Project 4 - Semantic Check Part 2 (Rev. v4)

2025 年 11 月 27 日

我们于 Nov 24 发布了 v2 版本的文档，对“3.2 预定义的语义错误模板”中常见问题进行了回答。

我们于 Nov 26 发布了 v3 版本的文档，对 Project 3 中一处文档错误进行修订，该错误会影响到本次 Project 的内容，请阅读第 5 章并修改你在 Project 3 中的实现代码。

我们于 Nov 27 发布了 v4 版本的文档，对文档中潜在的歧义和未清晰说明的地方进行了补充，请仔细阅读红色高亮部分。

1 项目要求

在 Project 3 中，你已经完成了符号表和对类型的表示。在 Project 4 中，你需要基于上述类型系统和符号表完成语义分析中的类型检查。

1.1 项目假设

在 Project 4 中，我们继承来自 Project 3 的如下假设：

- 样例可能存在**语义错误**，但不存在 Project 3 中规定的语义错误；
- `specifier` 中的**完整结构体**规则出现时，它在语法树中一定是**全局变量定义**或**全局结构体声明**的任意次子节点，即它不会出现在函数参数（包含函数声明和函数定义）、函数体内；

并新增如下假设：

- `specifier` 不再能够被推导为 `CHAR`，即 `char` 类型不再会出现；同时，字符常量 `Char` 也不会出现。
- 样例中出现的结构体均是被完整声明过的，即不存在 incomplete structure。
- **函数声明**与 **函数定义**一定相符，即函数的返回类型与参数列表一定相同，你不需要进行额外检查。
- 函数的返回类型**只能**为 `int`。

1.2 扩展要求

> 扩展部分在每个 Project 之间都是独立计分的，在后续的 Project 中移除对某扩展部分的支持不影响前序 Project 的分数。也就是说，你可以选择在前面的 Project 中完成较为简单的扩展任务；如果你发现在后续的 Project 中完成扩展部分过于困难，你可以选择不完成后续 Project 的扩展部分，这样不会影响你前面 Project 的分数。

本项目的扩展部分与前序 Project 保持一致，你需要确保你的类型检查能够正确处理**结构体**和**指针**相关的语法。

2 类型检查

2.1 类型相等

对于两个类型，它们被视为相等/相同，若它们均是：

- `int`；
- array，并且它们的 element type 相等。
- structure，并且它们是同一个结构体类型。
- pointer，并且它们的 reference type 相等。

2.2 `expression` 的语义与约束

每个 `expression` 具有两种属性：类型和值分类（Value Category, (表格中简写为 VC), lvalue 或 rvalue）；并且，我们对于其 operand 也有要求。

Rule	Return Type	VC	Constr.
标识符	与其声明一致	lvalue	[2.2.1]
数字常量	`int`	rvalue	
括号	与内部 `expression` 一致	与内部一致	
函数调用	函数返回类型	rvalue	[2.2.2]
数组访问	T（见下）	lvalue	[2.2.3]
结构体访问	该结构体成员的类型	lvalue	[2.2.4]
结构体指针访问	该结构体成员的类型	lvalue	[2.2.5]
取地址	$T \rightarrow$*pointer to* T	rvalue	[2.2.6]
解引用	*pointer to* $T \rightarrow T$	lvalue	[2.2.7]
后缀自增、后缀自减、前缀自增、前缀自减	与 Operand 一致（整型或指针）	rvalue	[2.2.8]
加法、减法	见下	rvalue	[2.2.9]
一元正号、一元负号、乘法、除法、取模	整型	rvalue	[2.2.10]
大于等于、大于、小于、小于等于	整型	rvalue	[2.2.11]
等于、不等	整型	rvalue	[2.2.12]
逻辑非、逻辑或、逻辑与	整型	rvalue	[2.2.13]
赋值	与右侧 Operand 一致	rvalue	[2.2.14]

- [2.2.1] Identifier 所引用的是一个 object (变量)，而不是函数。
- [2.2.2] Identifier 所引用的应是一个函数（可以被声明但是未被定义过），参数的类型应该与其声明一致。
- [2.2.3] **数组访问**：第一个 Operand 的类型应为 *array of* T 或者 *pointer to* T，若类型为 *array*，则要求它为 lvalue，第二个 Operand 应该为整型。
- [2.2.4] **结构体访问**：第一个 Operand 的类型应为 *complete structure type*，并且为 lvalue，Identifier 应是该结构体的成员。

- [2.2.5] **结构体指针访问**：第一个 Operand 的类型应为 *pointer to complete structure type*，Identifier 应是该结构体的成员。
- [2.2.6] **取地址**：Operand 应为一个 lvalue。
- [2.2.7] **解引用**：Operand 应为一个指针类型。
- [2.2.8] **自增、自减**：Operand 应是整型或指针，以及是一个 lvalue。指针对整数的加减法的语义见下。
- [2.2.9] **加减法**：若 **加法**、**减法** 两侧的 Operand 均为整型，则其结果为整型。**v4 修改：**
  - 对于**加法**：若其一侧 Operand 为指针，另一侧 Operand 为整型，则结果为指针 Operand 的类型，其语义为该指针后第 n 个元素的地址。
  - 对于**减法**：若其**左侧** Operand 为指针，**右侧** Operand 为整型，则结果为指针 Operand 的类型，其语义为该指针**前第 n 个**元素的地址。`p - n` 的语义与 `p + (-n)` 的语义一致。
  - 对于**减法**：若其**左侧** Operand 为整型，**右侧** Operand 为指针，该情况为语义错误。
  - 对于**减法**：若两侧均为同类型指针，则结果为整型，其语义为两个指针之间相差多少个该类型的对象。
  - 否则，是语义错误。
- [2.2.10] **算数运算**：Operand 应是整型。
- [2.2.11] **比较**：Operand 应是整型。
- [2.2.12] **相等比较**：Operand 应是整型或指针，并且两个 Operand 类型相同。
- [2.2.13] **逻辑运算**：Operand 应是整型或指针。指针在逻辑运算中的语义为与 0 比较，非 0 则为真。
- [2.2.14] **赋值**：Operand 应是整型或指针，并且两个 Operand 类型相同。左侧 Operand 应该是 lvalue。该表达式的返回值为右侧 Operand 的值。

2.3 0 与 null

在某些条件下，数字常量 0 可以被视为一个指针，它代表空指针。
在我们的 Project 中，上述条件被限制为：**直接**[1]出现在**赋值**的右侧，或者在**相等比较**的两侧。

示例 1

```
int main() {
    int *p;
    p = 0;      // OK
    p = 20;     // Error
}
```

2.4 statement 的语义和约束

- **局部变量定义**：若 `ASSIGN expression` 子句存在，**v4 修改：使用 [2.2.14] 规则**[2]对此处的 `expression` 与变量的定义类型进行检查。
- **返回语句**：返回值的类型应该与函数定义中的返回类型一致。
- **If 语句**与 **While 语句**：其中 `expression` 的类型要求为整型或指针。

[1]指该 `expression` 被推导为嵌套 0 个或多个**括号**的**数字常量** `0`，而不能经过其他任何 `expression` 产生式。
[2]**v4 新增**：同样的，若类型不符，使用 `unmatchedTypeForBinaryOP` 而不是 `unexpectedType` 来汇报错误。

3 Project 4 语义检查实现

初始代码位于 `https://github.com/sqlab-sustech/CS323-Compilers-2025F-Projects` 的 `project4-base` 分支。请参照 `Project Zero`，在你 Project 3 代码基础上合并来自 `project4-base` 分支的代码。

你需要完成 `impl.Compiler` 类，基础代码已经给出，注意不要绕过 Grader 打印内容。

若你需要修改 `framework` 包下的文件，请确保你的程序行为不依赖于你所修改的部分。在测评时，`framework` 包下所有文件均会被删除，然后替换为我们提供的版本。

3.1 报告语义错误

每个样例文件中可能存在零个、一个**或多个**语义错误。但是，每个 `statemennt` SubRule 下，至多存在一个语义错误。

你需要遍历所有 `statemennt`，并对其中每个 `expression` 按照要求进行语义检查，汇报遇到的语义错误。当你遇到一个语义错误时，你可以停止对整个 `expression` 树的剩余处理。

即你**不需要考虑如何进行错误恢复**，遇到错误时直接停止整个 `expression` 的检查，然后开始下一个 `statemennt` 的检查即可。

与 Project 3 类似，所有错误报告函数均已预定义在 `framework.project4.Project4SemanticError` 下。额外的，我们预定义了一种异常 `Project4Exception`，你可以在内部处理 `expression` 遇到语义错误时抛出它，在遍历 `statement` 处捕获它并向 `Grader` 汇报，然后开始处理下一个 `statemennt`。

Project 4 核心逻辑示例

```java
public class ExprVisitor extends SplcBaseVisitor<Void> {
    @Override
    public Void visitExprID(SplcParser.ExprIDContext ctx) {
        String id = ctx.Identifier().getText();
        // ...
        Project4SemanticError.identifierNotVariable(ctx, id).throwException();
        // ...
    }
}

// in Your Compiler class:
new SplcBaseVisitor<Void>() {
    @Override
    public Void visitStmtExpr(SplcParser.StmtExprContext ctx) {
        try {
            new ExprVisitor().visit(ctx.expression());
        } catch (Project4Exception ex) {
            grader.reportSemanticError(ex);
        }
        return null;
    }
}.visit(programContext);
```

3.2 预定义的语义错误模板

`Project4SemanticError` 中定义了如下语义错误的模板：

- `identifierNotVariable`、`identifierNotFunction`：当我们期望一个 Identifier 表示变量或函数时，但是它不是。[3]
- `unexpectedType`：Operator 要求 Operand(s) 是特定的类型，但给定的类型不符合文档要求。例如：
  - **结构体访问**左侧是一个 array type.
  - **v4 删除：加减法**~~左侧/右侧是一个 array type.~~ **加减法** [2.2.9] 、**相等比较** [2.2.12] 、**赋值** [2.2.14] 等运算始终使用 `unmatchedTypeForBinaryOP` 汇报类型错误。
  - **一元正号**、**乘法**等算术运算中出现非整型的类型。
- `unmatchedTypeForBinaryOP`：对于较为复杂的二元运算符，若左侧与右侧的 Operand 类型无法匹配。该错误仅适用于 **加减法** [2.2.9] 、**相等比较** [2.2.12] 、**赋值** [2.2.14] 、**局部变量定义**中的初始化。v2 新增：处理上述表达式规则的 **（v4 新增注释）**类型不匹配错误时[4]，只能使用该方法进行汇报错误，而不能使用 `unexpectedType`。例如：
  - 对于**加法**，左右侧分别是 pointer type 与 integer 是可以接受的；但是左右侧同时是 pointer type 则是不可接受的。
  - 对于**减法**，左右侧都是 pointer type，但是所指向的类型不一致，也是不可接受的。
- `badParamType`、`badParamCount`：在函数调用中，传入的参数个数与声明不符，以及第 n 个参数类型不符。**（v4 新增：）**
  - `badParamType` 中的 `ithParam` **从 1 开始计数**。由于文档缺乏清晰说明，第一次预测评中在此处出现了大量错误，第一次测评已将该值从结果比对中移出，即不论 +0 还是 +1 均视为正确。后续的评测将加回来。
  - `badParamCount` 中的 `requires` 表示函数需要多少个参数，即函数声明/定义处指定的参数数量；`given` 表示在函数调用处实际传入了多少个参数。
  - 优先检查参数个数（`badParamCount`），然后再检查每个参数的类型是否匹配（`badParamType`）。
- `badMember`：在结构体操作中，所给定的 Identifier 不是该结构体类型的成员。
- `lvalueRequired`：某些操作符只能作用于 lvalue 的操作数，但是操作数为 rvalue。

上述错误模板函数中的第一个参数均是 `SplcParser.ExpressionContext ctx`，该参数的目的是获取行号用于打印，而我们规定：样例中，每个 `statemennt` 中的 `expression` 均位于同一行。

所以，你不需要纠结应该传入哪一个 `expression` 对象（root、lhs 或 rhs），传哪个对于行号来说都是一样的。

4 评分

本次 Project 满分 100 分，其中基础部分 **80** 分，扩展 1 结构体 10 分，扩展 2 指针 10 分。

我们将运行 `framework.project3.Grader`，并比对你的程序输出与我们的标准输出，**完全一致**则视为通过测试。

在初始代码中，我们提供了一些测试用例，以及它们的参考答案。但是，它们并不涵盖所有可能的情况，你被鼓励自行设计更多测试用例来验证你的程序。

[3] v2 新增：`expression` 中有两处引用了 `Identifier`，分别是**标识符**和**函数引用**。对于前者，我们期望它在上下文中所指代的是一个对象；对于后者，我们期望它在上下文中所指代的是一个函数。若不符合，则分别使用 `identifierNotVariable` 和 `identifierNotFunction` 来报告错误。

[4] **v4 新增注解：** [2.2.14] **赋值**运算要求左侧 Operand 为 lvalue，若违反，使用 `lvalueRequired` 汇报错误而不是该函数。

5 对 Project 3 中 structure tag scope 的勘误

以下例子中，gcc 可以通过编译，而 Splc 可能会报错：`ptr2->qwq` 这一步可能会将 `b0.ptr2` 视为一个指向不完整结构体的指针，导致 [2.2.5] 这一条要求报错。这种样例会在 Project 3 中通过测评，但是在 Project 4 中导致潜在的歧义。

```
// struct a;
struct b {
    struct b *ptr1;
    struct a *ptr2;
};
struct a {
    struct b b0;
    int qwq;
};

int main0() {
    struct b b0;
    b0.ptr2->qwq = 1;
}
```

5.1 Root Cause

在 Project 3 中，[2.2.5] 条：If the declarator or type specifier that declares the identifier appears inside a block or within the list of parameter declarations in a function definition, the identifier has block scope, which terminates at the end of the associated block. 其中的 block 应特指：**函数定义**中的 `LBRACE statement* RBRACE` 与**语句**中的**代码块** `LBRACE statement* RBRACE`。

故：**完整结构体**中的大括号对并不是上述的 block。因此，Structure tag 应该被归类到 [2.2.3] 中的 [2.2.4] 下，**即 Structure tag 应该具有 file scope**。它们的 scope 从 specifier 结束开始 ([2.2.9])，到文件末尾结束 ([2.2.4])。

5.2 TLDR

如果你在 Project 3 中是按照以下逻辑处理结构体类型（注意红色内容），请修改你的实现：

- 第 4 行的 `struct a *ptr2`：根据 [2.1.14]：`a` 这个 tag 是第一次出现，故它声明 `a` 为一个不完整结构体类型，位于 `struct b { ... }` 的 block scope 下面。
- 第 6 行的 `struct a …`，它认为 `a` 是第一次出现，故它声明 `a` 为一个全新的结构体类型。
- 这导致了 `struct b` 里面的 `ptr2` 所指向的类型与外部的完整结构体 `a` 不是同一个类型。

请修改你的程序逻辑为：

- 第 4 行的 `struct a *ptr2`：根据 [2.1.14]：`a` 这个 tag 是第一次出现，故它声明 `a` 为一个不完整结构体类型，位于 **file scope** 下面。
- 第 6 行的 `struct a …`，它发现 `a` 是第二次出现，即它们是同一个类型，故它补全了 `struct a` 的声明，上述 `*ptr2` 也变为完整结构体。

在修改后，你的 Project 3 实现应该对以下案例通过，不再报错 Definition of incomplete type.

```
struct b {
    struct b *ptr1;
    struct a {
        int a;
        int b;
    } *ptr2;
};
struct a a0;    // Not Error: struct a is complete.
```

6 Revision

6.1 Nov 24, v2

在章节 3.2 预定义的语义错误模板中：

- 对 `identifierNotVariable` 与 `identifierNotFunction` 添加脚注。
- 对 `unmatchedTypeForBinaryOP` 和 `unexpectedType` 补充说明：处理上述表达式规则时，只能使用该方法进行汇报错误，而不能使用 `unexpectedType`。

6.2 Nov 26, v3

新增了章节 5：对 Project 3 中 structure tag scope 的勘误。

6.3 Nov 27, v4

所修改的部分已用红色高亮。