1. 技术选型
2. 编程语言：以Rust语言为主语言，利用其所有权模型确保内存安全，降低实验级编译器的调试复杂度。
3. 前端：支持Flex\Bison用于快速词法语法分析
4. llvm集成：通过inkwell库（Rust的LLVM安全绑定）或llvm-sys直接调用LLVM C API，生成优化中间代码（IR）,进行毕昇LLVM优化
5. 目标架构：优先支持risicv,扩展支持LoongArch
6. 工具配置
7. Rust工具链：nightly版本（支持FFI与LLVM绑定）
8. LLVM版本：毕昇编译器分支（需手动编译，启用RISCV后端）
9. RISC-V工具链：riscv-gnu-toolchain（GCC+Binutils）
10. 技术路线

核心模块分为前端、中端和后端三部分：

1. 前端

* 词法分析（Flex）：将源代码转换为Token流。
* 语法分析（Bison）：根据Token流构建语法树（AST），定义文法规则。
* FFI构建：将Bison产生的C结构体转换为Rust的AST结构，用于后续语义分析，或者使用bindgen从C头文件自动生成Rust的结构体与函数声明
* 语义分析（Rust）：遍历AST，检查变量定义、类型匹配、作用域规则。

1. 中端

* 中间层（毕昇）：Rust与LLVM绑定，使用llvm-sys，手动编译毕昇LLVM分支
* LLVM IR生成
* 优化pass：实现定制化优化（如针对RISC-V指令集的冗余操作消除），重点在于消除RISC-V无用的符号扩展，向量化循环优化，压缩指令模式选择（C扩展指令替换）

1. 后端

* 目标架构（risicv）：RV64GC标准扩展（保留LoongArch扩展）
* 目标三元组设置：LLVM的目标三元组，risicv64 linux-gnu
* 扩展支持：RISIC-V的-march，LLVM的features字符串
* 选择逻辑：通过TableGen定义模式匹配规则,手动优化相关场景以及后续的自定义指令
* 寄存器分配：分为整数寄存器、浮点寄存器、向量寄存器以及特殊寄存器，制定相关调用约定
* 代码生成：原子操作的生成，压缩指令的优化，向量指令代码生成
* LoongArch扩展计划：复用RISC-V后端框架，适配LoongArch的指令编码和ABI。基于LLVM的MC Layer实现自定义指令编码和调度策略。

1. 技术难点（部分）
2. Flex/Bison与Rust的FFI桥接：核心问题即为C指针到Rust的安全转换问题
3. LLVM版本兼容性：毕昇llvm分支与官方API的差异
4. RISIC-V的V扩展支持：LLVM官方并未稳定支持向量指令，需手动扩展