编译器设计与实现文档

2025.06

---

1. 概述

本文档旨在详细阐述一个特定编程语言 L310 的编译器的设计与实现。该编译器包含了一个词法分析器、一个递归下降的语法分析器、符号表管理器、中间代码（P-Code）生成器，以及一个用于执行该中间代码的虚拟机（VM）。此外，还提供了一个基于 Tkinter 的图形用户界面（GUI），方便用户编写、编译运行代码并查看结果。

该编译器在设计上采用了单遍编译（Single-Pass）策略。这意味着它在一次从前到后的源码扫描过程中，就完成了词法分析、语法分析、语义检查和代码生成等主要工作。这得益于以下关键技术的应用：

- **递归下降解析**：语法分析器的结构直接映射了语言的 EBNF 文法，通过函数间的相互调用来识别语法结构。
- **即时代码生成**：一旦识别出某个语法结构（如一个表达式或一条声明语句），编译器会立即生成相应的虚拟机指令，而无需构建一个完整的中间表示（如 AST）。

该编译器所针对的语言是一种过程式、静态作用域的语言，支持基本整型、函数、if-else、while 循环、I/O 操作，并特别地支持特定的 struct（结构体）类型、结构体数组以及结构体作为函数参数的传值调用，不支持浮点数、字符串、布尔和聚合类型直接运算。

2. 语言文法定义

语言的文法采用扩展巴科斯范式（EBNF）定义如下。

2.1 EBNF 文法（以程序中通过编译的语言为准）

EBNF

```
<program> ::= "program" <ident> "{" { <struct_def> } { <func_def> } "main"
"{" <stmt_list> "}" "}"

<struct_def> ::= "struct" <ident> "{" { <member_decl> } "}" ";"

<member_decl> ::= <type_specifier> <ident> [ "[" <number> "]" ] ";"

<type_specifier> ::= "int" | "struct" <ident>

<func_def> ::= "func" <ident> "(" [ <param_list> ] ")" "{" <stmt_list>
"return" <expr> ";" "}"

<param_list> ::= <param_decl> { "," <param_decl> }

<param_decl> ::= <ident> [ ":" <type_specifier> ]

<stmt_list> ::= <stmt> ";" { <stmt> ";" }

<stmt> ::= <declare_stmt>
```

```
     | <assign_stmt>
     | <if_stmt>
     | <while_stmt>
     | <input_stmt>
     | <output_stmt>
     | <func_call>

<declare_stmt> ::= "let" <ident> [ <declaration_options> ]

<declaration_options> ::= ( ":" <type_specifier> "[" <number> "]" )
           | ( ":" <type_specifier> [ "=" <expr> ] )
           | ( "=" <expr> )

<assign_stmt> ::= <designator> "=" <expr>

<if_stmt> ::= "if" "(" <bool_expr> ")" "{" <stmt_list> "}" [ "else" "{"
<stmt_list> "}" ]

<while_stmt> ::= "while" "(" <bool_expr> ")" "{" <stmt_list> "}"

<designator> ::= <ident> { ( "[" <expr> "]" ) | ( "." <ident> ) }

<func_call> ::= <ident> "(" [ <arg_list> ] ")"

<arg_list> ::= <expr> { "," <expr> } | <designator> { "," <designator> }

<input_stmt> ::= "input" "(" <ident> { "," <ident> } ")"

<output_stmt> ::= "output" "(" <expr> { "," <expr> } ")"

<bool_expr> ::= <expr> ( "==" | "!=" | "<" | "<=" | ">" | ">=" ) <expr>

<expr> ::= [ "+" | "-" ] <term> { ( "+" | "-" ) <term> }
<term> ::= <factor> { ( "*" | "/" ) <factor> }
<factor> ::= <designator>
     | <number>
     | "(" <expr> ")"
     | <func_call>

<ident> ::= <letter> { <letter> | <digit> }
<number> ::= <digit> { <digit> }
<letter> = "a" | "b" | ... | "z" | "A" | "B" | ... | "Z"
<digit> = "0" | "1" | ... | "9"
```

2.2 语言约束

除了 EBNF 文法，语言还遵循以下重要规则：

1. **返回语句**：func 体内部的最后一条语句必须是 return 语句，不允许提前返回。
2. **作用域限制**：main 或普通语句块（如 if/while 的 {...}）内不能使用 return。
3. **非空语句列表**：stmt_list 至少包含一条语句。
4. **函数定义**：函数定义不能嵌套在其他函数内部，所有函数都在程序顶层定义。
5. **调用顺序**：函数必须先定义后调用。
6. **函数传参**：当传递数组中的一个元素作为函数参数时，该数组的索引必须是数字字面量，不能是变量。数组的索引和大小必须为非负整数。
7. **变量作用域**：不允许全局变量。所有变量都在 main 或 func 的作用域内声明。变量的作用域遵循块级作用域（lexical scoping），内层作用域可以“遮蔽”（shadow）外层作用域的同名变量。
8. **层级**：main 函数体的层级（AR Level）为 0，每个 func 的层级均为 1。

---

3. 编译器设计与代码结构

编译器由四个核心模块组成，全部实现在 Lcompiler.py 文件中。

3.1 词法分析器（Lexer）

- **职责**：读取源代码文本，将其分解为一系列的 词法单元（Token）。
- **实现**：Lexer 类通过扫描输入字符串，识别出关键字（如 program，func，if）、标识符、数字、操作符（如 +，==，.）和界符（如 (，{，;），它能够自动跳过空格、换行符和单行注释（//）。

3.2 符号表（Symbol Table）

- **职责**：存储程序中所有标识符（变量、函数、结构体定义）的信息。
- **实现**：SymbolTable 和 Symbol 类协同工作。
  - Symbol 类表示一个符号，包含其 **名称(name)**、**类别(kind)**（如 var，proc，struct_def）、**类型信息(type)**、**作用域层级(block_level，ar_level)** 和 **在活动记录中的地址(addr)** 等关键信息。
  - SymbolTable 管理所有符号，提供添加和查找功能。它支持嵌套作用域，能够根据当前的 **作用域名称（scope_name）** 和 **块级别（block_level）** 正确解析标识符。
  - 它还专门管理 struct 的定义，包括其字段、每个字段的偏移量和总大小，为复杂数据类型的内存布局和访问提供支持。
- **ar_level 与 block_level 的含义**：
  - ar_level（Activation Record Level）：代表**函数的静态嵌套深度**。它被用来计算访问非局部变量时的层级差 L。在本编译器中，由于不允许函数嵌套定义，main 函数在 ar_level = 0 执行，而所有顶层 func 函数都在 ar_level = 1 执行。这个值在进入函数解析时确定，并在整个函数体内保持不变。
  - block_level（Block Level）：代表**同一函数内部的代码块（{...}）嵌套层级**。它用于处理变量的“遮蔽”（shadowing），例如，一个函数体的第一层是 block_level = 1，其内部的 if 或 while 语句会创建 block_level = 2 的新作用域。当查找变量时，编译器会从当前的 block_level 向外（递减）搜索，从而找到最近的定义。

- **同名变量（遮蔽）的处理**：
    - **存储方式**：当编译器遇到一个变量声明（如 let x）时，它会使用（name, scope_name, block_level）这个三元组作为键来在符号表中创建一个唯一的 Symbol 条目，例如，在 main 中（block_level=1）声明的 let num 和其内部 if 语句中（block_level=2）声明的 let num 会成为符号表里两个独立的条目，分别对应（'num', 'main', 1）和（'num', 'main', 2）。
    - **地址(A 值)分配**：每个被遮蔽的变量都是一个全新的变量，因此会通过 get_next_addr_offset 分配一个新的、独立的运行时栈地址（相对于 bp 的偏移量），即它们的 A 值（addr 属性）是不同的。
    - **访问时的 L 和 A 值计算**：当代码中使用一个变量名时，lookup_symbol 函数从当前 block_level 开始向 0 搜索。它会首先找到最内层的同名变量。生成 lod 或 sto 指令时，A 值就取自这个最内层符号的 addr 属性。L 值则根据 current_ar_level - symbol.ar_level 计算得出。这一机制确保了变量遮蔽的正确实现。

3.3 语法分析与代码生成（ParserAndGenerator）

- **职责**：这是编译器的核心。它采用 递归下降（Recursive Descent）的方法进行语法分析，并在分析过程中同步生成虚拟机的目标代码（P-Code）。
- **实现**：ParserAndGenerator 类：
    1. 从 Lexer 获取 Token。
    2. 拥有一系列与文法规则对应的 _parse_* 方法（如 _parse_stmt, _parse_expr）。
    3. 在语法分析的同时，与 SymbolTable 交互，填充和查询符号信息。
    4. 调用 _emit 方法生成 P-Code 指令，并将其存入一个列表。
    5. 处理地址回填，例如，if 或 while 语句中的跳转地址在解析完其代码块后才能确定。
- **实地址回填（A 值）的计算逻辑**：
  对于需要前向跳转的控制流语句（如 if, while），目标地址在生成跳转指令时是未知的。编译器采用“回填”技术解决此问题：
    1. **生成占位指令**：当解析到 if 或 while 的条件时，会立即生成一条 jpc 指令。此时，由于目标地址未知，指令的 A 值会被设置为一个临时的、唯一的字符串标签（如 _while_end_15）作为占位符。同时，保存这条指令在代码列表中的索引。
    2. **标记目标地址**：当解析器到达跳转的目标位置时（例如 while 循环体的末尾），它会调用 _mark_label 方法。该方法将当前的指令计数器（PC）值与该标签关联起来，存入一个字典中。
    3. **地址修补**：解析器随后调用 _patch_jump_address 方法，使用之前保存的指令索引找到那条 jpc 指令，并将其 A 值的占位符字符串替换为最终解析出的目标 PC 整数值，这个过程在解析完整个程序后的一个最终处理阶段完成，确保所有标签都转换为确切的地址。

3.4 虚拟机（VM）

- **职责**：执行由代码生成器产生的 P-Code 指令序列。
- **实现**：VM 类模拟一个基于栈的计算机。
    - **数据栈（stack）**：用于存储操作数、变量、函数调用的控制信息等。

- **寄存器**:
  - pc (Program Counter): 指向下一条要执行的指令。
  - bp (Base Pointer): 指向当前活动记录 (Activation Record) 的基地址。
  - sp (Stack Pointer): 指向栈顶。
- **执行循环**: run 方法循环读取 pc 指向的指令并执行，直到程序结束。
- **活动记录 (AR)**: 每次函数调用都会在栈上创建一个新的 AR，用于存放参数、局部变量和控制链（静态链、动态链、返回地址）。

---

4. 核心：P-Code 虚拟机指令集

P-Code 是一种为栈式虚拟机设计的中间代码。每条指令由一个操作码 (op) 和两个参数 (L 和 A) 组成。

4.1 指令格式

- {op, l, a, comment}
  - op: 指令操作码（字符串）。
  - l: 层级差 (Level difference)。一个整数，用于在静态作用域链上查找变量。
  - a: 地址或操作数 (Address/Argument)。一个整数或标签，具体含义取决于 op。
  - comment: 注释，用于解释该指令的生成目的，便于调试。

4.2 活动记录 (Activation Record) 结构

要理解 L 的含义，首先要了解函数调用时在栈上创建的活动记录 (AR) 的结构。当调用一个函数时，其 AR 从新的 bp 处开始，结构如下:

- stack[bp]: **静态链 (Static Link, SL)** - 指向定义该函数的上一层作用域的 AR 的基地址。
- stack[bp+1]: **动态链 (Dynamic Link, DL)** - 指向调用者 (caller) 的 AR 的基地址。
- stack[bp+2]: **返回地址 (Return Address, RA)** - 调用结束后应返回到的指令地址 (PC)。
- stack[bp+3]: **参数个数 (NARGS)** - 传递给该函数的标量参数总数。
- stack[bp+4]...: **参数和局部变量** - 按照符号表中分配的地址（偏移量）存放。

4.3 指令详解

操作码 (op)	L (Level)	A (Argument)	描述
lit	0	value	**Literal**: 将立即数 value 压入栈顶。
lod	L	offset	**Load**: 从 (L, offset) 指定的地址加载一个值到栈顶。计算方法: 首先通过 L 沿着静态链回溯 L 次找到正确的 AR 基地址 base_addr, 然后加载 stack[base_addr + offset] 的值。

sto	L	offset	**Store**: 将栈顶的值存入 (L, offset) 指定的地址。计算方法同 lod，但操作是存储。
cal	L	func_pc	**Call**: 调用函数。L 是调用者与被调用函数定义处的层级差。func_pc 是被调用函数的入口地址。执行时，会在栈上建立新的 AR (SL, DL, RA)，然后 pc 跳转到 func_pc。
int	0	size	**Increment Top**: 在当前 AR 中为局部变量预留 size 个存储单元的空间，即 sp = bp + size。
jmp	0	target_pc	**Jump**:无条件跳转。将 pc 设置为 target_pc。
jpc	0	target_pc	**Jump on Condition**: 弹出栈顶的值。如果为 0 (false)，则跳转到 target_pc。
opr	0	op_code	**Operator**: 执行操作，由 op_code 决定: - 0: **Return** - 从函数返回。恢复调用者的 pc, bp, sp，并将返回值放在新的栈顶。 - 2-5: +, -, *, / - 弹出两个操作数，计算后将结果压栈。 - 8-13: ==, !=, <, >=, >, <= - 弹出两个操作数，比较后将布尔结果 (1 或 0) 压栈。 - 14: **Output** - 弹出栈顶值并输出。 - 15: (无操作) - 用于在输出后格式化。 - 16: **Pop** - 弹出并丢弃栈顶值 (用于处理函数调用后未使用的返回值)。
lods	L	0	**Load Indirect**: 间接加载。用于访问 a[i] 或 s.f 这样的复杂左值。执行时，它会先从栈顶弹出一个计算好的**偏移量** computed_offset，然后通过 L 找到基地址 base_addr，最终加载 stack[base_addr + computed_offset] 的值并压入栈顶。
stos	L	0	**Store Indirect**: 间接存储。与 lods 类似，但它会弹出**值**和**偏移量**，将值存入 stack[base_addr + computed_offset]。
inp	L	offset	**Input**: 从用户获取一个整数输入，并将其存入 (L, offset) 指定的地址。

4.4 层级差 l 的说明

L（Level）的值用于处理不同静态嵌套深度函数之间的非局部变量访问。它帮助虚拟机沿着静态链（由活动记录中的静态链接 SL 构成）向上查找声明变量的那个函数的活动记录。l_val 指示向上跳多少级静态链。它的值是通过公式 L = current_ar_level - symbol_ar_level 计算得出的，其中 current_ar_level 是当前代码执行点的 AR 层级，symbol_ar_level 是被访问变量或被调用函数在其定义时的 AR 层级。

- **cal 指令的 L 值**：
  - **当 L = 1 时**：当一个函数 A（执行在 ar_level=1）调用另一个函数 B（定义在 ar_level=0）时，cal 指令的 L 为 1 - 0 = 1。
  - **当 L = 0 时**：main 函数（执行在 ar_level=0）调用一个函数（定义在 ar_level=0），cal 指令的 L 为 0 - 0 = 0。
- **lod/sto 指令的 L 值**：本语言的作用域规则有一个重要特点：**函数只能访问自身的局部变量或 main 作用域中的变量，不允许访问其他“兄弟”函数的内部变量**，这一设计简化了静态链的查找。同一函数内的不同块，即使嵌套，仍然属于同一个函数的静态层级（AR Level）。
  - **当 L = 1 时**：当一个函数（执行在 ar_level=1）需要访问在 main 中定义的变量（定义在 ar_level=0）时，lod/sto 指令的 L 为 1 - 0 = 1。
  - **当 L = 0 时**：当一个函数（执行在 ar_level=1）访问自己的局部变量或参数（也定义在 ar_level=1）时，lod/sto 指令的 L 为 1 - 1 = 0。

综上，由于语言不支持函数嵌套定义，L 的值只可能是 0 或 1，它精确地反映了当前执行点与目标符号定义点之间的静态层级关系。

4.5 地址或操作数 a 的说明

A（Argument）的值根据指令类型有不同的计算方法。

- **cal l a 指令中 A 的计算**：A 是被调用函数的入口地址。在代码生成阶段，解析器从符号表中查找函数符号，获取其 addr 属性，这个 addr 就是一个指向函数第一条指令的标签。在最终回填阶段，这个标签会被替换为具体的 PC 值，成为 cal 指令的 A 值。
- **函数调用时，加载参数的 lod 指令中 A 为负数的原因**：这发生在被调用函数（callee）的内部，目的是将调用者（caller）压入栈的**实参**复制到自己的**形参**存储区。调用栈是动态的，新分配区域的生命周期仅限于函数调用期间。
  1. **调用过程**：Caller 先将所有参数压栈，然后执行 cal 指令。
  2. **栈帧切换**：cal 指令执行后，VM 切换上下文，新的 bp 指向新栈帧的基址。之前由 Caller 压入的参数位于新 bp 的**下方**，因此需要用**负向偏移量**。
  3. **A 值计算**：在被调用函数的序言（prologue）中，编译器生成一系列 lod 指令。A 的值从 -N 开始（N 是标量参数总数），依次递增。例如，第一个参数的地址是 bp-N，第二个是 bp-N+1，以此类推。L 值为 0，因为这些参数就在当前 bp 的下方，无需跨越静态链。
- **图示：函数调用时参数的负向偏移量访问**：在被调用函数（callee）内部，需要使用 lod 指令将调用者（caller）压入栈的**实参**复制到自己的**形参**存储区。由于此时新的 bp 已经指向了 callee 栈帧的基址，而实参位于其下方，因此必须使用负向偏移量来访问。

```
func calc(a, b)
{
    let temp_val;let final_val;
    temp_val = a + b;
    final_val = temp_val * b;
    return final_val;
}

调用func calc(3, 5)为例
1. `cal` 指令执行前 (Caller 压入参数后)
| ...        | <-- 新的未使用空间
+------------+
|     5      | <-- 新的 sp (压入最后一个参数后)
+------------+
|     3      |
+------------+ <-- main 原本的 sp
| main vars |
| ...        | <-- bp
此时，存放 5 和 3 的这部分空间是一个临时的“参数传递区”。它在逻辑上既不属于 main 的局部变量区，也不属于即将被调用的函数的局部变量区。
--------------------------------------------------------------------------------
1. `cal` 指令执行后，`int` 执行后，真正为func分配大小为9的空间 (进入 Callee 内部，准备执行 lod)
| ...        |  <-- SP (栈顶，为局部变量分配空间后)
| final_val |  (bp+8)  <-- final_val 的地址
| temp_val  |  (bp+7)  <-- temp_val 的地址
| ret       |  (bp+6)  <-- ret 的地址
| b         |  (bp+5)  <-- 形参 b 的地址
| a         |  (bp+4)  <-- 形参 a 的地址
| RA        |  (bp+2)  <-- 返回地址 (Return Address)
| DL        |  (bp+1)  <-- 动态链 (Dynamic Link)
| SL        |  (bp+0)  <-- 静态链 (Static Link)，新的 BP 指向此处
| 5         |  (bp-1)  <-- lod 0 -1 读取的值 (第二个实参)
| 3         |  (bp-2)  <-- lod 0 -2 读取的值 (第一个实参)
| ...        |
当被调用函数执行 opr 0, 0 指令返回时，虚拟机会根据约定精确地恢复调用者的 bp，并将 sp 指针重置到参数区域之下。
```

---

5. 运行方式与图形用户界面 (GUI)

通过 GUI.py 启动的图形界面为用户提供了一个集成开发环境。

5.1 界面布局

GUI 界面 (CompilerGUI 类) 分为左右两个可调整的主窗格:

- **左侧面板** (垂直分割):
  - **源代码区**: 用户在此编写和编辑代码。
  - **符号表 & 结构体定义区**: 编译成功后, 详细显示符号表和结构体布局信息。
  - **底部标签页**:
    - **状态/错误**: 显示编译和运行过程中的状态信息或详细的错误回溯。
    - **程序输出 (ANS)**: 显示由 output 语句产生的程序运行结果。
  - **控制按钮区**:
    - 编译并运行: 核心按钮, 触发整个编译和执行流程。
    - 清空: 清空所有文本框。
    - +/-: 调整界面字体大小。
- **右侧面板**:
  - **生成的 VM 代码区**: 编译成功后, 显示生成的 P-Code 指令及其注释。

5.2 运行流程

1. 用户在“源代码区”输入代码。
2. 点击“编译并运行”按钮。
3. GUI 调用 compile_and_run 方法:

   a. 清空旧的输出内容。

   b. 实例化 Lexer, ParserAndGenerator。

   c. 调用 parser.parse(), 此过程会完成词法分析、语法分析、符号表构建和 P-Code 生成。

d. 如果编译成功:
    i. 将生成的 P-Code 格式化并显示在“VM 代码区”。
    ii. 捕获 symtable.display() 的输出并显示在“符号表区”。
    iii. 实例化 VM，并将生成的 P-Code 传入。
    iv. 将 GUI 的输入处理函数 gui_input_handler 设置给 VM 实例，以便在执行 inp 指令时弹出对话框。
    v. 调用 vm.run() 执行代码。
    vi. 将 VM 的输出结果显示在“程序输出 (ANS)”标签页。
    vii. 在“状态”区显示“编译和执行成功”。
e. 如果编译或运行过程中出现任何异常:
    i. 捕获异常信息和完整的 traceback。
    ii. 将详细的错误信息显示在“状态/错误”标签页。

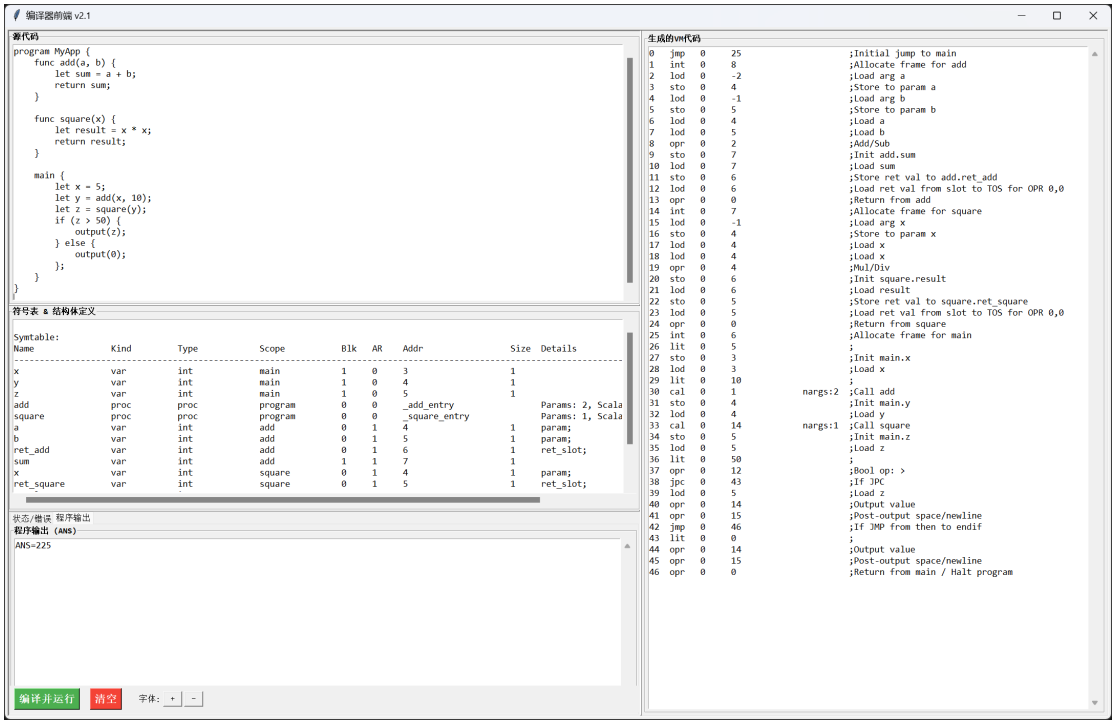
---

6. 样例分析

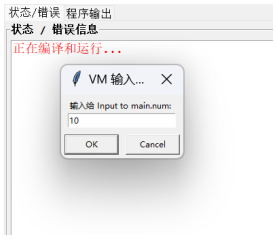
samples.ipynb 中提供了多个测试用例，每个用例都旨在测试编译器的特定功能。

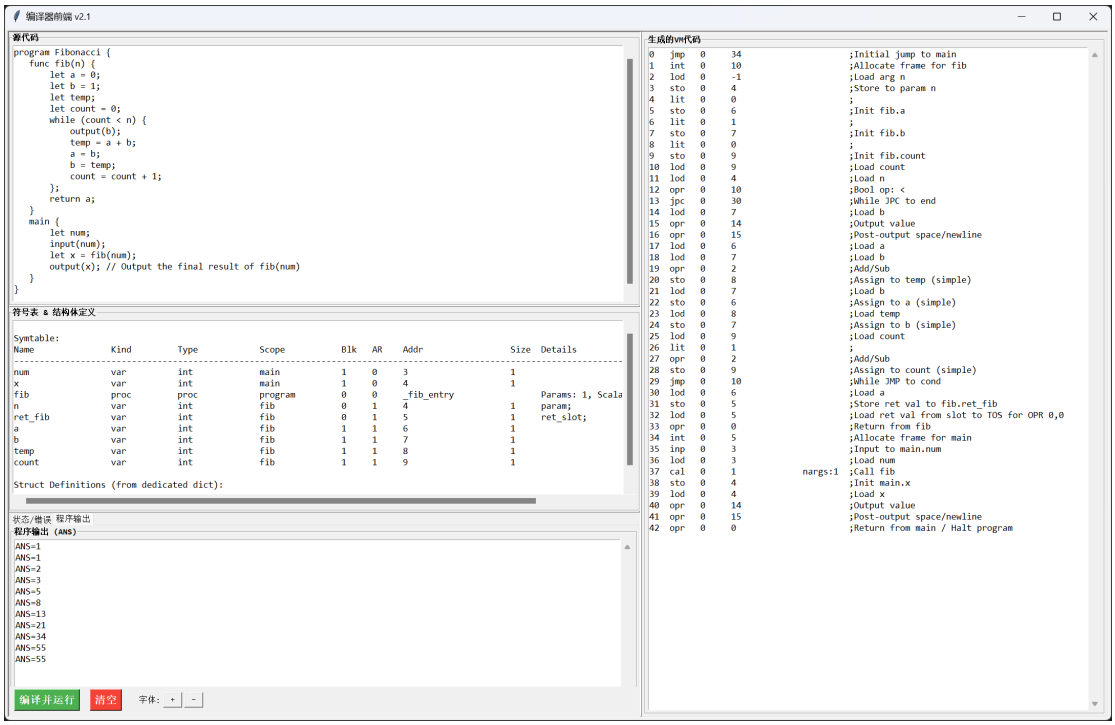
- MyApp: 测试基本的函数定义、调用、if-else 分支和 output 语句。
- Fibonacci: 测试 while 循环、变量的连续赋值和 input 功能。
- GCD: 测试更复杂的 while 循环逻辑和算术运算 (特别是除法和减法)。
- Factorial: 核心测试点是 **单一出口递归函数调用**。检验 cal 和 opr 0 指令能否正确构建和拆除多层活动记录。
- ChainedCallTest: 测试 **函数链式调用**，检验编译器的调用机制和运行时的栈管理是否正确，并验证 cal 指令的 L 值的动态计算。
    - 多层活动记录: 测试调用链能否正确创建和销毁各自的活动记录。
    - 返回值传递: 检验内层函数 (add_offset) 的返回值能否被其中间调用者 (calculate_series) 正确接收和使用，并最终将计算结果返回给顶层调用者 (main)。
- Shadowing: 重点测试 **词法作用域** 和 **变量遮蔽**。验证编译器能否在不同块级作用域 (if, else, while 内部) 正确创建和查找变量，lod/sto 指令的 L 参数是否计算正确。
- MyShape: 测试 **结构体 (struct)** 功能，包括:
    - 声明 struct 类型的变量 (pt1) 和 struct 类型的数组 (colors)。
    - 通过 . 和 [] 访问成员 (如 pt1.x, colors[i].r)，检验 lods/stos 间接寻址指令的正确性。
    - 将 struct 作为参数**按值传递**给函数，检验复杂数据类型在函数调用时的复制和访问机制。
- StudentGrades: 进一步验证 **结构体数组** 和 **将结构体作为参数** 的功能，特别是当结构体内部包含数组成员时 (s.scores[i])。
- PointOperations: 测试结构体，包含 **嵌套的复杂数据结构**。
    - 测试对深层嵌套成员 (如 sh.points[i].x) 的访问，高要求检验编译器的地址计算和 lods/stos 指令的生成逻辑。
    - 检验符号表对复杂类型大小和偏移的计算能力。
- BubbleSortTest: 测试 **冒泡排序** 算法和将 **数组作为结构体成员** 进行参数传递。
- DivideByZeroTest: 测试 **出错和异常管理**。

测试样例 1:

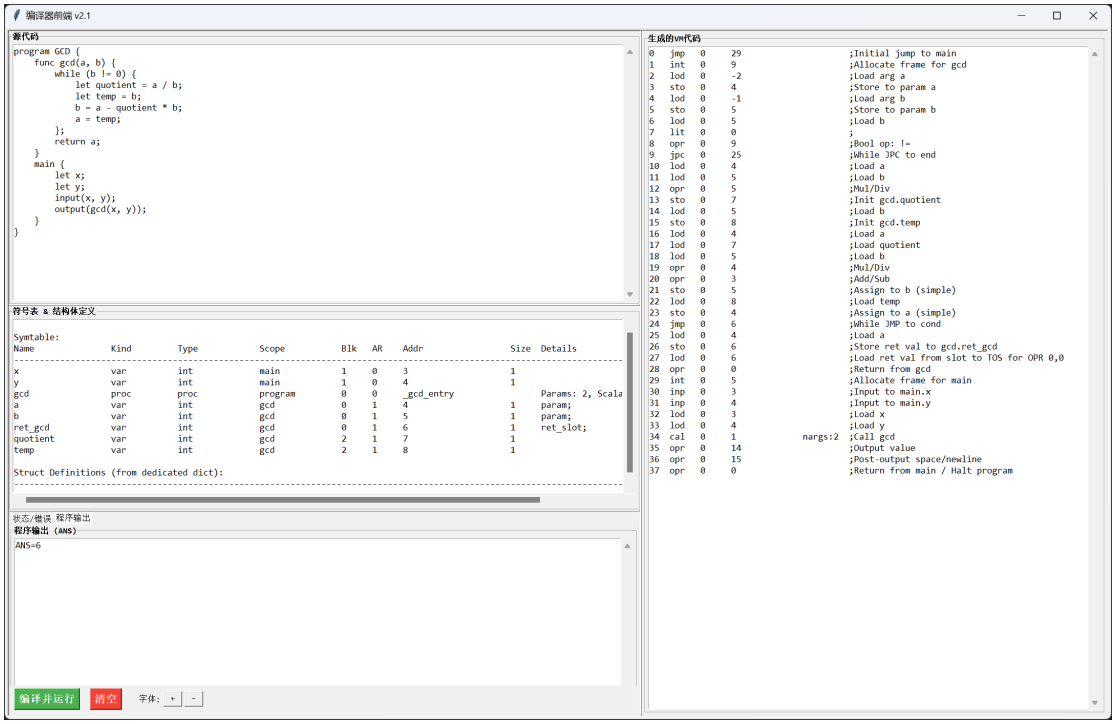

编译器前端 v2.1
源代码
program MyApp {
func add(a, b) {
let sum = a + b;
return sum;
}
func square(x) {
let result = x * x;
return result;
}
main {
let x = 5;
let y = add(x, 10);
let z = square(y);
if (z > 50) {
output(z);
} else {
output(0);
};
}
}
符号表 & 结构体定义
Symtable:
Name Kind Type Scope Blk AR Addr Size Details
x var int main 1 0 3 1
y var int main 1 0 4 1
z var int main 1 0 5 1
add proc proc program 0 0 _add_entry Params: 2, Scala
square proc proc program 0 0 _square_entry Params: 1, Scala
a var int add 0 1 4 1 param;
b var int add 0 1 5 1 param;
ret_add var int add 0 1 6 1 ret_slot;
sum var int add 1 1 7 1
x var int square 0 1 4 1 param;
ret_square var int square 0 1 5 1 ret_slot;
状态/错误 程序输出
程序输出 (ANS)
ANS=225
编译并运行
清空
字体: + -
生成的VM代码
0 jmp 0 25 ;Initial jump to main
1 int 0 8 ;Allocate frame for add
2 lod 0 -2 ;Load arg a
3 sto 0 4 ;Store to param a
4 lod 0 -1 ;Load arg b
5 sto 0 5 ;Store to param b
6 lod 0 4 ;Load a
7 lod 0 5 ;Load b
8 opr 0 2 ;Add/Sub
9 sto 0 7 ;Init add.sum
10 lod 0 7 ;Load sum
11 sto 0 6 ;Store ret val to add.ret_add
12 lod 0 6 ;Load ret val from slot to TOS for OPR 0,0
13 opr 0 0 ;Return from add
14 int 0 7 ;Allocate frame for square
15 lod 0 -1 ;Load arg x
16 sto 0 4 ;Store to param x
17 lod 0 4 ;Load x
18 lod 0 4 ;Load x
19 opr 0 4 ;Mul/Div
20 sto 0 6 ;Init square.result
21 lod 0 6 ;Load result
22 sto 0 5 ;Store ret val to square.ret_square
23 lod 0 5 ;Load ret val from slot to TOS for OPR 0,0
24 opr 0 0 ;Return from square
25 int 0 6 ;Allocate frame for main
26 lit 0 5 ;
27 sto 0 3 ;Init main.x
28 lod 0 3 ;Load x
29 lit 0 10 ;
30 cal 0 1 nargs:2 ;Call add
31 sto 0 4 ;Init main.y
32 lod 0 4 ;Load y
33 cal 0 14 nargs:1 ;Call square
34 sto 0 5 ;Init main.z
35 lod 0 5 ;Load z
36 lit 0 50 ;
37 opr 0 12 ;Bool op: >
38 jpc 0 43 ;If JPC
39 lod 0 5 ;Load z
40 opr 0 14 ;Output value
41 opr 0 15 ;Post-output space/newline
42 jmp 0 46 ;If JMP from then to endif
43 lit 0 0 ;
44 opr 0 14 ;Output value
45 opr 0 15 ;Post-output space/newline
46 opr 0 0 ;Return from main / Halt program

测试样例 2:

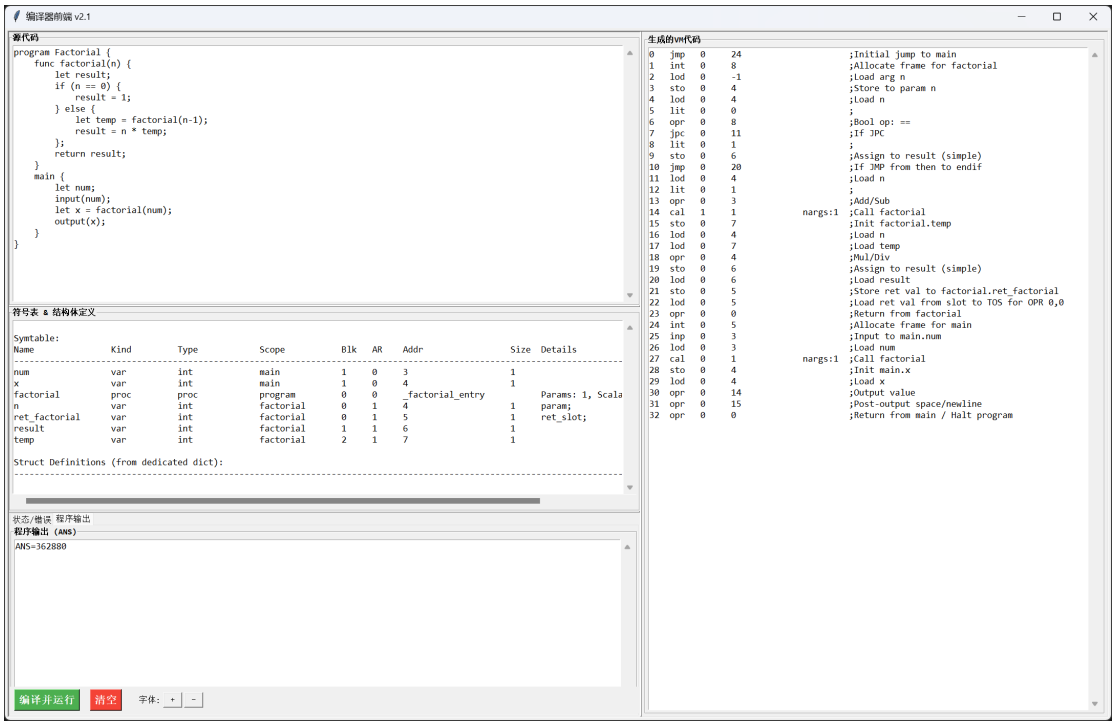

状态/错误 程序输出
状态 / 错误信息
正在编译和运行...
VM 输入...
输入给 Input to main.num:
10
OK
Cancel


编译器前端 v2.1
源代码
program Fibonacci {
func fib(n) {
let a = 0;
let b = 1;
let temp;
let count = 0;
while (count < n) {
output(b);
temp = a + b;
a = b;
b = temp;
count = count + 1;
};
return a;
}
main {
let num;
input(num);
let x = fib(num);
output(x); // Output the final result of fib(num)
}
}
符号表 & 结构体定义
Symtable:
Name Kind Type Scope Blk AR Addr Size Details
num var int main 1 0 3 1
x var int main 1 0 4 1
fib proc proc program 0 0 _fib_entry Params: 1, Scala
n var int fib 0 1 4 1 param;
ret_fib var int fib 0 1 5 1 ret_slot;
a var int fib 1 1 6 1
b var int fib 1 1 7 1
temp var int fib 1 1 8 1
count var int fib 1 1 9 1
Struct Definitions (from dedicated dict):
状态/错误 程序输出
程序输出 (ANS)
ANS=1
ANS=1
ANS=2
ANS=3
ANS=5
ANS=8
ANS=13
ANS=21
ANS=34
ANS=55
ANS=55
编译并运行
清空
字体: + -
生成的VM代码
0 jmp 0 34 ;Initial jump to main
1 int 0 10 ;Allocate frame for fib
2 lod 0 -1 ;Load arg n
3 sto 0 4 ;Store to param n
4 lit 0 0 ;
5 sto 0 6 ;Init fib.a
6 lit 0 1 ;
7 sto 0 7 ;Init fib.b
8 lit 0 0 ;
9 sto 0 9 ;Init fib.count
10 lod 0 9 ;Load count
11 lod 0 4 ;Load n
12 opr 0 10 ;Bool op: <
13 jpc 0 30 ;While JPC to end
14 lod 0 7 ;Load b
15 opr 0 14 ;Output value
16 opr 0 15 ;Post-output space/newline
17 lod 0 6 ;Load a
18 lod 0 7 ;Load b
19 opr 0 2 ;Add/Sub
20 sto 0 8 ;Assign to temp (simple)
21 lod 0 7 ;Load b
22 sto 0 6 ;Assign to a (simple)
23 lod 0 8 ;Load temp
24 sto 0 7 ;Assign to b (simple)
25 lod 0 9 ;Load count
26 lit 0 1 ;
27 opr 0 2 ;Add/Sub
28 sto 0 9 ;Assign to count (simple)
29 jmp 0 10 ;While JMP to cond
30 lod 0 6 ;Load a
31 sto 0 5 ;Store ret val to fib.ret_fib
32 lod 0 5 ;Load ret val from slot to TOS for OPR 0,0
33 opr 0 0 ;Return from fib
34 int 0 5 ;Allocate frame for main
35 inp 0 3 ;Input to main.num
36 lod 0 3 ;Load num
37 cal 0 1 nargs:1 ;Call fib
38 sto 0 4 ;Init main.x
39 lod 0 4 ;Load x
40 opr 0 14 ;Output value
41 opr 0 15 ;Post-output space/newline
42 opr 0 0 ;Return from main / Halt program

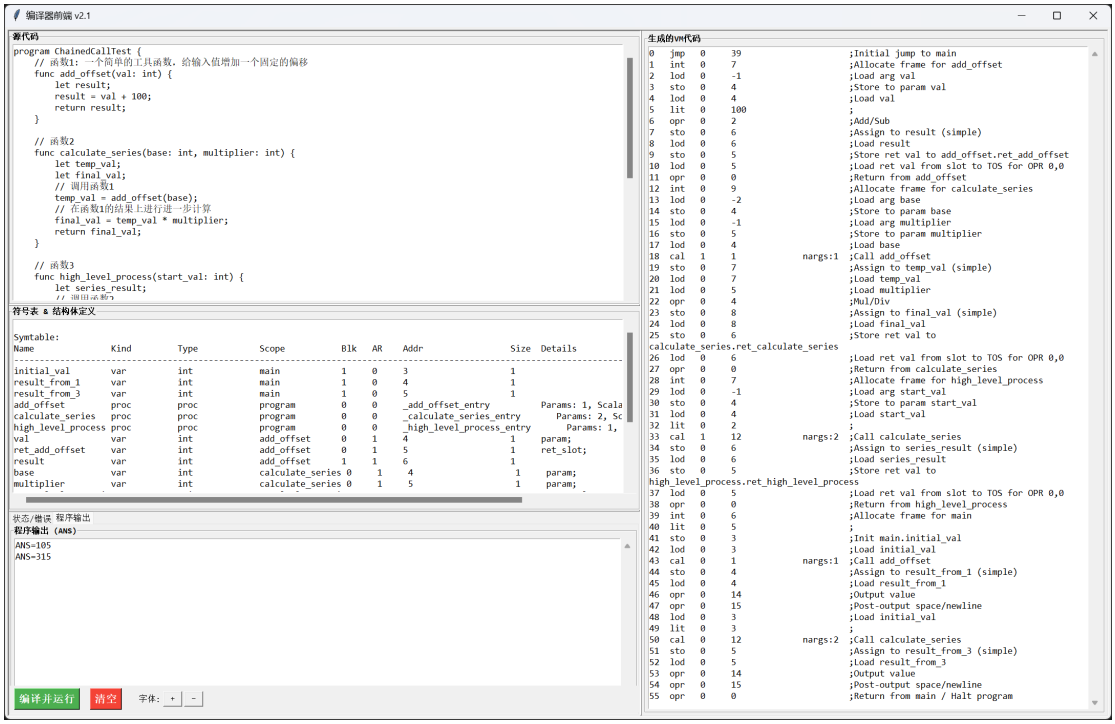
测试样例 3:


编译器前端 v2.1
源代码
program GCD {
func gcd(a, b) {
while (b != 0) {
let quotient = a / b;
let temp = b;
b = a - quotient * b;
a = temp;
};
return a;
}
main {
let x;
let y;
input(x, y);
output(gcd(x, y));
}
}
符号表 & 结构体定义
Symtable:
Name Kind Type Scope Blk AR Addr Size Details
x var int main 1 0 3 1
y var int main 1 0 4 1
gcd proc proc program 0 0 _gcd_entry Params: 2, Scala
a var int gcd 0 1 4 1 param;
b var int gcd 0 1 5 1 param;
ret_gcd var int gcd 0 1 6 1 ret_slot;
quotient var int gcd 2 1 7 1
temp var int gcd 2 1 8 1
Struct Definitions (from dedicated dict):
状态/错误 程序输出
程序输出 (ANS)
ANS=6
编译并运行
清空
字体: + -
生成的VM代码
0 jmp 0 29 ;Initial jump to main
1 int 0 9 ;Allocate frame for gcd
2 lod 0 -2 ;Load arg a
3 sto 0 4 ;Store to param a
4 lod 0 -1 ;Load arg b
5 sto 0 5 ;Store to param b
6 lod 0 5 ;Load b
7 lit 0 0 ;
8 opr 0 9 ;Bool op: !=
9 jpc 0 25 ;While JPC to end
10 lod 0 4 ;Load a
11 lod 0 5 ;Load b
12 opr 0 5 ;Mul/Div
13 sto 0 7 ;Init gcd.quotient
14 lod 0 5 ;Load b
15 sto 0 8 ;Init gcd.temp
16 lod 0 4 ;Load a
17 lod 0 7 ;Load quotient
18 lod 0 5 ;Load b
19 opr 0 4 ;Mul/Div
20 opr 0 3 ;Add/Sub
21 sto 0 5 ;Assign to b (simple)
22 lod 0 8 ;Load temp
23 sto 0 4 ;Assign to a (simple)
24 jmp 0 6 ;While JMP to cond
25 lod 0 4 ;Load a
26 sto 0 6 ;Store ret val to gcd.ret_gcd
27 lod 0 6 ;Load ret val from slot to TOS for OPR 0,0
28 opr 0 0 ;Return from gcd
29 int 0 5 ;Allocate frame for main
30 inp 0 3 ;Input to main.x
31 inp 0 4 ;Input to main.y
32 lod 0 3 ;Load x
33 lod 0 4 ;Load y
34 cal 0 1 nargs:2 ;Call gcd
35 opr 0 14 ;Output value
36 opr 0 15 ;Post-output space/newline
37 opr 0 0 ;Return from main / Halt program

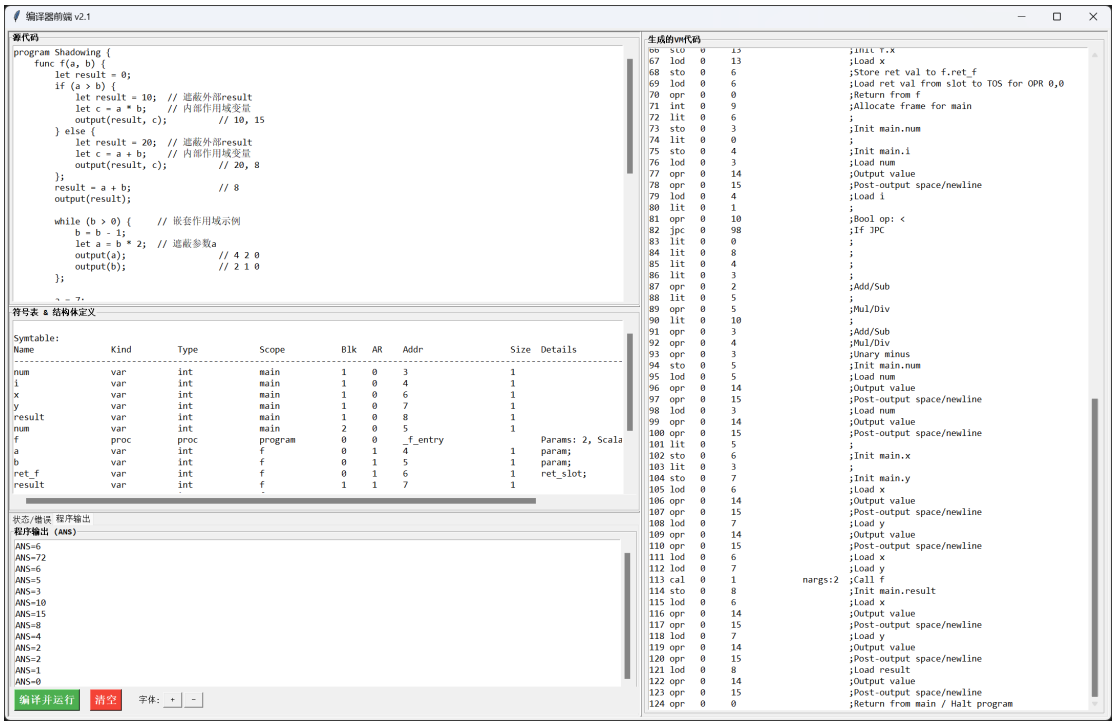
测试样例 4:


编译器前端 v2.1
源代码
program Factorial {
func factorial(n) {
let result;
if (n == 0) {
result = 1;
} else {
let temp = factorial(n-1);
result = n * temp;
};
return result;
}
main {
let num;
input(num);
let x = factorial(num);
output(x);
}
}
符号表 & 结构体定义
Symtable:
Name Kind Type Scope Blk AR Addr Size Details
num var int main 1 0 3 1
x var int main 1 0 4 1
factorial proc proc program 0 0 _factorial_entry Params: 1, Scala
n var int factorial 0 1 4 1 param;
ret_factorial var int factorial 0 1 5 1 ret_slot;
result var int factorial 1 1 6 1
temp var int factorial 2 1 7 1
Struct Definitions (from dedicated dict):
状态/错误 程序输出
程序输出 (ANS)
ANS=362880
编译并运行
清空
字体: + -
生成的VM代码
0 jmp 0 24 ;Initial jump to main
1 int 0 8 ;Allocate frame for factorial
2 lod 0 -1 ;Load arg n
3 sto 0 4 ;Store to param n
4 lod 0 4 ;Load n
5 lit 0 0 ;
6 opr 0 8 ;Bool op: ==
7 jpc 0 11 ;If JPC
8 lit 0 1 ;
9 sto 0 6 ;Assign to result (simple)
10 jmp 0 20 ;If JMP from then to endif
11 lod 0 4 ;Load n
12 lit 0 1 ;
13 opr 0 3 ;Add/Sub
14 cal 1 1 nargs:1 ;Call factorial
15 sto 0 7 ;Init factorial.temp
16 lod 0 4 ;Load n
17 lod 0 7 ;Load temp
18 opr 0 4 ;Mul/Div
19 sto 0 6 ;Assign to result (simple)
20 lod 0 6 ;Load result
21 sto 0 5 ;Store ret val to factorial.ret_factorial
22 lod 0 5 ;Load ret val from slot to TOS for OPR 0,0
23 opr 0 0 ;Return from factorial
24 int 0 5 ;Allocate frame for main
25 inp 0 3 ;Input to main.num
26 lod 0 3 ;Load num
27 cal 0 1 nargs:1 ;Call factorial
28 sto 0 4 ;Init main.x
29 lod 0 4 ;Load x
30 opr 0 14 ;Output value
31 opr 0 15 ;Post-output space/newline
32 opr 0 0 ;Return from main / Halt program

测试样例 5:

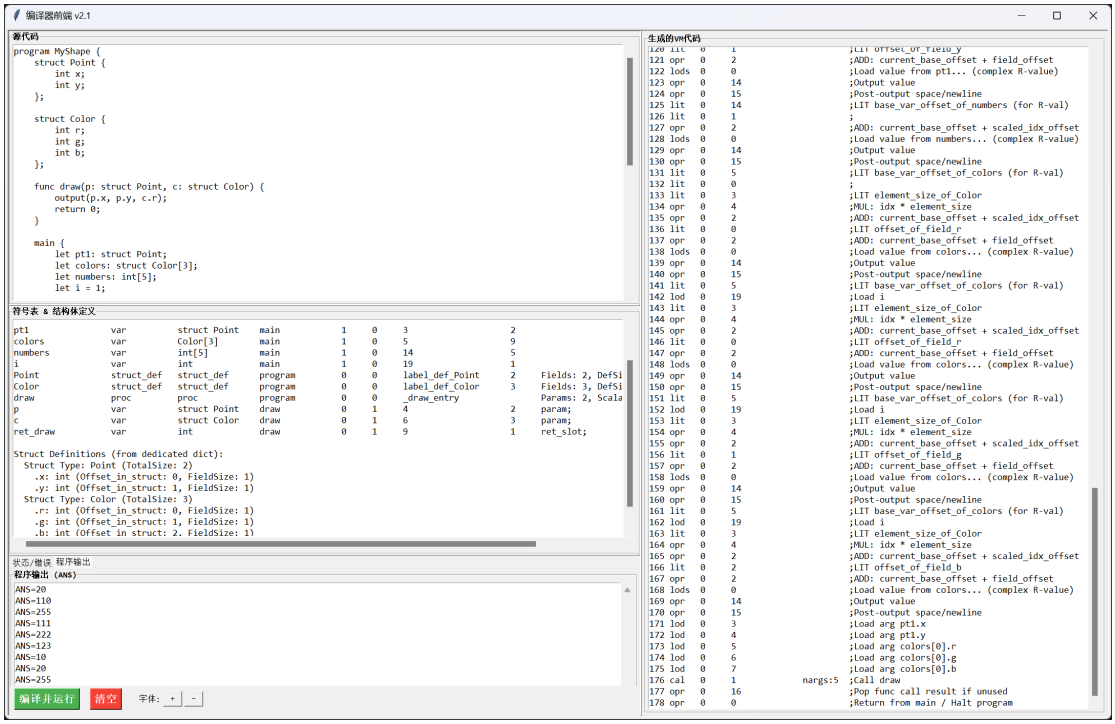


测试样例 6:

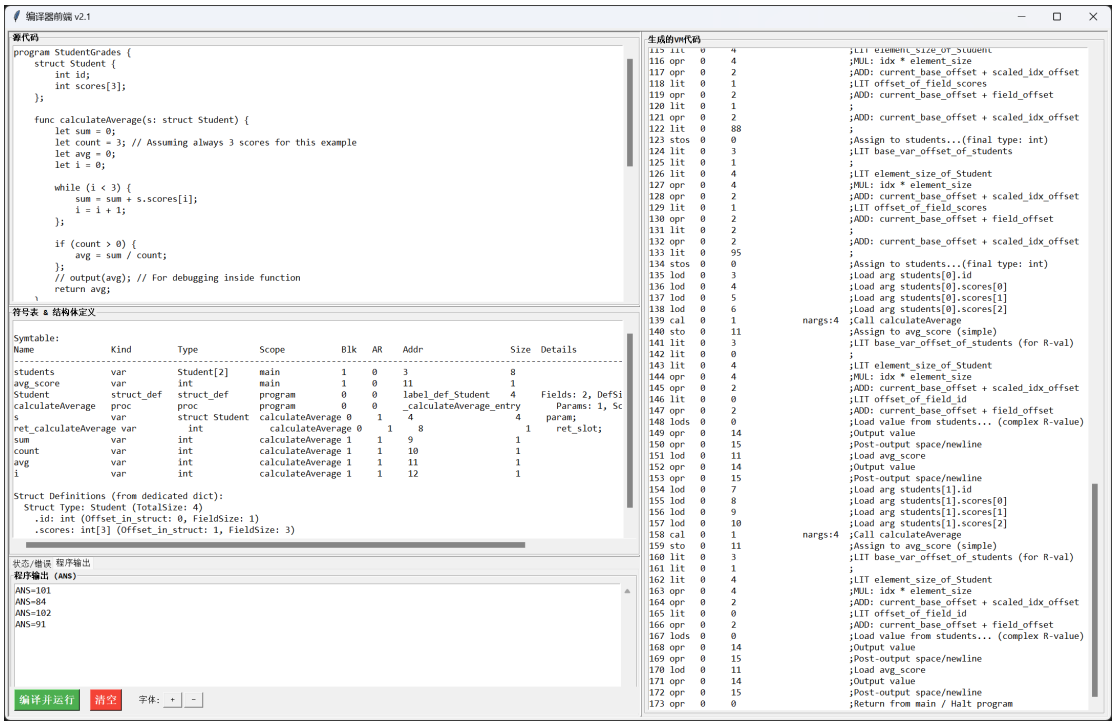



```
ANS=10
ANS=15
ANS=8
ANS=4
ANS=2
ANS=2
ANS=1
ANS=0
ANS=0
ANS=5
```

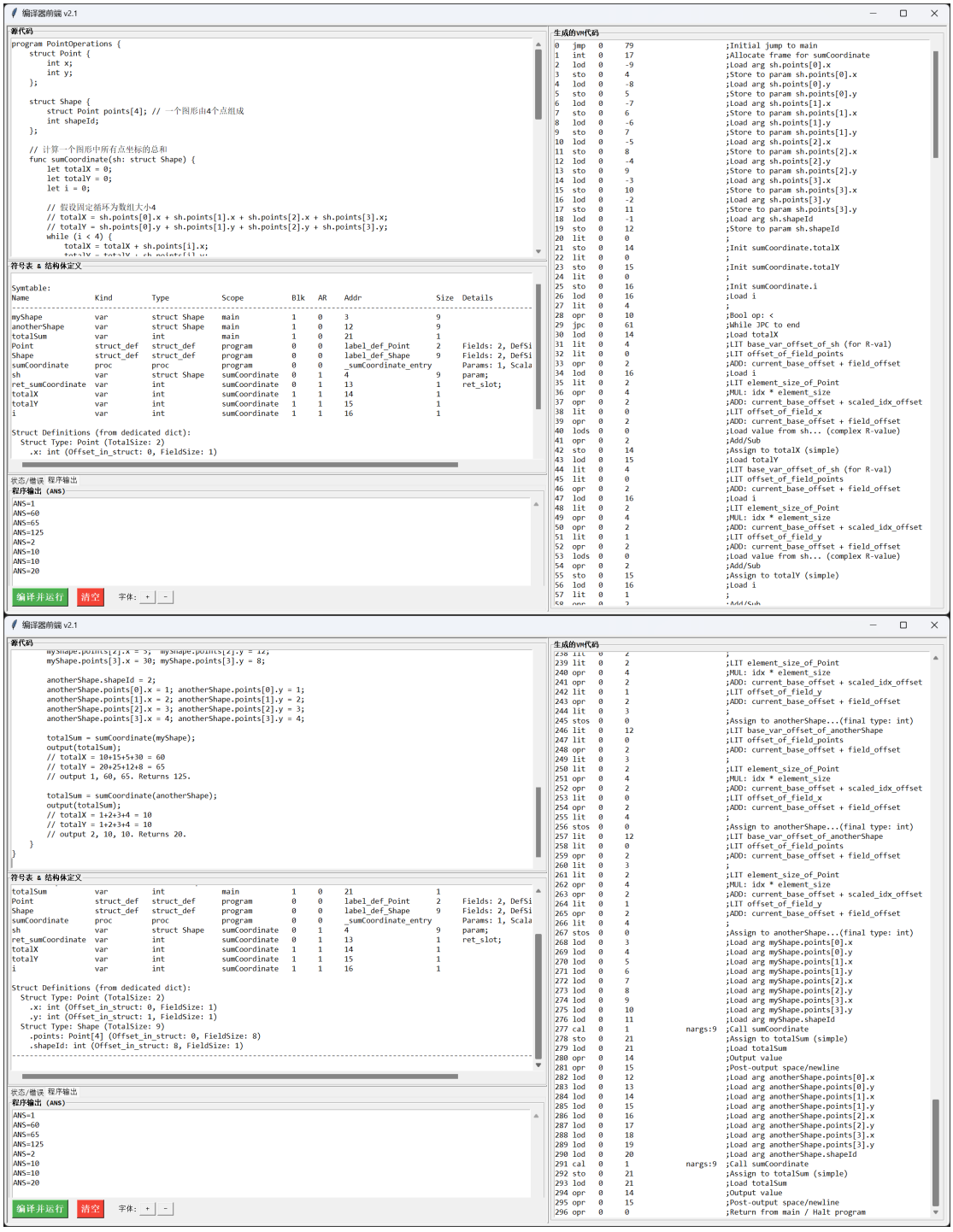
测试样例 7:


编译器前端 v2.1
源代码
program MyShape {
struct Point {
int x;
int y;
};
struct Color {
int r;
int g;
int b;
};
func draw(p: struct Point, c: struct Color) {
output(p.x, p.y, c.r);
return 0;
}
main {
let pt1: struct Point;
let colors: struct Color[3];
let numbers: int[5];
let i = 1;
符号表 & 结构体定义
pt1 var struct Point main 1 0 3 2
colors var Color[3] main 1 0 5 9
numbers var int[5] main 1 0 14 5
i var int main 1 0 19 1
Point struct_def struct_def program 0 0 label_def_Point 2 Fields: 2, DefSi
Color struct_def struct_def program 0 0 label_def_Color 3 Fields: 3, DefSi
draw proc proc program 0 0 _draw_entry Params: 2, Scala
p var struct Point draw 0 1 4 2 param;
c var struct Color draw 0 1 6 3 param;
ret_draw var int draw 0 1 9 1 ret_slot;
Struct Definitions (from dedicated dict):
Struct Type: Point (TotalSize: 2)
.x: int (Offset_in_struct: 0, FieldSize: 1)
.y: int (Offset_in_struct: 1, FieldSize: 1)
Struct Type: Color (TotalSize: 3)
.r: int (Offset_in_struct: 0, FieldSize: 1)
.g: int (Offset_in_struct: 1, FieldSize: 1)
.b: int (Offset_in_struct: 2, FieldSize: 1)
状态/错误 程序输出
程序输出 (ANS)
ANS=20
ANS=110
ANS=255
ANS=111
ANS=222
ANS=123
ANS=10
ANS=20
ANS=255
编译并运行
清空
字体: + -
生成的vm代码
121 opr 0 2 ;ADD: current_base_offset + field_offset
122 lods 0 0 ;Load value from pt1... (complex R-value)
123 opr 0 14 ;Output value
124 opr 0 15 ;Post-output space/newline
125 lit 0 14 ;LIT base_var_offset_of_numbers (for R-val)
126 lit 0 1 ;
127 opr 0 2 ;ADD: current_base_offset + scaled_idx_offset
128 lods 0 0 ;Load value from numbers... (complex R-value)
129 opr 0 14 ;Output value
130 opr 0 15 ;Post-output space/newline
131 lit 0 5 ;LIT base_var_offset_of_colors (for R-val)
132 lit 0 0 ;
133 lit 0 3 ;LIT element_size_of_Color
134 opr 0 4 ;MUL: idx * element_size
135 opr 0 2 ;ADD: current_base_offset + scaled_idx_offset
136 lit 0 0 ;LIT offset_of_field_r
137 opr 0 2 ;ADD: current_base_offset + field_offset
138 lods 0 0 ;Load value from colors... (complex R-value)
139 opr 0 14 ;Output value
140 opr 0 15 ;Post-output space/newline
141 lit 0 5 ;LIT base_var_offset_of_colors (for R-val)
142 lod 0 19 ;Load i
143 lit 0 3 ;LIT element_size_of_Color
144 opr 0 4 ;MUL: idx * element_size
145 opr 0 2 ;ADD: current_base_offset + scaled_idx_offset
146 lit 0 1 ;LIT offset_of_field_g
147 opr 0 2 ;ADD: current_base_offset + field_offset
148 lods 0 0 ;Load value from colors... (complex R-value)
149 opr 0 14 ;Output value
150 opr 0 15 ;Post-output space/newline
151 lit 0 5 ;LIT base_var_offset_of_colors (for R-val)
152 lod 0 19 ;Load i
153 lit 0 3 ;LIT element_size_of_Color
154 opr 0 4 ;MUL: idx * element_size
155 opr 0 2 ;ADD: current_base_offset + scaled_idx_offset
156 lit 0 2 ;LIT offset_of_field_b
157 opr 0 2 ;ADD: current_base_offset + field_offset
158 lods 0 0 ;Load value from colors... (complex R-value)
159 opr 0 14 ;Output value
160 opr 0 15 ;Post-output space/newline
161 lit 0 5 ;LIT base_var_offset_of_colors (for R-val)
162 lod 0 19 ;Load i
163 lit 0 3 ;LIT element_size_of_Color
164 opr 0 4 ;MUL: idx * element_size
165 opr 0 2 ;ADD: current_base_offset + scaled_idx_offset
166 lit 0 2 ;LIT offset_of_field_b
167 opr 0 2 ;ADD: current_base_offset + field_offset
168 lods 0 0 ;Load value from colors... (complex R-value)
169 opr 0 14 ;Output value
170 opr 0 15 ;Post-output space/newline
171 lod 0 3 ;Load arg pt1.x
172 lod 0 4 ;Load arg pt1.y
173 lod 0 5 ;Load arg colors[0].r
174 lod 0 6 ;Load arg colors[0].g
175 lod 0 7 ;Load arg colors[0].b
176 cal 0 1 nargs:5 ;Call draw
177 opr 0 16 ;Pop func call result if unused
178 opr 0 0 ;Return from main / Halt program

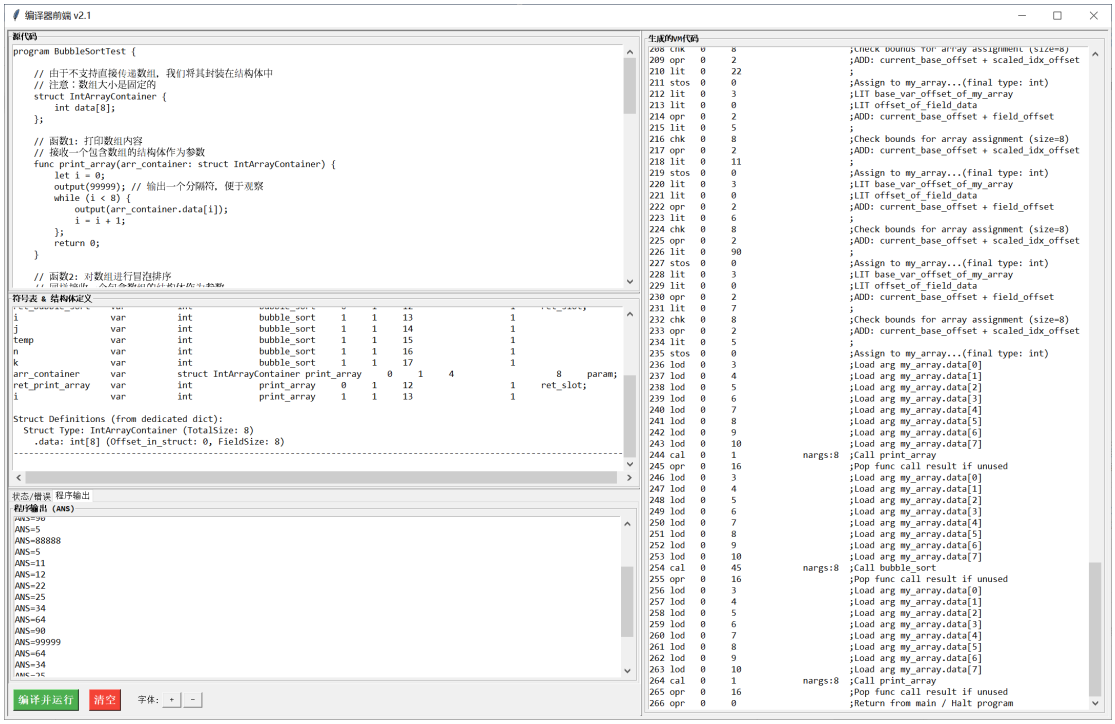
测试样例 8:


编译器前端 v2.1
源代码
program StudentGrades {
struct Student {
int id;
int scores[3];
};
func calculateAverage(s: struct Student) {
let sum = 0;
let count = 3; // Assuming always 3 scores for this example
let avg = 0;
let i = 0;
while (i < 3) {
sum = sum + s.scores[i];
i = i + 1;
};
if (count > 0) {
avg = sum / count;
};
// output(avg); // For debugging inside function
return avg;
符号表 & 结构体定义
Symtable:
Name Kind Type Scope Blk AR Addr Size Details
students var Student[2] main 1 0 3 8
avg_score var int main 1 0 11 1
Student struct_def struct_def program 0 0 label_def_Student 4 Fields: 2, DefSi
calculateAverage proc proc program 0 0 _calculateAverage_entry Params: 1, Sc
s var struct Student calculateAverage 0 1 4 4 param;
ret_calculateAverage var int calculateAverage 0 1 8 1 ret_slot;
sum var int calculateAverage 1 1 9 1
count var int calculateAverage 1 1 10 1
avg var int calculateAverage 1 1 11 1
i var int calculateAverage 1 1 12 1
Struct Definitions (from dedicated dict):
Struct Type: Student (TotalSize: 4)
.id: int (Offset_in_struct: 0, FieldSize: 1)
.scores: int[3] (Offset_in_struct: 1, FieldSize: 3)
状态/错误 程序输出
程序输出 (ANS)
ANS=101
ANS=84
ANS=102
ANS=91
编译并运行
清空
字体: + -
生成的vm代码
116 opr 0 4 ;MUL: idx * element_size
117 opr 0 2 ;ADD: current_base_offset + scaled_idx_offset
118 lit 0 1 ;LIT offset_of_field_scores
119 opr 0 2 ;ADD: current_base_offset + field_offset
120 lit 0 1 ;
121 opr 0 2 ;ADD: current_base_offset + scaled_idx_offset
122 lit 0 88 ;
123 stos 0 0 ;Assign to students...(final type: int)
124 lit 0 3 ;LIT base_var_offset_of_students
125 lit 0 1 ;
126 lit 0 4 ;LIT element_size_of_Student
127 opr 0 4 ;MUL: idx * element_size
128 opr 0 2 ;ADD: current_base_offset + scaled_idx_offset
129 lit 0 1 ;LIT offset_of_field_scores
130 opr 0 2 ;ADD: current_base_offset + field_offset
131 lit 0 2 ;
132 opr 0 2 ;ADD: current_base_offset + scaled_idx_offset
133 lit 0 95 ;
134 stos 0 0 ;Assign to students...(final type: int)
135 lod 0 3 ;Load arg students[0].id
136 lod 0 4 ;Load arg students[0].scores[0]
137 lod 0 5 ;Load arg students[0].scores[1]
138 lod 0 6 ;Load arg students[0].scores[2]
139 cal 0 1 nargs:4 ;Call calculateAverage
140 sto 0 11 ;Assign to avg_score (simple)
141 lit 0 3 ;LIT base_var_offset_of_students (for R-val)
142 lit 0 0 ;
143 lit 0 4 ;LIT element_size_of_Student
144 opr 0 4 ;MUL: idx * element_size
145 opr 0 2 ;ADD: current_base_offset + scaled_idx_offset
146 lit 0 0 ;LIT offset_of_field_id
147 opr 0 2 ;ADD: current_base_offset + field_offset
148 lods 0 0 ;Load value from students... (complex R-value)
149 opr 0 14 ;Output value
150 opr 0 15 ;Post-output space/newline
151 lod 0 11 ;Load avg_score
152 opr 0 14 ;Output value
153 opr 0 15 ;Post-output space/newline
154 lod 0 7 ;Load arg students[1].id
155 lod 0 8 ;Load arg students[1].scores[0]
156 lod 0 9 ;Load arg students[1].scores[1]
157 lod 0 10 ;Load arg students[1].scores[2]
158 cal 0 1 nargs:4 ;Call calculateAverage
159 sto 0 11 ;Assign to avg_score (simple)
160 lit 0 3 ;LIT base_var_offset_of_students (for R-val)
161 lit 0 1 ;
162 lit 0 4 ;LIT element_size_of_Student
163 opr 0 4 ;MUL: idx * element_size
164 opr 0 2 ;ADD: current_base_offset + scaled_idx_offset
165 lit 0 0 ;LIT offset_of_field_id
166 opr 0 2 ;ADD: current_base_offset + field_offset
167 lods 0 0 ;Load value from students... (complex R-value)
168 opr 0 14 ;Output value
169 opr 0 15 ;Post-output space/newline
170 lod 0 11 ;Load avg_score
171 opr 0 14 ;Output value
172 opr 0 15 ;Post-output space/newline
173 opr 0 0 ;Return from main / Halt program

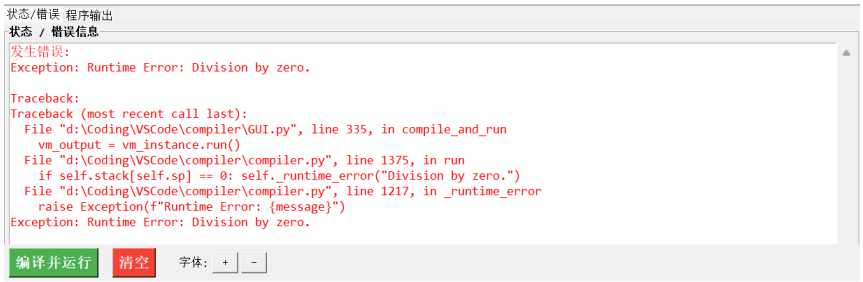
测试样例 9:


编译器前端 v2.1
源代码
program PointOperations {
struct Point {
int x;
int y;
};
struct Shape {
struct Point points[4]; // 一个图形由4个点组成
int shapeId;
};
// 计算一个图形中所有点坐标的总和
func sumCoordinate(sh: struct Shape) {
let totalX = 0;
let totalY = 0;
let i = 0;
// 假设固定循环为数组大小4
// totalX = sh.points[0].x + sh.points[1].x + sh.points[2].x + sh.points[3].x;
// totalY = sh.points[0].y + sh.points[1].y + sh.points[2].y + sh.points[3].y;
while (i < 4) {
totalX = totalX + sh.points[i].x;
符号表 & 结构体定义
Symtable:
Name Kind Type Scope Blk AR Addr Size Details
myShape var struct Shape main 1 0 3 9
anotherShape var struct Shape main 1 0 12 9
totalSum var int main 1 0 21 1
Point struct_def struct_def program 0 0 label_def_Point 2 Fields: 2, DefSi
Shape struct_def struct_def program 0 0 label_def_Shape 9 Fields: 2, DefSi
sumCoordinate proc proc program 0 0 _sumCoordinate_entry Params: 1, Scala
sh var struct Shape sumCoordinate 0 1 4 9 param;
ret_sumCoordinate var int sumCoordinate 0 1 13 1 ret_slot;
totalX var int sumCoordinate 1 1 14 1
totalY var int sumCoordinate 1 1 15 1
i var int sumCoordinate 1 1 16 1
Struct Definitions (from dedicated dict):
Struct Type: Point (TotalSize: 2)
.x: int (Offset_in_struct: 0, FieldSize: 1)
状态/错误 程序输出
程序输出 (ANS)
ANS=1
ANS=60
ANS=65
ANS=125
ANS=2
ANS=10
ANS=10
ANS=20
编译并运行
清空
字体: + -
生成的VM代码
0 jmp 0 79 ;Initial jump to main
1 int 0 17 ;Allocate frame for sumCoordinate
2 lod 0 -9 ;Load arg sh.points[0].x
3 sto 0 4 ;Store to param sh.points[0].x
4 lod 0 -8 ;Load arg sh.points[0].y
5 sto 0 5 ;Store to param sh.points[0].y
6 lod 0 -7 ;Load arg sh.points[1].x
7 sto 0 6 ;Store to param sh.points[1].x
8 lod 0 -6 ;Load arg sh.points[1].y
9 sto 0 7 ;Store to param sh.points[1].y
10 lod 0 -5 ;Load arg sh.points[2].x
11 sto 0 8 ;Store to param sh.points[2].x
12 lod 0 -4 ;Load arg sh.points[2].y
13 sto 0 9 ;Store to param sh.points[2].y
14 lod 0 -3 ;Load arg sh.points[3].x
15 sto 0 10 ;Store to param sh.points[3].x
16 lod 0 -2 ;Load arg sh.points[3].y
17 sto 0 11 ;Store to param sh.points[3].y
18 lod 0 -1 ;Load arg sh.shapeId
19 sto 0 12 ;Store to param sh.shapeId
20 lit 0 0 ;
21 sto 0 14 ;Init sumCoordinate.totalX
22 lit 0 0 ;
23 sto 0 15 ;Init sumCoordinate.totalY
24 lit 0 0 ;
25 sto 0 16 ;Init sumCoordinate.i
26 lod 0 16 ;Load i
27 lit 0 4 ;
28 opr 0 10 ;Bool op: <
29 jpc 0 61 ;While JPC to end
30 lod 0 14 ;Load totalX
31 lit 0 4 ;LIT base_var_offset_of_sh (for R-val)
32 lit 0 0 ;LIT offset_of_field_points
33 opr 0 2 ;ADD: current_base_offset + field_offset
34 lod 0 16 ;Load i
35 lit 0 2 ;LIT element_size_of_Point
36 opr 0 4 ;MUL: idx * element_size
37 opr 0 2 ;ADD: current_base_offset + scaled_idx_offset
38 lit 0 0 ;LIT offset_of_field_x
39 opr 0 2 ;ADD: current_base_offset + field_offset
40 lods 0 0 ;Load value from sh... (complex R-value)
41 opr 0 2 ;Add/Sub
42 sto 0 14 ;Assign to totalX (simple)
43 lod 0 15 ;Load totalY
44 lit 0 4 ;LIT base_var_offset_of_sh (for R-val)
45 lit 0 0 ;LIT offset_of_field_points
46 opr 0 2 ;ADD: current_base_offset + field_offset
47 lod 0 16 ;Load i
48 lit 0 2 ;LIT element_size_of_Point
49 opr 0 4 ;MUL: idx * element_size
50 opr 0 2 ;ADD: current_base_offset + scaled_idx_offset
51 lit 0 1 ;LIT offset_of_field_y
52 opr 0 2 ;ADD: current_base_offset + field_offset
53 lods 0 0 ;Load value from sh... (complex R-value)
54 opr 0 2 ;Add/Sub
55 sto 0 15 ;Assign to totalY (simple)
56 lod 0 16 ;Load i
57 lit 0 1 ;
编译器前端 v2.1
源代码
myShape.points[3].x = 30; myShape.points[3].y = 8;
anotherShape.shapeId = 2;
anotherShape.points[0].x = 1; anotherShape.points[0].y = 1;
anotherShape.points[1].x = 2; anotherShape.points[1].y = 2;
anotherShape.points[2].x = 3; anotherShape.points[2].y = 3;
anotherShape.points[3].x = 4; anotherShape.points[3].y = 4;
totalSum = sumCoordinate(myShape);
output(totalSum);
// totalX = 10+15+5+30 = 60
// totalY = 20+25+12+8 = 65
// output 1, 60, 65. Returns 125.
totalSum = sumCoordinate(anotherShape);
output(totalSum);
// totalX = 1+2+3+4 = 10
// totalY = 1+2+3+4 = 10
// output 2, 10, 10. Returns 20.
}
}
符号表 & 结构体定义
totalSum var int main 1 0 21 1
Point struct_def struct_def program 0 0 label_def_Point 2 Fields: 2, DefSi
Shape struct_def struct_def program 0 0 label_def_Shape 9 Fields: 2, DefSi
sumCoordinate proc proc program 0 0 _sumCoordinate_entry Params: 1, Scala
sh var struct Shape sumCoordinate 0 1 4 9 param;
ret_sumCoordinate var int sumCoordinate 0 1 13 1 ret_slot;
totalX var int sumCoordinate 1 1 14 1
totalY var int sumCoordinate 1 1 15 1
i var int sumCoordinate 1 1 16 1
Struct Definitions (from dedicated dict):
Struct Type: Point (TotalSize: 2)
.x: int (Offset_in_struct: 0, FieldSize: 1)
.y: int (Offset_in_struct: 1, FieldSize: 1)
Struct Type: Shape (TotalSize: 9)
.points: Point[4] (Offset_in_struct: 0, FieldSize: 8)
.shapeId: int (Offset_in_struct: 8, FieldSize: 1)
状态/错误 程序输出
程序输出 (ANS)
ANS=1
ANS=60
ANS=65
ANS=125
ANS=2
ANS=10
ANS=10
ANS=20
编译并运行
清空
字体: + -
生成的VM代码
239 lit 0 2 ;LIT element_size_of_Point
240 opr 0 4 ;MUL: idx * element_size
241 opr 0 2 ;ADD: current_base_offset + scaled_idx_offset
242 lit 0 1 ;LIT offset_of_field_y
243 opr 0 2 ;ADD: current_base_offset + field_offset
244 lit 0 3 ;
245 stos 0 0 ;Assign to anotherShape...(final type: int)
246 lit 0 12 ;LIT base_var_offset_of_anotherShape
247 lit 0 0 ;LIT offset_of_field_points
248 opr 0 2 ;ADD: current_base_offset + field_offset
249 lit 0 3 ;
250 lit 0 2 ;LIT element_size_of_Point
251 opr 0 4 ;MUL: idx * element_size
252 opr 0 2 ;ADD: current_base_offset + scaled_idx_offset
253 lit 0 0 ;LIT offset_of_field_x
254 opr 0 2 ;ADD: current_base_offset + field_offset
255 lit 0 4 ;
256 stos 0 0 ;Assign to anotherShape...(final type: int)
257 lit 0 12 ;LIT base_var_offset_of_anotherShape
258 lit 0 0 ;LIT offset_of_field_points
259 opr 0 2 ;ADD: current_base_offset + field_offset
260 lit 0 3 ;
261 lit 0 2 ;LIT element_size_of_Point
262 opr 0 4 ;MUL: idx * element_size
263 opr 0 2 ;ADD: current_base_offset + scaled_idx_offset
264 lit 0 1 ;LIT offset_of_field_y
265 opr 0 2 ;ADD: current_base_offset + field_offset
266 lit 0 4 ;
267 stos 0 0 ;Assign to anotherShape...(final type: int)
268 lod 0 3 ;Load arg myShape.points[0].x
269 lod 0 4 ;Load arg myShape.points[0].y
270 lod 0 5 ;Load arg myShape.points[1].x
271 lod 0 6 ;Load arg myShape.points[1].y
272 lod 0 7 ;Load arg myShape.points[2].x
273 lod 0 8 ;Load arg myShape.points[2].y
274 lod 0 9 ;Load arg myShape.points[3].x
275 lod 0 10 ;Load arg myShape.points[3].y
276 lod 0 11 ;Load arg myShape.shapeId
277 cal 0 1 nargs:9 ;Call sumCoordinate
278 sto 0 21 ;Assign to totalSum (simple)
279 lod 0 21 ;Load totalSum
280 opr 0 14 ;Output value
281 opr 0 15 ;Post-output space/newline
282 lod 0 12 ;Load arg anotherShape.points[0].x
283 lod 0 13 ;Load arg anotherShape.points[0].y
284 lod 0 14 ;Load arg anotherShape.points[1].x
285 lod 0 15 ;Load arg anotherShape.points[1].y
286 lod 0 16 ;Load arg anotherShape.points[2].x
287 lod 0 17 ;Load arg anotherShape.points[2].y
288 lod 0 18 ;Load arg anotherShape.points[3].x
289 lod 0 19 ;Load arg anotherShape.points[3].y
290 lod 0 20 ;Load arg anotherShape.shapeId
291 cal 0 1 nargs:9 ;Call sumCoordinate
292 sto 0 21 ;Assign to totalSum (simple)
293 lod 0 21 ;Load totalSum
294 opr 0 14 ;Output value
295 opr 0 15 ;Post-output space/newline
296 opr 0 0 ;Return from main / Halt program

测试样例 10:



测试样例 11:（出错管理）



---

7. 补充

增加了对数组越界访问的异常检测和管理。

- **新定义**：一条新的 P-Code 指令 `chk (check)`。格式：`chk 0 size`。功能：该指令执行时，会检查当前栈顶的值（即数组访问的索引 `index`），判断它是否在 `[0, size-1]` 的有效范围内。如果不在，就抛出运行时错误；否则程序继续执行。发射方式：在使用数组（读和写）的两个路径上都加入该检查逻辑。
- **澄清**：该编译器没有硬编码单个数组的最大长度，而是通过虚拟机的总栈空间限制。在 `Lcompiler.py` 的 `VM` 类中，虚拟机的栈被初始化为一个固定大小的数组：`self.stack = [0] * 8192`，超出后会发生运行时错误。