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

```
class Main {
  main(): IO {{
    let x : Int <- 5 in
      (new IO).out_int(x)
  }};
};

class A inherits IO {
  test(): Int {
    let y : String <- "hello" in
      y.length()
  };
};
```

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：

```
(*
 *  execute "coolc bad.cl" to see the error messages
     that the coolc parser
 *  generates
 *
 *  execute "mysemant bad.cl" to see the error messages
     that your semantic analyzer
 *  generates
 *)

(* error: 类型不匹配 *)
class A {
  test(): Int {
    "hello" + 5  -- 字符串不能与整数相加
  };
};

(* error: 未定义的类 *)
```

```
Class B inherits UndefinedClass {
};

(* error: 循环继承 *)
Class C inherits D {
};

Class D inherits C {
};

(* error: 方法重写不兼容 *)
Class E inherits IO {
  out_string(s : String): IO {
    new IO
  };
};

(* error: 变量未定义 *)
Class F {
  test(): Int {
    x + 1  -- x未定义
  };
};
```

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
    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：

```
enum SemanticErrorType {
  ERROR_UNDEFINED_CLASS,
  ERROR_UNDEFINED_METHOD,
  ERROR_UNDEFINED_VARIABLE,
  ERROR_TYPE_MISMATCH,
  ERROR_INHERITANCE_CYCLE,
  ERROR_METHOD_OVERRIDE,
  ERROR_SELF_TYPE_NEW,
  ERROR_SELF_TYPE_ASSIGN,
  ERROR_CASE_BRANCH_TYPES,
  ERROR_LET_REDEFINITION,
  ERROR_ATTRIBUTE_REDEFINITION
};
```

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
bool conforms_to(Symbol child, Symbol parent) {
  // 1. 相同类型总是兼容
  if (child == parent) return true;

  // 2. 任何类型与Object兼容
  if (parent == Object) return true;

  // 3. SELF_TYPE与自身类型兼容
  if (child == SELF\textunderscore TYPE && parent ==
     SELF\textunderscore TYPE) return true;

  // 4. 检查继承关系
  return is_subclass(child, parent);
}
```

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：

```
// 表达式类型检查
Symbol expr_class::tc_type_check(ClassTable* classTable,
      Symbol currentClass) {
  // 基本表达式类型检查逻辑
  return No_type;
}

// 赋值表达式类型检查
Symbol assign_class::tc_type_check(ClassTable*
     classTable, Symbol currentClass) {
  Symbol exprType = expr->tc_type_check(classTable,
     currentClass);
  Symbol varType = classTable->lookupVar(currentClass,
     name);

  if (varType == NULL) {
    classTable->semant_error(this) << "Assignment to
     undeclared variable "
                                   << name << ".\n";
    return Object;
  }

  if (!classTable->conforms_to(exprType, varType)) {
    classTable->semant_error(this) << "Type " <<
     exprType
                                   << " of assigned
     expression does not conform to "
                                   << "declared type " <<
      varType
                                   << " of variable " <<
     name << ".\n";
  }

  return exprType;
}

// 分派表达式类型检查
Symbol dispatch_class::tc_type_check(ClassTable*
     classTable, Symbol currentClass) {
  Symbol exprType = expr->tc_type_check(classTable,
     currentClass);

  // 检查方法是否存在
  MethodType methodType = classTable->lookupMethod(
     exprType, name);
  if (methodType == NULL) {
    classTable->semant_error(this) << "Dispatch to
     undefined method "
                                   << name << ".\n";
    return Object;
  }

  // 检查参数类型
  for (int i = actual->first(); actual->more(i); i =
     actual->next(i)) {
    Symbol actualType = actual->nth(i)->tc_type_check(
     classTable, currentClass);
    Symbol formalType = methodType->paramTypes[i];

    if (!classTable->conforms_to(actualType, formalType)
     ) {
      classTable->semant_error(this) << "In call of
     method " << name
                                     << ", type " <<
     actualType
                                     << " of parameter "
     << i
                                     << " does not
     conform to declared type "
                                     << formalType << ".\
     n";
    }
  }

  return methodType->returnType;
}
```

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：

```
// 类定义类型检查
Symbol class__class::tc_type_check(ClassTable* classTable) {
  // 检查父类是否存在
  if (parent != No_class && !classTable->hasClass(parent)) {
    classTable->semant_error(this) << "Class " << name
                                   << " inherits from an undefined class "
                                   << parent << ".\n";
    parent = Object;
  }

  // 检查属性和方法
  for (int i = features->first(); features->more(i); i = features->next(i)) {
    features->nth(i)->tc_type_check(classTable, name);
  }

  return name;
}

// 属性定义类型检查
Symbol attr_class::tc_type_check(ClassTable* classTable, Symbol currentClass) {
  // 检查属性类型是否存在
  if (!classTable->hasClass(type_decl)) {
    classTable->semant_error(this) << "Attribute " << name
                                   << " has undefined type "
                                   << type_decl << ".\n";
    return Object;
  }

  // 检查初始化表达式类型
  if (init) {
    Symbol initType = init->tc_type_check(classTable, currentClass);
    if (!classTable->conforms_to(initType, type_decl)) {
      classTable->semant_error(this) << "Inferred type " << initType
                                     << " of initialization of attribute "
                                     << name << " does not conform to "
                                     << "declared type " << type_decl << ".\n";
    }
  }

  return type_decl;
}

// 方法定义类型检查
Symbol method_class::tc_type_check(ClassTable* classTable, Symbol currentClass) {
  // 检查返回类型是否存在
  if (!classTable->hasClass(return_type)) {
    classTable->semant_error(this) << "Method " << name
                                   << " has undefined return type "
                                   << return_type << ".\n";
    return_type = Object;
  }

  // 检查参数类型
  for (int i = formals->first(); formals->more(i); i = formals->next(i)) {
    Symbol formalType = formals->nth(i)->get_type_decl();
    if (!classTable->hasClass(formalType)) {
      classTable->semant_error(this) << "Parameter " << formals->nth(i)->get_name()
                                     << " has undefined type "
                                     << formalType << ".\n";
    }
  }

  // 检查方法体类型
  Symbol bodyType = expr->tc_type_check(classTable, currentClass);
  if (!classTable->conforms_to(bodyType, return_type)) {
    classTable->semant_error(this) << "Inferred return type " << bodyType
                                   << " of method " << name
                                   << " does not conform to declared return type "
                                   << return_type << ".\n";
  }

  return return_type;
}
```

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

```

#include <stdlib.h>
#include <stdio.h>
#include <stdarg.h>
#include <set>
#include "semant.h"
#include "utilities.h"

// 辅助函数，用于访问类的protected成员
Symbol get_class_name(Class_ c) {
    class__class* class_ptr = dynamic_cast<class__class*>(c);
    if (class_ptr) {
        return class_ptr->get_name();
    }
    return NULL;
}

Symbol get_class_parent(Class_ c) {
    class__class* class_ptr = dynamic_cast<class__class*>(c);
    if (class_ptr) {
        return class_ptr->get_parent();
    }
    return NULL;
}

Features get_class_features(Class_ c) {
    class__class* class_ptr = dynamic_cast<class__class*>(c);
    if (class_ptr) {
        return class_ptr->get_features();
    }
    return NULL;
}

Symbol get_method_name(Feature f) {
    method_class* method = dynamic_cast<method_class*>(f);
    if (method) {
        return method->get_name();
    }
    return NULL;
}

Symbol get_method_return_type(Feature f) {
    method_class* method = dynamic_cast<method_class*>(f);
    if (method) {
        return method->get_return_type();
    }
    return NULL;
}

Formals get_method_formals(Feature f) {
    method_class* method = dynamic_cast<method_class*>(f);
    if (method) {
        return method->get_formals();
    }
    return NULL;
}

Expression get_method_expr(Feature f) {
    method_class* method = dynamic_cast<method_class*>(f);
    if (method) {
        return method->get_expr();
    }
    return NULL;
}

Symbol get_attr_name(Feature f) {
    attr_class* attr = dynamic_cast<attr_class*>(f);
    if (attr) {
        return attr->get_name();
    }
    return NULL;
}

Symbol get_attr_type_decl(Feature f) {
    attr_class* attr = dynamic_cast<attr_class*>(f);
    if (attr) {
        return attr->get_type_decl();
```

```
    }
    return NULL;
}

Expression get_attr_init(Feature f) {
    attr_class* attr = dynamic_cast<attr_class*>(f);
    if (attr) {
        return attr->get_init();
    }
    return NULL;
}

Symbol get_formal_name(Formal f) {
    formal_class* formal = dynamic_cast<formal_class*>(f);
    if (formal) {
        return formal->get_name();
    }
    return NULL;
}

Symbol get_formal_type_decl(Formal f) {
    formal_class* formal = dynamic_cast<formal_class*>(f);
    if (formal) {
        return formal->get_type_decl();
    }
    return NULL;
}

Symbol get_branch_type(Case branch) {
    branch_class* bc = dynamic_cast<branch_class*>(branch);
    if (bc) {
        return bc->get_type_decl();
    }
    return NULL;
}

Symbol get_branch_name(Case branch) {
    branch_class* bc = dynamic_cast<branch_class*>(branch);
    if (bc) {
        return bc->get_name();
    }
    return NULL;
}

Expression get_branch_expr(Case branch) {
    branch_class* bc = dynamic_cast<branch_class*>(branch);
    if (bc) {
        return bc->get_expr();
    }
    return NULL;
}

// 全局ClassTable指针，用于表达式类型检查
static ClassTable *classtable = NULL;



// 为Formals类型添加特化版本
int list_length(Formals formals) {
    int count = 0;
    if (formals) {
        for (int i = formals->first(); formals->more(i); i = formals->next(i)
    ) {
            count++;
        }
    }
    return count;
}

extern int semant_debug;
extern char *curr_filename;

// 当前类
static Symbol current_class = NULL;

// 当前方法
static Symbol current_method = NULL;

// 简单的Entry类，用于存储Symbol信息
class SymbolEntry : public Entry {
private:
    Symbol type;
public:
    SymbolEntry(Symbol t) : Entry("", 0, 0), type(t) {}
    Symbol get_type() { return type; }
};

// 对象符号表
static SymbolTable<Symbol, Entry> object_table;

// 辅助函数：检查类型是否兼容
bool conforms_to(Symbol type1, Symbol type2) {
    if (type1 == type2) return true;
    if (type1 == SELF_TYPE && type2 == SELF_TYPE) return true;
    if (type1 == SELF_TYPE) type1 = current_class;
    if (type2 == SELF_TYPE) type2 = current_class;
    return classtable->is_subclass(type1, type2);
}

// 辅助函数：获取最小公共祖先
Symbol join(Symbol type1, Symbol type2) {
    if (type1 == SELF_TYPE && type2 == SELF_TYPE) return SELF_TYPE;
    if (type1 == SELF_TYPE) type1 = current_class;
    if (type2 == SELF_TYPE) type2 = current_class;
    return classtable->least_common_ancestor(type1, type2);
}

// 辅助函数：查找方法
Feature lookup_method(Symbol class_name, Symbol method_name) {
    Class_ c = classtable->lookup_class(class_name);
    if (!c) return NULL;

    Features features = get_class_features(c);
    for (int i = features->first(); features->more(i); i = features->next(i))
        {
        Feature f = features->nth(i);
        method_class* method = dynamic_cast<method_class*>(f);
        if (method && get_method_name(method) == method_name) {
            return method;
        }
    }

    // 如果在当前类中找不到，递归查找父类
    Symbol parent = get_class_parent(c);
    if (parent != No_class) {
        return lookup_method(parent, method_name);
    }

    return NULL;
}

// 辅助函数：查找属性
Feature lookup_attribute(Symbol class_name, Symbol attr_name) {
    Class_ c = classtable->lookup_class(class_name);
    if (!c) return NULL;

    Features features = get_class_features(c);
    for (int i = features->first(); features->more(i); i = features->next(i))
        {
        Feature f = features->nth(i);
        attr_class* attr = dynamic_cast<attr_class*>(f);
        if (attr && get_attr_name(attr) == attr_name) {
            return attr;
        }
    }

    // 如果在当前类中找不到，递归查找父类
    Symbol parent = get_class_parent(c);
    if (parent != No_class) {
        return lookup_attribute(parent, attr_name);
    }

    return NULL;
}

// 实现program_class::semant()方法
void program_class::semant()
{
    // 初始化常量
    initialize_constants();

    // 初始化对象符号表 - 首先进入作用域
```

```
    if (semant_debug) {
        cerr << "Entering global scope" << endl;
    }
    object_table.enterscope();

    // 创建类表
    ::classtable = new ClassTable(classes);

    // 检查所有类的特性
    for (int i = classes->first(); classes->more(i); i = classes->next(i)) {
        current_class = get_class_name(classes->nth(i));
        if (semant_debug) {
            cerr << "Checking class features for " << current_class << endl;
        }
        classtable->check_class_features(classes->nth(i));
    }

    // 退出全局作用域
    if (semant_debug) {
        cerr << "Exiting global scope" << endl;
    }
    object_table.exitscope();

    // 如果有错误，退出
    if (classtable->errors()) {
        cerr << "Compilation halted due to static semantic errors." << endl;
        exit(1);
    }
}

// 实现各种表达式类型的semant方法

// 常量表达式
Symbol int_const_class::semant()
{
    set_type(Int);
    return Int;
}

Symbol bool_const_class::semant()
{
    set_type(Bool);
    return Bool;
}

Symbol string_const_class::semant()
{
    set_type(Str);
    return Str;
}

// 变量表达式
Symbol object_class::semant()
{
    // 查找变量在对象符号表中的类型
    SymbolEntry* entry = (SymbolEntry*)object_table.lookup(name);
    if (entry) {
        Symbol obj_type = entry->get_type();
        set_type(obj_type);
        return obj_type;
    }

    // 如果在对象符号表中找不到，查找属性
    Feature attr = lookup_attribute(current_class, name);
    if (attr) {
        Symbol attr_type = get_attr_type_decl(attr);
        set_type(attr_type);
        return attr_type;
    }

    // 如果都找不到，报错
    classtable->semant_error() << "Undeclared identifier " << name << "." <<
       endl;
    set_type(Object);
    return Object;
}

// 赋值表达式
Symbol assign_class::semant()
{
    // 检查变量是否已声明
    SymbolEntry* entry = (SymbolEntry*)object_table.lookup(name);
    Symbol var_type;
    if (!entry) {
        // 查找属性
        Feature attr = lookup_attribute(current_class, name);
        if (!attr) {
            classtable->semant_error() << "Undeclared identifier " << name <<
        "." << endl;
            set_type(Object);
            return Object;
        }
        var_type = get_attr_type_decl(attr);
    } else {
        var_type = entry->get_type();
    }

    // 检查表达式的类型
    Symbol expr_type = expr->semant();

    // 检查类型是否兼容
    if (!conforms_to(expr_type, var_type)) {
        classtable->semant_error() << "Type " << expr_type
            << " of assigned expression does not conform to declared type "
            << var_type << " of identifier " << name << "." << endl;
    }

    // 赋值表达式的类型是表达式的类型
    set_type(expr_type);
    return expr_type;
}

// new表达式
Symbol new__class::semant()
{
    // 检查类型是否存在
    if (type_name == SELF_TYPE) {
        set_type(SELF_TYPE);
        return SELF_TYPE;
    }

    Class_ c = classtable->lookup_class(type_name);
    if (!c) {
        classtable->semant_error() << "new is used with undefined class "
            << type_name << "." << endl;
        set_type(Object);
        return Object;
    }

    set_type(type_name);
    return type_name;
}

// isvoid表达式
Symbol isvoid_class::semant()
{
    e1->semant();
    set_type(Bool);
    return Bool;
}

// no_expr表达式
Symbol no_expr_class::semant()
{
    set_type(No_type);
    return No_type;
}

// 算术表达式
Symbol plus_class::semant()
{
    Symbol e1_type = e1->semant();
    Symbol e2_type = e2->semant();

    // 检查两个操作数是否都是Int类型
    if (e1_type != Int || e2_type != Int) {
        classtable->semant_error() << "non-Int arguments: "
            << e1_type << " + " << e2_type << endl;
        set_type(Object);
        return Object;
    }

```

```
    set_type(Int);
    return Int;
}

Symbol sub_class::semant()
{
    Symbol e1_type = e1->semant();
    Symbol e2_type = e2->semant();

    // 检查两个操作数是否都是Int类型
    if (e1_type != Int || e2_type != Int) {
        classtable->semant_error() << "non-Int arguments: "
            << e1_type << " - " << e2_type << endl;
        set_type(Object);
        return Object;
    }

    set_type(Int);
    return Int;
}

Symbol mul_class::semant()
{
    Symbol e1_type = e1->semant();
    Symbol e2_type = e2->semant();

    // 检查两个操作数是否都是Int类型
    if (e1_type != Int || e2_type != Int) {
        classtable->semant_error() << "non-Int arguments: "
            << e1_type << " * " << e2_type << endl;
        set_type(Object);
        return Object;
    }

    set_type(Int);
    return Int;
}

Symbol divide_class::semant()
{
    Symbol e1_type = e1->semant();
    Symbol e2_type = e2->semant();

    // 检查两个操作数是否都是Int类型
    if (e1_type != Int || e2_type != Int) {
        classtable->semant_error() << "non-Int arguments: "
            << e1_type << " / " << e2_type << endl;
        set_type(Object);
        return Object;
    }

    set_type(Int);
    return Int;
}

// 取负表达式
Symbol neg_class::semant()
{
    Symbol e1_type = e1->semant();

    // 检查操作数是否是Int类型
    if (e1_type != Int) {
        classtable->semant_error() << "Argument of '~' has type "
            << e1_type << " instead of Int." << endl;
        set_type(Object);
        return Object;
    }

    set_type(Int);
    return Int;
}

// 比较表达式
Symbol lt_class::semant()
{
    Symbol e1_type = e1->semant();
    Symbol e2_type = e2->semant();

    // 检查两个操作数是否都是Int类型
    if (e1_type != Int || e2_type != Int) {
        classtable->semant_error() << "non-Int arguments: "
            << e1_type << " < " << e2_type << endl;
        set_type(Object);
        return Object;
    }

    set_type(Bool);
    return Bool;
}

Symbol eq_class::semant()
{
    Symbol e1_type = e1->semant();
    Symbol e2_type = e2->semant();

    // 检查两个操作数是否是基本类型
    if ((e1_type == Int || e1_type == Str || e1_type == Bool) &&
        (e2_type == Int || e2_type == Str || e2_type == Bool) &&
        e1_type != e2_type) {
        classtable->semant_error() << "Illegal comparison with a basic type."
        << endl;
    }

    set_type(Bool);
    return Bool;
}

Symbol leq_class::semant()
{
    Symbol e1_type = e1->semant();
    Symbol e2_type = e2->semant();

    // 检查两个操作数是否都是Int类型
    if (e1_type != Int || e2_type != Int) {
        classtable->semant_error() << "non-Int arguments: "
            << e1_type << " <= " << e2_type << endl;
        set_type(Object);
        return Object;
    }

    set_type(Bool);
    return Bool;
}

// 逻辑表达式
Symbol comp_class::semant()
{
    Symbol e1_type = e1->semant();

    // 检查操作数是否是Bool类型
    if (e1_type != Bool) {
        classtable->semant_error() << "Argument of 'not' has type "
            << e1_type << " instead of Bool." << endl;
        set_type(Object);
        return Object;
    }

    set_type(Bool);
    return Bool;
}

// 控制流表达式

// if表达式
Symbol cond_class::semant()
{
    Symbol pred_type = pred->semant();

    // 检查条件表达式是否是Bool类型
    if (pred_type != Bool) {
        classtable->semant_error() << "Predicate of 'if' does not have type
       Bool." << endl;
    }

    Symbol then_type = then_exp->semant();
    Symbol else_type = else_exp->semant();

    // if表达式的类型是then分支和else分支的最小公共祖先
    Symbol result_type = join(then_type, else_type);
    set_type(result_type);
    return result_type;
}
```

```

// loop表达式
Symbol loop_class::semant()
{
    Symbol pred_type = pred->semant();

    // 检查条件表达式是否是Bool类型
    if (pred_type != Bool) {
        classtable->semant_error() << "Loop condition does not have type Bool." << endl;
    }

    body->semant();

    // loop表达式的类型是Object
    set_type(Object);
    return Object;
}

// block表达式
Symbol block_class::semant()
{
    Symbol last_type = Object;  // 默认返回Object类型

    // 依次检查每个表达式
    for (int i = body->first(); body->more(i); i = body->next(i)) {
        Expression expr = body->nth(i);
        last_type = expr->semant();
    }

    set_type(last_type);
    return last_type;
}

// let表达式
Symbol let_class::semant()
{
    // 保存当前对象符号表
    object_table.enterscope();

    // 检查初始化表达式
    Symbol init_type = No_type;
    if (init) {
        init_type = init->semant();
        // 检查初始化表达式类型是否与变量类型兼容
        if (!conforms_to(init_type, type_decl)) {
            classtable->semant_error() << "Inferred type " << init_type
                << " of initialization of " << identifier
                << " does not conform to declared type " << type_decl << "."
      << endl;
        }
    }

    // 将变量添加到对象符号表
    object_table.addid(identifier, new SymbolEntry(type_decl));

    // 检查body表达式
    Symbol body_type = body->semant();

    // 恢复对象符号表
    object_table.exitscope();

    set_type(body_type);
    return body_type;
}

// case表达式
Symbol typcase_class::semant()
{
    // 检查case表达式的类型
    Symbol expr_type = expr->semant();

    // 保存当前对象符号表
    object_table.enterscope();

    Symbol result_type = No_type;
    std::set<Symbol> case_types;  // 用于检查重复的case类型

    // 依次检查每个分支
    for (int i = cases->first(); cases->more(i); i = cases->next(i)) {
        Case branch = cases->nth(i);

        // 获取分支的类型和变量名
        Symbol branch_type = get_branch_type(branch);
        Symbol branch_name = get_branch_name(branch);
        Expression branch_expr = get_branch_expr(branch);

        // 检查case类型是否重复
        if (case_types.find(branch_type) != case_types.end()) {
            classtable->semant_error() << "Duplicate branch " << branch_type
      << " in case statement." << endl;
        } else {
            case_types.insert(branch_type);
        }

        // 将变量添加到对象符号表
        object_table.enterscope();
        object_table.addid(branch_name, new SymbolEntry(branch_type));

        // 检查分支表达式
        Symbol branch_expr_type = branch_expr->semant();

        // 更新结果类型
        if (result_type == No_type) {
            result_type = branch_expr_type;
        } else {
            result_type = join(result_type, branch_expr_type);
        }

        // 恢复对象符号表
        object_table.exitscope();
    }

    // 恢复对象符号表
    object_table.exitscope();

    // 如果没有分支，结果是Object
    if (result_type == No_type) {
        result_type = Object;
    }

    set_type(result_type);
    return result_type;
}

// 方法调用表达式

// 静态分派表达式
Symbol static_dispatch_class::semant()
{
    // 检查表达式类型
    Symbol expr_type = expr->semant();

    // 检查指定的类型是否存在
    if (type_name != SELF_TYPE && !classtable->lookup_class(type_name)) {
        classtable->semant_error() << "Type " << type_name
            << " of static dispatch is undefined." << endl;
        set_type(Object);
        return Object;
    }

    // 检查表达式类型是否是指定类型的子类
    if (expr_type != SELF_TYPE && !conforms_to(expr_type, type_name)) {
        classtable->semant_error() << "Expression type " << expr_type
            << " does not conform to declared static dispatch type " << type_name << "." << endl;
    }

    // 查找方法
    Feature method = lookup_method(type_name, name);
    if (!method) {
        classtable->semant_error() << "Dispatch to undefined method "
            << name << "." << endl;
        set_type(Object);
        return Object;
    }

    // 获取方法的参数类型和返回类型
    Formals formals = get_method_formals(method);
    Symbol return_type = get_method_return_type(method);

    // 检查参数数量
```

```
    if (list_length(formals) != list_length(actual)) {
        classtable->semant_error() << "Method " << name
            << " called with wrong number of arguments." << endl;
        set_type(Object);
        return Object;
    }

    // 检查每个参数的类型
    for (int i = actual->first(); actual->more(i); i = actual->next(i)) {
        Expression arg = actual->nth(i);
        Formal formal = formals->nth(i);

        Symbol arg_type = arg->semant();
        Symbol formal_type = get_formal_type_decl(formal);

        if (!conforms_to(arg_type, formal_type)) {
            classtable->semant_error() << "In call of method " << name
                << ", type " << arg_type << " of parameter " << get_formal_name(formal)
                << " does not conform to declared type " << formal_type << "." << endl;
        }
    }

    // 如果返回类型是SELF_TYPE，则返回表达式的类型
    if (return_type == SELF_TYPE) {
        if (expr_type == SELF_TYPE) {
            set_type(SELF_TYPE);
            return SELF_TYPE;
        } else {
            set_type(expr_type);
            return expr_type;
        }
    } else {
        set_type(return_type);
        return return_type;
    }
}

// 动态分派表达式
// dispatch_class的semant方法实现
Symbol dispatch_class::semant() {
    // 检查表达式类型
    Symbol expr_type = expr->semant();

    // 如果表达式类型是SELF_TYPE，则使用当前类
    Symbol dispatch_type = expr_type;
    if (dispatch_type == SELF_TYPE) {
        dispatch_type = current_class;
    }

    // 查找方法
    Feature method = lookup_method(dispatch_type, name);
    if (!method) {
        classtable->semant_error() << "Dispatch to undefined method "
            << name << "." << endl;
        set_type(Object);
        return Object;
    }

    // 获取方法的参数类型和返回类型
    Formals formals = get_method_formals(method);
    Symbol return_type = get_method_return_type(method);

    // 检查参数数量
    if (list_length(formals) != list_length(actual)) {
        classtable->semant_error() << "Method " << name
            << " called with wrong number of arguments." << endl;
        set_type(Object);
        return Object;
    }

    // 检查每个参数的类型
    for (int i = actual->first(); actual->more(i); i = actual->next(i)) {
        Expression arg = actual->nth(i);
        Formal formal = formals->nth(i);

        Symbol arg_type = arg->semant();
        Symbol formal_type = get_formal_type_decl(formal);

        if (!conforms_to(arg_type, formal_type)) {
            classtable->semant_error() << "In call of method " << name
                << ", type " << arg_type << " of parameter " << get_formal_name(formal)
                << " does not conform to declared type " << formal_type << "." << endl;
        }
    }

    // 如果返回类型是SELF_TYPE，则返回表达式的类型
    if (return_type == SELF_TYPE) {
        if (expr_type == SELF_TYPE) {
            set_type(SELF_TYPE);
            return SELF_TYPE;
        } else {
            set_type(expr_type);
            return expr_type;
        }
    } else {
        set_type(return_type);
        return return_type;
    }
}









ClassTable::ClassTable(Classes classes_) :
    semant_errors(0),
    error_stream(cerr),
    classes(classes_)
{
    // 初始化类符号表
    class_table.enterscope();

    // 安装基本类
    install_basic_classes();

    // 检查用户定义的类
    for (int i = classes->first(); classes->more(i); i = classes->next(i)) {
        Class_ c = classes->nth(i);

        // 检查类名是否重复
        if (class_table.probe(get_class_name(c))) {
            semant_error(c) << "Class " << get_class_name(c) << " was previously defined." << endl;
        } else {
            Class_ *class_ptr = new Class_(c);
            class_table.addid(get_class_name(c), class_ptr);
        }
    }

    // 检查继承关系
    for (int i = classes->first(); classes->more(i); i = classes->next(i)) {
        Class_ c = classes->nth(i);

        // 检查父类是否存在
        if (get_class_parent(c) != No_class && !class_table.lookup(get_class_parent(c))) {
            semant_error(c) << "Class " << get_class_name(c) << " inherits from an undefined class " << get_class_parent(c) << "." << endl;
        }

        // 检查是否继承自Int、Bool或Str
        if (get_class_parent(c) == Int || get_class_parent(c) == Bool || get_class_parent(c) == Str) {
            semant_error(c) << "Class " << get_class_name(c) << " cannot inherit class " << get_class_parent(c) << "." << endl;
        }
    }

    // 检查Main类是否存在
    if (!class_table.lookup(Main)) {
        semant_error() << "Class Main is not defined." << endl;
    } else {
        // 检查Main类中是否有main方法
        Class_ main_class = lookup_class(Main);
        Features features = get_class_features(main_class);
        bool has_main_method = false;
```

```
        for (int i = features->first(); features->more(i); i = features->next(i)) {
            Feature f = features->nth(i);
            method_class* method = dynamic_cast<method_class*>(f);
            if (method && get_method_name(method) == main_meth) {
                has_main_method = true;
                break;
            }
        }

        if (!has_main_method) {
            semant_error(main_class) << "No 'main' method in class Main." << endl;
        }
    }
}

void ClassTable::install_basic_classes() {

    // The tree package uses these globals to annotate the classes built below.
   // curr_lineno  = 0;
    Symbol filename = stringtable.add_string("<basic class>");

    // The following demonstrates how to create dummy parse trees to
    // refer to basic Cool classes.  There's no need for method
    // bodies -- these are already built in to the runtime system.

    // IMPORTANT: The results of the following expressions are
    // stored in local variables.  You will want to do something
    // with those variables at the end of this method to make this
    // code meaningful.

    //
    // The Object class has no parent class. Its methods are
    //        abort() : Object    aborts the program
    //        type_name() : Str   returns a string representation of class name
    //        copy() : SELF_TYPE  returns a copy of the object
    //
    // There is no need for method bodies in the basic classes---these
    // are already built in to the runtime system.

    Object_class =
   class_(Object,
           No_class,
           append_Features(
               append_Features(
                   single_Features(method(cool_abort, nil_Formals(), Object, no_expr())),
                   single_Features(method(type_name, nil_Formals(), Str, no_expr()))),
               single_Features(method(copy, nil_Formals(), SELF_TYPE, no_expr()))),
           filename);

    //
    // The IO class inherits from Object. Its methods are
    //        out_string(Str) : SELF_TYPE       writes a string to the output
    //        out_int(Int) : SELF_TYPE            "    an int    " "     "
    //        in_string() : Str                 reads a string from the input
    //        in_int() : Int                      "   an int     " "     "
    //
    IO_class =
   class_(IO,
           Object,
           append_Features(
               append_Features(
                   append_Features(
                       single_Features(method(out_string, single_Formals(formal(arg, Str)),
                           SELF_TYPE, no_expr())),
                       single_Features(method(out_int, single_Formals(formal(arg, Int)),
                           SELF_TYPE, no_expr()))),
                   single_Features(method(in_string, nil_Formals(), Str, no_expr()))),
               single_Features(method(in_int, nil_Formals(), Int, no_expr()))),
           filename);

    //
    // The Int class has no methods and only a single attribute, the
    // "val" for the integer.
    //
    Int_class =
   class_(Int,
           Object,
           single_Features(attr(val, prim_slot, no_expr())),
           filename);

    //
    // Bool also has only the "val" slot.
    //
    Bool_class =
   class_(Bool, Object, single_Features(attr(val, prim_slot, no_expr())),
           filename);

    //
    // The class Str has a number of slots and operations:
    //       val                                  the length of the string
    //       str_field                            the string itself
    //       length() : Int                       returns length of the string
    //       concat(arg: Str) : Str               self + arg
    //       substr(arg: Int, arg2: Int) : Str    substring from arg to arg2
    //
    Str_class =
   class_(Str,
           Object,
           append_Features(
               append_Features(
                   append_Features(
                       single_Features(attr(val, Int, no_expr())),
                       single_Features(attr(str_field, prim_slot, no_expr()))),
                   single_Features(method(length, nil_Formals(), Int, no_expr()))),
               append_Features(
                   single_Features(method(concat,
                        single_Formals(formal(arg, Str)),
                        Str, no_expr())),
                   single_Features(method(substr,
                    append_Formals(single_Formals(formal(arg, Int)),
                     single_Formals(formal(arg2, Int))),
                    Str, no_expr())))),
           filename);

    // 添加基本类到符号表
    class_table.addid(Object, &Object_class);
    class_table.addid(IO, &IO_class);
    class_table.addid(Int, &Int_class);
    class_table.addid(Bool, &Bool_class);
    class_table.addid(Str, &Str_class);
}

////////////////////////////////////////////////////////////////////
//
// semant_error is an overloaded function for reporting errors
// during semantic analysis.  There are three versions:
//
//    ostream& ClassTable::semant_error()
//    ostream& ClassTable::semant_error(Class_ c)
//       ostream& ClassTable::semant_error(Symbol filename, tree_node *t)
//
////////////////////////////////////////////////////////////////////

ostream& ClassTable::semant_error(Class_ c)
{
    return semant_error(c->get_filename(), c);
}

ostream& ClassTable::semant_error(Symbol filename, tree_node *t)
{
    error_stream << filename << ":" << t->get_line_number() << ": ";
    return semant_error();
}

ostream& ClassTable::semant_error()
{
    semant_errors++;
    return error_stream;
}
```

```
// 获取基本类
Class_ ClassTable::get_Object() { return Object_class; }
Class_ ClassTable::get_IO() { return IO_class; }
Class_ ClassTable::get_Int() { return Int_class; }
Class_ ClassTable::get_Bool() { return Bool_class; }
Class_ ClassTable::get_Str() { return Str_class; }

// 查找类
Class_ ClassTable::lookup_class(Symbol name) {
    Class_ *class_ptr = class_table.lookup(name);
    if (class_ptr) {
        return *class_ptr;
    }
    return nullptr;
}

/*  This is the entry point to the semantic checker.

    Your checker should do the following two things:

    1) Check that the program is semantically correct
    2) Decorate the abstract syntax tree with type information
       by setting the `type' field in each Expression node.

    You are free to first do 1), make sure you catch all semantic
    errors. Part 2) can be written in a check second pass using
    the reference to the information you set in 1).  */

// 检查是否是子类关系
bool ClassTable::is_subclass(Symbol child, Symbol parent) {
    // 如果child和parent相同，则是子类
    if (child == parent) {
        return true;
    }

    // 如果parent是Object，任何类都是Object的子类
    if (parent == Object) {
        return true;
    }

    // 获取child类
    Class_ child_class = lookup_class(child);
    if (!child_class) {
        return false;
    }

    // 获取child的父类
    Symbol child_parent = get_class_parent(child_class);

    // 如果child没有父类或者是Object，则不是parent的子类
    if (child_parent == No_class || child_parent == Object) {
        return false;
    }

    // 递归检查child的父类是否是parent的子类
    return is_subclass(child_parent, parent);
}

// 获取两个类的最小公共祖先
Symbol ClassTable::least_common_ancestor(Symbol type1, Symbol type2) {
    // 如果其中一个类型是No_type，返回另一个类型
    if (type1 == No_type) return type2;
    if (type2 == No_type) return type1;

    // 如果类型相同，返回该类型
    if (type1 == type2) return type1;

    // 如果type2是Object，返回Object
    if (type2 == Object) return Object;

    // 检查type1是否是type2的子类
    if (is_subclass(type1, type2)) {
        return type2;
    }

    // 检查type2是否是type1的子类
    if (is_subclass(type2, type1)) {
        return type1;
    }

    // 递归查找type1的父类与type2的最小公共祖先
    Class_ type1_class = lookup_class(type1);
    if (type1_class) {
        Symbol type1_parent = get_class_parent(type1_class);
        if (type1_parent != No_class) {
            return least_common_ancestor(type1_parent, type2);
        }
    }

    // 默认返回Object
    return Object;
}

void ClassTable::check_class_features(Class_ c)
{
    // 检查方法重写
    check_method_override(c);

    // 检查属性继承
    check_attribute_inheritance(c);

    // 检查类中的特性
    Features features = get_class_features(c);
    for (int i = features->first(); features->more(i); i = features->next(i))
        {
        Feature f = features->nth(i);

        // 检查是否是方法
        method_class* method = dynamic_cast<method_class*>(f);
        if (method) {
            // 设置当前类和方法
            current_class = get_class_name(c);
            current_method = get_method_name(method);

            // 创建新的作用域
            if (semant_debug) {
                cerr << "Entering scope for method " << current_method <<
    endl;
            }
            object_table.enterscope();

            // 添加self到作用域
            if (semant_debug) {
                cerr << "Adding self to scope in method " << current_method
    << endl;
            }
            object_table.addid(self, new SymbolEntry(SELF_TYPE));

            // 添加参数到作用域
            Formals formals = get_method_formals(method);
            for (int j = formals->first(); formals->more(j); j = formals->
    next(j)) {
                Formal formal = formals->nth(j);
                Symbol formal_name = get_formal_name(formal);
                Symbol formal_type = get_formal_type_decl(formal);
                object_table.addid(formal_name, new SymbolEntry(formal_type))
    ;
            }

            // 检查方法返回类型是否存在
            if (get_method_return_type(method) != SELF_TYPE) {
                Class_ return_class = lookup_class(get_method_return_type(
    method));
                if (!return_class) {
                    semant_error(c) << "Undefined return type " << get_method
    _return_type(method)
                        << " in method " << get_method_name(method) << "." <<
     endl;
                }
            }

            // 检查方法参数类型是否存在
            for (int j = formals->first(); formals->more(j); j = formals->
    next(j)) {
                Formal formal = formals->nth(j);
                if (get_formal_type_decl(formal) != SELF_TYPE) {
                    Class_ param_class = lookup_class(get_formal_type_decl(
    formal));
                    if (!param_class) {
                        semant_error(c) << "Undefined parameter type " << get
    _formal_type_decl(formal)
```

```
                        << " in method " << get_method_name(method) << "." << endl;
                }
            }
        }

        // 检查方法体的类型
        Expression method_body = get_method_expr(method);
        Symbol body_type = method_body->semant();
        Symbol return_type = get_method_return_type(method);

        // 检查方法体的类型是否与返回类型兼容
        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;
        }

        // 退出作用域
        if (semant_debug) {
            cerr << "Exiting scope for method " << current_method << endl;
        }
        object_table.exitscope();
    }

    // 检查是否是属性
    attr_class* attr = dynamic_cast<attr_class*>(f);
    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;
            }
        }

        // 检查属性初始化表达式
        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;
            }
        }
    }
}

void ClassTable::check_method_override(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
    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
    Features child_features = get_class_features(c);
    Features parent_features = get_class_features(parent_class);

    // 遍历当前类的方法
    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
```

```
    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特性
    Features child_features = get_class_features(c);
    Features parent_features = get_class_features(parent_class);

    // 遍历当前类的属性
    for (int i = child_features->first(); child_features->more(i); i = child_features->next(i)) {
        Feature child_feature = child_features->nth(i);

        // 检查是否是属性
        attr_class* child_attr = dynamic_cast<attr_class*>(child_feature);
        if (!child_attr) {
            continue; // 不是属性，跳过
        }

        // 在父类中查找同名属性
        attr_class* parent_attr = NULL;
        for (int j = parent_features->first(); parent_features->more(j); j = parent_features->next(j)) {
            Feature parent_feature = parent_features->nth(j);
            attr_class* attr = dynamic_cast<attr_class*>(parent_feature);
            if (attr && get_attr_name(attr) == get_attr_name(child_attr)) {
                parent_attr = attr;
                break;
            }
        }

        // 如果父类中有同名属性，报错
        if (parent_attr) {
            semant_error(c) << "Attribute " << get_attr_name(child_attr)
                << " is an attribute of an inherited class." << endl;
        }
    }
}



void initialize_constants(void)
{
    arg         = idtable.add_string("arg");
    arg2        = idtable.add_string("arg2");
    Bool        = idtable.add_string("Bool");
    concat      = idtable.add_string("concat");
    cool_abort  = idtable.add_string("abort");
    copy        = idtable.add_string("copy");
    Int         = idtable.add_string("Int");
    in_int      = idtable.add_string("in_int");
    in_string   = idtable.add_string("in_string");
    IO          = idtable.add_string("IO");
    length      = idtable.add_string("length");
    Main        = idtable.add_string("Main");
    main_meth   = idtable.add_string("main");
    //   _no_class is a symbol that can't be the name of any
    //   user-defined class.
    No_class    = idtable.add_string("_no_class");
    No_type     = idtable.add_string("_no_type");
    Object      = idtable.add_string("Object");
    out_int     = idtable.add_string("out_int");
    out_string  = idtable.add_string("out_string");
    prim_slot   = idtable.add_string("_prim_slot");
    self        = idtable.add_string("self");
    SELF_TYPE   = idtable.add_string("SELF_TYPE");
    Str         = idtable.add_string("String");
    str_field   = idtable.add_string("_str_field");
    substr      = idtable.add_string("substr");
    type_name    = idtable.add_string("type_name");
    val         = idtable.add_string("_val");
}

// 符号定义
Symbol arg;
Symbol arg2;
Symbol Bool;
Symbol concat;
Symbol cool_abort;
Symbol copy;
Symbol Int;
Symbol in_int;
Symbol in_string;
Symbol IO;
Symbol length;
Symbol Main;
Symbol main_meth;
Symbol No_class;
Symbol No_type;
Symbol Object;
Symbol out_int;
Symbol out_string;
Symbol prim_slot;
Symbol self;
Symbol SELF_TYPE;
Symbol Str;
Symbol str_field;
Symbol substr;
Symbol type_name;
Symbol val;



// 获取列表长度


int list_length(Expressions exprs) {
    int count = 0;
    if (exprs) {
        for (int i = exprs->first(); exprs->more(i); i = exprs->next(i)) {
            count++;
        }
    }
    return count;
}

int list_length(Cases cases) {
    int count = 0;
    if (cases) {
        for (int i = cases->first(); cases->more(i); i = cases->next(i)) {
            count++;
        }
    }
    return count;
}
```