Programming Assignment IV

15-李晓畅-凌之寒

1. 静态语义检查实现

1.1 静态语义检查实现简述

在 PA4 handout 中给出了实现静态语义检查的大概流程：

> We suggest that you divide your semantic analysis phase into two smaller components. First, check that the inheritance graph is well-defined, meaning that all the restrictions on inheritance are satisfied. If the inheritance graph is not well-defined, it is acceptable to abort compilation (after printing appropriate error messages, of course!). Second, check all the other semantic conditions. It is much easier to implement this second component if one knows the inheritance graph and that it is legal.

先进行继承检查，在满足继承关系的基础上再进行对具体语义条件的检查。

具体而言包含三个部分：

> **4 Inheritance** 检查类别和继承；
>
> **5 Naming and Scoping** 检查定义域内变量名；
>
> **6 Type Checking** 类型检查。

具体对应到实现上，也可以对应划分为三个层次上的检查：

第一，在类节点上的检查，其核心是基于构造的类表检查非法继承关系；

第二，在类方法节点上的检查，大概起到承接上下两种检查的作用，向上依靠类表及对应方法给出方法的函数签名，向下依靠符号表及相关方法给出表达式的返回值，结合实际所处环境，检查函数声明与函数实现之间是否存在矛盾；

第三，在表达式节点上的检查，主要通过递归调用检查子表达式并确定嵌套表达式的返回值，并以此为依据检查表达式是否符合要求及确定返回类型。

下面将具体描述各部分的实现细节。

1.2 类节点上的检查：类别和继承

1.2.1 数据结构简述

在这以层面上的检查核心是实现类表`28 > class ClassTable {···`，

而类表的核心是我们定义的两个数据结构：

```
std::map<Symbol, Class_> class_map;         // Maps class names to the class pointers
std::map<Symbol, Symbol> inheritance_graph; // Maps child classes to parents

// cool do not support pack, class name is unique
// so, inside, we use Symbol
// only to outside, we map to address
```

Inheritance_graph，继承图，采用双亲表示法而非常见的孩子表示法存储继承树，这是因为在面向对象中孩子类知道父类的存在，但反之则不然。

Class_map，类图，将类名映射到地址。

值得一提的是，我们在内部使用的都是类的名称，这是因为 COOL 不支持包，所以类名是类的唯一标识；这样的设计使得我们在内部检查时更加方便，毕竟输出处理名字还是更简单的，我们只在有必要的时候才将其映射为地址。

1.2.2 检查实现说明

错误处理相关流程已经由框架实现，此处不再赘述。

下面首先具体说明同本阶段检查直接相关的方法：

类加入类表时可以对其进行简单的检查，我们将其实现于`void add_to_class_table(Class_ c);`中。

在用初始化类表时我们将的插入操作都由改接口进行`add_to_class_table(classes->nth(i));`。

```
// no inheritance from final classes
if (parent == Bool || parent == SELF_TYPE || parent == Str) {…
// you inherite yourself?
else if (name == SELF_TYPE) {…
else if (class_map.count(name) == 0 && inheritance_graph.count(name) == 0) {…
// duplicate class definition
else {…
```

这可以检出一些问题，包括继承 final 类，重复定义和继承 SELF_TYPE。

另外，我们在这里将其从具体类转化为名字，这同我们设计的追求和数据结构定义一致。

核心检查工作实现在`bool is_valid();`函数上。

检查操作需要遍历继承树的每一个节点。

```
for (std::map<Symbol, Symbol>::iterator iter = inheritance_graph.begin();
    iter != inheritance_graph.end(); iter++) {
    Symbol child = iter->first;
    Symbol parent = iter->second;
```

因为我们采用的是双亲表示，所以遍历到一个节点就相当于进行对该类继承关系进行检查。正如前面提到的那样，我们使用类名检查就可以了。

因为在加入类表时我们的视野是有限的，只有机会看到已经加入的类，所以还是有一些检查要在构建出完整类表后进行。

此时对一个类还要进行两方面的检查：

其一，检查 Main 类`// check if Main  class is defined, and if main function is defined`
`if (child == Main){…`。虽然检查 main 方法按说应该放在下一步，但既然我们找到 Main 类了在这里做了会简单一些。

其二，检查全局继承信息。

通过沿着继承树向树根前进，我们可以检查出成环的情况：如果出现重复节点，那么路径上的类都为一个环所包含。

```
if (parent == child) {
    // Error - cycle detected
    ostream& err_stream = semant_error(class_map[child]);
    err_stream << "Class " << child << " inherits from itself.\n";
    valid = 0;
    break;
}
```

检查继承的父类是否存在，采用图结构可以很简单的实现查找。

```
else if (inheritance_graph.find(parent) == inheritance_graph.end())
{
    // Error - parent not found
    ostream& err_stream = semant_error(class_map[child]);
    err_stream << "Class " << child << " inherits from undefined class "
               << parent << ".\n";
    // return false;
    valid = 0;
    break;
}
```

最后报告 Main 类和 main 方法的情况，并返回正确性即可。

```
if (is_main_defined == false) {
    ostream& err_stream = semant_error();
    err_stream << "Class Main is not defined.\n";
    return false;
}
else if (has_main_fn == false){
    ostream& err_stream = semant_error();
    err_stream << "No 'main' method in class Main.\n";
    return false;
}
return valid;
```

至此，类上的错已经处理完毕，如有问题我们可以直接停止。

1.2.3 对其他支持方法的说明

构建好的 ClassTable 不仅要为自己服务，还要为其他阶段的检查服务。所以我们需要实现一些支持方法。

包括：

```
Symbol lca(Symbol c1, Symbol c2);
bool is_child(Symbol child, Symbol parent);
bool class_exists(Symbol c);
Class_ get_class(Symbol class_name);                                    // map to address, interface to outside
```

阐述类间关系的方法，在之前的类检查中，因为我们总之是要进行全部类表项目的遍历，没必要调用独特的方法；但对接下来的检查，他们或许只想知道一些具体类的情况。为此我们没必要让他们非得遍历表。这就是这些方法存在的意义。

LCA，即最小公共祖先，主要为 if-else 语句和 case-esac 语句使用，是确定这些表达式返回值类型的核心。顺便一提，因为使用的是双亲表示，这一函数是很高效的。

Is_child，判断是否是孩子，对面向对象的语言来说这是日常操作吧。

Class_exists，如名字所示，inherance_gragh 是私有的，提供检查接口。

Get_class，毕竟我们的 class_map 是私有的，需要一个同外界接口。

都是通用的树算法或简单返回一个值，此处不展开。

```
Formals get_formals(Symbol class_name, Symbol method_name);
Symbol get_return_type(Symbol class_name, Symbol method_name);
Symbol get_ancestor_method_class(Symbol class_name, Symbol method_name);    // find the class that defines the method
bool check_method_signature(Symbol c1, Symbol c2, Symbol method_name);
```

方法节点联动的方法。

可以看成是同下一阶段检查交互的接口吧，其功能就像名字阐述的那样。

这些方法总之就是做一件事，向上检查继承图，在真正实现了对应方法的类节点处停止，对应给出要求的输出信息即可。这些信息的给出又要求 method 类的支持，之后会具体说明这些方面。

比如可以看：

```
Symbol ClassTable::get_return_type(Symbol class_name, Symbol method_name) {
    Symbol cname = class_name;
    while (cname != No_class) {
        Class_ c = class_map[cname];
        Symbol r = c->get_return_type(method_name);
        if (r != NULL)
            return r;
        cname = inheritance_graph[cname];
    }
    return NULL;
}
```

就是向上爬。。。。。。。。

```
void display();
bool has_main(Symbol m);
```

他们是为 debug 准备的方法。 \ T_T \

1.2.4 类检查

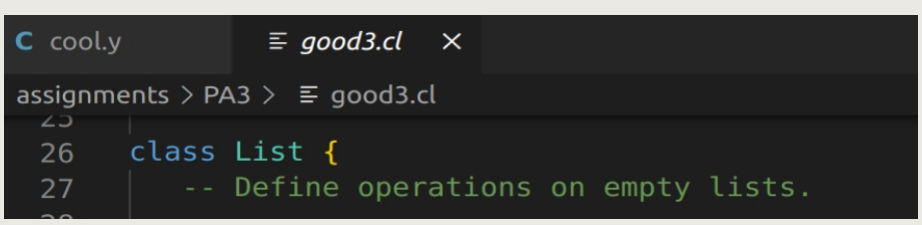
其实实现到这里他已经可以检查类的情况了，可以看：

```
void program_class::semant()
{
    initialize_constants();

    /* Initialize a new ClassTable inheritance graph and make sure it
       is well-formed. */
    ClassTable *classtable = new ClassTable(classes);
    // classtable->display();
    if (classtable->is_valid() && !classtable->errors() ) {…

    if (classtable->errors()) {
    cerr << "Compilation halted due to static semantic errors." << endl;
    exit(1);
    }
}
```

类检查对再继续 qwq。

2.1 good.cl 构造

其实我也不知道什么是好，所以在这一样例的构造上我们实际上主要参考了官方结构给出的一些代码，比如这段：



官方用于解释如何构造 list 的代码就是我们的一个测试。另一些事例代码同样在我们的测试程序范围内。

good.cl
good0.cl
good1.cl
good2.cl
good3.cl
goodorbad.cl

正确通过。

```
passed
wojtek@wojtek-virtual-machine:~/user/Compile-Lab/assignments/PA3$
```

这大概可以说明我们可以处理几乎所以常见的表达了。

我们就不为 good.cl 单独附加代码了，因为绝大多数您都可以在示例中找到。

除了这些外，我们还为示例代码未涉及的一些不太常用的语法构造了样例，比如：

cool.y | good1.cl

assignments > PA3 > good1.cl

```
1   class A {
2   a : Int;
3   b : Int <- if a = 1 then 2 else whatsoever@SOMECLASS.dontcare() fi;
4   good() : Int { {while (a < 0) loop {a = (a) + 1; b = b - 1;} pool;} };
5   };
```

同样可以通过。

```
wojtek@wojtek-virtual-machine:~/user/Compile-Lab/assignments/PA3$ ./checker good2.cl
passed
```

2.1 bad.cl 构造

样例的 bad.cl 包含关于类本身很多问题的检查，在此不做赘述。与之对应，我们将为其他部分进行检查，包括：

比如对于 expression,我们构造了很多样例包括：

```
class Bad {
b : int;
};
```

匹配不上；

```
--An empty block is the worst thing I have ever seen.

class Bad {
b : Int <- if a = 1 then 2 else {} fi;
};
```

```
1   --Notgood is good
2
3   class Bad {
4   b : Int <- if a = 1 then 2 else a@notgood.dontcare() fi;
5   };
```

```
1   --An empty block is the worst thing I have ever seen.
2
3   class Bad {
4   b : Int <- if a = 1 then 2 else {} fi;
5   };
```

```
1   --Why are you doing this? {I mean the semicolomn;}
2
3   class Bad {
4   b : Int <- if a = 1 then 2 else {c = 99999} fi;
5   };
```

等等。其他具体的错误情况可参考附录 2。

客观说，这些错误我自己写的时候也会犯 qwq。

1.3 方法（和属性）节点上的检查：一致性

1.3.1 从类到方法

下面我们考虑如何实现对具体语义的检查。

对具体语义的检查的核心是一个记录环境的结构：

```
// Environment struct <O, M, C> used in type checking
struct type_env_t {
    // Object map O<id name, id type>
    // or say : map a Symbol and type to it's pointer
    SymbolTable<Symbol, Symbol> *om;
    // Contains the class map and inheritance graph
    // These are used for class and method inheritance
    // See semant.h for more details
    ClassTable *ct;
    // Current class C
    Class_ curr;
};
```

包括：一个符号表，我们采用的是框架提供的，当然 qwq。

As discussed in class, a *symbol table* is a convenient data structure for managing names and scoping. You may use our implementation of symbol tables for your project. Our implementation provides methods for entering, exiting, and augmenting scopes as needed. You are also free to implement your own symbol table, of course.

一个类表，即之前我们实现的部分，我们稍后不难看到为什么结构正确的类表是本阶段检查的重要条件。

和一个类的指针。虽然似乎不是一个程序环境的一部分，但我们把它放进去处理上上要更方便，而且确实需要检查现在什么类中。

运用这一结构即可实现对具体语义的检查：

```
type_env_t env;
env.om = new SymbolTable<Symbol, Symbol>();
env.curr = NULL;
env.ct = classtable;

/* Recurisvely type check each class. */
for (int i = classes->first(); classes->more(i); i = classes->next(i)) {
    env.om->enterscope();
    env.curr = classes->nth(i);
    classes->nth(i)->init_class(env); // So the attributes are global
                                      // in the class environment/scope
    classes->nth(i)->type_check(env);
    env.om->exitscope();
}
```

在保证类关系正确之后，我们创建环境，然后以类为单位施加检查。

COOL 的简化是：所有属性私有，所有方法公有，初始化时添加属性到所在域。

```
/*
 * Initializes environment tables with class attributes.
 * Adds class attributes along the inheritance change,
 * adding parent class attributes first.
 */
void class__class::init_class(type_env_t env) {
    if (name != Object) {
        env.ct->get_class(parent)->init_class(env);
    }
    for (int i = features->first(); features->more(i); i = features->next(i)) {
        features->nth(i)->add_to_environment(env);
    }
}
```

或许存在继承，所以初始化要递归进行。

为所有属性和方法添加到环境，对属性而言需要为将其添加到符号表，表示局部可见，而方法则不需要。

之后即可进行检查：

```
Class_ class__class::type_check(type_env_t env) {
    for (int i = features->first(); features->more(i); i = features->next(i)) {
        features->nth(i)->type_check(env);
    }
    return this;
}
```

之后的检查处理就是 feature 类的工作了。

1.3.2 Feature 负责的检查

对 Feature 的检查可以分为两类：

其一，对属性的检查。这一检查稍微简单一些，初始化检查基本类似检查赋值，区别在于需要额外添加对 SELF_TYPE 的绑定。

```
Feature attr_class::type_check(type_env_t env) {
    env.om->enterscope();
    Symbol curr_class = env.curr->get_name();
    // always have self
    env.om->addid(self, &curr_class);
    Symbol t1 = init->type_check(env)->type;
    env.om->exitscope();
    if (t1 == SELF_TYPE) t1 = env.curr->get_name();
    if (t1 != No_type) {…
    if (name == self) {…
    return this;
}
```

Self 是为数不多的几个关键词，不可以作为符号，我们总是将其加入符号表。如果有 SELF_TYPE，我们将确定它的具体类型。

通过调用在表达式节点上的检查，我们可以检查初始值表达式的合法性，顺便还能得到表达式返回值的类型，在此条件下问题就变成了检查赋值合法性的问题了。调用前面描述过的 is_child 方法可以完成这一检查。

其二，对方法的检查。同属性类似，我们需要添加 SELF，并确定 SELF_TPYE 的类型。

```
Feature method_class::type_check(type_env_t env) {
    env.om->enterscope();
    Symbol curr_class = env.curr->get_name();
    // always have self in scope, pointing to the current class
    env.om->addid(self, &curr_class);
    // check fromals
    for (int i = formals->first(); formals->more(i); i = formals->next(i)) {…
    Symbol tret = expr->type_check(env)->type;

    Symbol ancestor = NULL;
    // check overriden methods
    if ((ancestor = env.ct->get_ancestor_method_class(curr_class, name)) != NULL) {…
    // check return type
    // really SELF_TYPE ?
    if (return_type == SELF_TYPE) {…
    // return type not defined
    else if (!env.ct->class_exists(return_type)) {…
    else {
        if (tret == SELF_TYPE)
            tret = env.curr->get_name();        // bind the actual return type
        if (!env.ct->is_child(tret, return_type)) {…
    }
    env.om->exitscope();
    return this;
}
```

除此之外，我们需要检查参数是否合法，只要保证参数名称不重复且合法、参数类型合法即可，合法的形参会添加到当前函数的符号表中。

为检查实际情况同声明是否一致，我们需要获取实际返回值类型。于是我们调用对表达式的检查，这在检查的同时也为我们给出了实际返回值的类型。

对于重载方法，要保证名字一致。

其他方法要保证返回值类型存在，且同声明一致，即为子类。

对方法和属性检查依赖于对表达式的检查和返回值的确定。下面我们将说明该部分。

1.4 表达式节点上的检查：基于具体语句和文法

本部分直接接收上一部分的调用，对对应的表达式节点施加检查，并返回它自己。

考察接口：

```
class Expression_class : public tree_node {
public:
   tree_node *copy()     { return copy_Expression(); }
   virtual Expression copy_Expression() = 0;
   virtual Expression type_check(type_env_t env) = 0;

#ifdef Expression_EXTRAS
   Expression_EXTRAS
#endif
};
```

下面具体以 let 表达式的检查加以说明：

```
class let_class : public Expression_class {
protected:
   Symbol identifier;
   Symbol type_decl;
   Expression init;
   Expression body;
public:
   let_class(Symbol a1, Symbol a2, Expression a3, Expression a4) {
      identifier = a1;
      type_decl = a2;
      init = a3;
      body = a4;
   }
   Expression copy_Expression();
   void dump(ostream& stream, int n);
   Expression type_check(type_env_t env);
```

Let 语句是一个四元组，包含变量声明、变量类型、初始化条件和主体表达式。这对应其变量和初始化方法。

```
Expression let_class::copy_Expression()
{
   return new let_class(copy_Symbol(identifier), copy_Symbol(type_decl), init->copy_Expression(), body->copy_Expression());
}
```

Dump 用于输出标记后的 AST，用于后续代码生成。

```
void let_class::dump(ostream& stream, int n)
{
   stream << pad(n) << "let\n";
   dump_Symbol(stream, n+2, identifier);
   dump_Symbol(stream, n+2, type_decl);
   init->dump(stream, n+2);
   body->dump(stream, n+2);
}
```

按照要求形式在流中添加输出即可。

Type_check 是本部分的核心内容。

首先，let 不可以定义 SELF_TYPE 类型变量。

```
Expression let_class::type_check(type_env_t env) {
    if (identifier == self) {
        ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
        err_stream << "'self' cannot be bound in a 'let' expression.\n";
        type = Object;
    }
```

因为初始化条件也是表达式，我们需要对它进行检查。

```
Symbol t0 = type_decl;
Symbol t1 = init->type_check(env)->type;
```

进一步检查分为有无初始化条件两种情况进行。

```
// No init
if (t1 == No_type) {
    env.om->enterscope();
    env.om->addid(identifier, &t0);
    Symbol t2 = body->type_check(env)->type;
    env.om->exitscope();
    type = t2;
}
```

对无初始化条件的，我们要创建 scope，为 scop 添加 let 声明的变量，并以此 scope 为基础，对其中的表达式进行检查。返回值类型也将由此检查确定。

对带初始化条件的情况，我们要对赋值的合法性进行检查：

```
// With init
else {
    if (env.ct->is_child(t1, t0)) {
        env.om->enterscope();
        env.om->addid(identifier, &t0);
        Symbol t2 = body->type_check(env)->type;
        env.om->exitscope();
        type = t2;
    }
    else {
        ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
        err_stream << "Expression must evaluate to a child of " << t0 << ".\n";
        type = Object;
    }
}
```

合法情况和上面相同，非法情况则将对应报错。

其他表达式节点的处理同上面类似。

2. 测试结果及说明

对 bad.cl 测试：

```
root@LAPTOP-R2BS8VQP:/home/Compile-Lab/assignments/PA4# ./checker bad.cl
1,2c1,2
< bad.cl:16: Formal parameter declared type Bool is not a subclass of Int.
< bad.cl:17: Number of declared formals doesn't match number checked.
---
> bad.cl:16: In call of method init, type Int of parameter y does not conform to declared type Bool.
> bad.cl:17: Method init called with wrong number of arguments.
```

输出不太一样，但内容正确。

对 good.cl

```
root@LAPTOP-R2BS8VQP:/home/Compile-Lab/assignments/PA4# ./checker good.cl
passed
```

其他构造样例的情况可以看下面说明。

3. 测试样例构造及说明

2.1 good.cl 构造

类似前一次实验中我们做的，我们的 good.cl 包含了一些样例代码。(1,2) 此处不详细说明。

另外，我们也为一些特殊情况构造了样例，比如：

```
--let let let LET

class LLLLLLLET {
  a : Int <- let x : Int, value : Int <- x + 1 in let value : Int <- 1 in x + value;
  getA() : Int {a};
};

Class Main {
  main() : Int {
    (new LLLLLLLET).getA()
  };
};
```

为 let 专门构造的样例。(good3.cl)

和一些其他情况，包括可以的继承和方法（good4.cl）：

```
--Class inherit and method

class A {
  compute(a:Int, b:Int) : Int {a+b};
  foo() : A {new A};
};

class A1 inherits A {
  i : Int <- 1;
  getI() : Int {i};
  foo() : A {new A1};
};

class A2 inherits A {
};

class A11 inherits A1 {
  compute(x:Int, y:Int) : Int {x-y+i};
};
```

```
class A12 inherits A1 {
  b : Bool;
  compute(b:Int, i:Int) : Int {b*i};
};

class A111 inherits A11 {
  j : Int <- 9;
  getJ() : Int {j};
};

class A121 inherits A12 {
  foo() : A {new A122};
};

class A122 inherits A12 {
  compute2(a:Bool) : Bool {a = b};
};

Class Main {
  a : A111;
  main() : Int{
    (new A121).foo().compute(a.getJ(), (new A121).getI())
  };
};
```

均可通过。

```
root@LAPTOP-R2BS8VQP:/home/Compile-Lab/assignments/PA4# ./checker good4.cl
passed
```

2.1 bad.cl 构造

不正确情况包括：

```
class A1 {};
class A1 {};

class B1 inherits B3 {};
class B2 inherits B1 {};
class B3 inherits B2 {};

class C1 {};
class C2 inherits C3 {};
class E  inherits E{};

class D1 inherits Bool {};
```

对类继承错误的检查，包括重复定义、环、继承不存在的类、试图继承 final 类和 Main 类缺乏。(bad1.cl)

与其类似，bad2.cl 包含各类方法错误。

```
--method

class A {
  compute(a:Int, b:Int) : Int {a+b};
  foo() : A {new A};
};

class A1 inherits A {
  i : Int;
  get() : Int {i};
  foo() : A1 {new A1};
};

class A2 inherits A {
};

class A11 inherits A1 {
  compute(x:A1, y:A1) : Int {x.get() + y.get()};
};

class A12 inherits A1 {
  b : Bool;
  compute(a:Int, b:Int): Int {c * i};
};
```

```
class A111 inherits A11 {
  i : Int <- 9;
};

class A121 inherits A12 {
  foo() : A122 {new A122};
};

class A122 inherits A12 {
  compute(a:Bool) : Bool {a = b};
};

Class Main {
  main() : Int{1};
  a() : Bool {(new A).compute(1,0)};
  b() : Int {(new A).compute(1)};
  c() : Int {(new A).compute(1,2,3)};
  d() : Int {(new A).compute(true, nothing)};
};
```

报错信息不一定和官方相同，但都类似或说得通。

```
root@LAPTOP-R2BS8VQP:/home/Compile-Lab/assignments/PA4# ./checker bad1.cl
1c1
< bad1.cl:2: Class A1 has already been defined.
---
> bad1.cl:2: Class A1 was previously defined.
3,8c3
< bad1.cl:4: Class B1 inherits from itself.
< bad1.cl:6: Class B3 inherits from itself.
< bad1.cl:5: Class B2 inherits from itself.
< bad1.cl:9: Class C2 inherits from undefined class C3.
< bad1.cl:10: Class E inherits from itself.
< Class Main is not defined.
---
> bad1.cl:9: Class C2 inherits from an undefined class C3.
```

所有成环我们都输出继承自己。。。。。

4.实验总结

在本次实验中我们实现了 COOL 的静态语义。

语义分析也称为类型检查、上下文相关分析。它负责检查程序（抽象语法树）的上下文。

一般来说，语言的定义包括： 词法，语法，语义和语用。

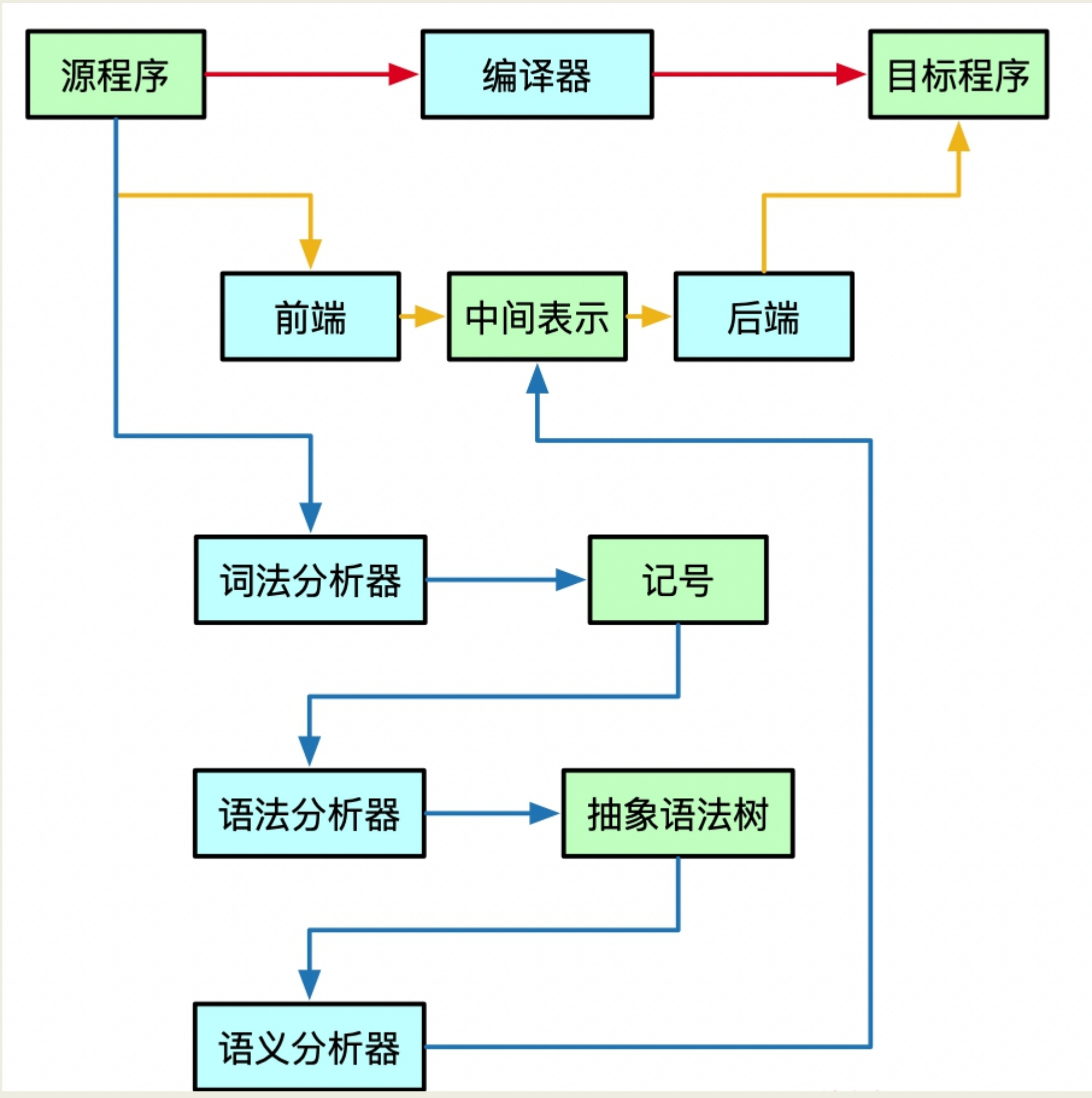
语义是单词，句子和语言的含义，语用说明了语言的用途。编译阶段即可确定的语义称为静态语义，目标代码运行时才能确定的叫动态语义。本次实验实现的就是静态语义检查。

在具体实现中，我们基于任务划分阶段，基于数据结构组织方法，最终实现了复杂的工程。

在具体任务的不同阶段，我们采取了不同的策略。在类检查，我们结合加入时的简单检查和需要全树遍历的检查，实现对继承信息的检查；而在名字和类检查阶段，我们采取递归检查的方法，通过合理设计接口与方法，实现对表达式、方法和属性的检查。

面向对象虽然很方便，但是实现起来会为我们一些麻烦。

在实现了静态语义后，我们的编译任务就进入后端了 qwq/。



附录

1. Semant.h 详细代码

```
#ifndef SEMANT_H_
#define SEMANT_H_

```

```
#include <assert.h>
#include <iostream>
#include "cool-tree.h"
#include "stringtab.h"
#include "symtab.h"
#include "list.h"

#define TRUE 1
#define FALSE 0

class ClassTable;
typedef ClassTable *ClassTableP;

/*
 * This class is used to check class and method inheritance and to print errors
 * encountered in semantic analysis. In type checking, it functions as the method
 * mapping M.
 * It contains a class map <class name, Class_ pointer> and an inheritance graph
 * <class name, parent class name>. The inheritance graph is constructed and
 * verified to ensure that the class declarations are valid (e.g., no cycles
 * in the inheritance tree). During type checking, helper methods use these maps
 * to get method definitions and check method inheritance.
 * See semant.cc for more details.
 */
class ClassTable {
private:
  int semant_errors;
  void install_basic_classes();
  ostream& error_stream;
  std::map<Symbol, Class_> class_map;         // Maps class names to the class pointers
  std::map<Symbol, Symbol> inheritance_graph; // Maps child classes to parents

  // cool do not support pack, class name is unique
  // so, inside, we use Symbol
  // only to outside, we map to address
public:
  ClassTable(Classes);
  int errors() { return semant_errors; }
  ostream& semant_error();
  ostream& semant_error(Class_ c);
  ostream& semant_error(Symbol filename, tree_node *t);

  void add_to_class_table(Class_ c);
  bool is_valid();                                                    // KEY

  Symbol lca(Symbol c1, Symbol c2);
  bool is_child(Symbol child, Symbol parent);
  bool class_exists(Symbol c);
  Class_ get_class(Symbol class_name);                                // map to address, interface to outside

  Formals get_formals(Symbol class_name, Symbol method_name);
  Symbol get_return_type(Symbol class_name, Symbol method_name);
  Symbol get_ancestor_method_class(Symbol class_name, Symbol method_name);    // find the class that defines the method
  bool check_method_signature(Symbol c1, Symbol c2, Symbol method_name);

  void display();
  bool has_main(Symbol m);
};


#endif
```

2. Semant.cc 详细代码

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

```

```
7.
8.  extern int semant_debug;
9.  extern char *curr_filename;
10.
11. //////////////////////////////////////////////////////////////////////
12. //
13. // Symbols
14. //
15. // For convenience, a large number of symbols are predefined here.
16. // These symbols include the primitive type and method names, as well
17. // as fixed names used by the runtime system.
18. //
19. //////////////////////////////////////////////////////////////////////
20. static Symbol
21.     arg,
22.     arg2,
23.     Bool,
24.     concat,
25.     cool_abort,
26.     copy,
27.     Int,
28.     in_int,
29.     in_string,
30.     IO,
31.     length,
32.     Main,
33.     main_meth,
34.     No_class,
35.     No_type,
36.     Object,
37.     out_int,
38.     out_string,
39.     prim_slot,
40.     self,
41.     SELF_TYPE,
42.     Str,
43.     str_field,
44.     substr,
45.     type_name,
46.     val;
47. //
48. // Initializing the predefined symbols.
49. //
50. static void initialize_constants(void)
51. {
52.     arg         = idtable.add_string("arg");
53.     arg2        = idtable.add_string("arg2");
54.     Bool        = idtable.add_string("Bool");
55.     concat      = idtable.add_string("concat");
56.     cool_abort  = idtable.add_string("abort");
57.     copy        = idtable.add_string("copy");
58.     Int         = idtable.add_string("Int");
59.     in_int      = idtable.add_string("in_int");
60.     in_string   = idtable.add_string("in_string");
61.     IO          = idtable.add_string("IO");
62.     length      = idtable.add_string("length");
63.     Main        = idtable.add_string("Main");
64.     main_meth   = idtable.add_string("main");
65.     //   _no_class is a symbol that can't be the name of any
66.     //   user-defined class.
67.     No_class    = idtable.add_string("_no_class");
68.     No_type     = idtable.add_string("_no_type");
69.     Object      = idtable.add_string("Object");
70.     out_int     = idtable.add_string("out_int");
71.     out_string  = idtable.add_string("out_string");
72.     prim_slot   = idtable.add_string("_prim_slot");
73.     self        = idtable.add_string("self");
74.     SELF_TYPE   = idtable.add_string("SELF_TYPE");
75.     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");
}

ClassTable::ClassTable(Classes classes) : semant_errors(0) , error_stream(cerr) {

    /* Construct inheritance graph, a graph of <child, parent> class mappings */
    /* Add basic classes */
    install_basic_classes();

    /* Add other classes */
    for (int i = classes->first(); classes->more(i); i = classes->next(i)) {
        add_to_class_table(classes->nth(i));
    }
}

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>");

      Class_ 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);

      Class_ 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.
      //
      Class_ Int_class =
      class_(Int,
             Object,
             single_Features(attr(val, prim_slot, no_expr())),
             filename);

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


      Class_ Str_class =
      class_(Str,
             Object,
             append_Features(
```

```
                        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()))),
                                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);

    add_to_class_table(Object_class);
    add_to_class_table(IO_class);
    add_to_class_table(Int_class);
    add_to_class_table(Bool_class);
    add_to_class_table(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)
//       print line number and filename for `c'
//
//    ostream& ClassTable::semant_error(Symbol filename, tree_node *t)
//       print a line number and filename
//
////////////////////////////////////////////////////////////////////

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;
}

/*
 * Other ClassTable methods
 */

void ClassTable::add_to_class_table(Class_ c) {
    Symbol name = c->get_name();
    Symbol parent = c->get_parent();
    // no inheritance from final classes
    if (parent == Bool || parent == SELF_TYPE || parent == Str) {
        ostream& err_stream = semant_error(c);
        err_stream << "Class " << name << " cannot inherit class " << parent << ".\n";
    }
    // you inherite yourself?
```

```
214.       else if (name == SELF_TYPE) {
215.           ostream& err_stream = semant_error(c);
216.           err_stream << "Redefinition of basic class " << name << ".\n";
217.       }
218.       else if (class_map.count(name) == 0 && inheritance_graph.count(name) == 0) {
219.           class_map[name] = c;
220.           inheritance_graph[name] = parent;
221.       }
222.       // duplicate class definition
223.       else {
224.           ostream& err_stream = semant_error(c);
225.           err_stream << "Class " << name << " has already been defined.\n";
226.       }
227.   }
228.
229.
230.   bool ClassTable :: has_main(Symbol m){
231.       return get_formals(m, main_meth) != NULL ;
232.   }
233.
234.   bool ClassTable::is_valid() {
235.       bool valid = true;
236.       bool is_main_defined = false;
237.       bool has_main_fn = false;
238.       // cout<<"WE check"<<endl;
239.       for (std::map<Symbol, Symbol>::iterator iter = inheritance_graph.begin();
240.           iter != inheritance_graph.end(); iter++) {
241.           Symbol child = iter->first;
242.           Symbol parent = iter->second;
243.           // cout<<"Checking "<< child->get_string();
244.           // ostream& sr = semant_error(class_map[child]);
245.           // sr<<child->get_string()<<" "<< parent->get_string()<<endl;
246.
247.           // check if Main  class is defined, and if main function is defined
248.           if (child == Main){
249.               is_main_defined = true;
250.               if (has_main(child)) has_main_fn = true;
251.           }
252.           while (parent != No_class) {
253.               if (parent == child) {
254.                   // Error - cycle detected
255.                   ostream& err_stream = semant_error(class_map[child]);
256.                   err_stream << "Class " << child << " inherits from itself.\n";
257.                   valid = 0;
258.                   break;
259.               }
260.               // else if (inheritance_graph.count(parent) == 0)
261.               else if (inheritance_graph.find(parent) == inheritance_graph.end())
262.               {
263.                   // Error - parent not found
264.                   ostream& err_stream = semant_error(class_map[child]);
265.                   err_stream << "Class " << child << " inherits from undefined class "
266.                              << parent << ".\n";
267.                   // return false;
268.                   valid = 0;
269.                   break;
270.               }
271.               else{
272.                   // cout<<"climb:" << child->get_string()<<" -> " << parent->get_string()<<endl;
273.                   parent = inheritance_graph[parent];
274.               }
275.           }
276.       }
277.       // cout<<"Check done ~"<<endl;
278.       if (is_main_defined == false) {
279.           ostream& err_stream = semant_error();
280.           err_stream << "Class Main is not defined.\n";
281.           return false;
282.       }
```

```
283.      else if (has_main_fn == false){
284.          ostream& err_stream = semant_error();
285.          err_stream << "No 'main' method in class Main.\n";
286.          return false;
287.      }
288.      return valid;
289.  }
290.
291.  Symbol ClassTable::lca(Symbol class1, Symbol class2) {
292.      Symbol c1 = class1;
293.      Symbol parent = Object;
294.      while (c1 != Object) {
295.          Symbol c2 = class2;
296.          while (c2 != Object) {
297.              if (c1 == c2) {
298.                  parent = c1;
299.                  goto finish;
300.              }
301.              c2 = inheritance_graph[c2];
302.          }
303.          c1 = inheritance_graph[c1];
304.      }
305.      finish:
306.          return parent;
307.  }
308.
309.  bool ClassTable::is_child(Symbol child, Symbol parent) {
310.      Symbol c = child;
311.      if (parent == Object)
312.          return true;
313.      while (c != Object) {
314.          if (c == parent)
315.              return true;
316.          c = inheritance_graph[c];
317.      }
318.      return false;
319.  }
320.
321.  bool ClassTable::class_exists(Symbol c) {
322.      return (inheritance_graph.count(c) > 0);
323.  }
324.
325.  Class_ ClassTable::get_class(Symbol class_name) {
326.      return class_map[class_name];
327.  }
328.
329.  Formals ClassTable::get_formals(Symbol class_name, Symbol method_name) {
330.      Symbol cname = class_name;
331.      while (cname != No_class) {
332.          Class_ c = class_map[cname];
333.          Formals f = c->get_formals(method_name);
334.          if (f != NULL)
335.              return f;
336.          cname = inheritance_graph[cname];
337.      }
338.      return NULL;
339.  }
340.
341.  Symbol ClassTable::get_return_type(Symbol class_name, Symbol method_name) {
342.      Symbol cname = class_name;
343.      while (cname != No_class) {
344.          Class_ c = class_map[cname];
345.          Symbol r = c->get_return_type(method_name);
346.          if (r != NULL)
347.              return r;
348.          cname = inheritance_graph[cname];
349.      }
350.      return NULL;
351.  }
```

```
352.
353.
354.  Symbol ClassTable::get_ancestor_method_class(Symbol class_name, Symbol method_name) {
355.      Symbol cname = inheritance_graph[class_name];
356.      while (cname != No_class) {
357.          Class_ c = class_map[cname];
358.          if (c->get_return_type(method_name) != NULL)
359.              return c->get_name();
360.          cname = inheritance_graph[cname];
361.      }
362.      return NULL;
363.  }
364.
365.  bool ClassTable::check_method_signature(Symbol c1, Symbol c2, Symbol method_name) {
366.      Class_ class1 = class_map[c1];
367.      Class_ class2 = class_map[c2];
368.      Formals f1 = class1->get_formals(method_name);
369.      Formals f2 = class2->get_formals(method_name);
370.      Symbol ret1 = class1->get_return_type(method_name);
371.      Symbol ret2 = class2->get_return_type(method_name);
372.      int i = f1->first();
373.      int j = f2->first();
374.      // Check formals
375.      while (f1->more(i) && f2->more(j)) {
376.          if (f1->nth(i)->get_type() != f2->nth(j)->get_type())
377.              return false;
378.          i = f1->next(i);
379.          j = f2->next(j);
380.      }
381.      if (f1->more(i) || f2->more(j))
382.          return false;
383.      // Check return type
384.      if (ret1 != ret2)
385.          return false;
386.      return true;
387.  }
388.
389.  void ClassTable::display(){
390.      typedef std::map<Symbol, Symbol> c;
391.      for(c::iterator i = inheritance_graph.begin(); i != inheritance_graph.end(); i++){
392.          cout <<  i->first->get_string() <<" -> "<< i->second->get_string() <<" ";
393.          if(inheritance_graph.find(i->second) != inheritance_graph.end()) cout<<"found";
394.          cout<<endl;
395.      }
396.  }
397.
398.
399.  void program_class::semant()
400.  {
401.      initialize_constants();
402.
403.      /* Initialize a new ClassTable inheritance graph and make sure it
404.         is well-formed. */
405.      ClassTable *classtable = new ClassTable(classes);
406.      // classtable->display();
407.      if (classtable->is_valid() && !classtable->errors() ) {
408.          // cout << "Class Table checked"<<endl;
409.          // create environment
410.          type_env_t env;
411.          env.om = new SymbolTable<Symbol, Symbol>();
412.          env.curr = NULL;
413.          env.ct = classtable;
414.
415.          /* Recurisvely type check each class. */
416.          for (int i = classes->first(); classes->more(i); i = classes->next(i)) {
417.              env.om->enterscope();
418.              env.curr = classes->nth(i);
419.              classes->nth(i)->init_class(env); // So the attributes are global
420.                                                // in the class environment/scope
```

```
421.                 classes->nth(i)->type_check(env);
422.                 env.om->exitscope();
423.             }
424.         }
425.
426.         if (classtable->errors()) {
427.         cerr << "Compilation halted due to static semantic errors." << endl;
428.         exit(1);
429.         }
430.     }
431.
432.     /*
433.      * Other semantic analysis helper methods defined in cool-tree.h.
434.      */
435.
436.     Symbol class__class::get_name() {
437.         return name;
438.     }
439.
440.     Symbol class__class::get_parent() {
441.         return parent;
442.     }
443.
444.     void class__class::init_class(type_env_t env) {
445.         if (name != Object) {
446.             env.ct->get_class(parent)->init_class(env);
447.         }
448.         for (int i = features->first(); features->more(i); i = features->next(i)) {
449.             features->nth(i)->add_to_environment(env);
450.         }
451.     }
452.
453.
454.     Formals class__class::get_formals(Symbol method) {
455.         for (int i = features->first(); features->more(i); i = features->next(i)) {
456.             Feature feature = features->nth(i);
457.             if (feature->is_method() && feature->get_name() == method)
458.                 return feature->get_formals();
459.         }
460.         return NULL;
461.     }
462.
463.
464.     Symbol class__class::get_return_type(Symbol method) {
465.         for (int i = features->first(); features->more(i); i = features->next(i)) {
466.             Feature feature = features->nth(i);
467.             if (feature->is_method() && feature->get_name() == method)
468.                 return feature->get_return_type();
469.         }
470.         return NULL;
471.     }
472.
473.     /*
474.      * Simple accessors for classes defined in cool-tree.h.
475.      */
476.     bool method_class::is_method() { return true; }
477.     bool attr_class::is_method() { return false; }
478.
479.     Formals method_class::get_formals() { return formals; }
480.     Symbol method_class::get_return_type() { return return_type; }
481.
482.     // These two methods should never be called.
483.     Formals attr_class::get_formals() { return NULL; }
484.     Symbol attr_class::get_return_type() { return NULL; }
485.
486.     Symbol method_class::get_name() { return name; };
487.     Symbol attr_class::get_name() { return name; };
488.
489.     // as method, noting to do when adding to environment table
```

```
490.  void method_class::add_to_environment(type_env_t env) { /* Nothing to do */ }
491.  void attr_class::add_to_environment(type_env_t env) {
492.      if (env.om->probe(name) == NULL)
493.          env.om->addid(name, &type_decl);
494.      else {
495.          ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
496.          err_stream << "Unable to add attribute " << name
497.                     << " to object map (already defined).\n";
498.      }
499.  }
500.
501.  Symbol formal_class::get_type() { return type_decl; }
502.  Symbol branch_class::get_type() { return type_decl; }
503.
504.
505.  Class_ class__class::type_check(type_env_t env) {
506.      for (int i = features->first(); features->more(i); i = features->next(i)) {
507.          features->nth(i)->type_check(env);
508.      }
509.      return this;
510.  }
511.
512.  Feature method_class::type_check(type_env_t env) {
513.      env.om->enterscope();
514.      Symbol curr_class = env.curr->get_name();
515.      // always have self in scope, pointing to the current class
516.      env.om->addid(self, &curr_class);
517.      // check fromals
518.      for (int i = formals->first(); formals->more(i); i = formals->next(i)) {
519.          formals->nth(i)->type_check(env);
520.      }
521.      Symbol tret = expr->type_check(env)->type;
522.
523.      Symbol ancestor = NULL;
524.      // check overriden methods
525.      if ((ancestor = env.ct->get_ancestor_method_class(curr_class, name)) != NULL) {
526.          if (!env.ct->check_method_signature(ancestor, curr_class, name)) {
527.              ostream &err_stream = env.ct->semant_error(env.curr->get_filename(), this);
528.              err_stream << "Overriding method signature of " << name << " for class "
529.                         << curr_class << " doesn't match method signature for ancestor "
530.                         << ancestor << ".\n";
531.          }
532.      }
533.      // check return type
534.      // really SELF_TYPE ?
535.      if (return_type == SELF_TYPE) {
536.          if (tret != SELF_TYPE) {
537.              ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
538.              err_stream << "Inferred return type " << tret << " of method " << name
539.                         << " does not conform to declared return type " << return_type << ".\n";
540.          }
541.      }
542.      // return type not defined
543.      else if (!env.ct->class_exists(return_type)) {
544.          ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
545.          err_stream << "Undefined return type " << return_type << " in method " << name << ".\n";
546.      }
547.      else {
548.          if (tret == SELF_TYPE)
549.              tret = env.curr->get_name();        // bind the actual return type
550.          if (!env.ct->is_child(tret, return_type)) {
551.              ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
552.              err_stream << "Method initialization " << tret
553.                         << " is not a subclass of " << return_type << ".\n";
554.          }
555.      }
556.      env.om->exitscope();
557.      return this;
558.  }
```

```
559.
560.  Feature attr_class::type_check(type_env_t env) {
561.      env.om->enterscope();
562.      Symbol curr_class = env.curr->get_name();
563.      // always have self
564.      env.om->addid(self, &curr_class);
565.      Symbol t1 = init->type_check(env)->type;
566.      env.om->exitscope();
567.      if (t1 == SELF_TYPE) t1 = env.curr->get_name();
568.      if (t1 != No_type) {
569.          if (!(env.ct->is_child(t1, type_decl))) {
570.              ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
571.              err_stream << "Attribute initialization " << t1
572.                         << " is not a subclass of " << type_decl << ".\n";
573.          }
574.      }
575.      if (name == self) {
576.          ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
577.          err_stream << "'self' cannot be the name of an attribute.\n";
578.      }
579.      return this;
580.  }
581.
582.  Formal formal_class::type_check(type_env_t env) {
583.      if (env.om->probe(name) != NULL) {
584.          ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
585.          err_stream << "Duplicate formal " << name << ".\n";
586.      }
587.      else if (type_decl == SELF_TYPE) {
588.          ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
589.          err_stream << "Formal parameter " << name << " cannot have type SELF_TYPE\n";
590.      }
591.      else
592.          env.om->addid(name, &type_decl);
593.      return this;
594.  }
595.
596.  Symbol branch_class::type_check(type_env_t env) {
597.      // The following condition should never be true because a branch identifier
598.      // is the first thing initialized in its own scope. This is just a sanity check.
599.      if (env.om->probe(name) != NULL) {
600.          ostream &err_stream = env.ct->semant_error(env.curr->get_filename(), this);
601.          err_stream << "Identifier " << name << " already defined in current scope.\n";
602.          return Object;
603.      }
604.      env.om->addid(name, &type_decl);
605.      return expr->type_check(env)->type;
606.  }
607.
608.  Expression assign_class::type_check(type_env_t env) {
609.      // t1 <- t2
610.      Symbol t1 = *env.om->lookup(name);
611.      Symbol t2 = expr->type_check(env)->type;
612.      // cout<< t1->get_string() << " <- "<< t2->get_string() << endl;
613.      if (t2 == SELF_TYPE) {
614.          ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
615.          err_stream << "Cannot assign to 'self'.\n";
616.          type = Object;
617.      }
618.      else if (env.ct->is_child(t2, t1))
619.          type = t2;
620.      else {
621.          ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
622.          err_stream << t2 << " is not a subclass of " << t1 << ".\n";
623.          type = Object;
624.      }
625.      return this;
626.  }
627.
```

```
Expression static_dispatch_class::type_check(type_env_t env) {
    std::vector<Symbol> eval_types; // Vector of parameter types after evaluation
    Symbol t0 = expr->type_check(env)->type;
    if (t0 == SELF_TYPE)
        t0 = env.curr->get_name();
    if (env.ct->is_child(t0, type_name)) {
        for (int i = actual->first(); actual->more(i); i = actual->next(i)) {
            Symbol tn = actual->nth(i)->type_check(env)->type;
            if (tn == SELF_TYPE)
                tn = env.curr->get_name();
            eval_types.push_back(tn);
        }
        Formals formals = env.ct->get_formals(t0, name);     // Declared formal types
        Symbol ret_type = env.ct->get_return_type(t0, name); // Declared return type
        if (formals == NULL || ret_type == NULL) {
            ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
            err_stream << "Dispatch to undefined method " << name << ".\n";
            type = Object;
            return this;
        }
        // Type check formal parameters
        std::vector<Symbol>::iterator iter = eval_types.begin();
        int fi = formals->first();
        while (iter != eval_types.end() && formals->more(fi)) {
            Symbol eval_type = *iter;
            Symbol declared_type = formals->nth(fi)->get_type();
            if (declared_type == SELF_TYPE) {
                ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
                err_stream << "Formal parameter cannot have type SELF_TYPE.\n";
            }
            else if (!env.ct->is_child(eval_type, declared_type)) {
                ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
                err_stream << "Formal parameter declared type " << declared_type
                           << " is not a subclass of " << eval_type << ".\n";
            }
            ++iter;
            fi = formals->next(fi);
        }
        if (iter != eval_types.end() || formals->more(fi)) {
            // If we're here, means the number of parameters didn't match
            // the expected number from the function definition. This should
            // not be possible as long as lexing and parsing is correct.
            ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
            err_stream << "Number of declared formals doesn't match number checked.\n";
        }

        if (ret_type == SELF_TYPE)
            type = t0;
        else
            type = ret_type;
    }
    else {
        ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
        err_stream << "Evaluated class " << t0 << " must be a child of declared class "
                   << type_name << " in static dispatch.\n";
        type = Object;
    }
    return this;
}

Expression dispatch_class::type_check(type_env_t env) {
    std::vector<Symbol> eval_types; // Vector of parameter types after evaluation
    Symbol t0 = expr->type_check(env)->type;
    Symbol curr = t0;
    if (t0 == SELF_TYPE)
        curr = env.curr->get_name();
    for (int i = actual->first(); actual->more(i); i = actual->next(i)) {
        Symbol tn = actual->nth(i)->type_check(env)->type;
        if (tn == SELF_TYPE)
```

```
697.                tn = env.curr->get_name();
698.            eval_types.push_back(tn);
699.        }
700.        Formals formals = env.ct->get_formals(curr, name);     // Declared formal types
701.        Symbol ret_type = env.ct->get_return_type(curr, name); // Declared return type
702.        if (formals == NULL || ret_type == NULL) {
703.            ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
704.            err_stream << "Dispatch to undefined method " << name << ".\n";
705.            type = Object;
706.            return this;
707.        }
708.
709.        // Type check formal parameters
710.        std::vector<Symbol>::iterator iter = eval_types.begin();
711.        int fi = formals->first();
712.        while (iter != eval_types.end() && formals->more(fi)) {
713.            Symbol eval_type = *iter;
714.            Symbol declared_type = formals->nth(fi)->get_type();
715.            if (declared_type == SELF_TYPE) {
716.                ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
717.                err_stream << "Formal parameter cannot have type SELF_TYPE.\n";
718.            }
719.            else if (!env.ct->is_child(eval_type, declared_type)) {
720.                ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
721.                err_stream << "Formal parameter declared type " << declared_type
722.                           << " is not a subclass of " << eval_type << ".\n";
723.            }
724.            ++iter;
725.            fi = formals->next(fi);
726.        }
727.        if (iter != eval_types.end() || formals->more(fi)) {
728.            // If we're here, means the number of parameters didn't match
729.            // the expected number from the function definition. This should
730.            // not be possible as long as lexing and parsing is correct.
731.            ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
732.            err_stream << "Number of declared formals doesn't match number checked.\n";
733.        }
734.
735.        if (ret_type == SELF_TYPE)
736.            type = t0;
737.        else
738.            type = ret_type;
739.        return this;
740.    }
741.
742.    /*
743.     * Recursively type checks the predicate, then the then expression, then the
744.     * else expression. Makes sure the predicate evaluates to Bool and returns
745.     * the least upper bound of the types of the other two expressions.
746.     */
747.    Expression cond_class::type_check(type_env_t env) {
748.        Symbol t1 = pred->type_check(env)->type;
749.        Symbol t2 = then_exp->type_check(env)->type;
750.        if (t2 == SELF_TYPE)
751.            t2 = env.curr->get_name();
752.        Symbol t3 = else_exp->type_check(env)->type;
753.        if (t3 == SELF_TYPE)
754.            t3 = env.curr->get_name();
755.        if (t1 == Bool)
756.            type = env.ct->lca(t2, t3);
757.        else {
758.            ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
759.            err_stream << "If condition did not evaluate to a boolean.\n";
760.            type = Object;
761.        }
762.        return this;
763.    }
764.
765.    /* Recurisvely type checks the predicate and body, makes sure predicate is a Bool. */
```

```
Expression loop_class::type_check(type_env_t env) {
    Symbol t1 = pred->type_check(env)->type;
    Symbol t2 = body->type_check(env)->type;
    if (t1 == Bool)
        type = Object;
    else {
        ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
        err_stream << "While condition did not evaluate to a boolean.\n";
        type = Object;
    }
    return this;
}

Expression typcase_class::type_check(type_env_t env) {
    Symbol t0 = expr->type_check(env)->type;

    // O(N^2) check to make sure there are no duplicate types.
    for (int i = cases->first(); cases->more(i); i = cases->next(i)) {
        for (int j = cases->first(); cases->more(j); j = cases->next(j)) {
            if (i != j && cases->nth(i)->get_type() == cases->nth(j)->get_type()) {
                ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
                err_stream << "Duplicate branch " << cases->nth(i)->get_type()
                           << " in case statement.\n";
                type = Object;
                return this;
            }
        }
    }

    Symbol tn = cases->nth(cases->first())->type_check(env);
    for (int i = cases->first(); cases->more(i); i = cases->next(i)) {
        if (i != cases->first()) {
            env.om->enterscope();
            tn = env.ct->lca(tn, cases->nth(i)->type_check(env));
            env.om->exitscope();
            // return type of Case will be the LCA of all branches
        }
    }
    type = tn;
    return this;
}

/* Type checks each enclosing expression. Imposes no type conditions. */
Expression block_class::type_check(type_env_t env) {
    Symbol t1;
    for (int i = body->first(); body->more(i); i = body->next(i)) {
        t1 = body->nth(i)->type_check(env)->type;
    }
    type = t1;
    return this;
}


Expression let_class::type_check(type_env_t env) {
    if (identifier == self) {
        ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
        err_stream << "'self' cannot be bound in a 'let' expression.\n";
        type = Object;
    }
    else {
        Symbol t0 = type_decl;
        Symbol t1 = init->type_check(env)->type;
        // No init
        if (t1 == No_type) {
            env.om->enterscope();
            env.om->addid(identifier, &t0);
            Symbol t2 = body->type_check(env)->type;
            env.om->exitscope();
            type = t2;
```

```
            }
            // With init
            else {
                if (env.ct->is_child(t1, t0)) {
                    env.om->enterscope();
                    env.om->addid(identifier, &t0);
                    Symbol t2 = body->type_check(env)->type;
                    env.om->exitscope();
                    type = t2;
                }
                else {
                    ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
                    err_stream << "Expression must evaluate to a child of " << t0 << ".\n";
                    type = Object;
                }
            }
        }
        return this;
    }

    /*
     * The following arithmetic classes are self-explanatory:
     * +, -, *, /, ~, <, =, !
     */
    Expression plus_class::type_check(type_env_t env) {
        Symbol t1 = e1->type_check(env)->type;
        Symbol t2 = e2->type_check(env)->type;
        if (t1 == Int && t2 == Int)
            type = Int;
        else {
            ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
            err_stream << "non-Int arguments " << t1 << " + " << t2 << ".\n";
            type = Object;
        }
        return this;
    }

    Expression sub_class::type_check(type_env_t env) {
        Symbol t1 = e1->type_check(env)->type;
        Symbol t2 = e2->type_check(env)->type;
        if (t1 == Int && t2 == Int)
            type = Int;
        else {
            ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
            err_stream << "non-Int arguments " << t1 << " - " << t2 << ".\n";
            type = Object;
        }
        return this;
    }

    Expression mul_class::type_check(type_env_t env) {
        Symbol t1 = e1->type_check(env)->type;
        Symbol t2 = e2->type_check(env)->type;
        if (t1 == Int && t2 == Int)
            type = Int;
        else {
            ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
            err_stream << "non-Int arguments " << t1 << " * " << t2 << ".\n";
            type = Object;
        }
        return this;
    }

    Expression divide_class::type_check(type_env_t env) {
        Symbol t1 = e1->type_check(env)->type;
        Symbol t2 = e2->type_check(env)->type;
        if (t1 == Int && t2 == Int)
            type = Int;
        else {
```

```
904.            ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
905.            err_stream << "non-Int arguments " << t1 << " / " << t2 << ".\n";
906.            type = Object;
907.        }
908.        return this;
909.    }
910.
911.    Expression neg_class::type_check(type_env_t env) {
912.        Symbol t1 = e1->type_check(env)->type;
913.        if (t1 == Int)
914.            type = Int;
915.        else {
916.            ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
917.            err_stream << "non-Int argument ~" << t1 << ".\n";
918.            type = Object;
919.        }
920.        return this;
921.    }
922.
923.    Expression lt_class::type_check(type_env_t env) {
924.        Symbol t1 = e1->type_check(env)->type;
925.        Symbol t2 = e2->type_check(env)->type;
926.        if (t1 == Int && t2 == Int)
927.            type = Bool;
928.        else {
929.            ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
930.            err_stream << "non-Int arguments " << t1 << " < " << t2 << ".\n";
931.            type = Object;
932.        }
933.        return this;
934.    }
935.
936.    Expression eq_class::type_check(type_env_t env) {
937.        Symbol t1 = e1->type_check(env)->type;
938.        Symbol t2 = e2->type_check(env)->type;
939.        // Any comparison is legal, except: if one argument is in {Int, Str, Bool},
940.        // the other must match.
941.        if ((t1 == Int && t2 != Int) || (t1 != Int && t2 == Int) ||
942.            (t1 == Str && t2 != Str) || (t1 != Str && t2 == Str) ||
943.            (t1 == Bool && t2 != Bool) || (t1 != Bool && t2 == Bool)) {
944.            ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
945.            err_stream << "Cannot compare arguments " << t1 << " = " << t2 << ".\n";
946.            type = Object;
947.        }
948.        else {
949.            type = Bool;
950.        }
951.        return this;
952.    }
953.
954.    Expression leq_class::type_check(type_env_t env) {
955.        Symbol t1 = e1->type_check(env)->type;
956.        Symbol t2 = e2->type_check(env)->type;
957.        if (t1 == Int && t2 == Int)
958.            type = Bool;
959.        else {
960.            ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
961.            err_stream << "non-Int arguments " << t1 << " <= " << t2 << ".\n";
962.            type = Object;
963.        }
964.        return this;
965.    }
966.
967.    Expression comp_class::type_check(type_env_t env) {
968.        Symbol t1 = e1->type_check(env)->type;
969.        if (t1 == Bool)
970.            type = Bool;
971.        else {
972.            ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
```

```
            err_stream << "non-Bool argument !" << t1 << ".\n";
            type = Object;
        }
        return this;
    }

    /* Primitive constants */
    Expression int_const_class::type_check(type_env_t env) {
        type = Int;
        return this;
    }

    Expression bool_const_class::type_check(type_env_t env) {
        type = Bool;
        return this;
    }

    Expression string_const_class::type_check(type_env_t env) {
        type = Str;
        return this;
    }

    Expression new__class::type_check(type_env_t env) {
        if (env.ct->class_exists(type_name) || type_name == SELF_TYPE)
            type = type_name;
        else {
            ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
            err_stream << "'new' used with undefined class " << type_name << ".\n";
            type = Object;
        }
        return this;
    }

    Expression isvoid_class::type_check(type_env_t env) {
        Symbol t1 = e1->type_check(env)->type;
        type = Bool;
        return this;
    }

    Expression no_expr_class::type_check(type_env_t env) {
        type = No_type;
        return this;
    }

    Expression object_class::type_check(type_env_t env) {
        if (name == self)
            type = SELF_TYPE;
        else if (env.om->lookup(name) != NULL)
            type = *(env.om->lookup(name));
        else {
            ostream& err_stream = env.ct->semant_error(env.curr->get_filename(), this);
            err_stream << "Could not find identifier " << name << " in current scope.\n";
            type = Object;
        }
        return this;
    }
```