🔧 Simple-C 编译器前端

编译原理课程设计

李易涵

302023315218

技术栈： Python + LLVM

词法分析 → 语法分析 → 语义分析 → 中间代码生成

目录

01

项目概述

Project Overview

项目目标

> 实现一个完整的 **编译器前端**，将自定义语言编译为中间代码

设计语言

Simple-C
类 C 语法

编译目标

LLVM IR
工业级中间代码

可执行

JIT 即时编译
直接运行

技术选型

组件	技术选择	选型理由
实现语言	Python 3.10+	开发效率高，易于理解
LLVM 绑定	llvmlite	轻量级，Numba 团队维护
词法分析	手写 Lexer	展示原理，完全掌控
语法分析	递归下降	LL 分析，简洁直观
中间代码	LLVM IR	工业标准，可直接执行

Simple-C 语言特性

支持的特性：

- 数据类型：int, float, bool, void
- 变量声明：let x: int = 10;
- 函数定义：func name() -> type
- 控制流：if/else, while
- 算术运算：+, -, *, /, %
- 比较运算：<, >, <=, >=, ==, !=
- 逻辑运算：&&, ||, !
- 递归函数调用

```
// 阶乘示例
func factorial(n: int) -> int {
    if (n <= 1) {
        return 1;
    }
    return n * factorial(n - 1);
}

func main() -> int {
    let result: int = factorial(5);
    print(result);
    return 0;
}
```

02

编译器原理

Compiler Theory

什么是编译器?

> 编译器是一个程序，将 **源语言** 程序翻译成 **目标语言** 程序

💡 类比：编译器 = 翻译官（把中文翻译成英文）

编译的六个阶段

- 绿色 = 本项目实现
- 蓝色 = LLVM 负责

前端 vs 后端

	前端 (Front-end)	后端 (Back-end)
阶段	词法、语法、语义、IR	优化、目标代码生成
关注点	源语言相关	目标平台相关
输出	中间表示 (IR)	可执行文件
本项目	✅ 完全实现	使用 LLVM

💡 前后端分离：一个前端可配合多个后端（跨平台）

为什么选择 LLVM?

LLVM 是什么?

- Low Level Virtual Machine
- 模块化的编译器框架
- 2003 年由 Chris Lattner 创建

谁在使用?

- Clang (C/C++)
- Rust
- Swift
- Julia

LLVM 的优势

- 🎯 **工业标准**：久经验证
- ⚡ **优化强大**：自动优化
- 🖥️ **多平台**：x86, ARM, ...
- 🔥 **JIT 支持**：可直接运行
- 📚 **文档完善**：易于学习

03

系统架构

System Architecture

数据流图


源代码
.sc 文件
词法分析
Lexer
语法分析
Parser
语义分析
Semantic
代码生成
LLVM IR
Token 流
AST
符号表
.ll 文件

项目目录结构

```
Compiler_Principles/
├── src/
│   ├── main.py                # 入口 + CLI
│   ├── lexer/
│   │   ├── tokens.py          # Token 类型定义 (45 种)
│   │   └── lexer.py           # 词法分析器 (250 行)
│   ├── parser/
│   │   ├── ast_nodes.py       # AST 节点定义 (20+ 种)
│   │   └── parser.py          # 递归下降解析器 (500 行)
│   ├── semantic/
│   │   ├── symbol_table.py    # 符号表 + 作用域
│   │   └── analyzer.py        # 语义分析器 (300 行)
│   └── codegen/
│       └── llvm_generator.py  # LLVM IR 生成 (450 行)
└── examples/                  # 4 个示例程序
```

代码统计

模块	文件数	代码行数	核心类
词法分析	2	~350	Lexer, Token, TokenType
语法分析	2	~800	Parser, 20+ ASTNode
语义分析	2	~400	SemanticAnalyzer, SymbolTable
代码生成	1	~500	LLVMCodeGenerator, JITEngine
合计	**7**	**~2000**	-

04

词法分析

Lexical Analysis

词法分析的作用

> 将 **字符流** 分解成 **Token（词法单元）**

输入（字符流）：

```
let x = 10 + 5;
```

输出（Token 流）：

```
[LET] [ID:x] [=] [10] [+] [5] [SEMI]
```

💡 类比：把一句话分解成一个个单词

Token 类型设计

类别	Token 类型	示例
关键字	FUNC, LET, IF, ELSE, WHILE, FOR, RETURN	func, let, if
类型	INT, FLOAT, BOOL, VOID	int, float
字面量	INT_LITERAL, FLOAT_LITERAL, TRUE, FALSE	123, 3.14
算术	PLUS, MINUS, STAR, SLASH, PERCENT	+, -, *, /, %
比较	EQ, NE, LT, GT, LE, GE	==, !=, <, >
逻辑	AND, OR, NOT	&&, ||, !
分隔符	LPAREN, RPAREN, LBRACE, RBRACE, SEMICOLON, ...	( ) { } ;

共定义 **45** 种 Token 类型

Token 数据结构

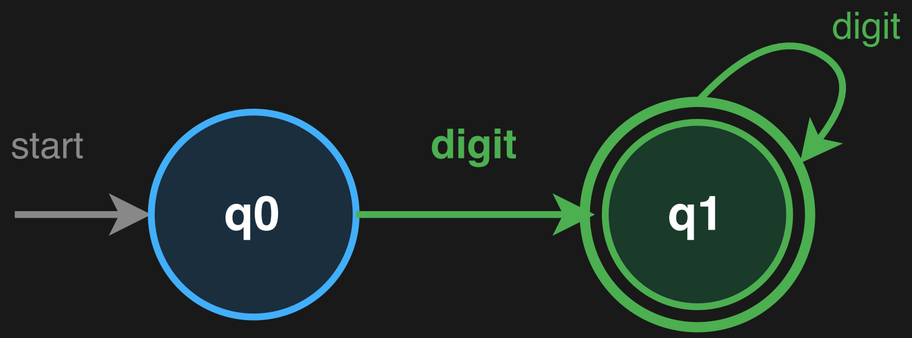
```
@dataclass
class Token:
    type: TokenType     # Token 类型
    value: Any          # 实际值
    line: int           # 行号（错误定位）
    column: int         # 列号（错误定位）

# 示例
Token(TokenType.INT_LITERAL, 42, line=3, col=10)
Token(TokenType.IDENTIFIER, "factorial", line=5, col=1)
```

词法分析器实现

```
class Lexer:
    def get_next_token(self) -> Token:
        while self.current_char:
            # 1. 跳过空白和注释
            if self.current_char.isspace():
                self.skip_whitespace()
                continue
            if self.current_char == '/' and self.peek() == '/':
                self.skip_comment()
                continue

            # 2. 识别数字
            if self.current_char.isdigit():
                return self.read_number()

            # 3. 识别标识符/关键字
            if self.current_char.isalpha():
                return self.read_identifier()

            # 4. 识别运算符
```

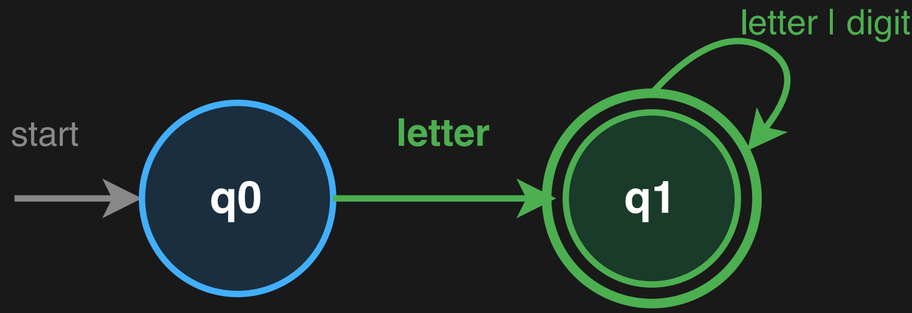
理论基础：有限自动机

识别整数的 DFA



正则：[0-9]+

识别标识符的 DFA



正则：[a-zA-Z][a-zA-Z0-9]*

○ 单圈 = 普通状态　◎ 双圈 = 接受状态

05

语法分析

Syntax Analysis / Parsing

语法分析的作用

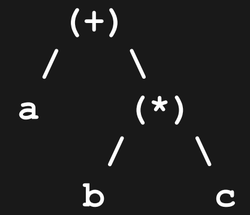
> 将 **Token 流** 按语法规则组织成 **抽象语法树 (AST)**

💡 类比：把单词按照语法规则组成句子结构（主语+谓语+宾语）

什么是 AST?

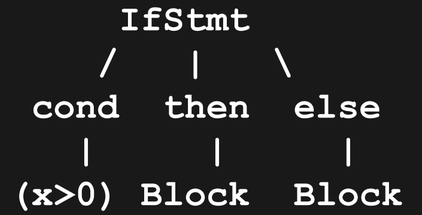
表达式：`a + b * c`

AST 表示优先级关系：



if 语句：

AST 表示控制流结构：



EBNF 语法规范

```
program     = { function_decl } ;
function    = "func" ID "(" params ")" "->" type block ;
block       = "{" { statement } "}" ;
statement   = var_decl | if_stmt | while_stmt | return_stmt | expr_stmt ;

(* 表达式优先级：低 → 高 *)
expression  = logic_or ;
logic_or    = logic_and { "||" logic_and } ;
logic_and   = equality { "&&" equality } ;
equality    = comparison { ("==" | "!=") comparison } ;
comparison  = term { ("<" | ">" | "<=" | ">=") term } ;
term        = factor { ("+" | "-") factor } ;
factor      = unary { ("*" | "/" | "%") unary } ;
unary       = ("!" | "-") unary | call ;
call        = primary [ "(" args ")" ] ;
primary     = INT | FLOAT | BOOL | ID | "(" expression ")" ;
```

递归下降分析法

核心思想：每条语法规则 = 一个函数

```
def parse_term(self) -> Expression:
    """term = factor { ("+" | "-") factor}"""
    left = self.parse_factor()      # 先解析更高优先级

    while self.check(PLUS, MINUS):  # 循环处理同优先级
        op = self.advance()
        right = self.parse_factor()
        left = BinaryExpr(op, left, right)

    return left
```

✅ 自顶向下分析 ✅ LL 分析器 ✅ 简洁直观

运算符优先级处理

优先级	运算符	解析函数	结合性
1 (最低)	||	parse_or()	左结合
2	&&	parse_and()	左结合
3	== !=	parse_equality()	左结合
4	< > <= >=	parse_comparison()	左结合
5	+ -	parse_term()	左结合
6	* / %	parse_factor()	左结合
7 (最高)	! - (一元)	parse_unary()	右结合

通过函数调用层级自然实现优先级

AST 节点设计

```
# 表达式节点
@dataclass
class BinaryExpr(Expression):    # a + b
    operator: str
    left: Expression
    right: Expression

@dataclass
class CallExpr(Expression):      # func(a, b)
    callee: str
    arguments: List[Expression]

# 语句节点
@dataclass
class IfStmt(Statement):
    condition: Expression
    then_branch: Block
    else_branch: Optional[Block]

@dataclass
```

LL vs LR 分析

	LL 分析（本项目）	LR 分析
方向	自顶向下	自底向上
推导	最左推导	最右推导的逆
实现	递归下降	移进-归约
工具	手写	yacc/bison
能力	较弱	更强大
复杂度	简单	复杂

06

语义分析

Semantic Analysis

语义分析的作用

> 检查程序的 **逻辑正确性**（语法正确 ≠ 语义正确）

```
// 语法正确，但语义错误的例子
int x = "hello";          // ❌ 类型不匹配
y = 10;                   // ❌ 变量 y 未声明
add(1);                   // ❌ 参数数量不对
func foo() -> int { }     // ❌ 缺少 return 语句
```

检查内容

检查项	说明	示例
类型检查	表达式类型兼容性	int + float → float
变量声明	使用前必须声明	print(x); // x 未定义
重复声明	同一作用域不能重复	let x = 1; let x = 2;
函数调用	参数数量和类型	add(1, 2) vs add(1)
返回类型	与声明一致	func foo() -> int { return "x"; }

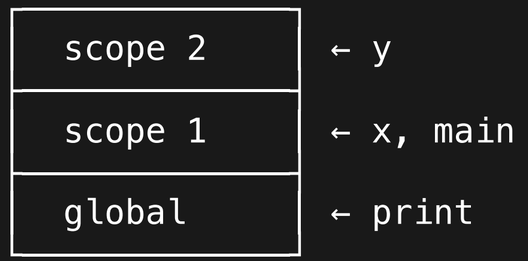
符号表设计

```
@dataclass
class Symbol:
    name: str           # 符号名
    kind: SymbolKind    # VARIABLE / FUNCTION / PARAMETER
    data_type: str      # int / float / bool
    # 函数额外信息
    params: List[Symbol] = []
    return_type: str = ""

class SymbolTable:
    def define(self, symbol: Symbol):
        """在当前作用域定义符号"""

    def lookup(self, name: str) -> Symbol:
        """向上查找符号（包括父作用域）"""

    def enter_scope(self):  # 进入新作用域（函数/块）
    def exit_scope(self):   # 退出当前作用域
```

作用域管理

```
func main() -> int {     // scope 1
    let x: int = 10;
    if (x > 0) {         // scope 2
        let y: int = 20;
        print(x);        // ✅ x 可见
        print(y);        // ✅ y 可见
    }
    print(x);            // ✅ x 可见
    print(y);            // ❌ y 已不可见
}
```

作用域栈：



类型系统

类型	LLVM 类型	大小	默认值
int	i32	32 位	0
float	float	32 位	0.0
bool	i1	1 位	false
void	void	-	-

隐式类型转换： int → float（自动提升）

07

代码生成

Code Generation (LLVM IR)

LLVM IR 简介

> LLVM IR 是一种 **强类型**、**SSA 形式** 的中间表示

- **强类型**：每个值都有明确类型（i32, float, ...）
- **SSA**：Static Single Assignment，每个变量只赋值一次
- **三地址码**：操作最多三个操作数

代码对比

Simple-C 源代码：

```
func add(a: int, b: int) -> int {
    let sum: int = a + b;
    return sum;
}
```

生成的 LLVM IR：

```
define i32 @add(i32 %a, i32 %b) {
entry:
  %sum = alloca i32
  %0 = add i32 %a, %b
  store i32 %0, i32* %sum
  %1 = load i32, i32* %sum
  ret i32 %1
}
```

指令对照表

Simple-C	LLVM IR	说明
a + b	add i32 %a, %b	整数加法
a - b	sub i32 %a, %b	整数减法
a * b	mul i32 %a, %b	整数乘法
a / b	sdiv i32 %a, %b	有符号除法
a < b	icmp slt i32 %a, %b	有符号小于比较
let x = ...	%x = alloca i32	栈上分配
x = val	store i32 %val, i32* %x	存储值
读取 x	load i32, i32* %x	加载值
return x	ret i32 %x	函数返回

控制流生成

if 语句：

```
if (x > 0) {
    y = 1;
} else {
    y = 2;
}
```

LLVM IR：

```
  %cmp = icmp sgt i32 %x, 0
  br i1 %cmp, label %then, label %else
then:
  store i32 1, i32* %y
  br label %endif
else:
  store i32 2, i32* %y
  br label %endif
endif:
  ; 继续执行...
```

代码生成器实现

```
class LLVMCodeGenerator(ASTVisitor):
    def visit_binary_expr(self, node: BinaryExpr):
        left = self.visit(node.left)    # 递归生成左操作数
        right = self.visit(node.right)  # 递归生成右操作数

        if node.operator == '+':
            return self.builder.add(left, right)
        elif node.operator == '-':
            return self.builder.sub(left, right)
        elif node.operator == '*':
            return self.builder.mul(left, right)
        elif node.operator == '<':
            return self.builder.icmp_signed('<', left, right)
        # ...
```

JIT 执行引擎

```
class JITEngine:
    def execute(self, llvm_ir: str) -> int:
        # 1. 解析 LLVM IR
        module = binding.parse_assembly(llvm_ir)
        module.verify()  # 验证正确性

        # 2. 创建执行引擎
        engine = binding.create_mcjit_compiler(module, tm)

        # 3. 获取 main 函数并执行
        main_ptr = engine.get_function_address("main")
        main_func = CFUNCTYPE(c_int)(main_ptr)
        return main_func()  # 调用执行!
```

08

运行演示

Live Demo

命令行接口

```
# 编译并执行（默认）
python -m src.main examples/factorial.sc

# 查看词法分析结果（Token 流）
python -m src.main examples/factorial.sc --tokens

# 查看语法分析结果（AST）
python -m src.main examples/factorial.sc --ast

# 输出 LLVM IR
python -m src.main examples/factorial.sc --emit-llvm

# 输出到文件
python -m src.main examples/factorial.sc --emit-llvm -o output.ll
```

测试结果

示例程序	功能	预期输出	状态
factorial.sc	递归阶乘	120 (5!)	✅ 通过
fibonacci.sc	斐波那契数列	0,1,1,2,3,5,8,13,21,34	✅ 通过
hello.sc	变量与输出	42	✅ 通过
arithmetic.sc	算术运算	13,7,30,3,1,0,1,15	✅ 通过

所有测试用例在 JIT 执行下均通过 ✅

错误处理演示

```
// 词法错误
$ python -m src.main error.sc
Lexer Error at line 3, column 5: Unexpected character: '@'

// 语法错误
$ python -m src.main error.sc
Parse Error at line 5, column 10: Expected ';', got RBRACE

// 语义错误
$ python -m src.main error.sc
Semantic Error at line 7, column 1: Undefined variable 'x'
Semantic Error at line 10, column 5: Type mismatch: expected 'int', got 'float'
```

🚀 交互式 Web IDE

为了更好的演示效果，项目集成了完整的 **Web 端集成开发环境**

进入 Web IDE 演示 →

✓ 语法高亮　✓ 交互式 AST　✓ 多标签页　✓ 即时编译运行

09

项目亮点

Highlights

核心技术亮点

工业级 IR

使用 LLVM IR
而非简单三地址码

可执行验证

JIT 即时编译
直接运行程序

模块化设计

各阶段独立
易于维护扩展

设计模式应用

模式	应用场景	优势
访问者模式	AST 遍历	新增操作无需修改节点类
工厂方法	Token 创建	统一 Token 构造
组合模式	AST 节点	统一处理叶子/组合节点

与课程知识的对应

课程知识点	项目实现
正则表达式 & DFA	词法分析器的 Token 识别
上下文无关文法 (CFG)	EBNF 语法规范
自顶向下分析 (LL)	递归下降解析器
属性文法 & 类型系统	语义分析的类型检查
符号表 & 作用域	SymbolTable 实现
中间代码 & SSA	LLVM IR 生成

10

总结与展望

Conclusion

项目成果

阶段	实现方法	输出	状态
词法分析	手写扫描器	Token 流	✅
语法分析	递归下降	AST	✅
语义分析	访问者 + 符号表	类型信息	✅
代码生成	llvmlite	LLVM IR	✅
执行	JIT	程序输出	✅

收获与体会

- 💡 深入理解了编译器工作原理
- 💡 掌握了递归下降语法分析技术
- 💡 学会了符号表和类型系统设计
- 💡 了解了 LLVM 工业级编译器框架
- 💡 体会了模块化软件设计的重要性

未来展望

- 🔮 支持更多数据类型（数组、结构体）
- 🔮 增加代码优化 Pass
- 🔮 生成独立可执行文件
- 🔮 增加更丰富的标准库
- 🔮 支持模块/包系统

Thank You!

Q & A

李易涵

302023315218

Simple-C Compiler

Computer Science

Designed with ♥ by **Li Yihan**