编译器设计入门

李睿远

Jul 05, 2026

编译器把高级语言程序转换成机器能直接执行的指令序列。本文用「前端—中端—后端」三阶段的思路，带读者从字符流开始，一步步生成能在 x86-64 平台上运行的二进制文件。目标读者只需具备 C、C++ 或 Python 的基本语法知识，无需任何编译原理背景。文章在每个关键步骤都给出可运行的代码片段，并对代码逐行解读，帮助读者在本地复现完整流程。

1　编译器三阶段概览

编译流程可以划分为前端、中端和后端。前端负责把源代码解析成结构化的抽象语法树，同时完成词法、语法、语义检查；中端把树形结构转换成更易于优化的中间表示；后端再把中间表示映射到具体目标指令集。本文演示的迷你编译器只实现最核心路径：输入 `int main(){return 42;}`，输出可执行文件。

2　词法分析：把字符流切成标记

词法分析器读取源文件中的一个个字符，根据正则规则把它们聚合成标记（token）。例如，关键字 `int`、标识符 `main`、数字 `42` 各是一个 token。手写词法分析器通常用一个巨大的 `match` 语句依次判断当前字符属于哪一类。

```
fn next_token(input: &str, pos: &mut usize) -> Option<Token> {
    while *pos < input.len() {
        let ch = input[*pos..].chars().next().unwrap();
        match ch {
            ' ' | '\n' => { *pos += ch.len_utf8(); continue; }
            '0'..='9' => { /* 解析整数并前进 */ }
            'a'..='z' | 'A'..='Z' | '_' => { /* 解析标识符或关键字 */ }
            _ => { /* 其他符号 */ }
        }
    }
    None
}
```

上面这段代码首先跳过空白字符，然后用字符范围匹配决定下一步动作。整数分支会继续读取后续数字，直到遇到非数字字符；标识符分支则把字母、数字、下划线连续吃掉，再查表判断是否为关键字。这样的实现虽然简单，但已能正确切分本文示例中的所有 token。

3 语法分析：递归下降构造抽象语法树

拿到 token 流后，语法分析器按照上下文无关文法把 token 组织成树形结构。递归下降是最直观的自顶向下方法：为每条产生式写一个函数，函数内部按顺序调用其他产生式函数或匹配终结符。

```
fn parse_func(tokens: &[Token], idx: &mut usize) -> Func {
    expect(TokenKind::KwInt, tokens, idx);
    let name = expect_ident(tokens, idx);
    expect(TokenKind::LParen, tokens, idx);
    expect(TokenKind::RParen, tokens, idx);
    expect(TokenKind::LBrace, tokens, idx);
    let stmt = parse_stmt(tokens, idx);
    expect(TokenKind::RBrace, tokens, idx);
    Func { name, stmt }
}
```

函数首先断言当前 token 必须是 `int`，然后依次吃掉函数名、左右括号、左右大括号，最后解析函数体语句。整个调用链清晰地对应文法规则，易于理解和调试。

4 表达式优先级：Pratt 解析

当文法中出现表达式时，普通递归下降容易在左递归或优先级上栽跟头。Pratt 解析用「当前运算符优先级」来驱动递归深度，从而在一次遍历内正确处理多种优先级。

```
fn parse_expr(tokens: &[Token], idx: &mut usize, min_prec: u8) -> Expr {
    let mut lhs = parse_atom(tokens, idx);
    while let Some(op) = peek_op(tokens, *idx) {
        let prec = precedence(op);
        if prec < min_prec { break; }
        *idx += 1;
        let rhs = parse_expr(tokens, idx, prec + 1);
        lhs = Expr::Binary(op, Box::new(lhs), Box::new(rhs));
    }
    lhs
}
```

代码先解析最基本的原子表达式，然后在循环中不断查看下一个运算符。若其优先级不低于当前阈值，就递归解析右侧子表达式，并把结果合并成新的二叉节点。优先级阈值随递归深度递增，自然实现了「先乘除后加减」的规则。

5 语义分析与最小符号表

在抽象语法树上做类型检查，需要知道每个名字的含义。符号表最简单的实现是用哈希表记录名字到类型的映射，同时用一个栈来处理嵌套作用域。

```
struct SymbolTable {
    scopes: Vec<HashMap<String, Type>>,
}

impl SymbolTable {
    fn enter_scope(&mut self) { self.scopes.push(HashMap::new()); }
    fn exit_scope(&mut self) { self.scopes.pop(); }
    fn insert(&mut self, name: String, ty: Type) {
        self.scopes.last_mut().unwrap().insert(name, ty);
    }
}
```

当编译器遇到函数定义时，先压入新作用域，再把形参插入表中；退出函数体时弹出作用域即可恢复外层可见性。类型检查阶段只需在符号表里查找名字，就能判断 `return 42;` 中的 42 是否与函数返回类型一致。

6 中间表示：把树拍平成指令序列

抽象语法树仍然带有嵌套结构，不利于后续优化和寄存器分配。线性中间表示把每个表达式或语句映射成一条或多条三地址指令。

```
enum Instr {
    Const { dst: u32, val: i32 },
    Ret { src: u32 },
}
```

对于 `return 42;`，转换函数会先给 42 分配一个虚拟寄存器，然后生成一条 `Const` 指令把立即数写入该寄存器，最后生成 `Ret` 指令把寄存器值返回。整个过程只需要一次递归遍历即可完成。

7 目标代码生成：从 IR 到 x86-64 汇编

寄存器分配最简单的策略是「全部溢出栈」，即把每个虚拟寄存器都映射到栈上的一个槽位。指令选择则把 IR 操作逐条翻译成对应的汇编助记符。

```
main:
    push rbp
    mov rbp, rsp
    sub rsp, 8 ; 为局部变量开辟空间
```

```
    mov dword [rbp-4], 42
    mov eax, dword [rbp-4]
    leave
    ret
```

序言负责保存旧栈帧指针并分配新栈空间；尾声则反向操作恢复调用者环境。mov 指令把立即数写入栈槽，再把它读到 eax 作为返回值，最后用 ret 结束函数。

8 完整流程：一键从源码到可执行

把词法、语法、语义、中间表示、代码生成五个模块串联后，只需在命令行执行一次 cargo run 即可完成全部流程。最终得到的 ELF 文件可以用 objdump -d 反汇编，验证 ret 42 确实出现在 .text 段中。

9 继续深入的方向

掌握最小编译器后，可以继续学习常量折叠、公共子表达式消除等中端优化；也可以研究图着色寄存器分配、ABI 调用约定等后端技术。若想直接对接工业级工具链，可尝试把 IR 替换成 LLVM IR，再用 inkwell 绑定生成目标文件。练习时可先扩展四则运算，再添加函数调用与参数传递，逐步丰富语言特性。