SCompiler

项目整体结构概览

SCompiler 项目包括**编译器**和**虚拟机**两大部分，源代码按功能划分在不同包中。编译器部分负责将源代码转换为中间表示和可执行的指令序列；虚拟机部分负责加载并解释执行这些指令。整体结构如下：

- **编译器部分（org.jcnc.snow.compiler）**：包含词法分析(`lexer` 子包)、语法分析(`parser` 子包)、语义分析(`semantic` 子包)、中间表示IR构建(`ir` 子包)、后端代码生成(`backend` 子包)以及命令行接口(`cli` 子包)。主要流程由 `SnowCompiler` 类驱动，按顺序调用各阶段模块完成从源代码到字节码的转换。
- **虚拟机部分（org.jcnc.snow.vm）**：实现自定义字节码的运行时支持。主要包括虚拟机引擎(`engine` 包)、指令集定义(`commands` 包)、执行模块(`execution` 包)、运行时数据结构(`module` 包，如操作数栈、调用栈、栈帧等)和辅助工具(`factories` 用于指令工厂， `utils` 提供日志等)。核心类 `VirtualMachineEngine` 负责载入由编译器生成的指令序列并逐条执行，借助 `VMCommandExecutor` 调度具体 `Command` 指令处理器完成运算。每种字节码指令在 `commands` 子包中实现为一个类，例如算术指令 `IAddCommand` （32位整数加法）、 `LAddCommand` （64位整数加法）等。

编译器部分结构

编译器前端按经典编译流程组织：

- **词法分析器（Lexer）**：由 `lexer.core.LexerEngine` 实现，负责读取源代码文本，将其拆解为有序的记号（Token）序列。LexerEngine 在构造时初始化一组 `TokenScanner` 子扫描器，每个扫描器识别一种类别的Token。例如：
  - `WhitespaceTokenScanner` 跳过空白字符（空格、制表符等）。
  - `CommentTokenScanner` 跳过注释文本。
  - `IdentifierTokenScanner` 识别标识符和关键字（例如关键字 `module` 、 `function` 、 `if` 、 `return` 、 `declare` 等）。
  - `NumberTokenScanner` 识别数字字面量（整数、浮点数）。
  - `StringTokenScanner` 识别字符串字面量。
  - `OperatorTokenScanner` 识别运算符符号（如 `+` ， `-` ， `*` ， `/` ， `==` ， `!=` 等）。
  - `SymbolTokenScanner` 识别分隔符和符号（如逗号、括号、冒号、换行等）。

  LexerEngine 按优先级运行上述扫描器，对源代码逐字符扫描，生成Token列表。输出的Token序列会附带类型和词素（lexeme）信息，并在代码中保留行列位置以便错误报告。扫描完成后还会在末尾插入文件结束标记 `EOF` 。

- **语法分析器（Parser）**：由 `parser.core.ParserEngine` 驱动，依据语言文法将Token序列解析为抽象语法树（AST）。ParserEngine 负责识别顶层结构并调用相应解析器：
  - **模块解析**： `ModuleParser` 处理 `module` 模块块。语法类似 `module 模块名`：开始，缩进块内包含多个函数定义，以缩进或显式 `end module` 标志结束模块。
  - **导入解析**： `ImportParser` 处理 `import` 声明，引入其它模块（目前仅语法支持，实际链接有限）。
  - **函数解析**： `FunctionParser` 处理函数定义。函数定义以关键字 `function` 起始，例如 `function foo(x: Int, y: Int): Int` ，后跟缩进的函数体，函数体由若干语句组成，函数可在缩进结束处结束（可能通过撤销缩进或 `end function` 标记）。函数头部包括函数名、参数列表（由 `ParameterNode` 表示每个形参及类型）、返回类型注释等。
  - **语句解析**：函数体内部，Parser使用 `StatementParserFactory` 根据每条语句起始的关键字选择相应的语句解析器：
    - `IfStatementParser` 解析条件语句（ `if ... else ...` 结构）。
    - `LoopStatementParser` 解析循环语句（ `loop ...` 结构，类似 `while` 或通用循环块）。
    - `ReturnStatementParser` 解析返回语句（ `return` 语句）。
    - `DeclarationStatementParser` 解析变量声明语句（ `declare` 开头的声明）。
    - 对于未匹配上述关键字的情况，默认使用 `ExpressionStatementParser` ，将整行解析为表达式语句。 `ExpressionStatementParser` 内部还处理赋值语句的特殊情况：如果行首是标识符且紧随 `=` ，则识别为赋值语句，将其解析为 `AssignmentNode` （左侧标识符和右侧表达式）。
  - **表达式解析**：对于语句中的表达式部分，编译器采用 Pratt 算法（运算符优先解析）。定义了一系列前缀和中缀解析子器(parselet)：例如 `NumberLiteralParselet` 处理数字字面量， `StringLiteralParselet` 处理字符串， `IdentifierParselet` 处理变量引用， `GroupingParselet` 处理括号表达式， `CallParselet` 处理函数调用（识别一对括号表示调用）， `MemberParselet` 处理成员访问（识别点号 `.` ）， `BinaryOperatorParselet` 根据运算符优先级解析二元运算符表达式等【7†】。Parser 根据定义的运算符优先级(`Precedence` 类)解析出嵌套表达式树。例如，一个算术表达式会被解析成由 `BinaryExpressionNode` 节点连接的AST子树。

语法分析结果是一个AST节点列表，通常整个源文件解析成一个 `ModuleNode` 根节点，其中包含模块名、导入列表、函数定义列表等子节点。如果在词法或语法阶段遇到非法输入，将抛出带有错误位置和原因的异常，中止编译。

- **语义分析器（Semantic Analyzer）**：语法阶段产生AST后，进入 `semantic` 包的语义检查阶段。 `SemanticAnalyzerRunner` 类会遍历AST节点，对各种语义规则进行验证，包括标识符引用解析、类型检查、函数签名检查等。主要组件有：
  - 符号表（ `SymbolTable` 类）：用于维护当前作用域内的符号（变量、函数、模块等）的信息。SymbolTable 支持嵌套作用域，每个符号具有名称、种类(`SymbolKind` ，区分变量、函数等)、类型(`Type`)等属性。语义分析过程中会根据作用域进入/离开情况创建或销毁符号表，例如进

入函数时新建函数级符号表，离开时销毁。对变量声明，分析器会将其添加到当前符号表；对标识符引用，则沿着当前作用域向上查找符号定义以解析引用是否有效。

- 内置类型系统： `BuiltinType` 枚举定义了语言支持的所有基础类型，包括**BYTE**(8位整数)、**SHORT**(16位整数)、**INT**(32位整数)、**LONG**(64位整数)、**FLOAT**(单精度浮点数)、**DOUBLE**(双精度浮点数)、**STRING**(字符串)、**VOID**(空类型，用于无返回值函数)【14†】。每个类型实现了 `Type` 接口并提供辅助方法，例如判断数值类型、类型兼容性检查(`isCompatible`)以及自动宽化转换（用于处理算术中小范围整数提升等）。
- 语义分析具体规则由各 `Analyzer` 类实现，分为表达式和语句两大类分析器：
  - **表达式分析器**：如 `BinaryExpressionAnalyzer` 检查二元运算两侧操作数类型是否兼容（例如算术运算要求操作数为数值类型，比较运算要求两侧类型可比等）， `CallExpressionAnalyzer` 检查函数调用的参数类型和数量是否匹配被调用函数签名， `IdentifierAnalyzer` 确保标识符已声明且在可见作用域内， `NumberLiteralAnalyzer` 和 `StringLiteralAnalyzer` 主要标注字面量的类型等。对于暂未支持的表达式形式，会使用 `UnsupportedExpressionAnalyzer` 占位或报告错误。
  - **语句分析器**：如 `DeclarationAnalyzer` 在变量声明时将新符号登记到符号表并检测重复定义， `AssignmentAnalyzer` 检查赋值目标是否已声明且类型匹配， `IfAnalyzer` 和 `LoopAnalyzer` 检查条件表达式类型是否正确（例如应该是布尔或可转换为布尔的值；目前由于没有独立布尔类型，可能规定为整数非零即真），并递归分析其内部语句块； `ReturnAnalyzer` 检查返回语句是否在函数内以及返回值类型是否与函数声明的返回类型一致。如果语义检查发现错误，会通过 `SemanticError` 机制记录错误信息（如“不匹配的类型”“未定义的变量”等），通常编译器会收集所有语义错误然后停止编译。

经过语义分析后，AST节点将附加必要的类型和符号引用信息，确保后续阶段可以安全地生成代码。

- **中间表示（IR）生成**：在 `compiler.ir` 包中，编译器将经过检查的AST转换为中间代码IR（Intermediate Representation）。IR采用**三地址码**风格的指令序列，抽象地表示计算过程。主要结构包括：
  - `IRValue` 及其子类：表示IR中的操作数，可以是常量(`IRConstant`)、虚拟寄存器(`IRVirtualRegister` ，用于表示中间计算结果)或标签(`IRLabel` ，用于控制流跳转目标)。
  - `IRInstruction` 及子类：表示各类中间指令。如算术运算 `IRAddInstruction` 、函数调用 `CallInstruction` 、条件跳转 `IRJumpInstruction` 、返回 `IRReturnInstruction` 等。这些IR指令通常是三地址形式（ `target = op (operand1, operand2)` ）。
  - IR构建由 `IRProgramBuilder` 驱动。它会遍历AST，针对每个函数定义创建一个 `IRFunction` 来容纳该函数的指令序列。IR生成过程借助多个Builder类： `ExpressionBuilder` 负责表达式节点转换， `StatementBuilder` 负责语句节点转换，二者利用 `InstructionFactory` 创建具体IR指令。例如，对一个变量声明节点，IR可能生成一条初始化赋值指令；对一个加法表达式AST，ExpressionBuilder会生成相应的取操作数指令和加法指令。所有指令追加到当前函数的IR列表中。

当前版本的编译器已实现基本的算术、函数调用等IR构造，但**对控制流语句(如 if/loop)的IR支持尚不完整**。如果IR生成过程中遇到 `IfNode` 或 `LoopNode` ，由于 `StatementBuilder` 没有相应处理逻辑，将触发默认分支抛出 “Unsupported statement” 异常【33†】。换言之，目前编译器无法将含有条件或循环的代码转换为IR。这是现阶段的一大局限，亟待在将来补充：例如为If节点生成条件判断和跳转的IR指令序列（类似三地址码的条件跳转），为Loop节点生成循环初始化、条件判断、跳转回loop开头等IR。项目中已经有基础的 `IRJumpInstruction` 等类，但尚未整合进高层语言结构的IR生成。

- **后端代码生成**：IR生成后，编译器后端(`compiler.backend` 包)会把IR转换为虚拟机可执行的字节码指令序列。主要步骤包括：
  - **寄存器分配**：由于IR中的 `IRVirtualRegister` 是无限的抽象寄存器，需要映射到虚拟机实际可用的局部变量槽位。 `RegisterAllocator` 负责为每个IR函数分配本地变量槽编号。通常采用线性扫描等策略为IR函数内的虚拟寄存器分配一个较小范围的索引。
  - **指令映射**： `VMCodeGenerator` 根据IR指令生成对应的虚拟机指令序列。通过 `IROpCodeMapper` 将IR操作码翻译为虚拟机操作码（ `VMOpCode` 枚举定义了虚拟机支持的操作码）。例如，一个IR加法指令会被映射为虚拟机的加载操作(`LOAD`)、加法操作(`ADD`)和保存结果操作(`STORE`)等一系列指令。又如函数调用IR会映射为虚拟机的 `Call` 指令，在生成时还需处理被调函数地址：如果被调函数在当前编译单元内，可能立即确定地址；否则留待链接或运行时通过 `CommandLoader` 解析。后端生成的结果通常是每个函数对应一段指令序列以及入口点信息，汇总为可装载到虚拟机的程序表示(`VMProgramBuilder` 可能负责构建最终指令列表)。

最终，编译器将产出包含虚拟机指令的程序对象，准备交由虚拟机执行。

虚拟机部分结构

SCompiler 的虚拟机（位于 `org.jcnc.snow.vm` 包）是一个**基于栈的解释器**，模拟执行编译器产生的指令。主要模块包括：

- **虚拟机引擎（VirtualMachineEngine）**：虚拟机的核心执行循环位于 `vm.engine.VirtualMachineEngine` 类中。它负责初始化运行时状态（如创建操作数栈、调用栈、全局/本地变量空间等），加载编译器生成的指令列表，然后进入主循环取指令执行。每条指令通过 `VMCommandExecutor` 分发给相应的 `Command` 处理类执行。VirtualMachineEngine 支持不同运行模式(`VMMode`)，并提供必要的调试输出（通过 `VMStateLogger` 记录执行过程）。
- **指令集及执行**：指令以**操作码（OpCode）+ 可选操作数**的形式表示，项目中每种指令实现为一个 `Command` 接口的实现类，位于 `vm.commands` 子包下，并按照功能分类：
  - **栈操作**：如 `Push` 压栈指令、 `Pop` 出栈、 `Dup` 复制栈顶、 `Swap` 交换栈顶等。其中又细分为不同数据类型版本，例如 `IPushCommand` / `LPushCommand` 分别将32位/64位整数压入栈， `BPushCommand` 压入字节，等等。虚拟机通过这些指令管理操作数栈上的数据。
  - **算术运算**：包括各整数类型（8/16/32/64位）和浮点类型的加减乘除、取模、取负、增量等。如 `IAddCommand` 、 `FSubCommand` 、 `LNegCommand` 、 `SDivCommand` 等等，每个类封装对应类型的运

算逻辑。还有类型转换指令在 `arithmetic.conversion` 包中，例如 `I2FCommand` （Int转Float）、 `F2DCommand` （Float转Double）等，实现数值类型之间的自动转换。
    - **按位运算**：如按位与、或、异或，当前实现了32位和64位整数的相应指令： `IAndCommand` / `LAndCommand` 、 `IOrCommand` / `LOrCommand` 、 `IXorCommand` / `LXorCommand` 。
    - **内存读写**：这里指访问本地变量槽位的指令，在 `commands.memory` 包下。如 `ILoadCommand` / `IStoreCommand` 用于读取/写入32位整数到局部变量， `FLoadCommand` / `FStoreCommand` 用于浮点数，等等。通用的 `MovCommand` 可能用于将栈顶数据移动到特定槽位或从槽位移动到栈顶。通过这些指令，虚拟机可以访问模拟的“栈帧”中的局部变量区域（类似JVM的局部变量表）。
    - **控制流**：包括无条件跳转 `JumpCommand` 以及条件分支跳转。目前仅针对32位整数实现了比较跳转指令，如 `ICECommand` （Int Compare Equal，比较两个整数若相等则跳转）、 `ICLCommand` （Int Compare Less，小于则跳转）等六种比较操作（==, !=, <, <=, >, >=）。这些指令通常由编译器在实现条件语句时利用（不过目前编译器尚未完成相应生成）。
    - **函数调用**： `CallCommand` 和 `RetCommand` 用于函数调用和返回。调用指令会处理新的栈帧的创建、参数传递和跳转至目标函数地址，返回指令则清理栈帧、恢复调用点等。
    - **虚拟机控制**： `commands.vm` 子包下可能包含虚拟机级别的指令，如 `HaltCommand` 用于停止程序执行等。
- **运行时数据结构**： `vm.module` 包定义了虚拟机执行所需的数据结构：
    - `OperandStack` （操作数栈）：后进先出栈，用于算术运算、函数调用等过程中临时存放操作数和结果，与各指令交互。
    - `LocalVariableStore` （局部变量存储）：模拟每个函数调用帧的本地变量区，通常以数组形式按索引存取。不同类型占据相应大小的槽位。
    - `CallStack` 和 `StackFrame` ：调用栈由一系列栈帧构成，每个 `StackFrame` 保存一次函数调用的上下文（包括局部变量存储、返回地址、上一帧引用等）。 `CallStack` 管理栈帧的压入弹出，实现函数调用链。
    - `MethodContext` ：可能封装一次方法执行过程中的上下文，方便指令访问操作数栈和本地变量等。
- **指令加载与执行流程**：当编译器产出字节码后，通常由 `VMInitializer` / `VMLauncher` 类加载程序。在运行时， `CommandLoader` 根据操作码构造相应 `Command` 对象（通过 `CommandFactory` 工厂映射操作码到具体 `Command` 类）。 `VirtualMachineEngine` 依次取出指令操作码，利用CommandFactory得到命令实例后，调用其执行方法。每个 `Command` 类实现了具体执行逻辑，比如二元运算指令会从OperandStack弹出两个操作数计算结果再压回，Load指令会从当前栈帧的LocalVariableStore取值压入栈顶，Jump指令会修改指令指针等。 `CommandExecutionHandler` 可能维护着指令指针和循环，逐条执行直到遇到 `HaltCommand` 或运行完所有指令。整个过程中 `VMUtils` 和 `LoggingUtils` 可以提供调试和日志支持。

总的来说，虚拟机按照**取指令-译码-执行**的循环运行，被设计为与编译器生成的指令集相契合。其指令集针对静态类型数据进行了划分，实现简单但可满足基本运算需求。

编译器支持的语法结构

当前版本的 SCompiler 编译器支持以下主要的语言元素和语法结构：

- **基本数据类型**：支持多种原生数据类型，包括整数（8位Byte, 16位Short, 32位Int, 64位Long）、浮点数（32位Float, 64位Double）、字符串(String)以及特殊的Void类型。变量和函数定义需要指明类型，编译器在语义分析时会检查类型一致性。
- **变量声明与赋值**：使用关键字 `declare` 进行变量声明，例如： `declare x: INT = 5` 。声明语句包含变量名、类型以及可选的初始化表达式。赋值则通过赋值运算符 `=` 实现，如 `x = x + 1;` 。赋值语句不需要特殊关键字，在语法解析时，如果一行以标识符开头并紧跟 `=` , 则被识别为赋值操作。编译器允许在函数体内声明局部变量（模块层可能也允许全局变量声明），并支持用表达式初始化。此后即可在后续代码中引用该变量。
- **表达式**：支持丰富的表达式类型：
  - 算术运算： `+` , `-` , `*` , `/` , `%` 等算术符，适用于数值类型。还支持一元负号（取反）。
  - 比较运算： `==` , `!=` , `<` , `<=` , `>` , `>=` 等比较符，通常产出一个表示真假的值（目前未明确布尔类型，用整数0/1表示真假）。
  - 字面量：整数（支持不同范围，如 `123` 默认推导为Int或Long等）、浮点数（如 `3.14` 推为Float/Double）、字符串字面量（如 `"hello"` ）。编译器能正确标注字面量类型并将其用作常量。
  - 标识符引用：对已有变量或常量的引用，可参与运算或赋值。
  - 函数调用：使用括号 `()` 调用函数，支持传递实参表达式列表。例如 `result = foo(a + 1, "test")` 。解析器识别函数名后遇到 `(` 即进入调用解析，将参数表达式列表构造成AST的 `CallExpressionNode` 。语义分析会检查被调用函数是否存在以及参数类型是否匹配定义。
  - 成员访问：语法支持点号 `.` 进行成员访问。例如 `moduleName.var` 或将来类的字段访问。目前由于未实现类/结构， `.` 主要用于模块命名空间引用（通过import导入模块后，可用ModuleName.element方式引用其公开符号）。
  - 括号分组：支持使用 `( ... )` 改变运算优先级或作为强制分组，例如 `(a + b) * c` 。括号中的子表达式被视为一个整体节点。
- **控制流**：语言设计包含基本的流程控制结构：
  - 条件分支（if-else）：可以编写 `if 条件:` 开始一个条件块，然后缩进编写分支语句。支持可选的 `else:` 块。语法分析会生成对应的 `IfNode` （含条件表达式、then块语句列表、以及可选的else块语句列表）。注意当前编译器能识别if语法并通过语义检查，但由于IR和虚拟机支持未完善，**含有if的代码无法正确编译执行**（详见后续不足部分）。
  - 循环（loop）：支持类似 `loop 条件:` 或无条件的循环结构（具体语法可能类似 `loop:` 表示无限循环，或 `loop 条件:` 表示当条件为真时循环）。解析产生 `LoopNode` ，其中包含循环条件表达式（或无条件）以及循环体语句列表。与if相似，目前语法和AST层面支持循环，但实际编译执行尚未完全实现。

- **函数定义与调用**：可以定义有参数和返回值的函数。函数定义以关键字 `function` 开始
  函数可以有多个参数（形参列表带类型注解），以及指定返回类型。函数体内部可以声明局部变量、编写控制流等。支持在函数体内使用 `return` 语句返回值或提前退出。编译器对函数调用进行了支持，包括解析调用表达式、检查参数、在IR中生成调用指令以及虚拟机执行函数调用机制。当前实现应允许运行时递归调用等基本功能。**注意**：函数参数在语义分析时会加入函数作用域符号表，并在调用时按值传递。返回类型为 `VOID` 表示不返回值， `return` 语句可省略值。
- **模块与导入**：语言支持将代码组织为模块。每个源文件可以声明 `module` 模块名：作为开头，模块内部可以定义多个函数和（可能的）全局变量，模块结尾以减少缩进（或 `end module` 标示）结束。可以使用 `import` 模块名 导入其他模块，以便调用其函数或使用其定义。当前实现的模块机制主要体现在解析和符号登记上： `ModuleParser` 和 `ImportParser` 能够构建模块AST和导入AST， `ModuleRegistry` 和 `ModuleInfo` 在语义阶段记录模块信息。然而，由于项目规模所限，模块间调用可能还没有完全实现链接（例如可能需要在编译或加载时解析跨模块引用）。但至少语法上模块化是支持的，未来可扩展为多文件编译。
- **其他语句**：包括 `return` 语句用于函数返回， `return` 后可跟表达式作为返回值或在void函数中单独使用。空语句或仅包含表达式求值（比如函数调用而不使用返回值）也是允许的作为Expression Statement。这些都由相应解析器和分析器处理。**注意**：目前没有显式的 `break` / `continue` 用于循环中断，也没有异常抛出/捕获语法。

综上，SCompiler目前提供了一个小型静态类型语言的基本要素：原生类型、变量、表达式、函数、模块、流程控制等核心结构在语法层面大多具备。但是，由于开发进度限制，有些虽然语法上存在但尚未完全支持执行（如if/loop），详见下节。

尚未支持的语法特性

尽管基础已经打好，但与现代编程语言相比，SCompiler当前缺少或未完善以下常见特性：

- **类和面向对象特性**：不支持类(class)、对象、继承、多态等OOP结构。代码中没有类定义语法，因而不能创建自定义复合类型或通过对象调用方法。这限制了用户只能使用过程式的模块和函数来组织代码。
- **更细粒度的作用域**：目前仅有模块和函数级作用域。缺乏块级作用域（Block Scope）的完善支持。例如，在 `if` 或 `loop` 内部声明的变量，按理应只在该块可见，但由于语义实现未明确区分，这些变量可能被当作函数级变量处理。这可能导致作用域规则的不严格，变量名冲突或生命周期管理不够健全。后续需要引入块作用域，使得进入 `if/else` 、循环体时能推入新作用域（SymbolTable），离开时弹出，以符合语言直觉。
- **数组和集合类型**：没有提供数组、列表等集合类型的数据结构及其语法。无法直接声明如 `int[]` 或使用下标访问元素。目前所有类型都是单值的基础类型和字符串，程序员无法方便地存储序列化数据。数组作为最基本的数据结构之一，应当在后续版本中考虑加入，包括其类型表示、字面量语法（如 `[1,2,3]` ）以及相关的IR和指令支持（如元素读取、写入）。

- **模块化的完善**：虽然有模块声明和import语法，但当前实现可能局限于单一编译单元，缺少真正**模块分离编译和链接**的机制。例如，导入的模块是否在编译时解析，还是需要运行时加载，目前不清晰。也没有命名空间管理机制来避免不同模块符号冲突。现代语言通常支持将模块编译为独立单元再链接，SCompiler 需要进一步完善模块系统，使多文件项目的编译成为可能，包括符号导出/导入、依赖解析等。
- **错误处理机制**：没有提供异常处理或其他错误处理语法。例如 `try-catch` 结构在语言中不存在，函数也没有声明抛出异常的概念。这意味着在运行时发生错误只能通过返回错误码或简单停止虚拟机。现代语言基本都有错误处理机制，SCompiler在这方面还是空白。将来可能需要设计 `throw/try/catch` 或者其它错误处理形式，并在虚拟机中支持异常的传播与捕获。
- **布尔类型和逻辑运算**：目前没有独立的Boolean布尔类型。条件判断使用整数代替布尔（非0为真0为假），这虽可行但不够直观和类型安全。另外，逻辑短路运算符如 `&&`，`||`，`!` 等未提及，估计未实现。如果需要编写复杂条件，用户只能使用按位运算或嵌套if代替，语义上不完善。应考虑加入布尔类型和逻辑运算支持，使条件表达式更加清晰。
- **其他高级特性**：诸如三目运算符(`a ? b : c`)、 `switch` / `match` 多分支选择结构、函数重载/泛型、lambda表达式、宏等，更高级的语言功能目前都未支持。这些虽然不是“最基本”特性，但值得在语言扩展时考虑其取舍和实现难度。

综上所述，SCompiler当前的语言特性集中在过程式编程的基本面，尚未涉及面向对象、异常处理等更高层次的概念。一些已经在语法层出现的元素（如if/loop、模块）也因后端缺失而暂无法使用。这些不足在下一节的改进建议中将重点讨论。

编译-执行流程说明

下面结合以上结构，梳理编译器将源代码转换为可执行指令并由虚拟机运行的整体流程：

1. **词法分析**： `LexerEngine` 读取源文件文本字符流，一段一段地应用各 `TokenScanner` 规则。每当识别出一个记号（如一个关键字、一串标识符、一个数字等），就产生相应的Token对象加入Token序列。连续空白和注释被跳过。最终得到按出现顺序排列的Token列表作为语法分析的输入。
2. **语法分析**： `ParserEngine` 从Token流构建抽象语法树(AST)。它首先识别文件是否声明了 `module`，如果有则调用 `ModuleParser` 生成一个 `ModuleNode` 作为AST根。然后依次读取后续Token：遇到 `import` 关键字则用 `ImportParser` 解析导入声明加入AST；遇到 `function` 关键字则调用 `FunctionParser` 解析函数定义节点(`FunctionNode`)并加入当前模块AST；直到Token流结束。函数内部，Parser会根据缩进层级处理嵌套结构：函数体内的每一行语句由 `StatementParserFactory` 选择合适的解析器生成AST节点（如 `IfNode`，`LoopNode`，`DeclarationNode`，`ReturnNode`，`AssignmentNode` 或一般 `ExpressionStatementNode` ）。同时表达式部分用运算符优先解析构造子树。例如，一个包含函数和控制流的源文件最终会得到：模块节点下有多个函数节点，函数节点下有若干子语句节点，每个语句节点可能还有表达式子节点，形成树状层次。

3. **语义分析**： `SemanticAnalyzerRunner` 遍历上述AST，检查静态语义正确性。比如：每个 `IdentifierNode` 标识符引用在当前或外层作用域中是否有声明（通过符号表查找）；函数调用的 `CallExpressionNode` 的目标函数是否存在、参数数量和类型是否匹配函数签名；算术 `BinaryExpressionNode` 两侧类型是否可计算（如不能把整数和字符串直接相加）； `ReturnNode` 是否出现在函数内且返回类型符合函数声明；控制流 `IfNode` / `LoopNode` 的条件部分应当是可判定真假的类型等等。语义分析阶段还会把符号（变量、函数）的类型信息附加到AST节点上，以方便后续使用。如果发现语义错误，则记录错误信息并可在编译器输出中报告。只有当AST通过所有语义检查后，才进入下一阶段。
4. **IR生成**：编译器将语义正确的AST转换为中间表示IR。 `IRProgramBuilder` 按函数进行处理：对每个函数的AST，创建对应的 `IRFunction` 容器，然后调用 `StatementBuilder` 遍历函数体的每个语句节点，逐一生成IR指令并加入该IRFunction。对表达式则使用 `ExpressionBuilder` 生成指令序列并返回表示结果值的 `IRValue` （通常是一个虚拟寄存器）。举例来说，对于语句 `declare x: INT = 5` ，IR生成可能如下：
   - 为常量5创建一个 `IRConstant` ，然后生成 `LoadConstInstruction` 将5加载到某个虚拟寄存器v1；
   - 再生成一条赋值指令（可能也是通过 `BinaryOperationInstruction` 实现，将v1赋给变量x对应的存储位置）。

     对于算术表达式如 `a + b * 2` ，IR可能顺序为：加载b常量2到v2，计算 `b*2` 结果存入v3，再加载a到v4，计算 `v4 + v3` 结果存入新的v5，最后v5即表达式结果。需要注意，当前版本**尚未实现 if/loop 等控制流**的IR生成，遇到这些节点会抛出异常中止IR构建【33†】。
5. **寄存器分配与指令生成**：得到每个函数的IR后，编译器后端开始生成最终字节码指令。首先 `RegisterAllocator` 扫描IR函数，统计其用到的虚拟寄存器，将它们映射到实际的局部变量槽编号。例如某函数IR用了v1..v5五个虚拟寄存器，则给它们分配0-4号槽位，对应虚拟机函数帧的局部变量0到4。接着 `VMCodeGenerator` 遍历IR指令列表，将每条IR指令翻译成等效的虚拟机指令序列：
   - 对于计算类IR指令，如加法 `IRAddInstruction(target = op1 + op2)` ，生成顺序可能是：先发出“将op1加载到操作数栈”的指令(例如对应类型的LOAD指令)，再“将op2加载栈”的指令，然后发出对应类型的ADD指令完成运算，最后把结果存入target的位置（如果target是一个已经分配的槽位，则用STORE指令保存栈顶到该槽）。
   - 对于控制转移类IR，如 `IRJumpInstruction` ，生成一个无条件跳转的VM指令（ `JumpCommand` 带目标地址）；对于将来可能实现的条件跳转IR，会生成类似 `ICECommand` 这类指令配合跳转。
   - 对于函数调用 `CallInstruction` ，生成 `CallCommand` 并在操作数中注明被调函数的入口地址或索引。如果被调用函数尚未确定地址，可能需要记录以便在链接或加载阶段回填。
   - 对于返回 `IRReturnInstruction` ，生成对应的 `RetCommand` ，可能包括返回值处理和栈帧清理。

   `VMProgramBuilder` 会收集所有函数的指令序列，构建程序对象。例如可以为每个函数保存起始指令的索引，用于函数调用时跳转。此外可生成一个全局指令数组（类似汇编的机器码序列），方便按索引跳转。
6. **虚拟机执行**：完成以上编译步骤后，最终产出的是虚拟机可执行的指令列表（类似字节码文件内容，但在内存中表示）。接下来将由虚拟机加载并运行：

- **加载**： `VirtualMachineEngine` 初始化时，通过 `CommandLoader` 读取指令列表，将每条指令的操作码翻译为具体的 `Command` 对象。可以理解为把纯操作码流“链接”成可执行的指令对象流。有的实现可能在首次执行遇到某指令操作码时才动态创建 `Command` 对象（按需加载）。
- **运行**：引擎设置好栈等环境后，设置指令指针（程序计数器）从入口点开始。循环执行：取出当前指令对象，调用其执行方法(`Command.execute(...)`)。这个执行过程中，会对虚拟机状态进行读写。例如算术指令从OperandStack弹出操作数、运算并压入结果；调用指令创建新StackFrame、调整指令指针到被调函数入口；返回指令弹出当前StackFrame、恢复调用者指令指针；跳转指令修改指令指针跳到目标位置等。每执行完一条指令，若未跳转则指令指针顺序加一。如此循环，直至遇到程序结束指令（例如 `HaltCommand` ）或指令指针越界。期间如果有异常指令（目前没有异常机制，则可能是虚拟机检测到非法操作如栈溢出等），虚拟机会中止执行并报告错误。
- **输出和调试**：如有需要，虚拟机可以输出计算结果到控制台或者通过 `FileIOUtils` 写文件等（视具体实现而定，项目中FileIOUtils可能提供简单IO操作）。调试方面， `VMStateLogger` 可以在每步执行后打印当前操作数栈、局部变量等状态，从而帮助开发者跟踪程序运行轨迹。

通过上述阶段，SCompiler将源码成功地翻译并运行。

编译器会：lexer分出 `module` , `function` , `declare` , `return` 等Token；parser建立ModuleNode("Demo")下挂一个FunctionNode("add")，里面有DeclarationNode(sum)和ReturnNode；semantic检查类型（a、b为Int，a+b结果Int赋给sum也为Int，返回类型匹配）；IR生成创建IRFunction("add")，发出指令将a和b加载、加法、存sum、再加载sum返回；后端转为虚拟机指令序列
如： `ILoad 0; ILoad 1; IAdd; IStore 2; ILoad 2; Ret` （假定a槽0, b槽1, sum槽2）；虚拟机执行这些指令，最终计算出正确结果返回给调用者。整个流程体现了编译各阶段各司其职，将高级源代码逐步细化为低级操作并执行的过程。

当前设计的不足和局限

虽然 SCompiler 已经搭建了编译器和虚拟机的雏形，但在功能实现上还存在一些不足和局限，需要我们注意：

1. 语言语法支持的完整性与一致性

- **控制流支持不完整**：如上所述， `if` 和 `loop` 等控制流结构仅停留在语法和语义层面，编译器并不能将其转换为可执行代码【33†】。这导致当前语言事实上**无法使用条件分支和循环**，大大限制了可编写程序的复杂度。这种半吊子的支持也破坏了一致性——开发者可能看到语法允许if/loop，却无法正常运行，造成困扰。
- **缺少布尔类型**：没有真正的Boolean类型使语言在逻辑判断上不够清晰和安全。目前是用整数代替布尔，这在语义上不严格，而且隐含约定0/非0的做法可能导致误用（比如不小心把普通整数当布

尔判断）。同时缺少逻辑运算符&&, ||会使复合条件判断变得繁琐。总体而言，基础类型系统还不完善。

- **面向对象特性缺失**：语言只支持过程抽象（函数）而无数据抽象（类）。现代语言的结构化和可扩展性往往来自类和模块体系，SCompiler仅有模块/函数，难以组织大型程序或表示复杂数据关系。同时，点运算符用于模块成员访问的场景有限，没有类的话 . 无法访问对象属性，这部分语法能力闲置，显得不一致。
- **标准库和内建函数**：目前未提及任何标准库函数或内建功能（如打印输出函数，字符串操作等）。一个语言通常会有一些内置支持，但SCompiler尚未提供，这使得即使虚拟机支持I/O，通过语言本身却无直接接口。功能匮乏会影响语言实用性。
- **表达式和语句局限**：例如没有三目运算符、switch语句等。这些并非必需但缺少会使某些逻辑表达不够简洁。另外，似乎也没有提供 `break` / `continue` ，导致无法提前跳出循环，只能依靠条件控制，减少了灵活性。错误处理如前述也完全没有，程序健壮性方面不足。

总的来说，语言目前只是**基本可用**状态，很多常见结构不是缺席就是不完善。这在设计上一方面是功能未做完，另一方面也反映出某些不一致：比如语法允许的东西语义/IR未跟上，实现存在断层。

2. 编译器模块结构的可拓展性

- **新语法扩展难度**：编译器采用较清晰的模块划分，但要增加一类新的语法元素，需要在多个地方做改动。以增加 `while` 循环为例（假设与loop语义不同）：需修改词法分析增加 `while` 关键字、语法分析增加WhileParser/AST节点、语义分析增加WhileAnalyzer、IR生成增加While支持（构造循环IR）、可能还要在虚拟机增加相应指令。这种多点修改流程对单人维护还好，但随着语言复杂度增长，手动同步多个阶段逻辑容易出错。因此目前编译器架构对于**快速拓展新特性**支持一般，没有提供更自动化或元编程的手段。
- **模块职责可能有混杂**：当前实现中，每一层都有许多类，结构清晰但也存在潜在问题。例如IR生成器里，StatementBuilder要识别各种AST类型然后调用不同生成过程，缺少一种**统一的访问者模式**来自动分派。这意味着每新增一种Statement节点，都要修改StatementBuilder代码，违反一定程度的开闭原则。类似地，语义分析通过注册Analyzer类来处理不同节点，这还算比较解耦，但IR这层次的处理稍显硬编码。
- **错误处理与调试支持不足**：编译器在Lexer/Parser阶段遇错会抛异常，Semantic阶段收集错误，但整个流程缺少一个统一的错误管理机制（比如可以收集多个阶段错误一并报告，而不是遇错立即终止）。拓展编译器功能时，错误和异常处理的分散可能导致健壮性问题。此外，缺少调试或日志接口使得排查编译问题要读代码，加大维护成本。
- **可重用性与模块化**：目前编译器各部分紧密协作，但如果想将某部分替换，例如换一种后端或不同IR，难度较高。例如IR到字节码的映射是硬编码在VMCodeGenerator里的，无法方便地替换成输出别的格式。再比如词法规则、语法规则写死在代码中，不能通过配置或脚本调整。这在一定程度上限制了编译器的**灵活性**。
- **性能优化欠缺**：虽然目前功能为主，但随着扩展，需要考虑编译性能和生成代码质量的问题。当前没有任何优化步骤（如常量折叠、死码消除等），IR也较原始。如果后续拓展语言特性，编译器结

构需要容纳优化模块，这方面的预留尚不明显。

3. 虚拟机指令设计的可扩展性

- **指令集合扩展繁琐**：虚拟机采用了一条指令一个类的设计，不同数据类型的同类操作也分开实现。这虽然直观，但当需要支持新类型或新操作时，需要成倍增加类和操作码。例如，若增加一个BOOL布尔类型，则可能需要增加BAnd, BOr, BXor等指令类；增加64位浮点比较，也要写对应的比较指令类。指令数量膨胀会增加维护负担，也容易出现不一致（忘记某型的某操作）。理想情况下，应该有更通用的方式描述指令逻辑，减少重复代码。
- **缺少复杂指令**：目前虚拟机指令偏底层，如算术、加载、跳转已经覆盖基本操作。但如果语言将来增加更复杂行为（如函数闭包、协程、面向对象调用等），现有指令可能无法直接支持，需要新增一系列指令。例如，面向对象需要指令处理堆内存和对象字段访问；异常机制需要指令来抛出和跳转到异常处理位置。当前VM设计较简单，新增复杂指令可能涉及调整整个调用栈或内存模型，这并非易事。
- **类型和操作强耦合**：正如前述，每种数据类型在VM里都是一套独立指令。这种设计在类型较少时问题不大，但扩展性差。如果将来支持用户定义类型（例如结构或类实例），虚拟机难以为每种用户类型都定制指令。需要一种更加类型无关或泛型化的指令体系。目前的设计更接近JVM的字节码风格（JVM也是区分int, long等指令集），但现代一些VM趋向于更统一的操作码或基于栈元素类型动态决定行为。SCompiler VM暂未体现这方面弹性。
- **内存管理简单**：虚拟机目前没有提及垃圾回收或内存分配指令。像字符串这类可能驻留在常量区，未涉及动态内存。此外，没有数组/对象，也回避了内存管理问题。但若后续加入这些，必须扩展虚拟机以支持动态分配、引用类型和垃圾回收算法。当前设计里没有预留这样的机制，这也是以后扩展的一大挑战。
- **性能与调优**：现有虚拟机以解释执行方式运行所有指令，没有JIT或优化执行的概念。如果字节码指令集膨胀，会进一步拖慢解释速度。而增加优化执行则需要对指令集和执行器做大改动。可见目前VM设计追求易懂实现，但当扩展功能后，性能和复杂度都会上升，需要提前考虑架构上的改进。

总结来说，虚拟机指令设计目前可满足简单程序的运行，但要扩展功能，**需要付出较多的手工工作**，且某些高级功能的加入会冲击现有设计。指令集的可扩展性有待提高，以便更从容地支持未来的语言增强。

优化与扩展建议

针对上述发现的问题，下面提出一些面向未来版本的改进方向，分别从语言特性、编译器架构和功能规划三方面给出具体建议。

1. 扩展语言语法支持

- **完善控制流实现**：优先补全 `if` 和 `loop` 的编译支持。具体做法是在IR生成阶段为If和Loop节点添加处理：
  - If语句：可以为条件表达式生成比较指令和条件跳转IR。例如构造两个IRLabel，一个指向else块（或结束），一个指向if块结束后位置。大致流程：计算条件到寄存器，生成 `IRJumpInstruction(condFalseLabel, JUMP_IF_FALSE)` 和块内指令，块结束时若有else则跳过else块的跳转，等等。然后在虚拟机指令映射时，利用已有的比较和Jump指令（如ICE等）实现跳转逻辑。
  - Loop循环：根据是 `while` 类似（有条件）还是无限循环，生成循环开始和结束Label。条件判断放在循环入口或尾部，根据语义决定。如果是 `loop condition:`（类似while），先判断条件跳转出循环或进入循环体；体内末尾再跳转回开头形成闭环。利用 `JumpCommand` 实现回跳。需注意break/continue等结构，暂时可以不支持或通过特殊跳转IR标识实现。
  - 完成上述，将使if/loop真正可用。同时**建议**加入 `break` 和 `continue` 简单实现：可以在LoopAnalyzer中识别这两个关键字（如果引入），在IR生成时将 `break` 转成跳转到循环后Label， `continue` 跳转到循环判定Label。
- **增加布尔类型和逻辑操作**：在 `BuiltinType` 中加入BOOLEAN类型，让语义分析和类型检查更明确（if条件要求Boolean类型）。Lexer和Parser增加布尔字面量（true/false关键字）和逻辑运算符词法。编译期将 `&&`，`||` 实现为短路逻辑：可在表达式解析时将它们优先级定义低于比较运算，然后在IR生成时特别处理——例如 `cond1 && cond2` 可生成：计算cond1，如果假直接跳过cond2计算输出假，否则计算cond2作为结果。这种短路逻辑可用条件跳转指令实现。VM也需新增布尔运算支持，比如直接按布尔处理AND/OR（或复用整数1/0实现，但最好还是类型独立）。引入Boolean有助于清晰表达条件，并为将来可能的布尔代数优化奠定基础。
- **支持数组和字符串操作**：数组方面，语法上可以增加数组类型表示和元素访问：例如类型 `INT[]`，字面量 `[1,2,3]`，以及表达式 `arr[index]`。实现上需要：
  - 在类型系统中引入数组类型表示，可设计 `ArrayType` 类包含元素类型和维度。
  - 解析器在遇到类型标注如 `TYPE[]` 时构造ArrayType，在表达式解析识别 `Identifier [ Expression ]` 模式为数组元素访问AST节点。
  - 语义检查数组越界等可能暂不做复杂分析，只需检查索引是整数类型。
  - IR和VM：增加指令 `NewArray` 用于创建数组（需要操作数指定长度），以及 `LoadElement` / `StoreElement` 用于按索引读写数组元素。因为当前VM没有动态内存，可能需要模拟一块连续内存，比如让 `OperandStack` 或一个全局堆来存储数组元素。实现难度较大但可以从一维定长数组开始。
  - 字符串目前只支持字面量，并未提到连接操作。可考虑支持用 `+` 连接字符串（很多语言如此）。在语义上检测 `String + String`，IR生成可调用运行时库函数或生成特殊指令。或者至少提供一些库函数如 `concat(str1, str2)`。
- **引入类和结构**：如有计划支持OOP，可以逐步来：

- 先支持**结构体(struct)**：允许用户定义简单数据类型组合。语法类似 `struct Point: x: INT, y: INT end struct` 。编译器将创建一个新的类型Symbol（Kind为STRUCT），记录字段列表。然后支持 `point.x` 这样的成员访问。VM需支持根据偏移访问结构字段的指令，或者将struct当做数组加字段索引映射。
- 真正的**class**：在结构基础上加入方法和封装。语义上更复杂，需要 `this` 引用、方法表等。由于工程量巨大，可暂不实现完整继承，只实现单纯的类实例创建和方法调用。方法调用可在编译时静态绑定（无继承多态时）。VM需要指令支持分配对象（在堆上）和调用实例方法（可能传入对象引用）。
- 由于涉及内存管理，必须设计堆和垃圾回收方案。可以先使用简单引用计数或手动内存释放的方法。指令上，需要指针/引用类型支持，比如LoadRef/StoreRef等操纵引用，以及一套新的调用机制支持虚方法（如多态时）。

- **完善模块系统**：使编译器真正支持多模块项目编译：
  - 允许编译器将每个模块单独编译为中间格式（例如每个模块产生自己的指令列表和符号表导出列表），然后有一个链接步骤将多个模块的符号解析、指令列表合并。可以引入**符号链接**阶段： `ModuleRegistry` 在所有模块编译后，解析import关系，检查所需符号是否在被导入模块导出，然后填充调用指令的目标地址等。
  - 支持模块初始化代码：比如模块级别的变量初始化或执行代码，需要在程序启动时统一跑一遍。可规定入口模块，并让VMLauncher按import依赖顺序执行模块初始化。
  - 命名空间处理：确保不同模块中可以有相同名字的符号互不冲突（通过模块名限定）。编译器在语义分析标识符时，如遇非本模块定义则尝试从导入模块符号表查找。这部分逻辑需要完善。
- **错误和异常处理**：为了提高程序健壮性，可考虑添加基本的异常抛出/捕获机制：
  - 语法：例如 `throw 表达式` 抛出异常；以及 `try: ... catch var: ... end catch` 块（或类似Python的缩进try-except结构）。初期可以限定只能抛出简单类型（如字符串或整数作为错误码）。
  - 语义：需要一个统一的异常类型或父类； `throw` 一定在函数有异常可抛的地方使用； `catch` 块需定义接收异常对象的变量。
  - IR/VM：增加 `ThrowInstruction` 对应VM的 `ThrowCommand` ，执行时会触发异常流程。虚拟机需要维护异常发生时的调用栈处理：可以在StackFrame增加异常捕获表，如果当前帧没有catch匹配则弹栈继续往上，直到找到catch或无则终止程序。实现完整异常较复杂，可以从简单的“非受检异常”开始（任何地方throw，上层不catch则终止）。
  - 当然，错误处理也可以一开始用简单方案：例如标准库一个 `panic(msg)` 函数直接终止程序，或者要求函数通过返回值反映错误。但长期看，语言层支持try-catch是值得的。
- **其他改进**：随着语言功能增强，可以逐步添加：
  - **条件表达式**：支持三目运算符 `condition ? expr1 : expr2` ，解析为特殊的三分支AST并在IR用条件跳转实现。或者像Python一样支持 `expr1 if cond else expr2` 。
  - **循环增强**：除了 `loop` (类似while)外，可增加 `for` 循环语法，支持迭代特定次数或范围；甚至支持 `foreach` 遍历数组（需容器支持后）。

- **函数默认参数/可变参数**：提升函数调用灵活性。不过这些需要在语义阶段处理默认值填充或参数列表打包解包，也需要审慎设计。
- **标准库**：哪怕语言本身不直接提供IO语法，也可以扩展标准库函数，比如print/println用于输出，len()用于取字符串或数组长度等等。这些函数可在编译器内置，编译时识别调用并映射到特殊指令或VM内置实现。

总之，语言层面的扩展应遵循**优先基础、逐步提高**的策略。先把现有语法真正落地可用，然后在此基础上添加新类型、新结构。在设计每个新特性时，要考虑与现有部分的兼容和交互，尽量保持语言的一致性。例如引入类后，函数如何作为方法处理、作用域如何影响等，都要有清晰规则。通过不断完善，SCompiler可以从一个小型脚本语言成长为具有现代语言基本特征的系统。

2. 优化编译器架构的模块化与可扩展性

- **采用设计模式提高扩展性**：可以运用更加模块化的设计模式来改进编译器结构。例如：
  - 在AST到IR的遍历中，引入**访问者模式(visitor)**：为AST节点增加接受访问者的方法，IR生成器实现一个IRVisitor，根据节点类型生成IR。这样新增节点类型时，只需添加相应visitor处理，而不用修改通用遍历逻辑。当前的 `StatementBuilder/ExpressionBuilder` 有些类似visitor但不是严格模式，可重构为visitor模式以提升一致性。
  - 语法和语义分析已用了工厂和注册表模式（如StatementParserFactory, AnalyzerRegistry），这很好地支持了通过注册添加新解析器/分析器。可以继续发扬：将关键字与解析器的绑定配置化，甚至通过读取配置文件/注解实现自动注册，这样添加新语法时减少改动集中在一处。
  - 对于指令和操作码映射，同样可以使用**工厂+反射**来减少硬编码。例如VM指令的操作码和Command类对应关系可在一个配置表中，新增指令时只需添加表项，不必修改大量代码。这在编译器后端(IROpCodeMapper)和虚拟机CommandFactory中都适用。
- **划清各阶段接口**：为每个编译阶段定义清晰的输入输出接口，有助于模块化和调试。例如：
  - Lexer输出Token流对象，Parser输入Token流输出AST对象，Semantic输入AST输出注释过的AST或符号表。IR生成输入AST输出IRProgram，后端输入IRProgram输出VMProgram。若能把这些接口抽象出来，甚至可以方便地插入或替换某个阶段实现（比如以后想支持不同前端语法，只要产出相同IR即可）。
  - 通过明确定义阶段边界，也可以插入**优化器**：例如在IR生成后，增加IR优化阶段，输入IRProgram输出优化后的IRProgram，然后再交给后端。因为接口稳定，所以对其他部分无影响。
- **增强错误处理和调试机制**：编译器应提供统一的错误报告系统。可以引入一个 `CompilationErrorReporter` 收集各阶段的错误和警告信息，并在最终输出。取代目前抛异常中断的做法，让编译尽可能检测所有问题一次性反馈给用户。对于调试编译过程，可加入**日志选项**：比如提供verbose模式，打印Lexer生成的token列表、Parser生成的AST树（可以借助现有 `ASTJsonSerializer` 将AST输出JSON）、语义分析的符号表、IR指令列表等。这对开发者理解编译器行为和定位问题很有帮助。

此外，考虑未来多人协作或用户自定义扩展，可设计错误码和文档，方便扩展者遵循统一规范添加新的错误类型。

- **提高复用性**：让编译器的一些组件更独立，可移植到其他项目或用途。例如Lexer可以设计成无需依赖整个编译器框架就能工作（输入源码输出tokens）。这样一来，如果将来语言有IDE插件或其他工具，可直接使用编译器组件而不必启动完整编译流程。再如符号表、类型系统等，也可作为独立模块提供API。这种松耦合设计能让扩展和重构更加从容。
- **配置与脚本化**：目前所有规则都硬编码在Java类中，可以考虑引入外部配置或脚本语言描述部分编译逻辑。例如，用一种元语言定义文法和生成解析器，而不是手写所有Parselet；或者用配置文件列出关键字和TokenScanner、StatementParser映射关系。这种改动前期成本较高，但对以后拓展新语法非常有利——可能只要修改配置就能支持简单的新关键字。此外，若有精力，还可以尝试**生成器**模式，比如根据BuiltinType列表自动生成各类型指令的代码框架，以减少手工重复。
- **面向未来的优化**：如果后续打算提升生成代码质量，可以将架构调整为便于插入优化算法：
  - IR可以考虑使用**控制流图(CFG)**形式，以更好地做数据流分析、死代码消除等优化。
  - 模块间优化（比如内联、常量传播）也需要编译器能够跨函数或模块看待，这会影响编译流程的架构。提前设计扩展点（如预留优化阶段钩子）可避免以后大改。
  - 另一个方向是支持多目标后端输出。也许未来不仅运行在自制VM上，还可以输出LLVM IR或Java字节码等。为了这个可能性，前端和后端的解耦要做好。可以设计接口 `CodeGenerator` ，当前实现一个生成Snow VM字节码，将来可以有生成其它平台代码的实现。

总之，优化编译器架构的关键词是**“解耦、扩展、可配置”**。通过更好的分层和约定，使得添加功能不需要大改现有代码，尽可能遵循开闭原则。同时，让编译器核心更加通用，以适应语言演进的需求。这一过程可以逐步进行，每次重构一种机制，使整体变得健壮灵活。

3. 后续功能实现规划

为了使 SCompiler 逐步具备现代编程语言的基本特性，建议按照由易到难、由基础到高级的顺序进行后续开发。以下是一个可能的规划路线：

1. **修复现有缺失**：首先解决当前明显的不完整之处，使语言最基本构造可正常工作。这一步重点是完成 if/loop 的IR和执行支持，以及添加Boolean类型和逻辑运算符。如前所述，这将显著提高语言可用性，让用户能编写条件和循环逻辑。
2. **增强类型系统**：在基本类型齐备后，可考虑引入更复杂类型如数组、结构体。数组相对独立，可以较快实现一维数组支持并提供基本库函数。结构体/类较复杂，可以留到稍后。同时，完善类型检查，比如实现自动类型提升规则（ `Type.widen` 方法已经定义，用于比如把Short提升为Int以参与运算【14†】），保证表达式类型推导符合期望。
3. **模块和作用域**：完善模块导入链接机制，实现多文件编译。确保符号表正确管理不同模块作用域和命名空间。同时，实现块级作用域：修改 Parser/Semantic，在进入新的缩进块时（如函数体、if/else块、loop块）创建子符号表，离开时销毁。测试在嵌套块中声明同名变量会屏蔽外层变量等行为是否正确。这一步完成后，作用域管理将更严谨，模块化也真正可用。

4. **异常处理和标准库**：加入基本的 `throw/try/catch` 机制，实现运行时错误捕获。可先支持在函数内捕获自己抛出的异常，不跨函数传播，逐步再实现跨栈传播。并增加一些标准库功能：如I/O（可以在VM里实现System.out风格输出指令，或提供标准库函数由底层IO支持）、字符串操作函数（子串、拼接等），数学函数库等。这些丰富程序功能，同时为后面支持更复杂应用打基础。
5. **面向对象支持**：如果计划支持类，在上述过程稳定后着手。先引入class定义语法和对象创建（可能用 `new` 关键字）。实现成员变量、方法的解析和符号管理。虚拟机需要重大升级支持对象引用和堆分配，这可能是最艰巨的一步。可考虑借鉴JVM或其他简易OO语言的运行时模型。逐步实现方法调用、 `this` 引用，最后考虑继承和多态。如果过于困难，也可以选择只支持简单类结构、不涉及继承（类似C的struct加函数指针风格）。
6. **优化和完善**：当语言主要特性都有后，可以回头优化编译器和VM性能。例如：
    - 编译器增加优化步骤，如常量折叠：在IR生成时把常量表达式直接计算，减少运行时开销。
    - 虚拟机可以考虑实现字节码直接线程化或JIT：例如把指令序列翻译为本地代码段执行。这需要深入的专业知识，可根据需要决定。
    - 增强调试支持：比如实现源码级调试功能，需要在编译时保留行号映射信息，在虚拟机执行时跟踪当前源码行，允许断点、单步。这对一个成熟语言实现很重要。
    - 补充更多标准库或与宿主系统交互功能，让语言更有用（比如文件操作、网络库等）。
7. **持续测试和文档**：在每个阶段，都需要编写大量测试用例验证新特性的正确性，并完善语言文档供用户参考。例如，当添加数组后，要在文档中说明其用法和限制。保持文档与实现同步能避免用户困惑，也方便开源协作时吸引他人参与。