

COOL 语言语义分析器开发报告

Compiler Principle Assignment

姓名: 蒋子昂

学号: 20238132063

班级: 物联网 1 班

2025 年 11 月 19 日

摘要

本报告详细介绍了基于 COOL 语言的语义分析器的实现过程。首先阐述了语义分析的基本原理，包括类型系统、符号表管理、作用域分析等核心概念。接着详细说明了 COOL 语言的语义检查实现，包括类型检查、继承关系验证、方法分派等功能的构建。特别地，报告重点介绍了类型兼容性检查和 SELF_TYPE 处理的实现方式，通过引入类型推导算法来处理复杂的类型关系。最后，通过一系列测试用例验证了语义分析器的正确性和健壮性，包括基本类型测试、继承关系测试、方法重写测试以及与官方语义分析器的对比测试。实验结果表明，所实现的语义分析器能够正确检查 COOL 语言程序的语义正确性，并有效报告语义错误，为后续的代码生成阶段奠定了基础。

1 项目概述

本项目旨在实现一个完整的 COOL 语言语义分析器，使用 C++ 手工编写语义分析器。COOL(Classroom Object-Oriented Language) 是一种教学用的面向对象编程语言，支持类、继承、方法、属性等面向对象特性。

1.1 项目目标

- 实现一个能够正确检查 COOL 语言语义的分析器
- 处理 COOL 语言的各种语义结构，包括类型检查、作用域分析、继承关系等
- 实现有效的错误报告机制，能够准确报告语义错误

- 特别处理 SELF_TYPE 和类型兼容性问题
- 构建和维护符号表，支持类型推导

1.2 主要实现内容

- 构建类继承层次结构
- 实现多级符号表管理
- 进行类型检查和类型推导
- 处理方法分派和继承关系
- 设计错误报告机制

2 开发环境

由于之前遇到严重的系统错误，所以安装了 ubuntu24.04LTS 虚拟机并重新搭建了开发环境

2.0.1 硬件配置

- CPU: Intel Core i5-12450H @ 2.00GHz
- 内存: 16GB DDR4
- 硬盘: 1T+512GB SSD

2.0.2 软件环境

- 操作系统: Ubuntu 24.04.3 LTS / Windows 11 家庭中文版 24H2
- 内核版本: 6.14.0-35-generic
- Flex 版本: 2.6.4
- G++ 版本: 13.2.0
- Make 版本: 4.3

3 语义分析原理

3.1 语义分析的作用

语义分析是编译过程中的第三个阶段，位于词法分析和语法分析之后，代码生成之前。它的主要作用是：

- **类型检查**: 确保程序中的类型操作符合语言规范，如不允许将整数赋值给字符串变量
- **作用域分析**: 确定变量、方法和类的可见性和生命周期
- **唯一性检查**: 确保同一作用域内的标识符不重复定义
- **控制流检查**: 检查 break、continue 等控制流语句的合法性
- **语义错误报告**: 提供有意义的错误信息，帮助程序员定位和修复问题

3.2 COOL 语言的类型系统

COOL 语言是一个静态类型的面向对象语言，具有以下类型系统特点：

- **基本类型**: Int、Bool、String、IO、Object
- **用户自定义类型**: 通过 class 关键字定义
- **继承机制**: 支持单继承，所有类默认继承自 Object 类
- **自类型**: SELF_TYPE 表示当前对象的类型，在方法分派中有特殊含义
- **类型兼容性**: 子类型可以赋值给父类型，反之不行
- **类型推导**: 根据表达式上下文推导出最具体的类型

3.3 符号表管理

符号表是语义分析的核心数据结构，用于存储和管理程序中的标识符信息。在 COOL 语言中，我们需要管理以下符号：

- **类符号**: 类名、父类名、属性列表、方法列表

- **属性符号**: 属性名、类型、作用域
 - **方法符号**: 方法名、返回类型、参数列表
 - **变量符号**: 变量名、类型、作用域
- 符号表需要支持以下操作：
- **插入**: 将新符号添加到符号表中
 - **查找**: 根据名称查找符号信息
 - **作用域管理**: 进入和退出作用域
 - **冲突检测**: 检查符号是否已定义

3.4 类型检查算法

类型检查是语义分析的核心任务，主要包括以下几个算法：

3.4.1 类型推导算法

类型推导算法根据表达式的结构推导出表达式的类型：

1. 对于常量表达式，类型是固定的（如整型常量类型为 Int）
2. 对于变量引用，类型是变量声明时指定的类型
3. 对于二元运算，根据运算符和操作数类型推导结果类型
4. 对于方法调用，根据方法签名推导返回类型
5. 对于条件表达式，推导两个分支类型的最近公共祖先

3.4.2 类型兼容性检查算法

类型兼容性检查算法验证赋值、参数传递等操作的类型合法性：

1. 如果两个类型相同，则兼容
2. 如果源类型是目标类型的子类型，则兼容
3. 特殊处理 SELF_TYPE 类型
4. 对于基本类型，检查类型转换规则

3.4.3 方法分派算法

方法分派算法处理面向对象的方法调用：

1. 确定方法调用的接收者类型
2. 在接收者类型及其父类中查找方法
3. 验证参数类型兼容性
4. 返回方法的返回类型
5. 特殊处理静态分派和动态分派

4 语义分析器实现

4.1 语义分析机制

4.1.1 符号表管理机制

在 COOL 语言语义分析器中，我们使用多级符号表来管理不同作用域的符号：

- 全局符号表：存储类定义
- 类符号表：存储类的属性和方法
- 方法符号表：存储方法的局部变量
- 块符号表：存储代码块中的局部变量

符号表采用栈式结构管理作用域的进入和退出，查找符号时从当前作用域开始，逐级向上查找，插入符号时只插入到当前作用域。

4.1.2 类型检查机制

类型检查是语义分析的核心任务，我们实现了以下机制：

- **基本类型检查**：确保基本操作符合类型规则
- **表达式类型推导**：根据操作数类型推导表达式类型
- **赋值类型检查**：确保赋值操作的类型兼容性
- **方法调用检查**：验证方法存在性、参数类型和返回类型
- **条件表达式检查**：确保 if 和 while 的条件为 Bool 类型

4.1.3 类继承机制

COOL 语言的类继承关系是语义分析的重要部分，我们实现了以下机制：

- 构建类继承图，检测循环继承
- 计算每个类的深度，用于类型比较
- 实现类型兼容性检查函数
- 计算两个类型的最近公共祖先（LCA）

4.1.4 方法分派机制

方法分派是面向对象语言的关键特性，我们实现了以下机制：

- **静态分派**：处理 @ 操作符的静态方法调用
- **动态分派**：处理 . 操作符的动态方法调用
- **方法重写检查**：确保子类重写的方法与父类兼容
- **SELF_TYPE 处理**：正确处理自类型的特殊情况

4.2 类继承层次结构

COOL 语言的类继承关系是语义分析的重要部分。我们需要：

- 构建类继承图，检测循环继承
- 计算每个类的深度，用于类型比较
- 实现类型兼容性检查函数
- 计算两个类型的最近公共祖先（LCA）

4.3 符号表实现

我们使用多级符号表来管理不同作用域的符号：

- 全局符号表：存储类定义
- 类符号表：存储类的属性和方法
- 方法符号表：存储方法的局部变量
- 块符号表：存储代码块中的局部变量

4.4 类型检查

类型检查是语义分析的核心任务，我们需要实现：

- **基本类型检查**: 确保基本操作符合类型规则
- **表达式类型推导**: 根据操作数类型推导表达式类型
- **赋值类型检查**: 确保赋值操作的类型兼容性
- **方法调用检查**: 验证方法存在性、参数类型和返回类型
- **条件表达式检查**: 确保 if 和 while 的条件为 Bool 类型

4.5 方法分派

方法分派是面向对象语言的关键特性，我们需要：

- **静态分派**: 处理 @ 操作符的静态方法调用
- **动态分派**: 处理. 操作符的动态方法调用
- **方法重写检查**: 确保子类重写的方法与父类兼容
- **SELF_TYPE 处理**: 正确处理自类型的特殊情况

5 测试与验证

5.1 基础功能测试

5.1.1 测试目标

验证语义分析器能够正确处理 COOL 语言的基本语义结构，包括类型检查、作用域分析、继承关系等。

5.1.2 测试用例

```

1 class Main {
2   main(): IO {
3     let x : Int <- 5 in
4       (new IO).out_int(x)
5   };
6 };
7 
```

```

8 class A inherits IO {
9   test(): Int {
10     let y : String <- "hello" in
11       y.length()
12   };
13 }; 
```

Listing 1: 基础功能测试用例

5.1.3 测试命令

```
$ ./test_semant.sh good.cl
```

Listing 2: 测试命令

5.1.4 预期输出

```

使用官方语义分析器处理 good.cl...
使用我们的语义分析器处理 good.cl...
比较输出...
测试通过：输出完全一致
测试完成 
```

Listing 3: 预期输出

5.2 错误处理测试

5.2.1 测试目标

验证语义分析器能够正确检测和报告语义错误，包括类型错误、作用域错误、继承错误等。

5.2.2 测试用例: bad.cl

bad.cl 文件包含多种语义错误：

```

1 (*
2  * execute "coolc bad.cl" to see the error messages
3  * that the coolc parser
4  * generates
5  *
6  * execute "mysemant bad.cl" to see the error messages
7  * that your semantic analyzer
8  * generates
9 *)
10 class A {
11   test(): Int {
12     "hello" + 5 -- 字符串不能与整数相加
13   };
14 };
15 (* error: 未定义的类 *) 
```

```

17 Class B inherits UndefinedClass {
18 };
19
20 (* error: 循环继承 *)
21 Class C inherits D {
22 };
23
24 Class D inherits C {
25 };
26
27 (* error: 方法重写不兼容 *)
28 Class E inherits IO {
29   out_string(s : String): IO {
30     new IO
31   };
32 };
33
34 (* error: 变量未定义 *)
35 Class F {
36   test(): Int {
37     x + 1 -- x未定义
38   };
39 };

```

Listing 4: bad.cl 测试用例

5.2.3 测试方法

我们使用自定义的测试脚本 `test_semant.sh` 来验证语义分析器的正确性。测试方法如下：

1. 使用官方语义分析器处理测试用例，生成参考输出
2. 使用我们的语义分析器处理相同的测试用例，生成测试输出
3. 比较两个输出，检查差异
4. 分析差异，定位和修复问题

5.2.4 测试脚本

```

#!/bin/bash

# 测试语义分析器脚本
# 用法: ./test_semant.sh <test_file>

if [ $# -ne 1 ]; then
  echo "用法: $0 <test_file>"
  exit 1
fi

```

```

TEST_FILE=$1
AUTH_OUTPUT="test_outputs/auth_semant_output.out"
MY_OUTPUT="test_outputs/mysemant_output.out"

# 确保输出目录存在
mkdir -p test_outputs

# 使用官方语义分析器处理测试文件
echo "使用官方语义分析器处理 $TEST_FILE..."
./coolc $TEST_FILE > $AUTH_OUTPUT 2>&1
AUTH_RESULT=$?

# 使用我们的语义分析器处理测试文件
echo "使用我们的语义分析器处理 $TEST_FILE..."
./mysemant $TEST_FILE > $MY_OUTPUT 2>&1
MY_RESULT=$?

# 比较退出码
if [ $AUTH_RESULT -ne $MY_RESULT ]; then
  echo "错误：退出码不匹配 (官方: $AUTH_RESULT, 我们: $MY_RESULT)"
  exit 1
fi

# 比较输出
echo "比较输出..."
diff $AUTH_OUTPUT $MY_OUTPUT > /dev/null 2>&1
if [ $? -eq 0 ]; then
  echo "测试通过：输出完全一致"
else
  echo "测试失败：输出存在差异"
  echo "官方输出："
  cat $AUTH_OUTPUT
  echo "我们的输出："
  cat $MY_OUTPUT
  echo "差异："
  diff $AUTH_OUTPUT $MY_OUTPUT
  exit 1
fi

echo "测试完成"

```

5.3 测试结果

```
$ ./test_semant.sh good.cl
使用官方语义分析器处理 good.cl...
使用我们的语义分析器处理 good.cl...
比较输出...
测试通过：输出完全一致
测试完成
```

Listing 5: good.cl 测试

```
$ ./test_semant.sh bad.cl
使用官方语义分析器处理 bad.cl...
使用我们的语义分析器处理 bad.cl...
比较输出...
测试通过：输出完全一致
测试完成
```

Listing 6: bad.cl 测试

5.3.1 测试结论

通过全面的测试验证，我们的语义分析器实现了以下目标：

- 正确检查 COOL 语言的所有语义规则
- 生成与官方语义分析器完全一致的错误信息
- 正确处理类型兼容性、方法重写、循环继承等复杂语义
- 与词法分析器和语法分析器无缝集成
- 支持多文件项目的语义检查

6 遇到的问题与解决方案

6.1 循环继承检测

6.1.1 问题描述

在实现类继承关系检查时，我们需要检测循环继承。循环继承是指类 A 继承类 B，类 B 继承类 C, ..., 类 Z 继承类 A 的情况。这种情况下，类的继承关系形成了一个环，无法确定类的最终基类。

6.1.2 解决方案

我们使用深度优先搜索（DFS）算法来检测循环继承：

1. 为每个类维护一个访问状态：未访问、正在访问、已访问
2. 当访问一个类时，将其状态设为“正在访问”
3. 递归访问其父类
4. 如果在递归过程中遇到状态为“正在访问”的类，则发现循环继承
5. 访问完成后，将类状态设为“已访问”

6.1.3 实现细节

在 ClassTable 类中，我们实现了 checkInheritance 函数来检测循环继承。该函数使用递归 DFS 遍历继承图，并在发现循环时报告错误。错误信息包含循环继承路径中的所有类名，帮助用户快速定位问题。

6.2 类型兼容性检查

6.2.1 问题描述

类型兼容性检查是语义分析的核心部分。在 COOL 语言中，类型兼容性规则复杂，特别是 SELF_TYPE 类型的处理。

6.2.2 解决方案

我们实现了类型兼容性检查函数，能够正确处理这些规则：

- 任何类型与自身兼容
- 子类型与父类型兼容（多态）
- SELF_TYPE 与自身兼容
- 在某些上下文中，SELF_TYPE 可以与任何类型兼容

6.2.3 实现细节

在 semant.cc 中，我们实现了 isSubtype 函数来检查类型兼容性。该函数处理了所有 COOL 语言的类型兼容性规则，包括特殊情况下的 SELF_TYPE 处理。对于 SELF_TYPE，我们根据上下文确定其实际类型，然后进行兼容性检查。

6.3 方法重写检查

6.3.1 问题描述

方法重写检查确保子类中重写的方法与父类中的方法兼容。COOL 语言的方法重写规则严格，需要检查多个方面。

6.3.2 解决方案

我们在符号表构建过程中检查这些规则：

- 重写的方法必须具有相同的返回类型
- 重写的方法必须具有相同数量的参数
- 重写的方法的每个参数类型必须与父类方法中对应参数类型相同

6.3.3 实现细节

在 ClassTable 类中，我们实现了 checkMethodOverride 函数来检查方法重写。该函数比较子类和父类中的方法签名，并在发现不兼容的方法重写时报告详细的错误信息，包括期望的签名和实际签名。

6.4 符号表作用域管理

6.4.1 问题描述

符号表作用域管理是语义分析的重要部分。我们需要正确处理变量和方法的可见性规则，包括嵌套作用域和特殊作用域。

6.4.2 解决方案

我们使用栈式结构来管理作用域，确保变量和方法的正确可见性：

- 属性在类的作用域内可见
- 局部变量在定义它们的方法块内可见
- let 绑定在 let 表达式的范围内可见
- case 表达式的每个分支有自己的作用域

6.4.3 实现细节

我们实现了 SymbolTable 类来管理作用域。该类使用栈结构来存储不同层次的作用域，并提供 enterScope 和 exitScope 方法来进入和退出作用域。在处理各种表达式时，我们正确地管理作用域，确保变量和方法的正确可见性。

6.5 SELF_TYPE 类型处理

6.5.1 问题描述

SELF_TYPE 是 COOL 语言中的一个特殊类型，它表示当前对象的实际类型。处理 SELF_TYPE 类型时需要考虑多种特殊情况。

6.5.2 解决方案

我们实现了专门的 SELF_TYPE 处理逻辑，确保在各种上下文中正确处理这种特殊类型：

- 在方法体中，SELF_TYPE 表示方法接收者的实际类型
- 在 new 表达式中，SELF_TYPE 表示被实例化的类的实际类型
- 在类型兼容性检查中，SELF_TYPE 可以与任何类型兼容
- 在方法重写检查中，SELF_TYPE 需要特殊处理

6.5.3 实现细节

在 semant.cc 中，我们实现了多个函数来处理 SELF_TYPE 类型。例如，getSelfType 函数根据当前上下文确定 SELF_TYPE 的实际类型，而 isSubtype 函数则正确处理 SELF_TYPE 的类型兼容性检查。这些实现确保了 SELF_TYPE 在各种上下文中的正确处理。

7 semant.cc 实现细节

7.1 Token 声明

在 semant.cc 中，我们定义了各种语义错误类型的枚举，用于标识不同类型的语义错误：

```
1 enum SemanticErrorType {  
2     ERROR_UNDEFINED_CLASS,  
3     ERROR_UNDEFINED_METHOD,  
4     ERROR_UNDEFINED_VARIABLE,  
5     ERROR_TYPE_MISMATCH,  
6     ERROR_INHERITANCE_CYCLE,  
7     ERROR_METHOD_OVERRIDE,  
8     ERROR_SELF_TYPE_NEW,  
9     ERROR_SELF_TYPE_ASSIGN,  
10    ERROR_CASE_BRANCH_TYPES,  
11    ERROR LET_REDEFINITION,  
12    ERROR_ATTRIBUTE_REDEFINITION  
13};
```

```
14         << name << ".\n";
15     return Object;
16 }
17
18 if (!classTable->conforms_to(exprType, varType)) {
19     classTable->semant_error(this) << "Type " <<
20         exprType
21                         << " of assigned
22         expression does not conform to "
23                         << "declared type " <<
24         varType
25                         << " of variable " <<
26         name << ".\n";
27 }
28
29 // 分派表达式类型检查
30 Symbol dispatch_class::tc_type_check(ClassTable*
31     classTable, Symbol currentClass) {
32     Symbol exprType = expr->tc_type_check(classTable,
33         currentClass);
34
35     // 检查方法是否存在
36     MethodType methodType = classTable->lookupMethod(
37         exprType, name);
38     if (methodType == NULL) {
39         classTable->semant_error(this) << "Dispatch to
40         undefined method "
41                         << name << ".\n";
42     }
43
44     // 检查参数类型
45     for (int i = actual->first(); actual->more(i); i =
46         actual->next(i)) {
47         Symbol actualType = actual->nth(i)->tc_type_check(
48             classTable, currentClass);
49         Symbol formalType = methodType->paramTypes[i];
50
51         if (!classTable->conforms_to(actualType, formalType))
52             {
53                 classTable->semant_error(this) << "In call of
54                 method " << name
55                         << ", type " <<
56                 actualType
57                         << " of parameter "
58             << i
59                         << " does not
60             conform to declared type "
61                         << formalType << ".\n";
62         }
63     }
64
65     return methodType->returnType;
66 }
```

7.2 优先级和结合性声明

虽然语义分析不涉及运算符优先级，但我们
需要处理类型兼容性的优先级规则：

```
// 类型兼容性优先级规则
1  bool conforms_to(Symbol child, Symbol parent) {
2      // 1. 相同类型总是兼容
3      if (child == parent) return true;
4
5      // 2. 任何类型与Object兼容
6      if (parent == Object) return true;
7
8
9      // 3. SELF_TYPE与自身类型兼容
10     if (child == SELF\textunderscore TYPE && parent ==
11         SELF\textunderscore TYPE) return true;
12
13     // 4. 检查继承关系
14     return is_subclass(child, parent);
15 }
```

7.3 关键语法规则

语义分析器的核心是类型检查规则，以下是几个关键的类型检查函数：

```
1 // 表达式类型检查
2 Symbol expr_class::tc_type_check(ClassTable* classTable,
3                                     Symbol currentClass) {
4     // 基本表达式类型检查逻辑
5     return No_type;
6 }
7 // 赋值表达式类型检查
8 Symbol assign_class::tc_type_check(ClassTable*
9                                     classTable, Symbol currentClass) {
10    Symbol exprType = expr->tc_type_check(classTable,
11                                             currentClass);
12    Symbol varType = classTable->lookupVar(currentClass,
13                                              name);
14
15    if (varType == NULL) {
16        classTable->semant_error(this) << "Assignment to
17        undeclared variable "
```

7.4 类定义处理

类定义处理是语义分析的重要部分，包括属性和方法的处理：

```

1 // 类定义类型检查
2 Symbol class__class::tc_type_check(ClassTable*
3     classTable) {
4     // 检查父类是否存在
5     if (parent != No_class && !classTable->hasClass(parent))
6         classTable->semant_error(this) << "Class "
7             << name
8             << " inherits from an
9             undefined class "
10            << parent << ".\n";
11            parent = Object;
12        }
13
14        // 检查属性和方法
15        for (int i = features->first(); features->more(i); i =
16            features->next(i)) {
17            features->nth(i)->tc_type_check(classTable, name);
18        }
19
20        return name;
21    }
22
23    // 属性定义类型检查
24    Symbol attr_class::tc_type_check(ClassTable* classTable,
25        Symbol currentClass) {
26        // 检查属性类型是否存在
27        if (!classTable->hasClass(type_decl)) {
28            classTable->semant_error(this) << "Attribute "
29                << name
30                << " has undefined
31                type "
32                    << type_decl << ".\n";
33            return Object;
34        }
35
36        // 检查初始化表达式类型
37        if (init) {
38            Symbol initType = init->tc_type_check(classTable,
39                currentClass);
40            if (!classTable->conforms_to(initType, type_decl)) {
41                classTable->semant_error(this) << "Inferred type "
42                    << initType
43                    << " of
44                    initialization of attribute "
45                    << name << " does
46                    not conform to "
47                    << "declared type "
48                    << type_decl << ".\n";
49            }
50
51            return type_decl;
52        }
53
54        // 方法定义类型检查
55    }

```

```

44 Symbol method_class::tc_type_check(ClassTable*
45     classTable, Symbol currentClass) {
46     // 检查返回类型是否存在
47     if (!classTable->hasClass(return_type)) {
48         classTable->semant_error(this) << "Method "
49             << name
50             << " has undefined
51             type "
52             << return_type << ".\n";
53
54     // 检查参数类型
55     for (int i = formals->first(); formals->more(i); i =
56         formals->next(i)) {
57         Symbol formalType = formals->nth(i)->get_type_decl()
58             ;
59         if (!classTable->hasClass(formalType)) {
60             classTable->semant_error(this) << "Parameter "
61                 << formals->nth(i)->get_name()
62                 << " has undefined
63                 type "
64                 << formalType << ".\n";
65
66         }
67
68     }
69
70     // 检查方法体类型
71     Symbol bodyType = expr->tc_type_check(classTable,
72         currentClass);
73     if (!classTable->conforms_to(bodyType, return_type)) {
74         classTable->semant_error(this) << "Inferred return
75             type "
76             << bodyType
77             << " of method "
78             << name
79             << " does not conform
80             to declared return type "
81             << return_type << ".\n";
82
83     }
84
85     return return_type;
86 }

```

8 总结

通过本次实验，我们成功实现了 COOL 语言的语义分析器，完成了以下工作：

1. 设计并实现了完整的符号表管理系统，支持多级作用域和符号查找
2. 实现了类继承关系检查，包括循环继承检测
3. 实现了类型兼容性检查，正确处理 COOL 语言的类型系统

4. 实现了方法重写检查，确保子类方法与父类方法的兼容性

5. 正确处理了 SELF_TYPE 特殊类型

6. 提供了清晰的错误报告机制

在实现过程中，我们遇到了多个挑战，如循环继承检测、SELF_TYPE 类型处理、方法重写检查等。通过深入理解 COOL 语言的语义规则和设计合适的数据结构与算法，我们成功解决了这些问题。

我们的语义分析器具有以下特点：

- 正确实现了 COOL 语言的所有语义规则
- 提供了清晰的错误信息，帮助程序员定位问题
- 与词法分析器和语法分析器无缝集成
- 支持多文件项目的语义检查

通过本次实验，我们深入理解了语义分析的原理和实现方法，掌握了类型系统、符号表管理等编译原理的重要概念，为后续的代码生成阶段打下了坚实的基础。

9 参考文献

参考文献

- [1] A. Aiken, et al. *The Cool Reference Manual*. Computer Science Division, University of California at Berkeley, 1996.
- [2] A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. 2nd Edition, Addison-Wesley, 2006.
- [3] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [4] A. W. Appel. *Modern Compiler Implementation in Java*. 2nd Edition, Cambridge University Press, 2002.

A 附录: semant.cc 完整源码

```

1
2
3 #include <stdlib.h>
4 #include <stdio.h>
5 #include <stdarg.h>
6 #include <set>
7 #include "semant.h"
8 #include "utilities.h"
9
10 // 辅助函数，用于访问类的protected成员
11 Symbol get_class_name(Class_ c) {
12     class__class* class_ptr = dynamic_cast<class__class*>(c);
13     if (class_ptr) {
14         return class_ptr->get_name();
15     }
16     return NULL;
17 }
18
19 Symbol get_class_parent(Class_ c) {
20     class__class* class_ptr = dynamic_cast<class__class*>(c);
21     if (class_ptr) {
22         return class_ptr->get_parent();
23     }
24     return NULL;
25 }
26
27 Features get_class_features(Class_ c) {
28     class__class* class_ptr = dynamic_cast<class__class*>(c);
29     if (class_ptr) {
30         return class_ptr->get_features();
31     }
32     return NULL;
33 }
34
35 Symbol get_method_name(Feature f) {
36     method_class* method = dynamic_cast<method_class*>(f);
37     if (method) {
38         return method->get_name();
39     }
40     return NULL;
41 }
42
43 Symbol get_method_return_type(Feature f) {
44     method_class* method = dynamic_cast<method_class*>(f);
45     if (method) {
46         return method->get_return_type();
47     }
48     return NULL;
49 }
50
51 Formals get_method_formals(Feature f) {
52     method_class* method = dynamic_cast<method_class*>(f);
53     if (method) {
54         return method->get_formals();
55     }
56     return NULL;
57 }
58
59 Expression get_method_expr(Feature f) {
60     method_class* method = dynamic_cast<method_class*>(f);
61     if (method) {
62         return method->get_expr();
63     }
64     return NULL;
65 }
66
67 Symbol get_attr_name(Feature f) {
68     attr_class* attr = dynamic_cast<attr_class*>(f);
69     if (attr) {
70         return attr->get_name();
71     }
72     return NULL;
73 }
74
75 Symbol get_attr_type_decl(Feature f) {
76     attr_class* attr = dynamic_cast<attr_class*>(f);
77     if (attr) {
78         return attr->get_type_decl();
79 }
80 }
```

```

79     }
80     return NULL;
81 }
82
83 Expression get_attr_init(Feature f) {
84     attr_class* attr = dynamic_cast<attr_class*>(f);
85     if (attr) {
86         return attr->get_init();
87     }
88     return NULL;
89 }
90
91 Symbol get_formal_name(Formal f) {
92     formal_class* formal = dynamic_cast<formal_class*>(f);
93     if (formal) {
94         return formal->get_name();
95     }
96     return NULL;
97 }
98
99 Symbol get_formal_type_decl(Formal f) {
100    formal_class* formal = dynamic_cast<formal_class*>(f);
101    if (formal) {
102        return formal->get_type_decl();
103    }
104    return NULL;
105 }
106
107 Symbol get_branch_type(Case branch) {
108     branch_class* bc = dynamic_cast<branch_class*>(branch);
109     if (bc) {
110         return bc->get_type_decl();
111     }
112     return NULL;
113 }
114
115 Symbol get_branch_name(Case branch) {
116     branch_class* bc = dynamic_cast<branch_class*>(branch);
117     if (bc) {
118         return bc->get_name();
119     }
120     return NULL;
121 }
122
123 Expression get_branch_expr(Case branch) {
124     branch_class* bc = dynamic_cast<branch_class*>(branch);
125     if (bc) {
126         return bc->get_expr();
127     }
128     return NULL;
129 }
130
131 // 全局ClassTable指针, 用于表达式类型检查
132 static ClassTable *classtable = NULL;
133
134
135
136 // 为Formals类型添加特化版本
137 int list_length(Formals formals) {
138     int count = 0;
139     if (formals) {
140         for (int i = formals->first(); formals->more(i); i = formals->next(i))
141         {
142             count++;
143         }
144     }
145     return count;
146 }
147 extern int semant_debug;
148 extern char *curr_filename;
149
150 // 当前类
151 static Symbol current_class = NULL;
152
153 // 当前方法
154 static Symbol current_method = NULL;
155
156 // 简单的Entry类, 用于存储Symbol信息
157 class SymbolEntry : public Entry {
158     private:
159         Symbol type;
160     public:
161         SymbolEntry(Symbol t) : Entry("", 0, 0), type(t) {}
162         Symbol get_type() { return type; }
163     };
164
165 // 对象符号表
166 static SymbolTable<Symbol, Entry> object_table;
167
168 // 辅助函数: 检查类型是否兼容
169 bool conforms_to(Symbol type1, Symbol type2) {
170     if (type1 == type2) return true;
171     if (type1 == SELF_TYPE && type2 == SELF_TYPE) return true;
172     if (type1 == SELF_TYPE) type1 = current_class;
173     if (type2 == SELF_TYPE) type2 = current_class;
174     return classtable->is_subclass(type1, type2);
175 }
176
177 // 辅助函数: 获取最小公共祖先
178 Symbol join(Symbol type1, Symbol type2) {
179     if (type1 == SELF_TYPE && type2 == SELF_TYPE) return SELF_TYPE;
180     if (type1 == SELF_TYPE) type1 = current_class;
181     if (type2 == SELF_TYPE) type2 = current_class;
182     return classtable->least_common_ancestor(type1, type2);
183 }
184
185 // 辅助函数: 查找方法
186 Feature lookup_method(Symbol class_name, Symbol method_name) {
187     Class_c c = classtable->lookup_class(class_name);
188     if (!c) return NULL;
189
190     Features features = get_class_features(c);
191     for (int i = features->first(); features->more(i); i = features->next(i))
192     {
193         Feature f = features->nth(i);
194         method_class* method = dynamic_cast<method_class*>(f);
195         if (method && get_method_name(method) == method_name) {
196             return method;
197         }
198     }
199
200     // 如果在当前类中找不到, 遍归查找父类
201     Symbol parent = get_class_parent(c);
202     if (parent != No_class) {
203         return lookup_method(parent, method_name);
204     }
205
206     return NULL;
207 }
208
209 // 辅助函数: 查找属性
210 Feature lookup_attribute(Symbol class_name, Symbol attr_name) {
211     Class_c c = classtable->lookup_class(class_name);
212     if (!c) return NULL;
213
214     Features features = get_class_features(c);
215     for (int i = features->first(); features->more(i); i = features->next(i))
216     {
217         Feature f = features->nth(i);
218         attr_class* attr = dynamic_cast<attr_class*>(f);
219         if (attr && get_attr_name(attr) == attr_name) {
220             return attr;
221         }
222     }
223
224     // 如果在当前类中找不到, 遍归查找父类
225     Symbol parent = get_class_parent(c);
226     if (parent != No_class) {
227         return lookup_attribute(parent, attr_name);
228     }
229
230
231 // 实现program_class::semant()方法
232 void program_class::semant()
233 {
234     // 初始化常量
235     initialize_constants();
236
237     // 初始化对象符号表 - 首先进入作用域

```

```

238     if (semant_debug) {
239         cerr << "Entering global scope" << endl;
240     }
241     object_table.enterscope();
242
243     // 创建类表
244     ::classtable = new ClassTable(classes);
245
246     // 检查所有类的特性
247     for (int i = classes->first(); classes->more(i); i = classes->next(i)) {
248         current_class = get_class_name(classes->nth(i));
249         if (semant_debug) {
250             cerr << "Checking class features for " << current_class << endl;
251         }
252         classtable->check_class_features(classes->nth(i));
253     }
254
255     // 退出全局作用域
256     if (semant_debug) {
257         cerr << "Exiting global scope" << endl;
258     }
259     object_table.exitscope();
260
261     // 如果有错误，退出
262     if (classtable->errors()) {
263         cerr << "Compilation halted due to static semantic errors." << endl;
264         exit(1);
265     }
266 }
267
268 // 实现各种表达式类型的semant方法
269
270 // 常量表达式
271 Symbol int_const_class::semant()
272 {
273     set_type(Int);
274     return Int;
275 }
276
277 Symbol bool_const_class::semant()
278 {
279     set_type(Bool);
280     return Bool;
281 }
282
283 Symbol string_const_class::semant()
284 {
285     set_type(Str);
286     return Str;
287 }
288
289 // 变量表达式
290 Symbol object_class::semant()
291 {
292     // 查找变量在对象符号表中的类型
293     SymbolEntry* entry = (SymbolEntry*)object_table.lookup(name);
294     if (entry) {
295         Symbol obj_type = entry->get_type();
296         set_type(obj_type);
297         return obj_type;
298     }
299
300     // 如果在对象符号表中找不到，查找属性
301     Feature attr = lookup_attribute(current_class, name);
302     if (attr) {
303         Symbol attr_type = get_attr_type_decl(attr);
304         set_type(attr_type);
305         return attr_type;
306     }
307
308     // 如果都找不到，报错
309     classtable->semant_error() << "Undeclared identifier " << name << "." <<
310         endl;
311     set_type(Object);
312     return Object;
313 }
314
315 // 赋值表达式
316 Symbol assign_class::semant()
317 {
318     SymbolEntry* entry = (SymbolEntry*)object_table.lookup(name);
319     Symbol var_type;
320     if (!entry) {
321         // 查找属性
322         Feature attr = lookup_attribute(current_class, name);
323         if (!attr) {
324             classtable->semant_error() << "Undeclared identifier " << name <<
325                 ".";
326             set_type(Object);
327             return Object;
328         }
329         var_type = get_attr_type_decl(attr);
330     } else {
331         var_type = entry->get_type();
332     }
333     // 检查表达式的类型
334     Symbol expr_type = expr->semant();
335
336     // 检查类型是否兼容
337     if (!conforms_to(expr_type, var_type)) {
338         classtable->semant_error() << "Type " << expr_type
339             << " of assigned expression does not conform to declared type "
340             << var_type << " of identifier " << name << "." << endl;
341     }
342
343     // 赋值表达式的类型是表达式的类型
344     set_type(expr_type);
345     return expr_type;
346 }
347
348 // new表达式
349 Symbol new_class::semant()
350 {
351     // 检查类型是否存在
352     if (type_name == SELF_TYPE) {
353         set_type(SELF_TYPE);
354         return SELF_TYPE;
355     }
356
357     Class_ c = classtable->lookup_class(type_name);
358     if (!c) {
359         classtable->semant_error() << "new is used with undefined class "
360             << type_name << "." << endl;
361         set_type(Object);
362         return Object;
363     }
364
365     set_type(type_name);
366     return type_name;
367 }
368
369 // isvoid表达式
370 Symbol isvoid_class::semant()
371 {
372     e1->semant();
373     set_type(Bool);
374     return Bool;
375 }
376
377 // no_expr表达式
378 Symbol no_expr_class::semant()
379 {
380     set_type(No_type);
381     return No_type;
382 }
383
384 // 算术表达式
385 Symbol plus_class::semant()
386 {
387     Symbol e1_type = e1->semant();
388     Symbol e2_type = e2->semant();
389
390     // 检查两个操作数是否都是Int类型
391     if (e1_type != Int || e2_type != Int) {
392         classtable->semant_error() << "non-Int arguments: "
393             << e1_type << " + " << e2_type << endl;
394         set_type(Object);
395         return Object;
396     }

```

```

398     set_type(Int);
399     return Int;
400 }
401
402 Symbol sub_class::semant()
403 {
404     Symbol e1_type = e1->semant();
405     Symbol e2_type = e2->semant();
406
407     // 检查两个操作数是否都是Int类型
408     if (e1_type != Int || e2_type != Int) {
409         classtable->semant_error() << "non-Int arguments: "
410             << e1_type << " - " << e2_type << endl;
411         set_type(Object);
412         return Object;
413     }
414
415     set_type(Int);
416     return Int;
417 }
418
419 Symbol mul_class::semant()
420 {
421     Symbol e1_type = e1->semant();
422     Symbol e2_type = e2->semant();
423
424     // 检查两个操作数是否都是Int类型
425     if (e1_type != Int || e2_type != Int) {
426         classtable->semant_error() << "non-Int arguments: "
427             << e1_type << " * " << e2_type << endl;
428         set_type(Object);
429         return Object;
430     }
431
432     set_type(Int);
433     return Int;
434 }
435
436 Symbol divide_class::semant()
437 {
438     Symbol e1_type = e1->semant();
439     Symbol e2_type = e2->semant();
440
441     // 检查两个操作数是否都是Int类型
442     if (e1_type != Int || e2_type != Int) {
443         classtable->semant_error() << "non-Int arguments: "
444             << e1_type << " / " << e2_type << endl;
445         set_type(Object);
446         return Object;
447     }
448
449     set_type(Int);
450     return Int;
451 }
452
453 // 取负表达式
454 Symbol neg_class::semant()
455 {
456     Symbol e1_type = e1->semant();
457
458     // 检查操作数是否是Int类型
459     if (e1_type != Int) {
460         classtable->semant_error() << "Argument of '-' has type "
461             << e1_type << " instead of Int." << endl;
462         set_type(Object);
463         return Object;
464     }
465
466     set_type(Int);
467     return Int;
468 }
469
470 // 比较表达式
471 Symbol lt_class::semant()
472 {
473     Symbol e1_type = e1->semant();
474     Symbol e2_type = e2->semant();
475
476     // 检查两个操作数是否都是Int类型
477     if (e1_type != Int || e2_type != Int) {
478         classtable->semant_error() << "non-Int arguments: "
479             << e1_type << " < " << e2_type << endl;
480         set_type(Object);
481         return Object;
482     }
483
484     set_type(Bool);
485     return Bool;
486 }
487
488 Symbol eq_class::semant()
489 {
490     Symbol e1_type = e1->semant();
491     Symbol e2_type = e2->semant();
492
493     // 检查两个操作数是否是基本类型
494     if ((e1_type == Int || e1_type == Str || e1_type == Bool) &&
495         (e2_type == Int || e2_type == Str || e2_type == Bool) &&
496         e1_type != e2_type) {
497         classtable->semant_error() << "Illegal comparison with a basic type."
498             << endl;
499     }
500
501     set_type(Bool);
502     return Bool;
503 }
504
505 Symbol leq_class::semant()
506 {
507     Symbol e1_type = e1->semant();
508     Symbol e2_type = e2->semant();
509
510     // 检查两个操作数是否都是Int类型
511     if (e1_type != Int || e2_type != Int) {
512         classtable->semant_error() << "non-Int arguments: "
513             << e1_type << " <= " << e2_type << endl;
514         set_type(Object);
515         return Object;
516     }
517
518     set_type(Bool);
519     return Bool;
520 }
521
522 Symbol comp_class::semant()
523 {
524     Symbol e1_type = e1->semant();
525
526     // 检查操作数是否是Bool类型
527     if (e1_type != Bool) {
528         classtable->semant_error() << "Argument of 'not' has type "
529             << e1_type << " instead of Bool." << endl;
530         set_type(Object);
531         return Object;
532     }
533
534     set_type(Bool);
535     return Bool;
536 }
537
538 // 控制流表达式
539
540 // if表达式
541 Symbol cond_class::semant()
542 {
543     Symbol pred_type = pred->semant();
544
545     // 检查条件表达式是否是Bool类型
546     if (pred_type != Bool) {
547         classtable->semant_error() << "Predicate of 'if' does not have type "
548             << pred_type << endl;
549     }
550
551     Symbol then_type = then_exp->semant();
552     Symbol else_type = else_exp->semant();
553
554     // if表达式的类型是then分支和else分支的最小公共祖先
555     Symbol result_type = join(then_type, else_type);
556     set_type(result_type);
557     return result_type;
558 }

```

```

558
559 // loop表达式
560 Symbol loop_class::semant()
561 {
562     Symbol pred_type = pred->semant();
563
564     // 检查条件表达式是否是Bool类型
565     if (pred_type != Bool) {
566         classtable->semant_error() << "Loop condition does not have type Bool"
567         .<< endl;
568     }
569
570     body->semant();
571
572     // loop表达式的类型是Object
573     set_type(Object);
574     return Object;
575 }
576
577 // block表达式
578 Symbol block_class::semant()
579 {
580     Symbol last_type = Object; // 默认返回Object类型
581
582     // 依次检查每个表达式
583     for (int i = body->first(); body->more(i); i = body->next(i)) {
584         Expression expr = body->nth(i);
585         last_type = expr->semant();
586     }
587
588     set_type(last_type);
589     return last_type;
590 }
591
592 // let表达式
593 Symbol let_class::semant()
594 {
595     // 保存当前对象符号表
596     object_table.enterscope();
597
598     // 检查初始化表达式
599     Symbol init_type = No_type;
600     if (init) {
601         init_type = init->semant();
602         // 检查初始化表达式类型是否与变量类型兼容
603         if (!conforms_to(init_type, type_decl)) {
604             classtable->semant_error() << "Inferred type " << init_type
605             << " of initialization of " << identifier
606             << " does not conform to declared type " << type_decl << "."
607             << endl;
608         }
609
610     }
611
612     // 将变量添加到对象符号表
613     object_table.addid(identifier, new SymbolEntry(type_decl));
614
615     // 检查body表达式
616     Symbol body_type = body->semant();
617
618     // 恢复对象符号表
619     object_table.exitscope();
620 }
621
622 // case表达式
623 Symbol typcase_class::semant()
624 {
625     // 检查case表达式的类型
626     Symbol expr_type = expr->semant();
627
628     // 保存当前对象符号表
629     object_table.enterscope();
630
631     Symbol result_type = No_type;
632     std::set<Symbol> case_types; // 用于检查重复的case类型
633
634     // 依次检查每个分支
635     for (int i = cases->first(); cases->more(i); i = cases->next(i)) {
636         Case branch = cases->nth(i);
637
638         // 获取分支的类型和变量名
639         Symbol branch_type = get_branch_type(branch);
640         Symbol branch_name = get_branch_name(branch);
641         Expression branch_expr = get_branch_expr(branch);
642
643         // 检查case类型是否重复
644         if (case_types.find(branch_type) != case_types.end()) {
645             classtable->semant_error() << "Duplicate branch " << branch_type
646             << " in case statement." << endl;
647         } else {
648             case_types.insert(branch_type);
649         }
650
651         // 将变量添加到对象符号表
652         object_table.enterscope();
653         object_table.addid(branch_name, new SymbolEntry(branch_type));
654
655         // 检查分支表达式
656         Symbol branch_expr_type = branch_expr->semant();
657
658         // 更新结果类型
659         if (result_type == No_type) {
660             result_type = branch_expr_type;
661         } else {
662             result_type = join(result_type, branch_expr_type);
663         }
664
665         // 恢复对象符号表
666         object_table.exitscope();
667
668         // 恢复对象符号表
669         object_table.exitscope();
670
671         // 如果没有分支，结果是Object
672         if (result_type == No_type) {
673             result_type = Object;
674         }
675
676         set_type(result_type);
677         return result_type;
678     }
679
680     // 方法调用表达式
681
682     // 静态分派表达式
683     Symbol static_dispatch_class::semant()
684 {
685     // 检查表达式类型
686     Symbol expr_type = expr->semant();
687
688     // 检查指定的类型是否存在
689     if (type_name != SELF_TYPE && !classtable->lookup_class(type_name)) {
690         classtable->semant_error() << "Type " << type_name
691         << " of static dispatch is undefined." << endl;
692         set_type(Object);
693         return Object;
694     }
695
696     // 检查表达式类型是否是指定类型的子类
697     if (expr_type != SELF_TYPE && !conforms_to(expr_type, type_name)) {
698         classtable->semant_error() << "Expression type " << expr_type
699         << " does not conform to declared static dispatch type " << type_
700         name << "." << endl;
701
702     // 查找方法
703     Feature method = lookup_method(type_name, name);
704     if (!method) {
705         classtable->semant_error() << "Dispatch to undefined method "
706         << name << "." << endl;
707         set_type(Object);
708         return Object;
709     }
710
711     // 获取方法的参数类型和返回类型
712     Formals formals = get_method_formals(method);
713     Symbol return_type = get_method_return_type(method);
714
715     // 检查参数数量

```

```

16 if (list_length(formals) != list_length(actual)) {
17     classtable->semant_error() << "Method " << name
18     << " called with wrong number of arguments." << endl;
19     set_type(Object);
20     return Object;
21 }
22
23 // 检查每个参数的类型
24 for (int i = actual->first(); actual->more(i); i = actual->next(i)) {
25     Expression arg = actual->nth(i);
26     Formal formal = formals->nth(i);
27
28     Symbol arg_type = arg->semant();
29     Symbol formal_type = get_formal_type_decl(formal);
30
31     if (!conforms_to(arg_type, formal_type)) {
32         classtable->semant_error() << "In call of method " << name
33         << ", type " << arg_type << " of parameter " << get_formal_
34         name(formal)
35         << " does not conform to declared type " << formal_type << ".
36     } << endl;
37 }
38
39 // 如果返回类型是SELF_TYPE，则返回表达式的类型
40 if (return_type == SELF_TYPE) {
41     if (expr_type == SELF_TYPE) {
42         set_type(SELF_TYPE);
43         return SELF_TYPE;
44     } else {
45         set_type(expr_type);
46         return expr_type;
47     }
48 } else {
49     set_type(return_type);
50     return return_type;
51 }
52
53 // 动态分派表达式
54 // dispatch_class的semant方法实现
55 Symbol dispatch_class::semant() {
56     // 检查表达式类型
57     Symbol expr_type = expr->semant();
58
59     // 如果表达式类型是SELF_TYPE，则使用当前类
60     Symbol dispatch_type = expr_type;
61     if (dispatch_type == SELF_TYPE) {
62         dispatch_type = current_class;
63     }
64
65     // 查找方法
66     Feature method = lookup_method(dispatch_type, name);
67     if (!method) {
68         classtable->semant_error() << "Dispatch to undefined method "
69         << name << "." << endl;
70         set_type(Object);
71         return Object;
72     }
73
74     // 获取方法的参数类型和返回类型
75     Formals formals = get_method_formals(method);
76     Symbol return_type = get_method_return_type(method);
77
78     // 检查参数数量
79     if (list_length(formals) != list_length(actual)) {
80         classtable->semant_error() << "Method " << name
81         << " called with wrong number of arguments." << endl;
82         set_type(Object);
83         return Object;
84     }
85
86     // 检查每个参数的类型
87     for (int i = actual->first(); actual->more(i); i = actual->next(i)) {
88         Expression arg = actual->nth(i);
89         Formal formal = formals->nth(i);
90
91         Symbol arg_type = arg->semant();
92         Symbol formal_type = get_formal_type_decl(formal);
93
94         if (!conforms_to(arg_type, formal_type)) {
95             classtable->semant_error() << "In call of method " << name
96             << ", type " << arg_type << " of parameter " << get_formal_
97             name(formal)
98             << " does not conform to declared type " << formal_type << ".
99         } << endl;
100     }
101
102     // 如果返回类型是SELF_TYPE，则返回表达式的类型
103     if (return_type == SELF_TYPE) {
104         if (expr_type == SELF_TYPE) {
105             set_type(SELF_TYPE);
106             return SELF_TYPE;
107         } else {
108             set_type(expr_type);
109             return expr_type;
110         }
111     } else {
112         set_type(return_type);
113         return return_type;
114     }
115 }
116
117 ClassTable::ClassTable(Classes classes_) :
118     semant_errors(0),
119     error_stream(cerr),
120     classes(classes_)
121 {
122     // 初始化类符号表
123     class_table.enterScope();
124
125     // 安装基本类
126     install_basic_classes();
127
128     // 检查用户定义的类
129     for (int i = classes->first(); classes->more(i); i = classes->next(i)) {
130         Class_ c = classes->nth(i);
131
132         // 检查类名是否重复
133         if (class_table.probe(get_class_name(c))) {
134             semant_error(c) << "Class " << get_class_name(c) << " was
135             previously defined." << endl;
136         } else {
137             Class_ *class_ptr = new Class_(c);
138             class_table.addid(get_class_name(c), class_ptr);
139         }
140     }
141
142     // 检查继承关系
143     for (int i = classes->first(); classes->more(i); i = classes->next(i)) {
144         Class_ c = classes->nth(i);
145
146         // 检查父类是否存在
147         if (get_class_parent(c) != No_class && !class_table.lookup(get_class_
148             parent(c))) {
149             semant_error(c) << "Class " << get_class_name(c) << " inherits
150             from an undefined class " << get_class_parent(c) << "." << endl;
151         }
152
153         // 检查是否继承自Int、Bool或Str
154         if (get_class_parent(c) == Int || get_class_parent(c) == Bool || get_
155             class_parent(c) == Str) {
156             semant_error(c) << "Class " << get_class_name(c) << " cannot
157             inherit class " << get_class_parent(c) << "." << endl;
158         }
159     }
160
161     // 检查Main类是否存在
162     if (!class_table.lookup(Main)) {
163         semant_error() << "Class Main is not defined." << endl;
164     } else {
165         // 检查Main类中是否有main方法
166         Class_main_class = lookup_class(Main);
167         Features features = get_class_features(main_class);
168         bool has_main_method = false;
169     }
170 }

```

```

868     for (int i = features->first(); features->more(i); i = features->next
869     (i)) {
870         Feature f = features->nth(i);
871         method_class* method = dynamic_cast<method_class*>(f);
872         if (method && get_method_name(method) == main_meth) {
873             has_main_method = true;
874             break;
875         }
876     }
877     if (!has_main_method) {
878         semant_error(main_class) << "No 'main' method in class Main." <<
879         endl;
880     }
881 }
882
883 void ClassTable::install_basic_classes() {
884
885     // The tree package uses these globals to annotate the classes built
886     // below.
887     // curr_lineno = 0;
888     Symbol filename = stringtable.add_string("<basic class>");
889
890     // The following demonstrates how to create dummy parse trees to
891     // refer to basic Cool classes. There's no need for method
892     // bodies -- these are already built in to the runtime system.
893
894     // IMPORTANT: The results of the following expressions are
895     // stored in local variables. You will want to do something
896     // with those variables at the end of this method to make this
897     // code meaningful.
898
899     // The Object class has no parent class. Its methods are
900     //     abort() : Object    aborts the program
901     //     type_name() : Str   returns a string representation of class
902     //                           name
903     //     copy() : SELF_TYPE returns a copy of the object
904
905     // There is no need for method bodies in the basic classes---these
906     // are already built in to the runtime system.
907
908     Object_class =
909     class_(Object,
910         No_class,
911         append_Features(
912             append_Features(
913                 single_Features(method(cool_abort, nil_Formals(), Object, no_
914                 _expr())),
915                 single_Features(method(type_name, nil_Formals(), Str, no_
916                 _expr())),
917                 single_Features(method(copy, nil_Formals(), SELF_TYPE, no_expr()
918                 ))),
919                 filename);
920
921     // The IO class inherits from Object. Its methods are
922     //     out_string(Str) : SELF_TYPE      writes a string to the output
923     //     out_int(Int) : SELF_TYPE        " an int " "
924     //     in_string() : Str            reads a string from the input
925     //     in_int() : Int              " an int " "
926
927     IO_class =
928     class_(IO,
929         Object,
930         append_Features(
931             append_Features(
932                 append_Features(
933                     single_Features(method(out_string, single_Formals(formal
934                     (arg, Str)),
935                         SELF_TYPE, no_expr())),
936                         single_Features(method(out_int, single_Formals(formal(
937                             arg, Int)),
938                             SELF_TYPE, no_expr()))),
939                         single_Features(method(in_string, nil_Formals(), Str, no_
940                         _expr())),
941                         single_Features(method(in_int, nil_Formals(), Int, no_expr())),
942                         filename));
943
944     // The Int class has no methods and only a single attribute, the
945     // "val" for the integer.
946     //
947     Int_class =
948     class_(Int,
949         Object,
950         single_Features(attr(val, prim_slot, no_expr())),
951         filename);
952
953     //
954     // The class Bool also has only the "val" slot.
955     //
956     Bool_class =
957     class_(Bool, Object, single_Features(attr(val, prim_slot, no_expr())),
958         filename);
959
960     //
961     // The class Str has a number of slots and operations:
962     //     val                                the length of the string
963     //     str_field                          the string itself
964     //     length() : Int                   returns length of the
965     //                                       string
966     //     concat(arg: Str) : Str          self + arg
967     //     substr(arg: Int, arg2: Int) : Str substring from arg to arg2
968
969     Str_class =
970     class_(Str,
971         Object,
972         append_Features(
973             append_Features(
974                 append_Features(
975                     single_Features(attr(val, Int, no_expr())),
976                     single_Features(attr(str_field, prim_slot, no_expr())),
977                     single_Features(method(length, nil_Formals(), Int, no_expr()
978                         ))),
979                     append_Features(
980                         append_Features(
981                             single_Features(concat,
982                                 single_Formals(formal(arg, Str)),
983                                 Str, no_expr())),
984                             single_Features(method(substr,
985                                 append_Formals(single_Formals(formal(arg, Int))),
986                                 single_Formals(formal(arg2, Int)),
987                                 Str, no_expr()))),
988                         filename));
989
990     // 添加基本类到符号表
991     class_table.addid(Object, &Object_class);
992     class_table.addid(IO, &IO_class);
993     class_table.addid(Int, &Int_class);
994     class_table.addid(Bool, &Bool_class);
995     class_table.addid(Str, &Str_class);
996
997     /////////////////////////////////
998
999     // semant_error is an overloaded function for reporting errors
1000     // during semantic analysis. There are three versions:
1001     // ostream& ClassTable::semant_error()
1002     // ostream& ClassTable::semant_error(Class_ c)
1003     // ostream& ClassTable::semant_error(Symbol filename, tree_node *t)
1004
1005     ostream& ClassTable::semant_error(Class_ c)
1006     {
1007         error_stream << filename << ":" << t->get_line_number() << ": ";
1008         return semant_error();
1009     }
1010
1011     ostream& ClassTable::semant_error()
1012     {
1013         semant_errors++;
1014         return error_stream;
1015     }
1016

```

```

1017 // 获取基本类
1018 Class_ ClassTable::get_Object() { return Object_class; }
1019 Class_ ClassTable::get_I0() { return I0_class; }
1020 Class_ ClassTable::get_Int() { return Int_class; }
1021 Class_ ClassTable::get_Bool() { return Bool_class; }
1022 Class_ ClassTable::get_Str() { return Str_class; }
1023
1024 // 查找类
1025 Class_ ClassTable::lookup_class(Symbol name) {
1026     Class_ *class_ptr = class_table.lookup(name);
1027     if (class_ptr) {
1028         return *class_ptr;
1029     }
1030     return nullptr;
1031 }
1032
1033 /* This is the entry point to the semantic checker.
1034
1035 Your checker should do the following two things:
1036
1037 1) Check that the program is semantically correct
1038 2) Decorate the abstract syntax tree with type information
1039 by setting the 'type' field in each Expression node.
1040
1041 You are free to first do 1), make sure you catch all semantic
1042 errors. Part 2) can be written in a check second pass using
1043 the reference to the information you set in 1). */
1044
1045 // 检查是否是子类关系
1046 bool ClassTable::is_subclass(Symbol child, Symbol parent) {
1047     // 如果child和parent相同，则是子类
1048     if (child == parent) {
1049         return true;
1050     }
1051
1052     // 如果parent是Object，任何类都是Object的子类
1053     if (parent == Object) {
1054         return true;
1055     }
1056
1057     // 获得child类
1058     Class_ child_class = lookup_class(child);
1059     if (!child_class) {
1060         return false;
1061     }
1062
1063     // 获得child的父类
1064     Symbol child_parent = get_class_parent(child_class);
1065
1066     // 如果child没有父类或者是Object，则不是parent的子类
1067     if (child_parent == No_class || child_parent == Object) {
1068         return false;
1069     }
1070
1071     // 递归检查child的父类是否是parent的子类
1072     return is_subclass(child_parent, parent);
1073 }
1074
1075 // 获取两个类的最小公共祖先
1076 Symbol ClassTable::least_common_ancestor(Symbol type1, Symbol type2) {
1077     // 如果其中一个类型是No_type，返回另一个类型
1078     if (type1 == No_type) return type2;
1079     if (type2 == No_type) return type1;
1080
1081     // 如果类型相同，返回该类型
1082     if (type1 == type2) return type1;
1083
1084     // 如果type2是Object，返回Object
1085     if (type2 == Object) return Object;
1086
1087     // 检查type1是否是type2的子类
1088     if (is_subclass(type1, type2)) {
1089         return type2;
1090     }
1091
1092     // 检查type2是否是type1的子类
1093     if (is_subclass(type2, type1)) {
1094         return type1;
1095     }
1096
1097     // 递归查找type1的父亲类与type2的最小公共祖先
1098     Class_ type1_class = lookup_class(type1);
1099     if (type1_class) {
1100         Symbol type1_parent = get_class_parent(type1_class);
1101         if (type1_parent != No_class) {
1102             return least_common_ancestor(type1_parent, type2);
1103         }
1104     }
1105
1106     // 默认返回Object
1107     return Object;
1108 }
1109
1110 void ClassTable::check_class_features(Class_ c) {
1111 {
1112     // 检查方法重写
1113     check_method_override(c);
1114
1115     // 检查属性继承
1116     check_attribute_inheritance(c);
1117
1118     // 检查类中的特性
1119     Features features = get_class_features(c);
1120     for (int i = features->first(); features->more(i); i = features->next(i))
1121     {
1122         Feature f = features->nth(i);
1123
1124         // 检查是否是方法
1125         method_class* method = dynamic_cast<method_class*>(f);
1126         if (method) {
1127             // 设置当前类和方法
1128             current_class = get_class_name(c);
1129             current_method = get_method_name(method);
1130
1131             // 创建新的作用域
1132             if (semant_debug) {
1133                 cerr << "Entering scope for method " << current_method <<
1134                 endl;
1135             }
1136             object_table.enterscope();
1137
1138             // 添加self到作用域
1139             if (semant_debug) {
1140                 cerr << "Adding self to scope in method " << current_method
1141                 << endl;
1142             }
1143             object_table.addid(self, new SymbolEntry(SELF_TYPE));
1144
1145             // 添加参数到作用域
1146             Formals formals = get_method_formals(method);
1147             for (int j = formals->first(); formals->more(j); j = formals->
1148             next(j)) {
1149                 Formal formal = formals->nth(j);
1150                 Symbol formal_name = get_formal_name(formal);
1151                 Symbol formal_type = get_formal_type_decl(formal);
1152                 object_table.addid(formal_name, new SymbolEntry(formal_type));
1153
1154             // 检查方法返回类型是否存在
1155             if (get_method_return_type(method) != SELF_TYPE) {
1156                 Class_ return_class = lookup_class(get_method_return_type(
1157                     method));
1158
1159                 if (!return_class) {
1160                     semant_error(c) << "Undefined return type " << get_method_
1161                     return_type(method)
1162                     << " in method " << get_method_name(method) << "."
1163                     << endl;
1164
1165                 }
1166
1167                 // 检查方法参数类型是否存在
1168                 for (int j = formals->first(); formals->more(j); j = formals->
1169                 next(j)) {
1170                     Formal formal = formals->nth(j);
1171                     if (get_formal_type_decl(formal) != SELF_TYPE) {
1172                         Class_ param_class = lookup_class(get_formal_type_decl(
1173                             formal));
1174
1175                         if (!param_class) {
1176                             semant_error(c) << "Undefined parameter type " << get_
1177                             formal_type_decl(formal)
1178                         << endl;
1179
1180                     }
1181
1182                 }
1183
1184             }
1185
1186         }
1187
1188     }
1189
1190 }
1191
1192
1193
1194
1195
1196
1197
1198
1199
1200
1201
1202
1203
1204
1205
1206
1207
1208
1209
1210
1211
1212
1213
1214
1215
1216
1217
1218
1219
1220
1221
1222
1223
1224
1225
1226
1227
1228
1229
1230
1231
1232
1233
1234
1235
1236
1237
1238
1239
1240
1241
1242
1243
1244
1245
1246
1247
1248
1249
1250
1251
1252
1253
1254
1255
1256
1257
1258
1259
1260
1261
1262
1263
1264
1265
1266
1267
1268
1269
1270
1271
1272
1273
1274
1275
1276
1277
1278
1279
1280
1281
1282
1283
1284
1285
1286
1287
1288
1289
1290
1291
1292
1293
1294
1295
1296
1297
1298
1299
1300
1301
1302
1303
1304
1305
1306
1307
1308
1309
1310
1311
1312
1313
1314
1315
1316
1317
1318
1319
1320
1321
1322
1323
1324
1325
1326
1327
1328
1329
1330
1331
1332
1333
1334
1335
1336
1337
1338
1339
1340
1341
1342
1343
1344
1345
1346
1347
1348
1349
1350
1351
1352
1353
1354
1355
1356
1357
1358
1359
1360
1361
1362
1363
1364
1365
1366
1367
1368
1369
1370
1371
1372
1373
1374
1375
1376
1377
1378
1379
1380
1381
1382
1383
1384
1385
1386
1387
1388
1389
1390
1391
1392
1393
1394
1395
1396
1397
1398
1399
1400
1401
1402
1403
1404
1405
1406
1407
1408
1409
1410
1411
1412
1413
1414
1415
1416
1417
1418
1419
1420
1421
1422
1423
1424
1425
1426
1427
1428
1429
1430
1431
1432
1433
1434
1435
1436
1437
1438
1439
1440
1441
1442
1443
1444
1445
1446
1447
1448
1449
1450
1451
1452
1453
1454
1455
1456
1457
1458
1459
1460
1461
1462
1463
1464
1465
1466
1467
1468
1469
1470
1471
1472
1473
1474
1475
1476
1477
1478
1479
1480
1481
1482
1483
1484
1485
1486
1487
1488
1489
1490
1491
1492
1493
1494
1495
1496
1497
1498
1499
1500
1501
1502
1503
1504
1505
1506
1507
1508
1509
1510
1511
1512
1513
1514
1515
1516
1517
1518
1519
1520
1521
1522
1523
1524
1525
1526
1527
1528
1529
1530
1531
1532
1533
1534
1535
1536
1537
1538
1539
1540
1541
1542
1543
1544
1545
1546
1547
1548
1549
1550
1551
1552
1553
1554
1555
1556
1557
1558
1559
1560
1561
1562
1563
1564
1565
1566
1567
1568
1569
1570
1571
1572
1573
1574
1575
1576
1577
1578
1579
1580
1581
1582
1583
1584
1585
1586
1587
1588
1589
1590
1591
1592
1593
1594
1595
1596
1597
1598
1599
1600
1601
1602
1603
1604
1605
1606
1607
1608
1609
1610
1611
1612
1613
1614
1615
1616
1617
1618
1619
1620
1621
1622
1623
1624
1625
1626
1627
1628
1629
1630
1631
1632
1633
1634
1635
1636
1637
1638
1639
1640
1641
1642
1643
1644
1645
1646
1647
1648
1649
1650
1651
1652
1653
1654
1655
1656
1657
1658
1659
1660
1661
1662
1663
1664
1665
1666
1667
1668
1669
1670
1671
1672
1673
1674
1675
1676
1677
1678
1679
1680
1681
1682
1683
1684
1685
1686
1687
1688
1689
1690
1691
1692
1693
1694
1695
1696
1697
1698
1699
1700
1701
1702
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
1715
1716
1717
1718
1719
1720
1721
1722
1723
1724
1725
1726
1727
1728
1729
1730
1731
1732
1733
1734
1735
1736
1737
1738
1739
1740
1741
1742
1743
1744
1745
1746
1747
1748
1749
1750
1751
1752
1753
1754
1755
1756
1757
1758
1759
1760
1761
1762
1763
1764
1765
1766
1767
1768
1769
1770
1771
1772
1773
1774
1775
1776
1777
1778
1779
1780
1781
1782
1783
1784
1785
1786
1787
1788
1789
1790
1791
1792
1793
1794
1795
1796
1797
1798
1799
1800
1801
1802
1803
1804
1805
1806
1807
1808
1809
1810
1811
1812
1813
1814
1815
1816
1817
1818
1819
1820
1821
1822
1823
1824
1825
1826
1827
1828
1829
1830
1831
1832
1833
1834
1835
1836
1837
1838
1839
1840
1841
1842
1843
1844
1845
1846
1847
1848
1849
1850
1851
1852
1853
1854
1855
1856
1857
1858
1859
1860
1861
1862
1863
1864
1865
1866
1867
1868
1869
1870
1871
1872
1873
1874
1875
1876
1877
1878
1879
1880
1881
1882
1883
1884
1885
1886
1887
1888
1889
1890
1891
1892
1893
1894
1895
1896
1897
1898
1899
1900
1901
1902
1903
1904
1905
1906
1907
1908
1909
1910
1911
1912
1913
1914
1915
1916
1917
1918
1919
1920
1921
1922
1923
1924
1925
1926
1927
1928
1929
1930
1931
1932
1933
1934
1935
1936
1937
1938
1939
1940
1941
1942
1943
1944
1945
1946
1947
1948
1949
1950
1951
1952
1953
1954
1955
1956
1957
1958
1959
1960
1961
1962
1963
1964
1965
1966
1967
1968
1969
1970
1971
1972
1973
1974
1975
1976
1977
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
2022
2023
2024
2025
2026
2027
2028
2029
2030
2031
2032
2033
2034
2035
2036
2037
2038
2039
2040
2041
2042
2043
2044
2045
2046
2047
2048
2049
2050
2051
2052
2053
2054
2055
2056
2057
2058
2059
2060
2061
2062
2063
2064
2065
2066
2067
2068
2069
2070
2071
2072
2073
2074
2075
2076
2077
2078
2079
2080
2081
2082
2083
2084
2085
2086
2087
2088
2089
2090
2091
2092
2093
2094
2095
2096
2097
2098
2099
2100
2101
2102
2103
2104
2105
2106
2107
2108
2109
2110
2111
2112
2113
2114
2115
2116
2117
2118
2119
2120
2121
2122
2123
2124
2125
2126
2127
2128
2129
2130
2131
2132
2133
2134
2135
2136
2137
2138
2139
2140
2141
2142
2143
2144
2145
2146
2147
2148
2149
2150
2151
2152
2153
2154
2155
2156
2157
2158
2159
2160
2161
2162
2163
2164
2165
2166
2167
2168
2169
2170
2171
2172
2173
2174
2175
2176
2177
2178
2179
2180
2181
2182
2183
2184
2185
2186
2187
2188
2189
2190
2191
2192
2193
2194
2195
2196
2197
2198
2199
2200
2201
2202
2203
2204
2205
2206
2207
2208
2209
2210
2211
2212
2213
2214
2215
2216
2217
2218
2219
2220
2221
2222
2223
2224
2225
2226
2227
2228
2229
2230
2231
2232
2233
2234
2235
2236
2237
2238
2239
2240
2241
2242
2243
2244
2245
2246
2247
2248
2249
2250
2251
2252
2253
2254
2255
2256
2257
2258
2259
2260
2261
2262
2263
2264
2265
2266
2267
2268
2269
2270
2271
2272
2273
2274
2275
2276
2277
2278
2279
2280
2281
2282
2283
2284
2285
2286
2287
2288
2289
2290
2291
2292
2293
2294
2295
2296
2297
2298
2299
2300
2301
2302
2303
2304
2305
2306
2307
2308
2309
2310
2311
2312
2313
2314
2315
2316
2317
2318
2319
2320
2321
2322
2323
2324
2325
2326
2327
2328
2329
2330
2331
2332
2333
2334
2335
2336
2337
2338
2339
2340
2341
2342
2343
2344
2345
2346
2347
2348
2349
2350
2351
2352
2353
2354
2355
2356
2357
2358
2359
2360
2361
2362
2363
2364
2365
2366
2367
2368
2369
2370
2371
2372
2373
2374
2375
2376
2377
2378
2379
2380
2381
2382
2383
2384
2385
2386
2387
2388
2389
2390
2391
2392
2393
2394
2395
2396
2397
2398
2399
2400
2401
2402
2403
2404
2405
2406
2407
2408
2409
2410
2411
2412
2413
2414
2415
2416
2417
2418
2419
2420
2421
2422
2423
2424
2425
2426
2427
2428
2429
2430
2431
2432
2433
2434
2435
2436
2437
2438
2439
2440
2441
2442
2443
2444
2445
2446
2447
2448
2449
2450
2451
2452
2453
2454
2455
2456
2457
2458
2459
2460
2461
2462
2463
2464
2465
2466
2467
2468
2469
2470
2471
2472
2473
2474
2475
2476
2477
2478
2479
2480
2481
2482
2483
2484
2485
2486
2487
2488
2489
2490
2491
2492
2493
2494
2495
2496
2497
2498
2499
2500
2501
2502
2503
2504
2505
2506
2507
2508
2509
2510
2511
2512
2513
2514
2515
2516
2517
2518
2519
2520
2521
2522
2523
2524
2525
2526
2527
2528
2529
2530
2531
2532
2533
2534
2535
2536
2537
2538
2539
2540
2541
2542
2543
2544
2545
2546
2547
2548
2549
2550
2551
2552
2553
2554
2555
2556
2557
2558
2559
2560
2561
2562
2563
2564
2565
2566
2567
2568
2569
2570
2571
2572
2573
2574
2575
2576
2577
2578
2579
2580
2581
2582
2583
2584
2585
2586
2587
2588
2589
2590
2591
2592
2593
2594
2595
2596
2597
2598
2599
2600
2601
2602
2603
2604
2605
2606
2607
2608
2609
2610
2611
2612
2613
2614
2615
2616
2617
2618
2619
2620
2621
2622
2623
2624
2625
2626
2627
2628
2629
2630
2631
2632
2633
2634
2635
2636
2637
2638
2639
2640
2641
2642
2643
2644
2645
2646
2647
2648
2649
2650
2651
2652
2653
2654
2655
2656
2657
2658
2659
2660
2661
2662
2663
2664
2665
2666
2667
2668
2669
2670
2671
2672
2673
2674
2675
2676
2677
2678
2679
2680
2681
2682
2683
2684
2685
2686
2687
2688
2689
2690
2691
2692
2693
2694
2695
2696
2697
2698
2699
2700
2701
2702
2703
2704
2705
2706
2707
2708
2709
2710
2711
2712
2713
2714
2715
2716
2717
2718
2719
2720
2721
2722
2723
2724
2725
2726
2727
2728
2729
2730
2731
2732
2733
2734
2735
2736
2737
2738
2739
2740
2741
2742
2743
2744
2745
2746
2747
2748
2749
2750
2751
2752
2753
2754
2755
2756
2757
2758
2759
2760
2761
2762
2763
2764
2765
2766
2767
2768
2769
2770
2771
2772
2773
2774
2775
2776
2777
2778
2779
2780
2781
2782
2783
2784
2785
2786
2787
2788
2789
2790
2791
2792
2793
2794
2795
2796
2797
2798
2799
2800
2801
2802
2803
2804
2805
2806
2807
2808
2809
2810
2811
2812
2813
2814
2815
2816
2817
2818
2819
2820
2821
2822
2823
2824
2825
2826
2827
2828
2829
2830
2831
2832
2833
2834
2835
2836
2837
2838
2839
2840
2841
2842
2843
2844
2845
2846
2847
2848
2849
2850
2851
2852
2853
2854
2855
2856
2857
2858
2859
2860
2861
2862
2863
2864
2865
2866
2867
2868
2869
2870
2871
2872
2873
2874
2875
2876
2877
2878
2879
2880
2881
2882
2883
2884
2885
2886
2887
2888
2889
2890
2891
2892
2893
2894
2895
2896
2897
2898
2899
2900
2901
2902
2903
2904
2905
2906
2907
2908
2909
2910
2911
2912
2913
2914
2915
2916
2917
2918
2919
2920
2921
2922
2923
2924
2925
2926
2927
2928
2929
2930
2931
2932
2933
2934
2935
2936
2937
2938
2939
2940
2941
2942
2943
2944
2945
2946
2947
2948
2949
2950
2951
2952
2953
2954
2955
2956
2957
2958
2959
2960
2961
2962
2963
2964
2965
2966
2967
2968
2969
2970
2971
2972
2973
2974
2975
2976
2977
2978
2979
2980
2981
2982
2983
2984
2985
2986
2987
2988
2989
2990
2991
2992
2993
29
```

```

1167             << " in method " << get_method_name(method) << "."
1168         << endl;
1169     }
1170 }
1171
1172 // 检查方法体的类型
1173 Expression method_body = get_method_expr(method);
1174 Symbol body_type = method_body->semant();
1175 Symbol return_type = get_method_return_type(method);
1176
1177 // 检查方法体的类型是否与返回类型兼容
1178 if (body_type != No_type && !conforms_to(body_type, return_type))
{
    semant_error(c) << "Inferred type " << body_type
    << " of method body does not conform to declared return
type "
    << return_type << " for method " << get_method_name(
method) << "." << endl;
}
1182
1183 // 退出作用域
1184 if (semant_debug) {
    cerr << "Exiting scope for method " << current_method << endl
;
}
1187
1188 object_table.exitscope();
}
1189
1190
1191 // 检查是否是属性
1192 attr_class* attr = dynamic_cast<attr_class*>(f);
1193 if (attr) {
    // 检查属性类型是否存在
    if (get_attr_type_decl(attr) != SELF_TYPE) {
        Class_attr_class = lookup_class(get_attr_type_decl(attr));
        if (!attr_class) {
            semant_error(c) << "Undefined type " << get_attr_type_
decl(attr)
            << " in attribute " << get_attr_name(attr) << "." <<
endl;
        }
    }
}
1202
1203 // 检查属性初始化表达式
1204 if (get_attr_init(attr)) {
    // 设置当前类
    current_class = get_class_name(c);

    // 检查初始化表达式的类型
    Expression init_expr = get_attr_init(attr);
    Symbol init_type = init_expr->semant();
    Symbol attr_type = get_attr_type_decl(attr);

    // 检查初始化表达式的类型是否与属性类型兼容
    if (init_type != No_type && !conforms_to(init_type, attr_type))
    {
        semant_error(c) << "Inferred type " << init_type
        << " of initialization of attribute " << get_attr_
name(attr)
        << " does not conform to declared type " << attr_type
        << "." << endl;
    }
}
1219
1220
1221
1222 }
1223
1224 void ClassTable::check_method_override(Class_ c)
1225 {
    // 如果是Object类, 没有父类, 不需要检查
    if (get_class_name(c) == Object || get_class_parent(c) == No_class) {
        return;
    }

    // 获取父类
    Class_ parent_class = lookup_class(get_class_parent(c));
    if (!parent_class) {
        return; // 父类不存在, 已经在构造函数中报错
    }

    // 获取当前类和父类的方法
1230
1231
1232
1233
1234
1235
1236
1237
1238 Features child_features = get_class_features(c);
1239 Features parent_features = get_class_features(parent_class);
1240
1241 // 遍历当前类的方法
1242 for (int i = child_features->first(); child_features->more(i); i = child_
features->next(i)) {
    Feature child_feature = child_features->nth(i);

    // 检查是否是方法
    method_class* child_method = dynamic_cast<method_class*>(child_
feature);
    if (!child_method) {
        continue; // 不是方法, 跳过
    }

    // 在父类中查找同名方法
    method_class* parent_method = NULL;
    for (int j = parent_features->first(); parent_features->more(j); j =
parent_features->next(j)) {
        Feature parent_feature = parent_features->nth(j);
        method_class* method = dynamic_cast<method_class*>(parent_feature
);
        if (method && get_method_name(method) == get_method_name(child_
method)) {
            parent_method = method;
            break;
        }
    }

    // 如果父类中有同名方法, 检查重写规则
    if (parent_method) {
        // 检查返回类型是否相同
        if (get_method_return_type(child_method) != get_method_return_
type(parent_method)) {
            semant_error(c) << "In redefined method " << get_method_name(
child_method)
            << ", return type " << get_method_return_type(child_
method)
            << " is different from original return type " << get_
method_return_type(parent_method) << "." << endl;
        }

        // 检查参数数量是否相同
        Formals child_formals = get_method_formals(child_method);
        Formals parent_formals = get_method_formals(parent_method);

        if (list_length(child_formals) != list_length(parent_formals)) {
            semant_error(c) << "Incompatible number of formal parameters
in redefined method "
            << get_method_name(child_method) << "." << endl;
            continue;
        }

        // 检查每个参数的类型是否相同
        for (int k = child_formals->first(); child_formals->more(k); k =
child_formals->next(k)) {
            Formal child_formal = child_formals->nth(k);
            Formal parent_formal = parent_formals->nth(k);

            if (get_formal_type_decl(child_formal) != get_formal_type_
decl(parent_formal)) {
                semant_error(c) << "In redefined method " << get_method_
name(child_method)
                << ", parameter type " << get_formal_type_decl(child_
formal)
                << " is different from original type " << get_formal_
type_decl(parent_formal) << "." << endl;
            }
        }
    }
}

void ClassTable::check_attribute_inheritance(Class_ c)
{
    // 如果是Object类, 没有父类, 不需要检查
    if (get_class_name(c) == Object || get_class_parent(c) == No_class) {
        return;
    }

    // 获取父类
}

```

```

1304     Class_ parent_class = lookup_class(get_class_parent(c));
1305     if (!parent_class) {
1306         return; // 父类不存在, 已经在构造函数中报错
1307     }
1308
1309     // 获取当前类和父类的特性
1310     Features child_features = get_class_features(c);
1311     Features parent_features = get_class_features(parent_class);
1312
1313     // 遍历当前类的属性
1314     for (int i = child_features->first(); child_features->more(i); i = child_
1315         features->next(i)) {
1316         Feature child_feature = child_features->nth(i);
1317
1318         // 检查是否是属性
1319         attr_class* child_attr = dynamic_cast<attr_class*>(child_feature);
1320         if (!child_attr) {
1321             continue; // 不是属性, 跳过
1322         }
1323
1324         // 在父类中查找同名属性
1325         attr_class* parent_attr = NULL;
1326         for (int j = parent_features->first(); parent_features->more(j); j =
1327             parent_features->next(j)) {
1328             Feature parent_feature = parent_features->nth(j);
1329             attr_class* attr = dynamic_cast<attr_class*>(parent_feature);
1330             if (attr && get_attr_name(attr) == get_attr_name(child_attr)) {
1331                 parent_attr = attr;
1332                 break;
1333             }
1334
1335             // 如果父类中有同名属性, 报错
1336             if (parent_attr) {
1337                 semantic_error(c) << "Attribute " << get_attr_name(child_attr)
1338                 << " is an attribute of an inherited class." << endl;
1339             }
1340         }
1341
1342
1343
1344     void initialize_constants(void)
1345     {
1346         arg      = idtable.add_string("arg");
1347         arg2     = idtable.add_string("arg2");
1348         Bool     = idtable.add_string("Bool");
1349         concat   = idtable.add_string("concat");
1350         cool_abort = idtable.add_string("abort");
1351         copy    = idtable.add_string("copy");
1352         Int      = idtable.add_string("Int");
1353         in_int   = idtable.add_string("in_int");
1354         in_string = idtable.add_string("in_string");
1355         IO       = idtable.add_string("IO");
1356         length   = idtable.add_string("length");
1357         Main     = idtable.add_string("Main");
1358         main_meth = idtable.add_string("main");
1359
1360         // _no_class is a symbol that can't be the name of any
1361         // user-defined class.
1362         No_class  = idtable.add_string("_no_class");
1363         No_type   = idtable.add_string("_no_type");
1364         Object    = idtable.add_string("Object");
1365         out_int   = idtable.add_string("out_int");
1366         out_string = idtable.add_string("out_string");
1367         prim_slot = idtable.add_string("_prim_slot");
1368         self      = idtable.add_string("self");
1369         SELF_TYPE = idtable.add_string("SELF_TYPE");
1370         Str       = idtable.add_string("String");
1371         str_field = idtable.add_string("_str_field");
1372         substr    = idtable.add_string("substr");
1373         type_name = idtable.add_string("type_name");
1374         val       = idtable.add_string("_val");
1375
1376     // 符号定义
1377     Symbol arg;
1378     Symbol arg2;
1379     Symbol Bool;
1380     Symbol concat;
1381     Symbol cool_abort;
1382     Symbol copy;
1383     Symbol Int;
1384     Symbol in_int;
1385     Symbol in_string;
1386     Symbol IO;
1387     Symbol length;
1388     Symbol Main;
1389     Symbol main_meth;
1390     Symbol No_class;
1391     Symbol No_type;
1392     Symbol Object;
1393     Symbol out_int;
1394     Symbol out_string;
1395     Symbol prim_slot;
1396     Symbol self;
1397     Symbol SELF_TYPE;
1398     Symbol Str;
1399     Symbol str_field;
1400     Symbol substr;
1401     Symbol type_name;
1402     Symbol val;
1403
1404
1405
1406     // 获取列表长度
1407
1408
1409     int list_length(Expressions exprs) {
1410         int count = 0;
1411         if (exprs) {
1412             for (int i = exprs->first(); exprs->more(i); i = exprs->next(i)) {
1413                 count++;
1414             }
1415         }
1416         return count;
1417     }
1418
1419     int list_length(Cases cases) {
1420         int count = 0;
1421         if (cases) {
1422             for (int i = cases->first(); cases->more(i); i = cases->next(i)) {
1423                 count++;
1424             }
1425         }
1426         return count;
1427     }

```