# 编译器设计文档

**参考编译器介绍**

- **总体结构**: 参考编译器采用经典的前中后端划分：`frontend`（词法、语法、语义）、`midend`（中间表示、优化、符号表）、`backend`（寄存器分配、目标代码生成）。顶层入口由`Compiler.java`/`FrontEnd.java`/`MidEnd.java`/`BackEnd.java`等协调。
- **接口设计**: 各阶段通过明确的数据结构和接口进行解耦：词法器输出`Token`流，解析器构建`AST`，语义分析/符号表在`AST`上附加属性并产生中间表示（IR），优化器以 IR 为输入并产生优化后的 IR，后端读取 IR 生成目标代码。每个模块以少量公共类型（Token、AST 节点基类、Symbol、IR 节点）作为契约。
- **文件组织**: 参考实现把前端、midend、后端分别放在子目录下，且各子目录内部再细分子模块（如`lexer`、`parser`、`ast`、`llvm`、`mips`）。工具类和错误处理（`ErrorRecorder`、`ErrorType`）独立成包，便于所有阶段调用。

**编译器总体设计**

- **总体模块划分**
  - `frontend`：`lexer/`、`parser/`、`ast/`、`semantic/`。
  - `midend`：`irgen/`、`optimize/`（子模块：SSA、Mem2Reg、常量传播、死代码消除、支配分析）和`symbol/`。
  - `backend`：`MipsGenerator.*`（目标指令选择、指令选择模板、寄存器分配接口、调用约定实现）。
  - `error/`：统一错误记录与报告（`ErrorRecorder`）。
- **主要接口契约**
  - 词法器：提供`Token nextToken()`与`peek()`，可重置输入位置，错误通过`ErrorRecorder`报告。
  - 解析器：基于递归下降，暴露`parse()`返回`AST*`。
  - 语义分析器：遍历`AST`，维护符号表栈，接口`void analyze(AST*)`，在 AST 上标注类型/偏移/属性并生成中间表示或调用 IR 生成器。
  - IR 接口：定义统一的中间指令与函数/基本块结构，支持向后端或 LLVM 后端输出。优化器接受`IRModule`并按 pass 顺序变换它。
  - 后端：消费`IRModule`，接口`emit(Module, OutputStream)`，内部使用寄存器分配器与指令选择器。
- **文件组织**
  - `src/frontend/lexer/`：词法相关源文件。
  - `src/frontend/parser/`：语法分析器与语法驱动代码。
  - `src/frontend/ast/`：AST 节点定义与打印/序列化工具。
  - `src/midend/irgen/`：从 AST->IR 的生成。
  - `src/optimize/`：各优化 pass 文件（如`DominanceAnalysis.cpp`、`Mem2RegPass.cpp`、`Optimizer.cpp`）。
  - `src/backend/`：`MipsGenerator.cpp/hpp`等。
  - `src/error/`：`ErrorRecorder.cpp/hpp`、`ErrorType.hpp`。

**词法分析设计**

- 编码前的设计：
  - 使用状态机（手写 DFA）实现高效的流式扫描，词法单元结构包含 `type`、`lexeme`、`line`、`column`、可选 `literalValue`。支持回退和 peek。注释、空白、换行被正确处理以更新位置信息。
  - 错误策略：遇到非法字符或不完整字面量（如未闭合字符串）调用 `ErrorRecorder` 记录并尝试同步到下一个可识别的分隔符（如分号或换行）。
  - 测试用例：覆盖关键字、标识符、数字字面量、字符串、运算符以及注释边界条件。
- 编码完成后的修改：
  - 实现情况：词法器为手写扫描器（`frontend/lexer/lexer.cpp`），实现了数字、字符串、注释、关键字识别及复合运算符；提供 `GenerateTokenList()` 与 `TokenStream`（`TokenStream.hpp`）用于 peek/回溯，满足编码前设计的核心契约。
  - 差异与改进：实现中未记录列号（仅记录行号），字符字面量被标为 `ERROR_T`；对单字符不完整逻辑运算符 `&`/`|` 会记录非法符号错误但仍产生可让解析继续的 token。

**语法分析设计**

- 编码前的设计：
  - 优先采用 LL(1)/递归下降。若文法存在左递归或二义性，先做改写（消除左递归，提取左公因子）。
  - AST 构造：每个语法产生式对应 AST 节点构造逻辑，节点类型包括 `FunctionDecl`、`VarDecl`、`BinaryExpr`、`IfStmt`、`WhileStmt` 等，并保留源码位置用于错误定位和后续优化/代码生成。
  - 错误恢复：实现同步策略并记录语法错误以尽量继续解析以报告更多错误。
- 编码完成后的修改：
  - 实现情况：语法分析器为递归下降实现（`frontend/parser/Parser.cpp`），按文法构造明确的嵌套 AST 节点，并实现了语法错误检测与定位（通过 `ErrorRecorder`）；对左递归的表达式采用手工构造嵌套节点来保留文法形态。
  - 差异与改进：实现没有采用 Pratt 解析器，但通过显式的子节点嵌套正确处理表达式优先级；解析器在缺失分隔符时会记录错误并尽量同步，满足原设计中的错误恢复目标。

**语义分析设计**

- 编码前的设计：
  - 符号表：实现基于栈的符号表（支持嵌套作用域），`Symbol` 记录名字、类别（变量/函数/类型）、类型信息、偏移/存储位置。
  - 类型检查：实现表达式与语句的类型检查规则，支持基础类型与数组/函数类型，检查函数调用签名与参数匹配。
  - 语义动作：对变量的生命周期、作用域、声明-使用检查、未初始化使用和重定义进行检测并记录错误。
  - 与 IR 的连接：语义分析阶段负责将高层语义信息传递给 IR 生成器。
- 编码完成后的修改：

- 实现情况：`midend/analysis/SemanticAnalyzer.cpp` 实现了基于作用域栈的符号管理（配合 `SymbolManager`）、函数/参数检查、未定义名检测、常见语义错误（赋值给常量、break/continue 使用位置、printf 参数匹配、缺失 return）的完整检测与报告；并能将符号表导出为 `symbol.txt`，符合设计目标。
- 差异与改进：语义分析已在一定程度上支持常量求值（`IRGenerator::evaluateConstExp` 依赖符号信息），但更高级的语义阶的常量传播/折叠主要在 IR 层进行；符号查找已基于符号表实现并能满足当前测试规模。

**代码生成设计**

- 编码前的设计：
  - 目标架构：选择 MIPS（参考已有 `MipsGenerator.cpp`）或保留生成 LLVM IR 的后端以便复用 LLVM 的寄存器分配/优化。
  - 指令选择：采用模板或模式匹配将 IR 指令映射到目标指令序列，明确调用约定（参数传递、返回值、栈帧布局、保存寄存器规则）。
  - 寄存器分配接口：设计一个抽象 `RegisterAllocator`，用线性扫描或图着色算法实现。支持溢出（spill）到栈。
  - 输出格式：支持文本汇编输出与可选二进制/汇编器脚本。
- 编码完成后的修改：
  - 实现情况：IR 已由 `midend/irgen/IRGenerator.cpp` 产出，后端 `backend/MipsGenerator.cpp` 直接消费 IR 生成 MIPS 汇编；后端实现了栈帧布局、参数传递（前 4 个参数放 `$a0-$a3`，其余入栈）、常用内建函数映射与系统调用，以及基于 `stackOffsets` 的临时与局部变量分配。
  - 差异与改进：原计划中的抽象 `RegisterAllocator` 未单独实现；当前后端通过将临时/局部值映射到栈槽并在需要时载入寄存器来生成代码（即没有独立的图着色或线性扫描寄存器分配器）。调用约定与栈对齐为手工实现并已适配常见用例，但尚未处理浮点寄存器或复杂 SIMD/平台特性。

**代码优化设计**

- 编码前的设计：
  - Pass 管理器：实现统一的 `PassManager`，按序对 `IRModule` 执行多个优化 pass（如 `Mem2Reg`、`ConstProp`、`DeadCodeElim`、`CommonSubexprElim`、`DominanceAnalysis`）。
  - SSA/寄存器提升：实现 `Mem2Reg` 将内存局部变量提升为 SSA 寄存器值，减少内存访问。
  - 数据流分析：实现支配树、活性分析、可达性分析等为下游优化提供信息。
  - 可配置性：允许按命令行/配置文件选择启用的 pass 与优化级别。
- 编码完成后的修改：
  - 实现情况：存在 `optimize/Optimizer.cpp` 管理 pass 的执行，且已实现 `DominanceAnalysis` 与 `Mem2RegPass`（完整的 phi 插入、重命名与移除 load/store），表明 SSA 构造与寄存器提升（promote allocas to registers）工作流已经实现并与 IR 对接。
  - 差异与改进：目前实现的 pass 主要集中在 Mem2Reg/支配分析，尚未找到完整实现的常量传播、死代码消除或循环不变代码外提等 pass（这些可以随后加入并注册到 `Optimizer`）。

**测试与验证**

- 单元测试：为词法、语法、语义各模块建立单元测试（输入代码片段 -> 预期 tokens / AST / 错误列表）。

- 集成测试：使用仓库中的 `test/` 目录下样例作为回归测试，确保从源代码到生成代码的端到端正确性。