 压栈
0x12	`load.32`	-	1:addr	1:val	从 addr 加载 32 位 value 压栈
0x13	`load.64`	-	1:addr	1:val	从 addr 加载 64 位 value 压栈

指令	指令名	操作数	弹栈	压栈	介绍
0x14	`store.8`	-	1:addr, 2:val	-	把 val 截断到 8 位存入 addr
0x15	`store.16`	-	1:addr, 2:val	-	把 val 截断到 16 位存入 addr
0x16	`store.32`	-	1:addr, 2:val	-	把 val 截断到 32 位存入 addr
0x17	`store.64`	-	1:addr, 2:val	-	把 val 存入 addr
0x18	`alloc`	-	1:size	1:addr	在堆上分配 size 字节的内存
0x19	`free`	-	1:addr	-	释放 addr 指向的内存块
0x1a	`stackalloc`	size:u32	-	-	在当前栈顶分配 size 个 slot，初始化为 0
0x20	`add.i`	-	1:lhs, 2:rhs	1:res	计算 res = lhs + rhs，参数为整数
0x21	`sub.i`	-	1:lhs, 2:rhs	1:res	计算 res = lhs - rhs，参数为整数
0x22	`mul.i`	-	1:lhs, 2:rhs	1:res	计算 res = lhs * rhs，参数为整数
0x23	`div.i`	-	1:lhs, 2:rhs	1:res	计算 res = lhs / rhs，参数为有符号整数
0x24	`add.f`	-	1:lhs, 2:rhs	1:res	计算 res = lhs + rhs，参数为浮点数
0x25	`sub.f`	-	1:lhs, 2:rhs	1:res	计算 res = lhs - rhs，参数为浮点数
0x26	`mul.f`	-	1:lhs, 2:rhs	1:res	计算 res = lhs * rhs，参数为浮点数
0x27	`div.f`	-	1:lhs, 2:rhs	1:res	计算 res = lhs / rhs，参数为浮点数
0x28	`div.u`	-	1:lhs, 2:rhs	1:res	计算 res = lhs / rhs，参数为无符号整数
0x29	`shl`	-	1:lhs, 2:rhs	1:res	计算 res = lhs << rhs
0x2a	`shr`	-	1:lhs, 2:rhs	1:res	计算 res = lhs >> rhs （算术右移）
0x2b	`and`	-	1:lhs, 2:rhs	1:res	计算 res = lhs & rhs
0x2c	`or`	-	1:lhs, 2:rhs	1:res	计算 res = lhs | rhs
0x2d	`xor`	-	1:lhs, 2:rhs	1:res	计算 res = lhs ^ rhs
0x2d	`not`	-	1:lhs	1:res	计算 res = !lhs

指令	指令名	操作数	弹栈	压栈	介绍
0x2e	`inv`	-	1:lhs	1:res	计算 res = ~lhs（按位反转）
0x30	`cmp.i`	-	1:lhs, 2:rhs	1:res	比较有符号整数 lhs 和 rhs 大小
0x31	`cmp.f`	-	1:lhs, 2:rhs	1:res	比较浮点数 lhs 和 rhs 大小
0x32	`cmp.u`	-	1:lhs, 2:rhs	1:res	比较无符号整数 lhs 和 rhs 大小
0x34	`neg.i`	-	1:lhs	1:res	对 lhs 取反
0x35	`neg.f`	-	1:lhs	1:res	对 lhs 取反
0x36	`itof`	-	1:lhs	1:res	把 lhs 从整数转换成浮点数
0x37	`ftoi`	-	1:lhs	1:res	把 lhs 从浮点数转换成整数
0x38	`shrl`	-	1:lhs, 2:rhs	1:res	计算 res = lhs >>> rhs （逻辑右移）
0x39	`set.lt`	-	1:lhs	1:res	如果 lhs < 0 则推入 1，否则 0
0x3a	`set.gt`	-	1:lhs	1:res	如果 lhs > 0 则推入 1，否则 0
0x41	`br`	off:i32			无条件跳转偏移 `off`
0x42	`br.false`	off:i32		1:test	如果 `test` 是 0 则跳转偏移 `off`
0x43	`br.true`	off:i32		1:test	如果 `test` 非 0 则跳转偏移 `off`
0x48	`call`	id:u32	见栈帧介绍	-	调用编号为 id 的函数
0x49	`ret`	-	-	见栈帧介绍	从当前函数返回
0x4a	`callname`	id:u32	见栈帧介绍	-	调用名称与编号为 id 的全局变量相同的函数
0x50	`scan.i`	-	-	1:n	从标准输入读入一个整数 n
0x51	`scan.c`	-	-	1:c	从标准输入读入一个字符 c
0x52	`scan.f`	-	-	1:f	从标准输入读入一个浮点数 f
0x54	`print.i`	-	1:x	-	向标准输出写入一个有符号整数 x
0x55	`print.c`	-	1:c	-	向标准输出写入字符 c
0x56	`print.f`	-	1:f	-	向标准输出写入浮点数 f
0x57	`print.s`	-	1:i	-	向标准输出写入全局变量 i 代表的字符串
0x58	`println`	-	-	-	向标准输出写入一个换行
0xfe	`panic`				恐慌（强行退出）

`cmp.T` 指令

指令会在 `lhs < rhs` 时压入 `-1`，`lhs > rhs` 时压入 `1`，`lhs == rhs` 时压入 `0`。浮点数无法比较时压入 `0`。

`load.8/16/32/64` 指令

指令会从 `addr` 处取 `T` 长度的数据压入栈中。如果 `addr` 不是 `T` 的倍数，将会产生 `UnalignedAccess` 错误。如果 `T` 小于 64，多余的数位将会被补成 0。

`store.8/16/32/64` 指令

指令会将 `T` 长度的数据弹栈并存入 `addr` 地址处。如果 `addr` 不是 `T` 的倍数，将会产生 `UnalignedAccess` 错误。如果 `T` 小于 64，数据将被截断至 `T` 长度。

应该比较常见的问题

条件跳转

如果需要实现条件跳转，请使用以下指令的组合（`T` 代表 `u`、`f` 或 `i`：

- 等于：`cmp.T`, `br.false`
- 不等于：`cmp.T`, `br.true`
- 大于：`cmp.T`, `set.gt`, `br.true`
- 小于：`cmp.T`, `set.lt`, `br.true`
- 大于等于：`cmp.T`, `set.lt`, `br.false`
- 小于等于：`cmp.T`, `set.gt`, `br.false`

局部变量和参数的存取

在 navm 中，局部变量和参数是分开存储的。其中，参数和返回值（`arg`）存储在一起，从栈底方向开始顺序编号。局部变量（`loc`）存储在另一个位置，也从栈底开始顺序编号。比如：

```
| d             | ↑          loc.1
| c             | 局部变量   loc.0
|===============|
| 1             | ↑
| %ip           |
| %bp           | 虚拟机数据
|===============|
| b             | ↑          arg.2
| a             | 参数       arg.1
| _ret          | 返回值     arg.0
| ...           |
```

此时执行 `loca 1` 获得的就是变量 `d` 的地址，执行 `arga 0` 获得的就是返回值的地址。

获取到地址之后，就可以执行存取操作了。我们用的基本都是 64 位数据类型，所以使用 `load.64` 和 `store.64` 指令就可以了。

```
# 加载局部变量 1
loca 1
load.64

# 存储 0 到参数 0
arga 0
push 0
store.64

# 将局部变量 1 拷贝到局部变量 0
loca 0
loca 1
load.64
store.64
```

提交与评测说明

C0 大作业的提交方式是自动评测，评测的方式之后会写在这里。

程序要求

你提交的程序应当至少能通过命令行参数接收一个 c0 代码文件的输入，并输出一个 s0 二进制文件。

如果编译过程中出现了错误（语法、语义、编译过程错误等），你的程序应当**以非 0 的返回值退出**。否则如果一切正常，你的程序应当以返回值 0 退出。

提交要求

> 在线评测的相关内容将在第 13 周前发布，请大家不要着急。

参考实现

这个仓库 中保存了一个实现了全部扩展 C0 的编译器参考实现，以及相关的工具链。你可以在 Release 中下载相关程序自己测试（尚未发布），也可以在 https://c0.karenia.cc/playground 使用浏览器版本测试。

当然，参考实现真的只是给你参考用的。敢抄的话有你好果子吃（

参考资料