西安交通大學
XI'AN JIAOTONG UNIVERSITY

2022-2023 学年第二学期

《编译器设计专题实验》

实验报告 7

学　　院：电信学部

班　　级：[redacted]

学　　号：[redacted]

姓　　名：[redacted]

二〇二三年　五月

目录

《实验 7-语义分析(一), 实现 COOL 语义分析》

一、实验内容（必做）

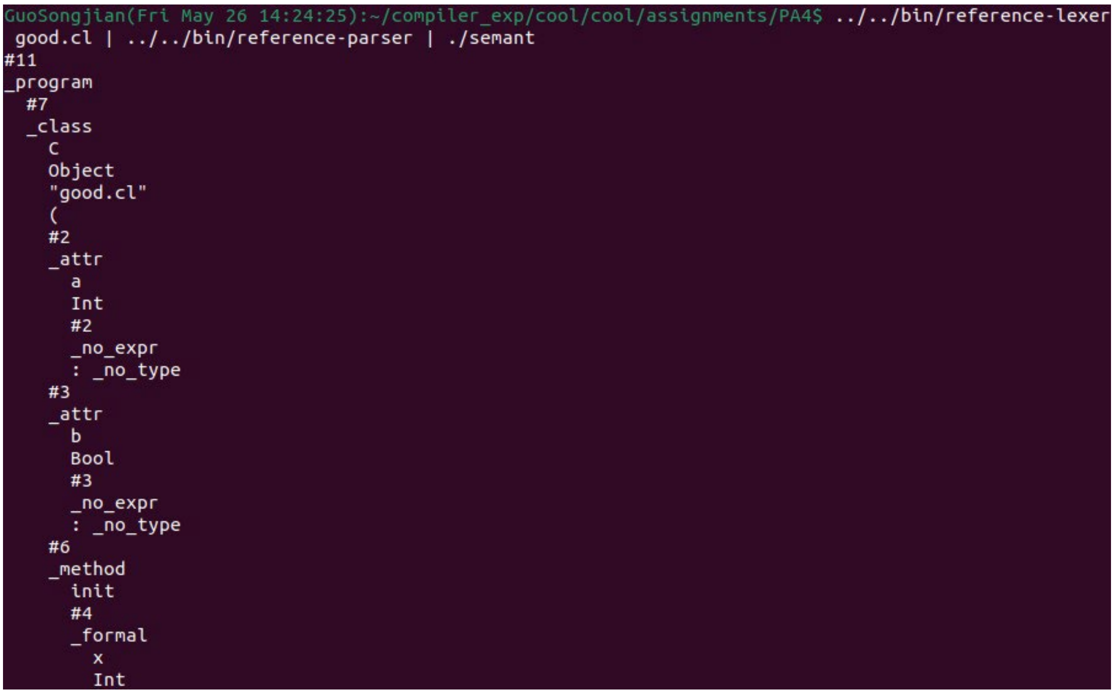
1. 对 class 进行基本检查，如：类名合法、类间循环继承等；
2. 在 good.cl 中编写正确的 COOL 程序，bad.cl 中编写类名以小写字母开头的、存在循环继承、函数重载非法的 COOL 程序，并对两个程序进行语义分析。

二、实验内容（选做）

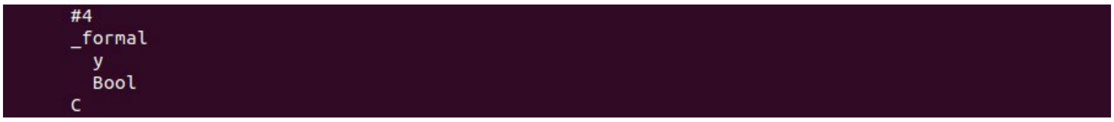
根据时间实现 COOL 语言的 class、feature、method、expr 等的语义检查。

三、实验结果

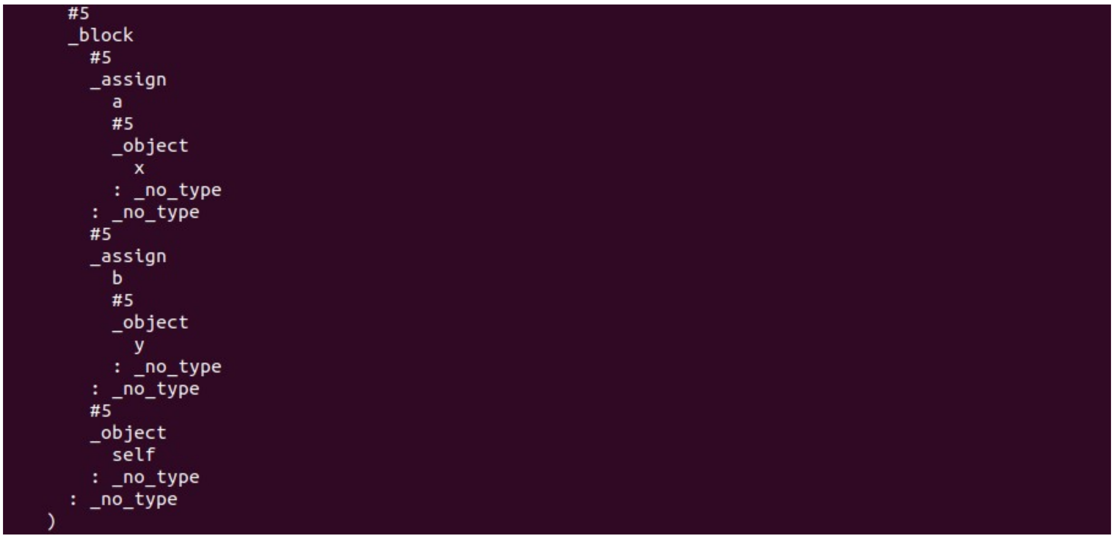
（1）good.cl 语义分析



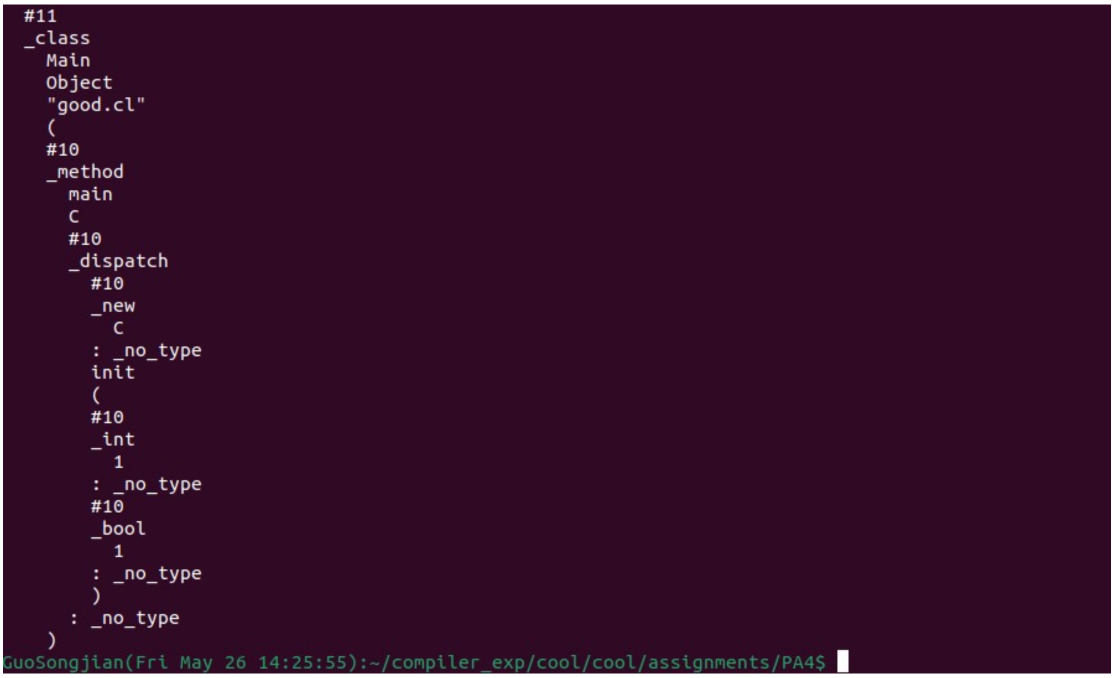
```
GuoSongjian(Fri May 26 14:24:25):~/compiler_exp/cool/cool/assignments/PA4$ ../../bin/reference-lexer
 good.cl | ../../bin/reference-parser | ./semant
#11
_program
  #7
  _class
    C
    Object
    "good.cl"
    (
    #2
    _attr
      a
      Int
      #2
      _no_expr
      : _no_type
    #3
    _attr
      b
      Bool
      #3
      _no_expr
      : _no_type
    #6
    _method
      init
      #4
      _formal
        x
        Int
```



```
      #4
      _formal
        y
        Bool
      C
```

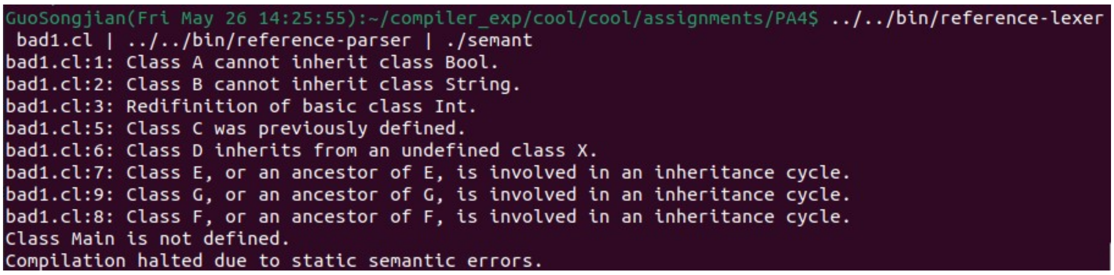


```
      #5
      _block
        #5
        _assign
          a
          #5
          _object
            x
          : _no_type
        : _no_type
        #5
        _assign
          b
          #5
          _object
            y
          : _no_type
        : _no_type
        #5
        _object
          self
        : _no_type
      : _no_type
    )
```



```
  #11
  _class
    Main
    Object
    "good.cl"
    (
    #10
    _method
      main
      C
      #10
      _dispatch
        #10
        _new
          C
        : _no_type
        init
        (
        #10
        _int
          1
        : _no_type
        #10
        _bool
          1
        : _no_type
        )
      : _no_type
    )
GuoSongjian(Fri May 26 14:25:55):~/compiler_exp/cool/cool/assignments/PA4$
```

（2）bad1.cl 语义分析



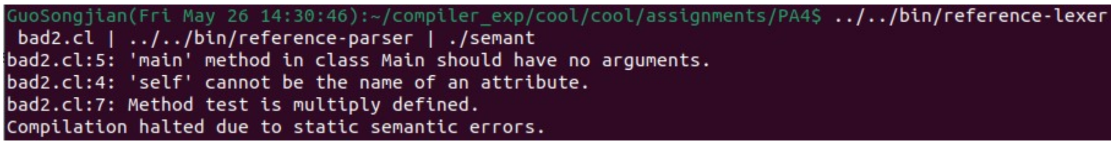
```
GuoSongjian(Fri May 26 14:25:55):~/compiler_exp/cool/cool/assignments/PA4$ ../../bin/reference-lexer
 bad1.cl | ../../bin/reference-parser | ./semant
bad1.cl:1: Class A cannot inherit class Bool.
bad1.cl:2: Class B cannot inherit class String.
bad1.cl:3: Redifinition of basic class Int.
bad1.cl:5: Class C was previously defined.
bad1.cl:6: Class D inherits from an undefined class X.
bad1.cl:7: Class E, or an ancestor of E, is involved in an inheritance cycle.
bad1.cl:9: Class G, or an ancestor of G, is involved in an inheritance cycle.
bad1.cl:8: Class F, or an ancestor of F, is involved in an inheritance cycle.
Class Main is not defined.
Compilation halted due to static semantic errors.
```

成功实现了对 class 的语义分析，包括：不能继承或定义 Int、Bool、

String 等基本类；不能重复定义同一个类；检查所继承的父类是否定义；检查类的继承是否存在环；检查 Main 类是否存在。

此外，实验要求中的类名合法性检查（大写字母开头）已在语法分析中实现，故这里不再重复。

（3）bad2.cl 语义分析



```
GuoSongjian(Fri May 26 14:30:46):~/compiler_exp/cool/cool/assignments/PA4$ ../../bin/reference-lexer
 bad2.cl | ../../bin/reference-parser | ./semant
bad2.cl:5: 'main' method in class Main should have no arguments.
bad2.cl:4: 'self' cannot be the name of an attribute.
bad2.cl:7: Method test is multiply defined.
Compilation halted due to static semantic errors.
```

成功实现了对 method 的部分语义分析，包括：类的属性名不能为 self；Main 类中是否存在 main 方法；Main 类中的 main 方法是否有参数；同一个类中的方法是否重复定义。

四、源代码

（1）cool-tree.handcode.h

```
// The following include files must come first.

#ifndef COOL_TREE_HANDCODE_H
#define COOL_TREE_HANDCODE_H

#include <iostream>
#include "tree.h"
#include "cool.h"
#include "stringtab.h"
#define yylineno curr_lineno;
extern int yylineno;

inline Boolean copy_Boolean(Boolean b) { return b; }
inline void assert_Boolean(Boolean) {}
inline void dump_Boolean(std::ostream& stream, int padding, Boolean
b)
    { stream << pad(padding) << (int) b << "\n"; }

void dump_Symbol(std::ostream& stream, int padding, Symbol b);
void assert_Symbol(Symbol b);
```

```
Symbol copy_Symbol(Symbol b);

class Program_class;
typedef Program_class *Program;
class Class__class;
typedef Class__class *Class_;
class Feature_class;
typedef Feature_class *Feature;
class Formal_class;
typedef Formal_class *Formal;
class Expression_class;
typedef Expression_class *Expression;
class Case_class;
typedef Case_class *Case;

typedef list_node<Class_> Classes_class;
typedef Classes_class *Classes;
typedef list_node<Feature> Features_class;
typedef Features_class *Features;
typedef list_node<Formal> Formals_class;
typedef Formals_class *Formals;
typedef list_node<Expression> Expressions_class;
typedef Expressions_class *Expressions;
typedef list_node<Case> Cases_class;
typedef Cases_class *Cases;

#define Program_EXTRAS                     \
virtual void semant() = 0;                  \
virtual void dump_with_types(std::ostream&, int) = 0;

#define program_EXTRAS                     \
void semant();                            \
void dump_with_types(std::ostream&, int);

#define Class__EXTRAS                      \
virtual Symbol get_filename() = 0;           \
virtual Symbol get_name() = 0;               \
virtual Symbol get_parent() = 0;             \
virtual Features get_features() = 0;          \
virtual void dump_with_types(std::ostream&, int) = 0;

#define class__EXTRAS                      \
Symbol get_filename() { return filename; }     \
Symbol get_name()    { return name; }        \
Symbol get_parent() { return parent; }        \
Features get_features() { return features; }   \
void dump_with_types(std::ostream&, int);

#define Feature_EXTRAS                     \
virtual bool is_method() = 0;               \
virtual void dump_with_types(std::ostream&, int) = 0;

#define Feature_SHARED_EXTRAS               \
void dump_with_types(std::ostream&, int);

#define method_EXTRAS                      \
bool is_method() { return true; }            \
Formals get_formals() { return formals; }     \
Symbol get_name() { return name; }

#define attr_EXTRAS                        \
bool is_method() { return false; }           \
Symbol get_name() { return name; }

#define Formal_EXTRAS                      \
virtual void dump_with_types(std::ostream&, int) = 0;

#define formal_EXTRAS                      \
void dump_with_types(std::ostream&, int);

#define Case_EXTRAS                        \
virtual void dump_with_types(std::ostream&, int) = 0;

#define branch_EXTRAS                      \
void dump_with_types(std::ostream& ,int);

#define Expression_EXTRAS                   \
Symbol type;                              \
Symbol get_type() { return type; }           \
Expression set_type(Symbol s) { type = s; return this; } \
virtual void dump_with_types(std::ostream&, int) = 0;    \
void dump_type(std::ostream&, int);           \
Expression_class() { type = (Symbol) NULL; }

#define Expression_SHARED_EXTRAS             \
void dump_with_types(std::ostream&, int);

#endif
```

(2) semant.h

```
#ifndef SEMANT_H_
#define SEMANT_H_

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

#define TRUE 1
#define FALSE 0

class ClassTable;
typedef ClassTable *ClassTableP;

// This is a structure that may be used to contain the semantic
// information such as the inheritance graph.
// You may use it or not as you like: it is only here to provide
// a container for the supplied methods.

class ClassTable {
private:
   int semant_errors;
   void install_basic_classes();
   void install_classes(Classes &classes);
   void check_inheritance();
   std::ostream& error_stream;
   std::map<Symbol, Class_> symbol_table;
   std::map<Class_, std::vector<method_class*>> method_table;

public:
   ClassTable(Classes);
   void check_main();
   void install_methods();
   int errors() { return semant_errors; }
   std::ostream& semant_error();
   std::ostream& semant_error(Class_ c);
   std::ostream& semant_error(Symbol filename, tree_node *t);
};
```

```
#endif
```

(3) semant.cc

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

extern int semant_debug;
extern char *curr_filename;

///////////////////////////////////////////////////////////////////
///
//
// Symbols
//
// For convenience, a large number of symbols are predefined here.
// These symbols include the primitive type and method names, as
well
// as fixed names used by the runtime system.
//
///////////////////////////////////////////////////////////////////
///
static Symbol
   arg,
   arg2,
   Bool,
   concat,
   cool_abort,
   copy,
   Int,
   in_int,
   in_string,
   IO,
   length,
   Main,
   main_meth,
   No_class,
   No_type,
   Object,
   out_int,
   out_string,
   prim_slot,
   self,
   SELF_TYPE,
   Str,
   str_field,
   substr,
   type_name,
   val;
//
// Initializing the predefined symbols.
//
static 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");
}
```

```
ClassTable::ClassTable(Classes classes) : semant_errors(0) ,
error_stream(std::cerr) {
    /* Fill this in */
    install_basic_classes();
    install_classes(classes);
    check_inheritance();
}

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 into 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.

    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);

   //
   // 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     "  "    "
   //
   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);

   //
   // 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           performs string
concatenation
   //       substr(arg: Int, arg2: Int): Str    substring selection
   //
   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);
   symbol_table[Object_class->get_name()] = Object_class;
   symbol_table[IO_class->get_name()] = IO_class;
```

```
    symbol_table[Int_class->get_name()] = Int_class;
    symbol_table[Bool_class->get_name()] = Bool_class;
    symbol_table[Str_class->get_name()] = Str_class;
}

void ClassTable::install_classes(Classes &classes) {
    Class_ curr_class;
    Symbol curr_name, parent_name;
    for (int i = classes->first(); classes->more(i); i =
classes->next(i)) {
        curr_class = classes->nth(i);
        curr_name = curr_class->get_name();
        parent_name = curr_class->get_parent();
        if (curr_name == SELF_TYPE || curr_name == Int || curr_name
== Bool || curr_name == Str || curr_name == IO || curr_name ==
Object) {
            semant_error(curr_class) << "Redifinition of basic class
" << curr_name << ".\n";
        } else if (symbol_table.find(curr_name) !=
symbol_table.end()) {
            semant_error(curr_class) << "Class " << curr_name << "
was previously defined.\n";
        } else if (parent_name == Int || parent_name == Bool ||
parent_name == Str || parent_name == SELF_TYPE) {
            semant_error(curr_class) << "Class " << curr_name << "
cannot inherit class " << parent_name << ".\n";
        } else {
            symbol_table[curr_name] = curr_class;
        }
    }
}

void ClassTable::check_inheritance() {
    Symbol curr_name;
    Symbol parent_name;
    std::set<Symbol> tmp_set;
    Symbol first_symbol;
    Class_ curr_class;
    for (std::map<Symbol, Class_>::iterator it =
symbol_table.begin(); it != symbol_table.end(); it++){
        curr_name = it->first;
        curr_class = it->second;
        parent_name = curr_class->get_parent();
        //check if parent defined
        if (curr_name != Object && parent_name != Object){
            if (symbol_table.find(parent_name) ==
symbol_table.end()){
                semant_error(curr_class) << "Class " << curr_name <<
" inherits from an undefined class " << parent_name << ".\n";
                continue;
            }
            //check cycle
            first_symbol = curr_name;
            while(curr_name != Object){
                parent_name = curr_class->get_parent();
                if(tmp_set.find(curr_name) != tmp_set.end()){
                    semant_error(symbol_table[first_symbol]) << "Class
" << first_symbol << ", or an ancestor of " << first_symbol << ",
is involved in an inheritance cycle.\n";
                    break;
                }else{
                    tmp_set.insert(curr_name);
                    curr_name = parent_name;
                    curr_class = symbol_table[curr_name];
                }
            }
            tmp_set.clear();
        }
    }
}

void ClassTable::check_main(){
    //check Main
    if (symbol_table.find(Main) == symbol_table.end()){
        semant_error() << "Class Main is not defined.\n";
        return;
    }

    //check main() method
    Features feature_list = symbol_table[Main]->get_features();
    bool find_flag = false;
    bool para_flag = false;
    method_class* curr_method;

    for (int i = feature_list->first(); feature_list->more(i); i =
feature_list->next(i)){
        if(feature_list->nth(i)->is_method() &&
static_cast<method_class*>(feature_list->nth(i))->get_name() ==
main_meth){
            find_flag = true;
            curr_method =
static_cast<method_class*>(feature_list->nth(i));
            Formals formals = curr_method->get_formals();
            if ((formals->len()) >= 1){
                para_flag = true;
            }
        }
    }

    if(!find_flag){
        semant_error(symbol_table[Main]) << "No 'main' method in
class Main.\n";
        return;
    }
    if(para_flag){
        semant_error(symbol_table[Main]->get_filename(),
curr_method) << "'main' method in class Main should have no
arguments.\n";
    }
}

void ClassTable::install_methods(){
    Symbol curr_nmae;
    Features features;
    std::vector<method_class*> methodlist;
    method_class* tmp_method;
    attr_class* tmp_attr;
    Class_ curr_class;
    for (std::map<Symbol, Class_>::iterator it =
symbol_table.begin(); it != symbol_table.end(); it++){
        curr_class = it->second;
        //install every method
        features = curr_class->get_features();
        for (int i = features->first(); features->more(i); i =
features->next(i)){
            if(features->nth(i)->is_method()){
                //method
                tmp_method =
static_cast<method_class*>(features->nth(i));
                bool find_flag = false;
                for (std::vector<method_class*>::iterator itv =
methodlist.begin(); itv != methodlist.end(); itv++){
                    if ((*itv)->get_name() == tmp_method->get_name()){
                        find_flag = true;
                    }
                }
                if (find_flag){
                    semant_error(curr_class->get_filename(),
tmp_method) << "Method " << tmp_method->get_name() << " is multiply
defined.\n";
                }
                else{
                    methodlist.push_back(tmp_method);
                }
            }else{
                tmp_attr =
static_cast<attr_class*>(features->nth(i));
                if (tmp_attr->get_name() == self){
                    semant_error(curr_class->get_filename(), tmp_attr)
<< "'self' cannot be the name of an attribute.\n";
                }
            }
        }
        method_table[curr_class] = methodlist;
        methodlist.clear();
    }
}

///////////////////////////////////////////////////////////////////
//
// 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
//
///////////////////////////////////////////////////////////////////
```

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

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

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

/*   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.
       (see `tree.h')

    You are free to first do 1), make sure you catch all semantic
    errors. Part 2) can be done in a second stage, when you want
    to build mycoolc.
*/
void program_class::semant()
{
   initialize_constants();

   /* ClassTable constructor may do some semantic analysis */
   ClassTable *classtable = new ClassTable(classes);

   /* some semantic analysis code may go here */
   classtable->check_main();
   classtable->install_methods();

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

(4) good.cl

```
class C {
    a : Int;
    b : Bool;
    init(x : Int, y : Bool) : C {
        { a <- x; b <- y; self; }
    };
};

Class Main {
    main():C { (new C).init(1,true) };
};
```

(5) bad1.cl

```
Class A Inherits Bool {};
Class B Inherits String {};
Class Int {};
Class C {};
Class C {};
Class D Inherits X {};
Class E Inherits G {};
Class F Inherits E {};
Class G Inherits F {};
```

(6) bad2.cl

```
Class Main {
    a : Int;
    b: Bool;
    self: String;
    main(a:Int): Object {new Object};
    test(): IO {new IO};
    test(): Bool {new Bool};
};
```