

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 (* error: 类型不匹配 *)
11 class A {
12   test(): Int {
13     "hello" + 5 -- 字符串不能与整数相加
14   };
15 };
16 (* error: 未定义类 *)
```

```

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: $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 and 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 };

```

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：

```

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

```

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

```

```

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

```


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

```

```

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

```

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     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             count++;
142         }
143     }
144     return count;
145 }
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 = 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         Feature f = features->nth(i);
193         method_class* method = dynamic_cast<method_class*>(f);
194         if (method && get_method_name(method) == method_name) {
195             return method;
196         }
197     }
198
199     // 如果在当前类中找不到, 递归查找父类
200     Symbol parent = get_class_parent(c);
201     if (parent != No_class) {
202         return lookup_method(parent, method_name);
203     }
204
205     return NULL;
206 }
207
208 // 辅助函数: 查找属性
209 Feature lookup_attribute(Symbol class_name, Symbol attr_name) {
210     Class_c = classtable->lookup_class(class_name);
211     if (!c) return NULL;
212
213     Features features = get_class_features(c);
214     for (int i = features->first(); features->more(i); i = features->next(i)) {
215         Feature f = features->nth(i);
216         attr_class* attr = dynamic_cast<attr_class*>(f);
217         if (attr && get_attr_name(attr) == attr_name) {
218             return attr;
219         }
220     }
221
222     // 如果在当前类中找不到, 递归查找父类
223     Symbol parent = get_class_parent(c);
224     if (parent != No_class) {
225         return lookup_attribute(parent, attr_name);
226     }
227
228     return NULL;
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 << "." << endl;
310     set_type(Object);
311     return Object;
312 }
313
314 // 赋值表达式
315 Symbol assign_class::semant()
316 {
317     // 检查变量是否已声明

```

```

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                 "." << endl;
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
334     // 检查表达式的类型
335     Symbol expr_type = expr->semant();
336
337     // 检查类型是否兼容
338     if (!conforms_to(expr_type, var_type)) {
339         classtable->semant_error() << "Type " << expr_type
340             << " of assigned expression does not conform to declared type "
341             << var_type << " of identifier " << name << "." << endl;
342     }
343
344     // 赋值表达式的类型是表达式的类型
345     set_type(expr_type);
346     return expr_type;
347 }
348
349 // new表达式
350 Symbol new__class::semant()
351 {
352     // 检查类型是否存在
353     if (type_name == SELF_TYPE) {
354         set_type(SELF_TYPE);
355         return SELF_TYPE;
356     }
357
358     Class_ c = classtable->lookup_class(type_name);
359     if (!c) {
360         classtable->semant_error() << "new is used with undefined class "
361             << type_name << "." << endl;
362         set_type(Object);
363         return Object;
364     }
365
366     set_type(type_name);
367     return type_name;
368 }
369
370 // isvoid表达式
371 Symbol isvoid_class::semant()
372 {
373     e1->semant();
374     set_type(Bool);
375     return Bool;
376 }
377
378 // no_expr表达式
379 Symbol no_expr_class::semant()
380 {
381     set_type(No_type);
382     return No_type;
383 }
384
385 // 算术表达式
386 Symbol plus_class::semant()
387 {
388     Symbol e1_type = e1->semant();
389     Symbol e2_type = e2->semant();
390
391     // 检查两个操作数是否都是Int类型
392     if (e1_type != Int || e2_type != Int) {
393         classtable->semant_error() << "non-Int arguments: "
394             << e1_type << " + " << e2_type << endl;
395         set_type(Object);
396         return Object;
397     }

```

```

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 // 逻辑表达式
523 Symbol comp_class::semant()
524 {
525     Symbol e1_type = e1->semant();
526
527     // 检查操作数是否是Bool类型
528     if (e1_type != Bool) {
529         classtable->semant_error() << "Argument of 'not' has type "
530             << e1_type << " instead of Bool." << endl;
531         set_type(Object);
532         return Object;
533     }
534
535     set_type(Bool);
536     return Bool;
537 }
538
539 // 控制流表达式
540
541 // if 表达式
542 Symbol cond_class::semant()
543 {
544     Symbol pred_type = pred->semant();
545
546     // 检查条件表达式是否是Bool类型
547     if (pred_type != Bool) {
548         classtable->semant_error() << "Predicate of 'if' does not have type
549             Bool." << endl;
550     }
551
552     Symbol then_type = then_exp->semant();
553     Symbol else_type = else_exp->semant();
554
555     // if 表达式的类型是then分支和else分支的最小公共祖先
556     Symbol result_type = join(then_type, else_type);
557     set_type(result_type);
558     return result_type;
559 }

```

```

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     // 将变量添加到对象符号表
612     object_table.addid(identifier, new SymbolEntry(type_decl));
613
614     // 检查body表达式
615     Symbol body_type = body->semant();
616
617     // 恢复对象符号表
618     object_table.exitscope();
619
620     set_type(body_type);
621     return body_type;
622 }
623
624 // case表达式
625 Symbol typcase_class::semant()
626 {
627     // 检查case表达式的类型
628     Symbol expr_type = expr->semant();
629
630     // 保存当前对象符号表
631     object_table.enterscope();
632
633     Symbol result_type = No_type;
634     std::set<Symbol> case_types; // 用于检查重复的case类型
635
636     // 依次检查每个分支
637     for (int i = cases->first(); cases->more(i); i = cases->next(i)) {
638         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
669 // 恢复对象符号表
670 object_table.exitscope();
671
672 // 如果没有分支, 结果是Object
673 if (result_type == No_type) {
674     result_type = Object;
675 }
676
677 set_type(result_type);
678 return result_type;
679 }
680
681 // 方法调用表达式
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给定类型的子类
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
703     // 查找方法
704     Feature method = lookup_method(type_name, name);
705     if (!method) {
706         classtable->semant_error() << "Dispatch to undefined method "
707         << name << "." << endl;
708         set_type(Object);
709         return Object;
710     }
711
712     // 获取方法的参数类型和返回类型
713     Formals formals = get_method_formals(method);
714     Symbol return_type = get_method_return_type(method);
715
716     // 检查参数数量

```

```

716     if (list_length(formals) != list_length(actual)) {
717         classtable->semant_error() << "Method " << name
718         << " called with wrong number of arguments." << endl;
719         set_type(Object);
720         return Object;
721     }
722
723     // 检查每个参数的类型
724     for (int i = actual->first(); actual->more(i); i = actual->next(i)) {
725         Expression arg = actual->nth(i);
726         Formal formal = formals->nth(i);
727
728         Symbol arg_type = arg->semant();
729         Symbol formal_type = get_formal_type_decl(formal);
730
731         if (!conforms_to(arg_type, formal_type)) {
732             classtable->semant_error() << "In call of method " << name
733             << ", type " << arg_type << " of parameter " << get_formal_
734             name(formal)
735             << " does not conform to declared type " << formal_type << ".
736             " << endl;
737         }
738     }
739
740     // 如果返回类型是SELF_TYPE, 则返回表达式的类型
741     if (return_type == SELF_TYPE) {
742         if (expr_type == SELF_TYPE) {
743             set_type(SELF_TYPE);
744             return SELF_TYPE;
745         } else {
746             set_type(expr_type);
747             return expr_type;
748         }
749     } else {
750         set_type(return_type);
751         return return_type;
752     }
753
754     // 动态分派表达式
755     // dispatch_class的semant方法实现
756     Symbol dispatch_class::semant() {
757         // 检查表达式类型
758         Symbol expr_type = expr->semant();
759
760         // 如果表达式类型是SELF_TYPE, 则使用当前类
761         Symbol dispatch_type = expr_type;
762         if (dispatch_type == SELF_TYPE) {
763             dispatch_type = current_class;
764         }
765
766         // 查找方法
767         Feature method = lookup_method(dispatch_type, name);
768         if (!method) {
769             classtable->semant_error() << "Dispatch to undefined method "
770             << name << "." << endl;
771             set_type(Object);
772             return Object;
773         }
774
775         // 获取方法的参数类型和返回类型
776         Formals formals = get_method_formals(method);
777         Symbol return_type = get_method_return_type(method);
778
779         // 检查参数数量
780         if (list_length(formals) != list_length(actual)) {
781             classtable->semant_error() << "Method " << name
782             << " called with wrong number of arguments." << endl;
783             set_type(Object);
784             return Object;
785         }
786
787         // 检查每个参数的类型
788         for (int i = actual->first(); actual->more(i); i = actual->next(i)) {
789             Expression arg = actual->nth(i);
790             Formal formal = formals->nth(i);
791
792             Symbol arg_type = arg->semant();
793             Symbol formal_type = get_formal_type_decl(formal);
794
795             if (!conforms_to(arg_type, formal_type)) {

```

```

795         classtable->semant_error() << "In call of method " << name
796         << ", type " << arg_type << " of parameter " << get_formal_
797         name(formal)
798         << " does not conform to declared type " << formal_type << ".
799         " << endl;
800     }
801
802     // 如果返回类型是SELF_TYPE, 则返回表达式的类型
803     if (return_type == SELF_TYPE) {
804         if (expr_type == SELF_TYPE) {
805             set_type(SELF_TYPE);
806             return SELF_TYPE;
807         } else {
808             set_type(expr_type);
809             return expr_type;
810         }
811     } else {
812         set_type(return_type);
813         return return_type;
814     }
815
816     }
817
818     // 初始化类符号表
819     class_table.enterscope();
820
821     // 安装基本类
822     install_basic_classes();
823
824     // 检查用户定义的类型
825     for (int i = classes->first(); classes->more(i); i = classes->next(i)) {
826         Class_ c = classes->nth(i);
827
828         // 检查类名是否重复
829         if (class_table.probe(get_class_name(c))) {
830             semant_error(c) << "Class " << get_class_name(c) << " was
831             previously defined." << endl;
832         } else {
833             Class_ *class_ptr = new Class(c);
834             class_table.addid(get_class_name(c), class_ptr);
835         }
836     }
837
838     // 检查继承关系
839     for (int i = classes->first(); classes->more(i); i = classes->next(i)) {
840         Class_ c = classes->nth(i);
841
842         // 检查父类是否存在
843         if (get_class_parent(c) != No_class && !class_table.lookup(get_class_
844         parent(c))) {
845             semant_error(c) << "Class " << get_class_name(c) << " inherits
846             from an undefined class " << get_class_parent(c) << "." << endl;
847         }
848
849         // 检查是否继承自Int, Bool或Str
850         if (get_class_parent(c) == Int || get_class_parent(c) == Bool || get_
851         class_parent(c) == Str) {
852             semant_error(c) << "Class " << get_class_name(c) << " cannot
853             inherit class " << get_class_parent(c) << "." << endl;
854         }
855     }
856
857     // 检查Main类是否存在
858     if (!class_table.lookup(Main)) {
859         semant_error() << "Class Main is not defined." << endl;
860     } else {
861         // 检查Main类中是否有main方法
862         Class_ main_class = lookup_class(Main);
863         Features features = get_class_features(main_class);
864         bool has_main_method = false;

```



```

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     // There is no need for method bodies in the basic classes---these
905     // are already built in to the runtime system.
906
907     Object_class =
908     class_(Object,
909     No_class,
910     append_Features(
911     append_Features(
912     single_Features(method(cool_abort, nil_Formals(), Object, no
913     _expr()),
914     single_Features(method(type_name, nil_Formals(), Str, no
915     _expr()))),
916     single_Features(method(copy, nil_Formals(), SELF_TYPE, no_expr()
917     )),
918     filename);
919
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     single_Features(method(out_string, single_Formals(formal
933     (arg, Str)),
934     SELF_TYPE, no_expr()),
935     single_Features(method(out_int, single_Formals(formal(
936     arg, Int)),
937     SELF_TYPE, no_expr()))),
938     single_Features(method(in_string, nil_Formals(), Str, no
939     _expr()))),
940     single_Features(method(in_int, nil_Formals(), Int, no_expr()))),
941     filename);
942
943     //

```

```

939     // The Int class has no methods and only a single attribute, the
940     // "val" for the integer.
941     //
942     Int_class =
943     class_(Int,
944     Object,
945     single_Features(attr(val, prim_slot, no_expr()),
946     filename);
947
948     //
949     // Bool also has only the "val" slot.
950     //
951     Bool_class =
952     class_(Bool, Object, single_Features(attr(val, prim_slot, no_expr()),
953     filename);
954
955     //
956     // The class Str has a number of slots and operations:
957     // val                the length of the string
958     // str_field          the string itself
959     // length() : Int     returns length of the
960     // string
961     // concat(arg: Str) : Str    self + arg
962     // substr(arg: Int, arg2: Int) : Str    substring from arg to arg2
963     //
964     Str_class =
965     class_(Str,
966     Object,
967     append_Features(
968     append_Features(
969     append_Features(
970     single_Features(attr(val, Int, no_expr()),
971     single_Features(attr(str_field, prim_slot, no_expr()))),
972     single_Features(method(length, nil_Formals(), Int, no_expr()
973     )),
974     append_Features(
975     single_Features(method(concat,
976     single_Formals(formal(arg, Str)),
977     Str, no_expr()),
978     single_Features(method(substr,
979     append_Formals(single_Formals(formal(arg, Int)),
980     single_Formals(formal(arg2, Int))),
981     Str, no_expr()))),
982     filename);
983
984     // 添加基本类到符号表
985     class_table.addid(Object, &Object_class);
986     class_table.addid(IO, &IO_class);
987     class_table.addid(Int, &Int_class);
988     class_table.addid(Bool, &Bool_class);
989     class_table.addid(Str, &Str_class);
990 }
991
992 // semant_error is an overloaded function for reporting errors
993 // during semantic analysis. There are three versions:
994 //
995 // ostream& ClassTable::semant_error()
996 // ostream& ClassTable::semant_error(Class_ c)
997 // ostream& ClassTable::semant_error(Symbol filename, tree_node *t)
998 //
999
1000 ostream& ClassTable::semant_error(Class_ c)
1001 {
1002     return semant_error(c->get_filename(), c);
1003 }
1004
1005 ostream& ClassTable::semant_error(Symbol filename, tree_node *t)
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_IO() { return IO_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     // 获取child类
1058     Class_ child_class = lookup_class(child);
1059     if (!child_class) {
1060         return false;
1061     }
1062
1063     // 获取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 << endl;
1134             }
1135             object_table.enterscope();
1136
1137             // 添加self到作用域
1138             if (semant_debug) {
1139                 cerr << "Adding self to scope in method " << current_method << endl;
1140             }
1141             object_table.addid(self, new SymbolEntry(SELF_TYPE));
1142
1143             // 添加参数到作用域
1144             Formals formals = get_method_formals(method);
1145             for (int j = formals->first(); formals->more(j); j = formals->next(j)) {
1146                 Formal formal = formals->nth(j);
1147                 Symbol formal_name = get_formal_name(formal);
1148                 Symbol formal_type = get_formal_type_decl(formal);
1149                 object_table.addid(formal_name, new SymbolEntry(formal_type));
1150             }
1151
1152             // 检查方法返回类型是否存在
1153             if (get_method_return_type(method) != SELF_TYPE) {
1154                 Class_ return_class = lookup_class(get_method_return_type(
1155                     method));
1156                 if (!return_class) {
1157                     semant_error(c) << "Undefined return type " << get_method_return_type(method) << " in method " << get_method_name(method) << " " << endl;
1158                 }
1159             }
1160
1161             // 检查方法参数类型是否存在
1162             for (int j = formals->first(); formals->more(j); j = formals->next(j)) {
1163                 Formal formal = formals->nth(j);
1164                 if (get_formal_type_decl(formal) != SELF_TYPE) {
1165                     Class_ param_class = lookup_class(get_formal_type_decl(
1166                         formal));
1167                     if (!param_class) {
1168                         semant_error(c) << "Undefined parameter type " << get_formal_type_decl(formal)

```

```

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))
1179 {
1180     semant_error(c) << "Inferred type " << body_type
1181     << " of method body does not conform to declared return
1182     type "
1183     << return_type << " for method " << get_method_name(
1184     method) << "." << endl;
1185 }
1186
1187 // 退出作用域
1188 if (semant_debug) {
1189     cerr << "Exiting scope for method " << current_method << endl
1190 ;
1191 }
1192 object_table.exitscope();
1193 }
1194
1195 // 检查是否是属性
1196 attr_class* attr = dynamic_cast<attr_class*>(f);
1197 if (attr) {
1198     // 检查属性类型是否存在
1199     if (get_attr_type_decl(attr) != SELF_TYPE) {
1200         Class_attr_class = lookup_class(get_attr_type_decl(attr));
1201         if (!attr_class) {
1202             semant_error(c) << "Undefined type " << get_attr_type_
1203             decl(attr)
1204             << " in attribute " << get_attr_name(attr) << "." <<
1205             endl;
1206         }
1207     }
1208 }
1209
1210 // 检查属性初始化表达式
1211 if (get_attr_init(attr)) {
1212     // 设置当前类
1213     current_class = get_class_name(c);
1214
1215     // 检查初始化表达式的类型
1216     Expression init_expr = get_attr_init(attr);
1217     Symbol init_type = init_expr->semant();
1218     Symbol attr_type = get_attr_type_decl(attr);
1219
1220     // 检查初始化表达式的类型是否与属性类型兼容
1221     if (init_type != No_type && !conforms_to(init_type, attr_type
1222 )) {
1223         semant_error(c) << "Inferred type " << init_type
1224         << " of initialization of attribute " << get_attr_
1225         name(attr)
1226         << " does not conform to declared type " << attr_type
1227         << "." << endl;
1228     }
1229 }
1230 }
1231
1232 void ClassTable::check_method_override(Class_ c)
1233 {
1234     // 如果是Object类, 没有父类, 不需要检查
1235     if (get_class_name(c) == Object || get_class_parent(c) == No_class) {
1236         return;
1237     }
1238
1239     // 获取父类
1240     Class_parent_class = lookup_class(get_class_parent(c));
1241     if (!parent_class) {
1242         return; // 父类不存在, 已经在构造函数中报错
1243     }
1244
1245     // 获取当前类和父类的方法

```

```

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_
1243     features->next(i)) {
1244         Feature child_feature = child_features->nth(i);
1245
1246         // 检查是否是方法
1247         method_class* child_method = dynamic_cast<method_class*>(child_
1248         feature);
1249         if (!child_method) {
1250             continue; // 不是方法, 跳过
1251         }
1252
1253         // 在父类中查找同名方法
1254         method_class* parent_method = NULL;
1255         for (int j = parent_features->first(); parent_features->more(j); j =
1256         parent_features->next(j)) {
1257             Feature parent_feature = parent_features->nth(j);
1258             method_class* method = dynamic_cast<method_class*>(parent_feature
1259 );
1260             if (method && get_method_name(method) == get_method_name(child_
1261             method)) {
1262                 parent_method = method;
1263                 break;
1264             }
1265         }
1266
1267         // 如果父类中有同名方法, 检查重写规则
1268         if (parent_method) {
1269             // 检查返回类型是否相同
1270             if (get_method_return_type(child_method) != get_method_return_
1271             type(parent_method)) {
1272                 semant_error(c) << "In redefined method " << get_method_name(
1273                 child_method)
1274                 << ", return type " << get_method_return_type(child_
1275                 method)
1276                 << " is different from original return type " << get_
1277                 method_return_type(parent_method) << "." << endl;
1278             }
1279
1280             // 检查参数数量是否相同
1281             Formals child_formals = get_method_formals(child_method);
1282             Formals parent_formals = get_method_formals(parent_method);
1283
1284             if (list_length(child_formals) != list_length(parent_formals)) {
1285                 semant_error(c) << "Incompatible number of formal parameters
1286                 in redefined method "
1287                 << get_method_name(child_method) << "." << endl;
1288                 continue;
1289             }
1290
1291             // 检查每个参数的类型是否相同
1292             for (int k = child_formals->first(); child_formals->more(k); k =
1293             child_formals->next(k)) {
1294                 Formal child_formal = child_formals->nth(k);
1295                 Formal parent_formal = parent_formals->nth(k);
1296
1297                 if (get_formal_type_decl(child_formal) != get_formal_type_
1298                 decl(parent_formal)) {
1299                     semant_error(c) << "In redefined method " << get_method_
1300                     name(child_method)
1301                     << ", parameter type " << get_formal_type_decl(child_
1302                     formal)
1303                     << " is different from original type " << get_formal_
1304                     type_decl(parent_formal) << "." << endl;
1305                 }
1306             }
1307         }
1308     }
1309 }
1310
1311 void ClassTable::check_attribute_inheritance(Class_ c)
1312 {
1313     // 如果是Object类, 没有父类, 不需要检查
1314     if (get_class_name(c) == Object || get_class_parent(c) == No_class) {
1315         return;
1316     }
1317
1318     // 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_
    features->next(i)) {
1315     Feature child_feature = child_features->nth(i);
1316
1317     // 检查是否是属性
1318     attr_class* child_attr = dynamic_cast<attr_class*>(child_feature);
1319     if (!child_attr) {
1320         continue; // 不是属性, 跳过
1321     }
1322
1323     // 在父类中查找同名属性
1324     attr_class* parent_attr = NULL;
1325     for (int j = parent_features->first(); parent_features->more(j); j =
        parent_features->next(j)) {
1326         Feature parent_feature = parent_features->nth(j);
1327         attr_class* attr = dynamic_cast<attr_class*>(parent_feature);
1328         if (attr && get_attr_name(attr) == get_attr_name(child_attr)) {
1329             parent_attr = attr;
1330             break;
1331         }
1332     }
1333
1334     // 如果父类中有同名属性, 报错
1335     if (parent_attr) {
1336         semant_error(c) << "Attribute " << get_attr_name(child_attr)
1337             << " is an attribute of an inherited class." << endl;
1338     }
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     // _no_class is a symbol that can't be the name of any
1360     // user-defined class.
1361     No_class  = idtable.add_string("_no_class");
1362     No_type   = idtable.add_string("_no_type");
1363     Object    = idtable.add_string("Object");
1364     out_int   = idtable.add_string("out_int");
1365     out_string = idtable.add_string("out_string");
1366     prim_slot = idtable.add_string("_prim_slot");
1367     self      = idtable.add_string("self");
1368     SELF_TYPE = idtable.add_string("SELF_TYPE");
1369     Str       = idtable.add_string("String");
1370     str_field = idtable.add_string("_str_field");
1371     substr    = idtable.add_string("substr");
1372     type_name = idtable.add_string("type_name");
1373     val       = idtable.add_string("_val");
1374 }
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 // 获取列表长度
1406
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 }

```