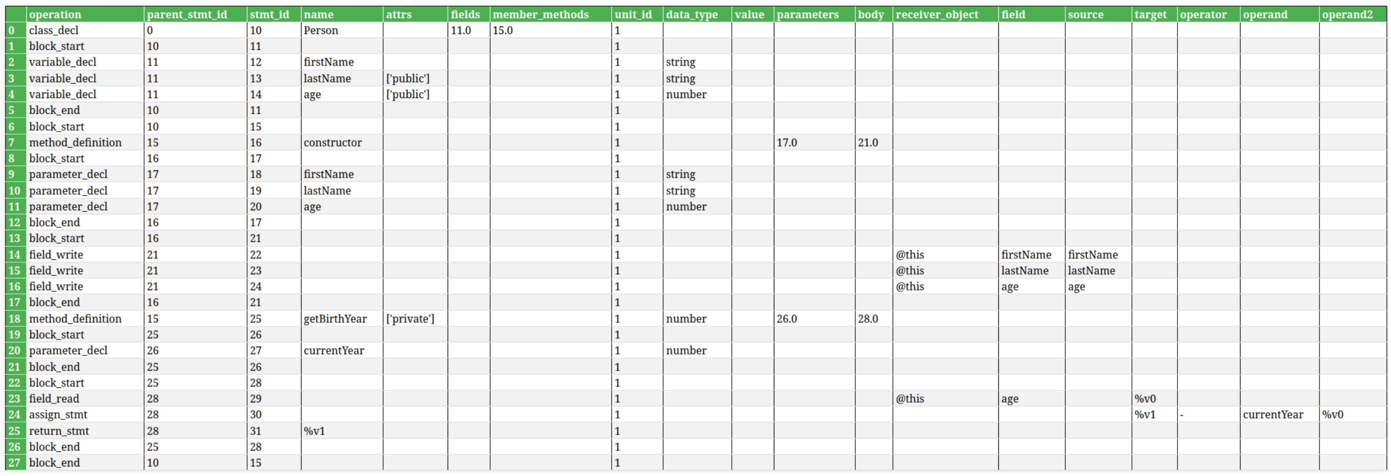
Lab2 Week3

王宁森 周子轩
22307130058 22307130401

截图



	operation	parent_stmt_id	stmt_id	name	attrs	fields	member_methods	unit_id	data_type	value	parameters	body	receiver_object	field	source	target	operator	operand	operand2
0	class_decl	0	10	Person		11.0	15.0	1											
1	block_start	10	11					1											
2	variable_decl	11	12	firstName				1	string										
3	variable_decl	11	13	lastName	['public']			1	string										
4	variable_decl	11	14	age	['public']			1	number										
5	block_end	10	11					1											
6	block_start	10	15					1											
7	method_definition	15	16	constructor				1			17.0	21.0							
8	block_start	16	17					1											
9	parameter_decl	17	18	firstName				1	string										
10	parameter_decl	17	19	lastName				1	string										
11	parameter_decl	17	20	age				1	number										
12	block_end	16	17					1											
13	block_start	16	21					1											
14	field_write	21	22					1					@this	firstName	firstName				
15	field_write	21	23					1					@this	lastName	lastName				
16	field_write	21	24					1					@this	age	age				
17	block_end	16	21					1											
18	method_definition	15	25	getBirthYear	['private']			1	number		26.0	28.0							
19	block_start	25	26					1											
20	parameter_decl	26	27	currentYear				1	number										
21	block_end	25	26					1											
22	block_start	25	28					1											
23	field_read	28	29					1					@this	age		%v0			
24	assign_stmt	28	30					1								%v1	-	currentYear	%v0
25	return_stmt	28	31	%v1				1											
26	block_end	25	28					1											
27	block_end	10	15					1											

解析class

完善assign_stmt

完善assign_stmt,使其支持左侧为object.property的情况，对应GIR指令为field_write

```python
def parse_field(self, node: Node, statements: list):
    myobject = self.find_child_by_field(node, "object")
    field = self.find_child_by_field(node, "property")
    shadow_object = self.parse(myobject, statements)
    shadow_field = self.parse(field, statements)
    return (shadow_object, shadow_field)

def assignment_expression(self, node: Node, statements: list):
        #week2任务
        #week3任务，需要支持left为object.property的形式，可以用parser_field函数帮助解析
        # week3任务: 支持 object.property 左值

        left_node = node.child_by_field_name("left")
        right_node = node.child_by_field_name("right")


        if not left_node or not right_node:
            return None

        value = self.parse(right_node, statements)

        if left_node.type == "member_expression" or left_node.type == "subscript_expression":
            parsed_field_result = self.parse_field(left_node, statements)
            if parsed_field_result is None or parsed_field_result == (None, None):
                 return None

            shadow_object, shadow_field = parsed_field_result

            stmt = {
                "field_write": {
                    "receiver_object": shadow_object,
                    "field": shadow_field,
                    "value": value,
                }
            }
            statements.append(stmt)
            return value

        elif self.is_identifier(left_node) or left_node.type == "private_property_identifier":
            target = self.read_node_text(left_node)

            stmt = {
                "assign_stmt": {
                    "target": target,
                    "operand": value,
                }
            }
            statements.append(stmt)
            return value

        else:
            self.parse(left_node, statements)
            return value
```

现在的`assignment_expression`方法在继承了上周处理简单变量（标识符）到简单值（标识符或字面量）赋值并生成 `assign_stmt` 指令的基础上，扩展了左值的支持范围,能够识别并处理`object.property`形式的赋值。通过判断左侧 `AST` 节点的类型 (`member_expression` 或 `subscript_expression`)，该方法会调用专门的辅助函数 `self.parse_field` 来解析出要操作的对象（`receiver_object`）和要访问的属性（`field`），并结合右侧表达式解析得到的 `value`，生成更具体的 `field_write` 中间表示指令，从而实现对对象属性或集合元素的赋值的 `GIR` 转换。

operation	parent_stmt_id	stmt_id	attrs	data_type	name	body	unit_id	reciever_object	field	target	operand	operator	operand2
field_read	28	29	None	None	None	None	1	@this	age	%v0	None	None	None
assign_stmt	28	30	None	None	None	None	1	None		%v1	currentYear	-	%v0

`field_read`直接读取`this.age`并存入`%v0`，然后再计算`currentYear - this.age`并将结果存入`%v1`作为函数返回值。

完善method_definition

完善method_definition, 使其支持简单参数与类型，不要求支持可选参数、默认参数等复杂参数情况， 不要求支持复杂类型（联合类型、泛型等）

```
    def method_declaration(self, node, statements):
        # week2任务
        # week3任务，需要支持参数，可以用formal_parameter函数帮助解析
        # 函数名
        # week2 & week3任务: 支持简单参数与类型
        name_node = node.child_by_field_name("name")
        fallback_name = f"<{node.type}_{node.start_byte}_{node.end_byte}>"
        func_name = self.read_node_text(name_node) if name_node else fallback_name
        if node.type == 'method_definition' and func_name == 'constructor':
            pass  # Keep 'constructor' name

        # 修饰符 attrs (Modifiers)
        attrs = self.parse_modifiers(node)
        params_node = node.child_by_field_name("parameters")
        params_gir_list = []
        if params_node and params_node.type == 'formal_parameters':
            for param_node in params_node.named_children:
                parameter_info = self.formal_parameter(param_node, statements)
                if parameter_info:
                    params_gir_list.append(parameter_info)


        # 返回值类型 data_type
        return_type_node = node.child_by_field_name("return_type")
        data_type = self.parse_type_annotation(return_type_node)

        # 递归解析函数体 (Body)
        body_stmts: list = []
        body_node = node.child_by_field_name("body")
        if body_node is not None:
            if body_node.type == 'statement_block':
                self.statement_block(body_node, body_stmts)
            elif node.type == 'arrow_function' and self.is_expression(body_node):
                result_expr = self.parse(body_node, body_stmts)
                body_stmts.append({"return_stmt": {"value": result_expr}})
            # Method/Function signatures/abstract methods have no body
            elif node.type in ('method_signature', 'function_signature', 'abstract_method_signature'):
                pass

        # 组装 GIR 节点
        gir_key = node.type
        func_ir = {
            gir_key: {
                "attrs": attrs,
                "data_type": data_type,
                "name": func_name,
                "parameters": params_gir_list,
                "body": body_stmts,
            }
        }
        statements.append(func_ir)
        return func_name

    def formal_parameter(self, node: Node, statements: list):


        param_name = None
        data_type = None
        name_node = None
        type_node = None

        # Find name and type nodes based on expected structures
        if node.type == 'required_parameter':
            name_node = node.child_by_field_name("pattern")
            type_node = node.child_by_field_name("type")
        elif node.type == 'identifier':
            name_node = node
            type_node = None

        if name_node:
            if name_node.type == 'identifier' or name_node.type == 'this':
                param_name = self.read_node_text(name_node)
            elif name_node.type in ('object_pattern', 'array_pattern'):
                return None  # Skip complex parameters
            else:
                param_name = self.read_node_text(name_node)


        if type_node is None and node.type != 'identifier':
            type_node = node.child_by_field_name("type")
        data_type = self.parse_type_annotation(type_node)

        if param_name:
            return {
                "parameter_decl": {
                    "name": param_name,
                    "data_type": data_type,
                }

            }

        return None

    def parse_type_annotation(self, type_node: Node):
        if type_node and (type_node.type == 'type_annotation' or type_node.type == 'predefined_type'):
            actual_type_text_parts = []
            start_collecting = False
            if type_node.type == 'predefined_type':
                start_collecting = True
                actual_type_text_parts.append(self.read_node_text(type_node))
            else:
                for child in type_node.children:
                    if start_collecting:
                        actual_type_text_parts.append(self.read_node_text(child))
                    elif self.read_node_text(child) == ':':
                        start_collecting = True

            type_str = "".join(actual_type_text_parts).strip()
            return type_str if type_str else None
        elif type_node:
            return self.read_node_text(type_node).strip()
        return None
```

在完成这种的任务之前，先把上周那个又臭又长的代码做了一些分解。

解析函数名 (`func_name`) 和修饰符 (`attrs`)用`self.parse_modifiers(node)` 这个辅助函数进行了封装。解析函数体 (`body_stmts`)，处理 `statement_block` 和箭头函数的表达式体时，对后者增加了隐式的 `return_stmt GIR`。对返回值类型的解析也进行了优化，拆分出 `self.parse_type_annotation(return_type_node)` 辅助函数来处理，替换了上周手动查找冒号后拼接文本的逻辑，这里就不去细讲了，实际上就是把上次的部分代码拆了出去。

下面是这种的参数解析：

参数解析 (新增核心功能):查找名为 `parameters` 的子节点，创建一个新的列表 `params_gir_list` 来收集解析后的参数 `GIR`。 遍历 `parameters_node` 的命名子节点 (`named_children`)。对于每一个独立的参数节点调用了新建的辅助方法 `self.formal_parameter(param_node, statements)` （下面会有讲解）来进行处理。`self.formal_parameter` 返回一个表示单个参数的 GIR 字典添加到 `params_gir_list` 中。

`formal_parameter` 方法：它接收一个表示单个参数的 `AST` 节点 根据传入的参数节点的类型， 查找参数的名称节点 (`name_node`) 和类型节点 (`type_node`)。然后提取参数名: 从 `name_node` 中读取文本作为 `param_name`。此过程中跳过复杂参数: 明确检查 `name_node` 是否是 `object_pattern` 或 `array_pattern` （解构赋值参数）。如果是，则直接返回 `None`。 最后调用 `self.parse_type_annotation(type_node)` 辅助方法来解析参数的类型注解。

class解析

该部分需要解析`class`，支持成员变量声明和成员函数声明，且需要支持`public/private`修饰符。

首先，定义`class_declaration`方法，解析类的声明。该方法用于获取类名`name`和类体`body`的节点。

```
    def class_declaration(self, node: Node, statements: list):
        # week3任务，解析class,class_body部分可用class_body函数帮助解析
        name_node = node.child_by_field_name("name")
        body_node = node.child_by_field_name("body")

        class_name = self.read_node_text(name_node) if name_node else f"<anonymous_class_{node.start_byte}>"

        attrs = self.parse_modifiers(node)

        fields_gir = []
        methods_gir = []

        if body_node and body_node.type == "class_body":
            self.class_body(body_node, fields_gir, methods_gir)

        class_ir = {
            "class_decl": {
                "name": class_name,
                "attrs": attrs,
                "fields": fields_gir,
                "member_methods": methods_gir,
            }
        }

        statements.append(class_ir)
        return class_name
```

解析类体`body`部分具体通过调用`class_body`方法进行，在`class_declaration`方法中主要完成对类名的解析。若`name_node`存在，调用`read_node_text`将其转换为字符串；若类是匿名的，则生成一个临时类名`<anonymous_class_314_{node.start_byte}>`作为唯一标识。 随后调用`parse_modifiers`函数解析修饰符。
类体`body`部分通过`class_body`解析。首先创建字段列表`fields_gie`和方法列表`methods_gir`，分别用于保存类字段`GIR`和成员函数`GIR`，然后调用`self.class_body`函数，该函数会遍历类体中的没一个字段和方法，并将这些内容加入`fields_gir`和`methods_gir`。
最后，用解析得到的各个字段构造类的`GIR`，并添加到语句列表中。

下面是`class_body`方法的解释。

```
    def class_body(self, node, fields_list: list, methods_list: list):
        # week3任务，解析class_body部分，需要解析类的字段与成员函数
        for member in node.named_children:
            if self.is_comment(member):
                continue

            if member.type == "method_definition":
                self.method_declaration(member, methods_list)
            elif member.type == "public_field_definition" or member.type == "field_definition":
                self.public_field_definition(member, fields_list)
            else:
                pass
```

在该方法中，`node`传入`class_body`类型的语法树节点，包括整个大括号内部分，并对该部分进行解析。`class_body`节点的`named_children`是类体中所有地显示成员。

- 方法成员`method_definition`： 如果是方法定义，则调用`method_declaration()`处理，并将结果加入到`method_list`中。
- 字段成员`public_field_definition`或`field_definition`： 其中`public_field_definition`有显示带有修饰符，调用`public_field_definition()`处理，而`field_definition`不带有修饰符字段。

上述用于处理`public_field_definition`字段成员的方法`public_field_definition()`如下:

```
    def public_field_definition(self, node: Node, statements: list):
        # week3任务，解析类的字段
        name_node = node.child_by_field_name("name")
        type_node = node.child_by_field_name("type")
        value_node = node.child_by_field_name("value")
        field_name = self.read_node_text(name_node) if name_node else f"<unknown_field_{node.start_byte}>"

        attrs = self.parse_modifiers(node)

        data_type = self.parse_type_annotation(type_node)

        value = self.parse(value_node, statements) if value_node else None

        field_ir = {
            "variable_decl": {
                "name": field_name,
                "attrs": attrs,
                "data_type": data_type,
                "value": value,
            }
        }
        statements.append(field_ir)
        return field_name
```

在该方法中，首先获取字段的名name_node、类型type_node和初始值value_node。通过的道德名name_node获取字段名field_name，若语法树异常（例如name_node为None），则生成一个临时名占位。该解析支持私有字段（如#name）和普通字段（name）。然后调用parse_modifiers()方法解析修饰符attrs，调用parse_type_annotation()方法解析类型data_type。接下来解析初始值表达式，若有=...表达式，则调用parse()方法进行处理。该parse方法可能在statements列表中插入一些中间临时变量定义来支持复杂的右值。最后，根据上述解析内容构建GIR，并插入语句列表。

经解析，得到的GIR表中与类解析相关结果如下：

stmt_id	operation	parent_stmt_id	name	attrs	fields	member_methods
10	class_decl	0	Person		11.0	15.0
11	block_start	10				
12	variable_decl	11	firstName	['public']		
13	variable_decl	11	lastName	['public']		
14	variable_decl	11	age			
15	block_end	10				