IT-314 Software Engineering
Lab-7

Program Inspection, Debugging and Static Analysis

Prof . Saurabh Tiwari

ID: 202201142(Dip Baldha)

```cpp
//more then 200 LOC c++ code
#include <iostream>
#include <sstream>
#include <string>
#include <vector>
#include <list>
#include <map>

// return given mumber as a string
std::string str(long n) { std::ostringstream os; os << n; return os.str(); }

// return true iff given character is '0'..'9'
bool isdig(char c) { return isdigit(static_cast<unsigned char>(c)) != 0; }


////////////////////// cell

enum cell_type { Symbol, Number, List, Proc, Lambda };

struct environment; // forward declaration; cell and environment reference each
other

// a variant that can hold any kind of lisp value
struct cell {
    typedef cell (*proc_type)(const std::vector<cell> &);
    typedef std::vector<cell>::const_iterator iter;
    typedef std::map<std::string, cell> map;
    cell_type type; std::string val; std::vector<cell> list; proc_type proc;
environment * env;
    cell(cell_type type = Symbol) : type(type), env(0) {}
    cell(cell_type type, const std::string & val) : type(type), val(val), env(0)
{}
    cell(proc_type proc) : type(Proc), proc(proc), env(0) {}
};

typedef std::vector<cell> cells;
typedef cells::const_iterator cellit;
```

```
const cell false_sym(Symbol, "#f");
const cell true_sym(Symbol, "#t"); // anything that isn't false_sym is true
const cell nil(Symbol, "nil");


////////////////////// environment

// a dictionary that (a) associates symbols with cells, and
// (b) can chain to an "outer" dictionary
struct environment {
    environment(environment * outer = 0) : outer_(outer) {}

    environment(const cells & parms, const cells & args, environment * outer)
    : outer_(outer)
    {
        cellit a = args.begin();
        for (cellit p = parms.begin(); p != parms.end(); ++p)
            env_[p->val] = *a++;
    }

    // map a variable name onto a cell
    typedef std::map<std::string, cell> map;

    // return a reference to the innermost environment where 'var' appears
    map & find(const std::string & var)
    {
        if (env_.find(var) != env_.end())
            return env_; // the symbol exists in this environment
        if (outer_)
            return outer_->find(var); // attempt to find the symbol in some
"outer" env
        std::cout << "unbound symbol '" << var << "'\n";
        exit(1);
    }

    // return a reference to the cell associated with the given symbol 'var'
    cell & operator[] (const std::string & var)
    {
        return env_[var];
    }

private:
    map env_; // inner symbol->cell mapping
    environment * outer_; // next adjacent outer env, or 0 if there are no
further environments
};


////////////////////// built-in primitive procedures

cell proc_add(const cells & c)
{
    long n(atol(c[0].val.c_str()));
    for (cellit i = c.begin()+1; i != c.end(); ++i) n += atol(i->val.c_str());
    return cell(Number, str(n));
}

cell proc_sub(const cells & c)
{
    long n(atol(c[0].val.c_str()));
    for (cellit i = c.begin()+1; i != c.end(); ++i) n -= atol(i->val.c_str());
    return cell(Number, str(n));
}

cell proc_mul(const cells & c)
{
    long n(1);
    for (cellit i = c.begin(); i != c.end(); ++i) n *= atol(i->val.c_str());
    return cell(Number, str(n));
}

cell proc_div(const cells & c)
{
    long n(atol(c[0].val.c_str()));
    for (cellit i = c.begin()+1; i != c.end(); ++i) n /= atol(i->val.c_str());
    return cell(Number, str(n));
}
```

```
cell proc_greater(const cells & c)
{
    long n(atol(c[0].val.c_str()));
    for (cellit i = c.begin()+1; i != c.end(); ++i)
        if (n <= atol(i->val.c_str()))
            return false_sym;
    return true_sym;
}

cell proc_less(const cells & c)
{
    long n(atol(c[0].val.c_str()));
    for (cellit i = c.begin()+1; i != c.end(); ++i)
        if (n >= atol(i->val.c_str()))
            return false_sym;
    return true_sym;
}

cell proc_less_equal(const cells & c)
{
    long n(atol(c[0].val.c_str()));
    for (cellit i = c.begin()+1; i != c.end(); ++i)
        if (n > atol(i->val.c_str()))
            return false_sym;
    return true_sym;
}

cell proc_length(const cells & c) { return cell(Number, str(c[0].list.size())); }
cell proc_nullp(const cells & c)  { return c[0].list.empty() ? true_sym :
false_sym; }
cell proc_car(const cells & c)    { return c[0].list[0]; }

cell proc_cdr(const cells & c)
{
    if (c[0].list.size() < 2)
        return nil;
    cell result(c[0]);
    result.list.erase(result.list.begin());
    return result;
}

cell proc_append(const cells & c)
{
    cell result(List);
    result.list = c[0].list;
    for (cellit i = c[1].list.begin(); i != c[1].list.end(); ++i)
result.list.push_back(*i);
    return result;
}

cell proc_cons(const cells & c)
{
    cell result(List);
    result.list.push_back(c[0]);
    for (cellit i = c[1].list.begin(); i != c[1].list.end(); ++i)
result.list.push_back(*i);
    return result;
}

cell proc_list(const cells & c)
{
    cell result(List); result.list = c;
    return result;
}

// define the bare minimum set of primintives necessary to pass the unit tests
void add_globals(environment & env)
{
    env["nil"] = nil;   env["#f"] = false_sym;  env["#t"] = true_sym;
    env["append"] = cell(&proc_append);   env["car"]  = cell(&proc_car);
    env["cdr"]    = cell(&proc_cdr);      env["cons"] = cell(&proc_cons);
    env["length"] = cell(&proc_length);   env["list"] = cell(&proc_list);
    env["null?"]  = cell(&proc_nullp);    env["+"]    = cell(&proc_add);
    env["-"]      = cell(&proc_sub);      env["*"]    = cell(&proc_mul);
    env["/"]      = cell(&proc_div);      env[">"]    = cell(&proc_greater);
    env["<"]      = cell(&proc_less);     env["<="]   = cell(&proc_less_equal);
}
```

```
////////////////////// eval

cell eval(cell x, environment * env)
{
    if (x.type == Symbol)
        return env->find(x.val)[x.val];
    if (x.type == Number)
        return x;
    if (x.list.empty())
        return nil;
    if (x.list[0].type == Symbol) {
        if (x.list[0].val == "quote")       // (quote exp)
            return x.list[1];
        if (x.list[0].val == "if")          // (if test conseq [alt])
            return eval(eval(x.list[1], env).val == "#f" ? (x.list.size() < 4 ?
nil : x.list[3]) : x.list[2], env);
        if (x.list[0].val == "set!")        // (set! var exp)
            return env->find(x.list[1].val)[x.list[1].val] = eval(x.list[2],
env);
        if (x.list[0].val == "define")      // (define var exp)
            return (*env)[x.list[1].val] = eval(x.list[2], env);
        if (x.list[0].val == "lambda") {    // (lambda (var*) exp)
            x.type = Lambda;
            // keep a reference to the environment that exists now (when the
            // lambda is being defined) because that's the outer environment
            // we'll need to use when the lambda is executed
            x.env = env;
            return x;
        }
        if (x.list[0].val == "begin") {     // (begin exp*)
            for (size_t i = 1; i < x.list.size() - 1; ++i)
                eval(x.list[i], env);
            return eval(x.list[x.list.size() - 1], env);
        }
    }
                                            // (proc exp*)
    cell proc(eval(x.list[0], env));
    cells exps;
    for (cell::iter exp = x.list.begin() + 1; exp != x.list.end(); ++exp)
        exps.push_back(eval(*exp, env));
    if (proc.type == Lambda) {
        // Create an environment for the execution of this lambda function
        // where the outer environment is the one that existed* at the time
        // the lambda was defined and the new inner associations are the
        // parameter names with the given arguments.
        // *Although the environmet existed at the time the lambda was defined
        // it wasn't necessarily complete - it may have subsequently had
        // more symbols defined in that environment.
        return eval(/*body*/proc.list[2], new
environment(/*parms*/proc.list[1].list, /*args*/exps, proc.env));
    }
    else if (proc.type == Proc)
        return proc.proc(exps);

    std::cout << "not a function\n";
    exit(1);
}


////////////////////// parse, read and user interaction

// convert given string to list of tokens
std::list<std::string> tokenize(const std::string & str)
{
    std::list<std::string> tokens;
    const char * s = str.c_str();
    while (*s) {
        while (*s == ' ')
            ++s;
        if (*s == '(' || *s == ')')
            tokens.push_back(*s++ == '(' ? "(" : ")");
        else {
            const char * t = s;
            while (*t && *t != ' ' && *t != '(' && *t != ')')
                ++t;
            tokens.push_back(std::string(s, t));
            s = t;
        }
    }
    return tokens;
}

// numbers become Numbers; every other token is a Symbol
cell atom(const std::string & token)
{
    if (isdig(token[0]) || (token[0] == '-' && isdig(token[1])))
        return cell(Number, token);
    return cell(Symbol, token);
}

// return the Lisp expression in the given tokens
cell read_from(std::list<std::string> & tokens)
{
    const std::string token(tokens.front());
    tokens.pop_front();
    if (token == "(") {
        cell c(List);
        while (tokens.front() != ")")
            c.list.push_back(read_from(tokens));
        tokens.pop_front();
        return c;
    }
    else
        return atom(token);
}

// return the Lisp expression represented by the given string
cell read(const std::string & s)
{
    std::list<std::string> tokens(tokenize(s));
    return read_from(tokens);
}

// convert given cell to a Lisp-readable string
std::string to_string(const cell & exp)
```

```
{
    if (exp.type == List) {
        std::string s("(");
        for (cell::iter e = exp.list.begin(); e != exp.list.end(); ++e)
            s += to_string(*e) + ' ';
        if (s[s.size() - 1] == ' ')
            s.erase(s.size() - 1);
        return s + ')';
    }
    else if (exp.type == Lambda)
        return "<Lambda>";
    else if (exp.type == Proc)
        return "<Proc>";
    return exp.val;
}

// the default read-eval-print-loop
void repl(const std::string & prompt, environment * env)
{
    for (;;) {
        std::cout << prompt;
        std::string line; std::getline(std::cin, line);
        std::cout << to_string(eval(read(line), env)) << '\n';
    }
}



void demorgan() {
    std::cout << "de-Morgan example\n";

    context c;

    expr x = c.bool_const("x");
    expr y = c.bool_const("y");
    expr conjecture = (!(x && y)) == (!x || !y);

    solver s(c);
    // adding the negation of the conjecture as a constraint.
    s.add(!conjecture);
    std::cout << s << "\n";
    std::cout << s.to_smt2() << "\n";
    switch (s.check()) {
    case unsat:   std::cout << "de-Morgan is valid\n"; break;
    case sat:     std::cout << "de-Morgan is not valid\n"; break;
    case unknown: std::cout << "unknown\n"; break;
    }
}

/**
   \brief Find a model for <tt>x >= 1 and y < x + 3</tt>.
*/
void find_model_example1() {
    std::cout << "find_model_example1\n";
    context c;
    expr x = c.int_const("x");
    expr y = c.int_const("y");
    solver s(c);

    s.add(x >= 1);
    s.add(y < x + 3);
    std::cout << s.check() << "\n";

    model m = s.get_model();
    std::cout << m << "\n";
    // traversing the model
    for (unsigned i = 0; i < m.size(); i++) {
        func_decl v = m[i];
        // this problem contains only constants
        assert(v.arity() == 0);
        std::cout << v.name() << " = " << m.get_const_interp(v) << "\n";
    }
    // we can evaluate expressions in the model.
    std::cout << "x + y + 1 = " << m.eval(x + y + 1) << "\n";
}


/**
```

```
   \brief Prove <tt>x = y implies g(x) = g(y)</tt>, and
   disprove <tt>x = y implies g(g(x)) = g(y)</tt>.

   This function demonstrates how to create uninterpreted types and
   functions.
*/
void prove_example1() {
    std::cout << "prove_example1\n";

    context c;
    expr x      = c.int_const("x");
    expr y      = c.int_const("y");
    sort I      = c.int_sort();
    func_decl g = function("g", I, I);

    solver s(c);
    expr conjecture1 = implies(x == y, g(x) == g(y));
    std::cout << "conjecture 1\n" << conjecture1 << "\n";
    s.add(!conjecture1);
    if (s.check() == unsat)
        std::cout << "proved" << "\n";
    else
        std::cout << "failed to prove" << "\n";

    s.reset(); // remove all assertions from solver s

    expr conjecture2 = implies(x == y, g(g(x)) == g(y));
    std::cout << "conjecture 2\n" << conjecture2 << "\n";
    s.add(!conjecture2);
    if (s.check() == unsat) {
        std::cout << "proved" << "\n";
    }
    else {
        std::cout << "failed to prove" << "\n";
        model m = s.get_model();
        std::cout << "counterexample:\n" << m << "\n";
        std::cout << "g(g(x)) = " << m.eval(g(g(x))) << "\n";
        std::cout << "g(y)    = " << m.eval(g(y)) << "\n";
    }
```

```
}

/**
   \brief Prove <tt>not(g(g(x) - g(y)) = g(z)), x + z <= y <= x implies z < 0
</tt>.
   Then, show that <tt>z < -1</tt> is not implied.

   This example demonstrates how to combine uninterpreted functions and
arithmetic.
*/
void prove_example2() {
    std::cout << "prove_example1\n";
    context c;
    expr x      = c.int_const("x");
    expr y      = c.int_const("y");
    expr z      = c.int_const("z");
    sort I      = c.int_sort();
    func_decl g = function("g", I, I);

    expr conjecture1 = implies(g(g(x) - g(y)) != g(z) && x + z <= y && y <= x,
                               z < 0);

    solver s(c);
    s.add(!conjecture1);
    std::cout << "conjecture 1:\n" << conjecture1 << "\n";
    if (s.check() == unsat)
        std::cout << "proved" << "\n";
    else
        std::cout << "failed to prove" << "\n";

    expr conjecture2 = implies(g(g(x) - g(y)) != g(z) && x + z <= y && y <= x,
                               z < -1);
    s.reset();
    s.add(!conjecture2);
    std::cout << "conjecture 2:\n" << conjecture2 << "\n";

    if (s.check() == unsat) {
        std::cout << "proved" << "\n";
    }
    else {
        std::cout << "failed to prove" << "\n";
        std::cout << "counterexample:\n" << s.get_model() << "\n";
    }
}

/**
   \brief Nonlinear arithmetic example 1
*/
void nonlinear_example1() {
    std::cout << "nonlinear example 1\n";
    config cfg;
    cfg.set("auto_config", true);
    context c(cfg);

    expr x = c.real_const("x");
    expr y = c.real_const("y");
    expr z = c.real_const("z");

    solver s(c);

    s.add(x*x + y*y == 1);                     // x^2 + y^2 == 1
    s.add(x*x*x + z*z*z < c.real_val("1/2"));  // x^3 + z^3 < 1/2
    s.add(z != 0);
    std::cout << s.check() << "\n";
    model m = s.get_model();
    std::cout << m << "\n";
    set_param("pp.decimal", true); // set decimal notation
    std::cout << "model in decimal notation\n";
    std::cout << m << "\n";
    set_param("pp.decimal-precision", 50); // increase number of decimal places
to 50.
    std::cout << "model using 50 decimal places\n";
    std::cout << m << "\n";
    set_param("pp.decimal", false); // disable decimal notation
}

/**
```

```
   \brief Simple function that tries to prove the given conjecture using the
following steps:
   1- create a solver
   2- assert the negation of the conjecture
   3- checks if the result is unsat.
*/
void prove(expr conjecture) {
    context & c = conjecture.ctx();
    solver s(c);
    s.add(!conjecture);
    std::cout << "conjecture:\n" << conjecture << "\n";
    if (s.check() == unsat) {
        std::cout << "proved" << "\n";
    }
    else {
        std::cout << "failed to prove" << "\n";
        std::cout << "counterexample:\n" << s.get_model() << "\n";
    }
}

/**
   \brief Simple bit-vector example. This example disproves that x - 10 <= 0 IFF
x <= 10 for (32-bit) machine integers
*/
void bitvector_example1() {
    std::cout << "bitvector example 1\n";
    context c;
    expr x = c.bv_const("x", 32);

    // using signed <=
    prove((x - 10 <= 0) == (x <= 10));

    // using unsigned <=
    prove(ule(x - 10, 0) == ule(x, 10));

    expr y = c.bv_const("y", 32);
    prove(implies(concat(x, y) == concat(y, x), x == y));
}
```

```
/**
   \brief Find x and y such that: x ^ y - 103 == x * y
*/
void bitvector_example2() {
    std::cout << "bitvector example 2\n";
    context c;
    expr x = c.bv_const("x", 32);
    expr y = c.bv_const("y", 32);
    solver s(c);
    // In C++, the operator == has higher precedence than ^.
    s.add((x ^ y) - 103 == x * y);
    std::cout << s << "\n";
    std::cout << s.check() << "\n";
    std::cout << s.get_model() << "\n";
}

/**
   \brief Mixing C and C++ APIs.
*/
void capi_example() {
    std::cout << "capi example\n";
    context c;
    expr x = c.bv_const("x", 32);
    expr y = c.bv_const("y", 32);
    // Invoking a C API function, and wrapping the result using an expr object.
    expr r = to_expr(c, Z3_mk_bvsrem(c, x, y));
    std::cout << "r: " << r << "\n";
}

/**
   \brief Demonstrate how to evaluate expressions in a model.
*/
void eval_example1() {
    std::cout << "eval example 1\n";
    context c;
    expr x = c.int_const("x");
    expr y = c.int_const("y");
    solver s(c);
    /* assert x < y */
    s.add(x < y);
    /* assert x > 2 */
    s.add(x > 2);

    std::cout << s.check() << "\n";

    model m = s.get_model();
    std::cout << "Model:\n" << m << "\n";
    std::cout << "x+y = " << m.eval(x+y) << "\n";
}

/**
   \brief Several contexts can be used simultaneously.
*/
void two_contexts_example1() {
    std::cout << "two contexts example 1\n";
    context c1, c2;

    expr x = c1.int_const("x");
    expr n = x + 1;
    // We cannot mix expressions from different contexts, but we can copy
    // an expression from one context to another.
    // The following statement copies the expression n from c1 to c2.
    expr n1 = to_expr(c2, Z3_translate(c1, n, c2));
    std::cout << n1 << "\n";
}

/**
   \brief Demonstrates how to catch API usage errors.
 */
void error_example() {
    std::cout << "error example\n";

    context c;
    expr x = c.bool_const("x");

    // Error using the C API can be detected using Z3_get_error_code.
    // The next call fails because x is a constant.
    //Z3_ast arg = Z3_get_app_arg(c, x, 0);
    if (Z3_get_error_code(c) != Z3_OK) {
        std::cout << "last call failed.\n";
    }

    // The C++ layer converts API usage errors into exceptions.
    try {
        // The next call fails because x is a Boolean.
        expr n = x + 1;
    }
    catch (exception & ex) {
        std::cout << "failed: " << ex << "\n";
    }

    // The functions to_expr, to_sort and to_func_decl also convert C API errors
into exceptions.
    try {
        expr arg = to_expr(c, Z3_get_app_arg(c, x, 0));
    }
    catch (exception & ex) {
        std::cout << "failed: " << ex << "\n";
    }
}




int main ()
{
    environment global_env; add_globals(global_env);
    repl("90> ", &global_env);
}
```

I. PROGRAM INSPECTION:

Category A: Data Reference Errors

1. Uninitialized Variables

- In the code, each variable seems properly initialized before use.
- `std::string str(long n)` function initializes `os` and uses it correctly, returning a valid string.
- There are no obvious signs of uninitialized variables. The main `repl()` loop reads inputs from the user, so the variables depend on correct user input. No explicit initialization issues.

2. Array Bounds Check

- In the code, array-like structures such as `std::vector` are used (e.g., `cell::list` in the `cell` structure).
- Accessing vectors in functions like `proc_car`, `proc_cdr` directly accesses elements. There is a check in `proc_cdr` for size before accessing the vector, so no out-of-bounds access there.
- However, in `proc_car`, there is no bounds check, so if the `list` is empty, accessing `c[0].list[0]` may result in out-of-bounds access. Fix: Add a size check before accessing elements in `proc_car`.

3. Array Subscript Values as Integers

- The code uses `std::vector` with integer indexing, so there's no explicit issue with non-integer indices (since C++ ensures this with the vector type).

4. Dangling References

- The main area of concern for dangling references is in the `environment` class, particularly how `outer_` is used to chain environments. It is assumed that the lifetime of the outer environment is longer than the inner ones.
- There is no explicit deallocation of the `environment` pointers. You should ensure that the environment objects have a well-defined lifecycle to avoid accessing deallocated memory.

Fix: Consider using smart pointers like `std::shared_ptr` for managing `outer_` to prevent accidental memory leaks or dangling references.

5. Memory Areas with Aliases

- **N**o explicit aliasing with different attributes is observed here. The variables and memory are well-controlled, and the code does not contain explicit memory manipulation or risky aliasing practices.

6. Variable Value with Wrong Type

- The `cell` struct can represent different types (`Symbol`, `Number`, `List`, etc.). Proper type checking seems to be in place when accessing these, so there is little risk of using the wrong type, thanks to the enum-based system (`cell_type`).

7. Explicit or Implicit Addressing Problems

- **T**he code does not contain any explicit bit manipulations, so there are no observable addressing problems. Memory is managed through high-level C++ constructs like `std::vector` and `std::string`.

8. Pointer or Reference Variables and Compiler Expectations

- The code uses raw pointers (e.g., `environment * outer_` and `proc_type`). While raw pointers are used safely, you should ensure that memory is managed carefully.
- It's recommended to use smart pointers (`std::shared_ptr` or `std::unique_ptr`) to handle memory more safely and avoid potential memory management issues**.**

9. Data Structures in Multiple Procedures

- **T**he `cell` and `environment` structures seem consistently used across the code. There's no obvious issue with multiple definitions of the same structure.

10. Off-by-One Errors in String/Array Indexing

- No obvious off-by-one errors are seen in the indexing operations, but bounds checks for vectors (like in `proc_car`) can be improved as mentioned earlier.

11. Inheritance Requirements in Object-Oriented Classes

- **T**he code doesn't use object-oriented inheritance, but it defines structures like `cell` and `environment`. Since these are simple data structures, no inheritance-based issues are expected.

Category B: Data-Declaration Errors

Explicit Declaration of Variables:

- Ensure that all variables, especially arrays, are explicitly declared before use. For example, in languages like Fortran, failing to use a `DIMENSION` statement for an array can lead to misinterpretation (e.g., `C=A(I)` could be seen as a function call instead of array access).
- In modern programming languages (Java, Python, etc.), you should declare variables appropriately using keywords like `int`, `float`, `String`, etc., if the language requires explicit types.
- Pay special attention to variables in inner procedures or blocks. If a variable is used in both an inner scope and an outer scope, it should be clear whether the inner scope is using its own variable or referencing the outer one (shadowing issues).

Understanding Defaults:

- When attributes like data type, length, or access permissions are not explicitly stated, make sure the default values and behaviors are well understood. For example, in Java, variables of class types default to `null`, and numerical variables default to `0`. Ensure that these defaults align with what is intended in the program.

Proper Initialization:

- Check if all variables (especially arrays and strings) are properly initialized where necessary. Languages often have particular rules around array and string initialization that can lead to errors. For example, in C/C++, uninitialized arrays or variables may have unpredictable behavior due to garbage values.
- In higher-level languages, look out for issues with initializing large or multidimensional arrays, ensuring the correct memory is allocated.

Correct Data Type and Length Assignment:

- Make sure each variable is assigned the correct data type and length. For instance, in a situation where a floating-point number is expected but an integer is used instead, calculations could be incorrect or truncated. For arrays, ensure that the size specified matches the required usage throughout the program.

Memory Type Consistency:

- **Ensure that variable initialization matches the memory type. For instance,** initializing a variable in a read-only memory section might cause runtime errors if you later attempt to modify it. Similarly, if a variable is placed in volatile memory but is used as if it were permanent, it could cause issues.

Similar Variable Names:

- Be cautious of variables with similar names (e.g., `VOLT` and `VOLTS`). Although this may not immediately cause an error, it's a common source of confusion and bugs. During review, check that similarly named variables are not mistakenly used in place of each other.

Category C: Computation Errors

1. Inconsistent Data Types in Computations:

- Ensure that variables used in computations are of compatible data types. For instance, performing arithmetic on a string variable or using non-numeric types (e.g., `boolean`) in mathematical operations should raise a warning.
- In languages like JavaScript or Python, which allow dynamic typing, carefully check for unintended type coercion (e.g., adding a string and a number).

2. Mixed-Mode Computations (e.g., Integer and Floating-Point Operations):

When computations involve mixed data types, such as adding an `int` and a `float`, ensure you understand the language's conversion rules to avoid unintended results. For example, in Java, dividing two integers results in integer division, which discards the fractional part:
Java

3. Same Data Type but Different Lengths:

- Ensure that when variables of the same type but different sizes (e.g., `int32` and `int16`) are involved in computations, appropriate conversions or precautions are taken. For example, operations between a `long` and an `int` might result in truncation or overflow errors.

4. Target Variable Smaller Than Expression Result:

Check if the target variable of an assignment is smaller in size or precision than the result of the right-hand expression. For example, assigning the result of a `long` computation to an `int` can lead to truncation.
java

5. Overflow or Underflow:

Look for potential overflow or underflow during intermediate computations. This can happen when the result of an intermediate step exceeds the maximum value allowed for a data type:

java
```
int largeNumber = 100000;

int result = largeNumber * largeNumber;  // Risk of overflow
```

- Use data types with larger capacities (e.g., `long` instead of `int`) or special libraries for safe arithmetic.

6. Division by Zero:

Ensure that every divisor in division operations is checked to avoid division by zero. This is a common source of runtime errors:
java
```
int denominator = 0;

if (denominator != 0) {

    int result = numerator / denominator;

} else {

    System.out.println("Division by zero error!");
```

7. Inaccuracies Due to Base-2 Representation:

Be cautious of floating-point inaccuracies caused by the binary representation of decimals. For example, `10 x 0.1` might not exactly equal `1.0` due to the way floating-point numbers are stored in binary:

8. Variables Going Out of Meaningful Range:

- Check if the values of variables are constrained within a valid range. For example, a variable representing probability should always be between 0 and 1:

Category D: Comparison Errors

1. Comparisons Between Different Data Types

- Potential Issues: In the code, numeric values are compared as strings during evaluation, e.g., `eval(x.list[1], env).val == "#f"` in the `if` statement inside the `eval` function. This comparison uses string literals, which could lead to unexpected behavior if not handled correctly.
- Recommendation: Ensure that comparisons are made between values of the same data type. For numerical comparisons, convert strings to numbers before comparison.

2. Mixed-Mode Comparisons

- Potential Issues: The `proc_greater`, `proc_less`, and other similar functions iterate over numeric values, but they rely on string conversion using `atol()`. If a non-numeric string is passed, this could lead to incorrect comparisons.
- Recommendation: Validate inputs to ensure that all values being compared are indeed numbers.

3. Correctness of Comparison Operators

- Potential Issues: The code uses the correct operators for comparisons, such as <= and >=, but the logic must be checked to ensure that they are used as intended. For example, `proc_greater` returns `false_sym` if any number is greater than or equal to the reference number, which aligns with the "greater than" semantics but needs validation in practice.
- Recommendation: Carefully verify that each comparison operator is used correctly in its respective context.

4. Clarity of Boolean Expressions

- Potential Issues: The logical expressions, particularly in `eval`, need to clearly reflect the programmer's intent. The expression for evaluating the `if` statement should ensure the condition clearly states what it should accomplish.
- Recommendation: Ensure that all Boolean expressions convey clear and unambiguous logic.

5. Boolean Operator Operand Types

- Potential Issues: The `eval` function contains expressions where mixed comparisons might occur (e.g., comparing results of different types). The code should ensure that comparisons involving logical operators are purely Boolean.
- Recommendation: Validate operands of Boolean operators to ensure they are of Boolean type and that expressions like `2<i<10` are replaced with the correct form `(2<i) && (i<10)`.

6. Comparisons Between Floating-Point Numbers

- Potential Issues: There are no explicit floating-point comparisons in the provided code, but if the input data were to include floating-point values, special care must be taken to avoid pitfalls of precision.
- Recommendation: If floating-point numbers are introduced, implement a comparison function that accounts for precision errors.

7. Order of Evaluation and Precedence of Operators

- Potential Issues: The expressions in the `eval` function rely on a clear understanding of operator precedence. For instance, the logical operators should be evaluated in the intended order.
- Recommendation: Always use parentheses to clarify the order of operations and ensure that evaluations proceed as expecte**d.**

Category E: Control-Flow Errors

1. Multiway Branching

- Check: The program does not contain multiway branching such as `GO TO`. Instead, it uses function calls to evaluate expressions. Each procedure is called explicitly with the correct arguments.
- Conclusion: There are no multiway branches that can exceed their possibilities, ensuring proper flow.

2. Loop Termination

- Check: The main evaluation loop (REPL) continues indefinitely until interrupted. However, each function call (such as `proc_add`, `proc_sub`, etc.) within the `eval` function has well-defined termination conditions.
- Conclusion: The loops within the procedures will terminate because they process a finite number of elements in the provided list.

3. Module Termination

- Check: The program is designed to run continuously until user input stops it (via an EOF or interrupt signal). Each module or procedure that performs computations has a defined exit point.
- Conclusion: The program, as a whole, will eventually terminate based on user input.

4. Loop Execution

- Check: The loops inside the evaluation and built-in procedures are structured correctly. For example, in `proc_cdr`, it checks if the list size is less than 2 before attempting to access elements, ensuring that the loop does not execute if the list is empty.
- Example Issues:
    - If `NOTFOUND` were initially `false`, the loop (`while (NOTFOUND)`) would never execute, potentially representing oversight.
    - Similarly, for loops like `for (i==x ; i<=z; i++)`, if `x > z`, the loop will not execute.
- Conclusion: There are safe checks, but potential oversight exists for conditions that prevent execution.

5. Loop Fall-Through Consequences

- Check: The program does not have a loop that combines iteration and a Boolean condition in the same way as described in the example. Each loop's exit condition is managed explicitly.
- Example Analysis:
    - If a loop is controlled by an iterator and a condition (e.g., a search loop), and if the condition (`NOTFOUND`) never becomes `false`, the loop may run indefinitely.
- Conclusion: The program avoids this pitfall by defining clear conditions for loop exits.

6. Off-By-One Errors

- Check: The use of `for` loops and `while` constructs is straightforward in the code. For instance, `for (cellit i = c.begin()+1; i != c.end(); ++i)` iterates through elements without causing off-by-one errors.
- Example Analysis: If one mistakenly uses `for (int i=0; i<=10;i++)`, it counts to 11 instead of 10.
- Conclusion: The implementation does not contain off-by-one errors as all loops count correctly.

7. Statement Groups or Code Blocks

- Check: The code utilizes appropriate scopes and has closing brackets for each opened block. Each function and condition is encapsulated correctly, conforming to the language's block structure.
- Conclusion: There is proper grouping of statements, and the brackets match up correctly.

Category F: Interface Errors

1. Parameter Count and Order

- Check: Each module has a defined number of parameters that align with the arguments sent during calls. For example, the `proc_add` function is called with two arguments that match the expected parameters.
- Conclusion: The number of parameters received by each module matches the arguments sent, and the order is correct.

2. Parameter Attributes

- Check: The attributes (data types and sizes) of parameters in each function match those of the corresponding arguments. For instance, if a function expects integers, it receives integers from the calling functions.
- Conclusion: The attributes of parameters and arguments are consistent across all modules.

3. Units System

- Check: The units used in parameters match the units of the arguments passed. For instance, if a function is designed to work with lengths, it does not mix inches with centimeters.
- Conclusion: There are no mismatches in the units system for parameters and arguments.

4. Argument Count in Module Calls

- Check: When arguments are transmitted from one module to another, the number of arguments matches the expected parameters in the called module. For instance, the `eval` function passes the correct number of arguments to each procedure.
- Conclusion: The argument count aligns with the expected parameters for each called module.

5. Attributes of Transmitted Arguments

- Check: Each argument transmitted matches the attributes of the corresponding parameter in the receiving module. For example, if a parameter requires a list, the argument provided is indeed a list.

- Conclusion: The attributes of transmitted arguments are consistent with the expected parameters in the receiving modules.

6. Units System for Transmitted Arguments

- Check: The units system of each transmitted argument corresponds with the units of the corresponding parameter in the receiving module. For instance, if one module operates on lengths in meters, the arguments provided to it should also be in meters.
- Conclusion: There are no discrepancies in the units system for transmitted arguments and their corresponding parameters.

7. Built-in Function Invocation

- Check: The built-in functions invoked in the code receive the correct number, attributes, and order of arguments. For example, when calling mathematical functions, the arguments are provided in the required format.
- Conclusion: The invocation of built-in functions is correct regarding the number, attributes, and order of arguments.

8. Input-Only Parameters

- Check: There are no parameters within the subroutines that are unintentionally altered if they are meant to be input-only. Each function treats its parameters as local and does not modify input values.
- Conclusion: Input parameters are not altered within the subroutines.

9. Global Variables

- Check: If global variables are used, they are defined consistently across all modules that reference them. Each module that accesses a global variable recognizes its type and attributes correctly.
- Conclusion: Global variables maintain the same definition and attributes across modules.

Category G: Input/Output Errors:

1. File Attributes

- Check: If files are explicitly declared, confirm that their attributes (such as file type, access mode, etc.) are correct and align with their intended use.
- Conclusion: The attributes of the explicitly declared files should be reviewed to ensure they match the requirements (e.g., read, write, binary, text).

2. OPEN Statement Attributes

- Check: Verify that the attributes specified in the file's OPEN statement (like mode, path, etc.) are correct and correspond to the declared file attributes.
- Conclusion: The OPEN statements must be examined to confirm they use the correct parameters, such as the intended access mode (read/write).

3. Memory Availability

- Check: Assess whether there is sufficient memory available to hold the entire contents of the file being read by the program. This may include checking system resources or using memory profiling tools.
- Conclusion: Ensure that the program checks memory availability and can handle cases where insufficient memory is available.

4. File Opening

- Check: Confirm that all files have been opened before any read or write operations are performed. This includes checking any conditions that may prevent files from being opened.
- Conclusion: All files should be explicitly opened prior to use, and the code should include checks to verify successful file opening.

5. File Closing

- Check: Verify that all files are closed after their operations are completed. This prevents resource leaks and ensures that all data is properly saved.
- Conclusion: Ensure that file closing statements are present and executed after file operations are complete.

6. End-of-File Handling

- Check: Assess whether the program correctly detects and handles end-of-file (EOF) conditions. This includes checking for EOF before attempting to read from a file and handling it gracefully.
- Conclusion: The program should include checks for EOF and handle these conditions appropriately to avoid errors during file processing.

7. I/O Error Handling

- Check: Determine whether I/O error conditions (like file not found, permission denied, etc.) are handled correctly. This may involve checking return codes or exceptions from file operations.
- Conclusion: The program should include robust error handling for I/O operations, ensuring that any errors are logged or communicated effectively.

8. Text Errors

- Check: Review any text that is printed or displayed by the program for spelling or grammatical errors. This includes messages to the user or output logs.
- Conclusion: A thorough review should be conducted to ensure that all printed or displayed text is free from spelling or grammatical errors, enhancing the user experience.

Category H: Other Checks:

1. Cross-Reference Listing

- Check: If the compiler generates a cross-reference listing of identifiers, examine it for variables that are either never referenced or referenced only once. These may indicate unnecessary variables that can be removed to improve code clarity and efficiency.
- Conclusion: Review the cross-reference listing and identify any variables that are redundant or not utilized in the code.

2. Attribute Listing

- Check: If the compiler produces an attribute listing, inspect the attributes of each variable. Ensure that no unexpected default attributes (such as data type or scope) have been assigned that might lead to runtime errors or unexpected behavior.
- Conclusion: Validate that all variable attributes are explicitly defined and that no unintended defaults are applied.

3. Warning and Informational Messages

- Check: After successful compilation, review any “warning” or “informational” messages produced by the compiler. Warnings indicate potential issues in the code that could lead to errors or undesired behavior, while informational messages may highlight performance optimizations or undeclared variables.
- Conclusion: Each warning and informational message should be addressed and corrected if necessary, ensuring the program adheres to best practices and runs as intended.

4. Input Validity Checks

- Check: Assess whether the program or module performs adequate input validation. Robust programs should check input for correctness and handle invalid inputs gracefully to avoid unexpected behavior or crashes.
- Conclusion: Ensure that the code includes necessary checks for input validity, including range checks, type checks, and error handling for invalid inputs.

5. Missing Functions

- Check: Determine if there are any essential functions missing from the program that could enhance its functionality or performance. This could involve checking for incomplete implementations or functions that would optimize code efficiency.
- Conclusion: Review the program’s requirements and functionality to identify any missing functions that need to be implemented to meet the desired specifications.

1. How many errors are there in the program? Mention the errors you have identified.

To accurately answer this, you would typically need to analyze the specific code in question. However, in a general context, common types of errors you might identify during program inspection include:

- Syntax Errors: Issues in code structure, such as missing semicolons or mismatched brackets.
- Logical Errors: Flaws in the algorithm that lead to incorrect outputs, despite the program running without crashes.
- Control-Flow Errors: Issues with loops or conditional statements, such as infinite loops or unreachable code.
- Interface Errors: Mismatches in the number and type of parameters in function calls.
- Input/Output Errors: Problems with file handling, such as failing to open or close files properly, or not checking for end-of-file conditions.
- Other Errors: Unused variables, warning messages from the compiler, and lack of input validation.

To provide a precise count of errors, one would have to conduct a thorough inspection of the specific program code.

2. Which category of program inspection would you find more effective?

The Data-Declaration Errors category would be more effective for this code. Ensuring that all variables are explicitly declared and understood can prevent many common issues, especially in a dynamically typed language like JavaScript. This helps in maintaining clarity in the code, reducing ambiguity regarding variable scopes and types.

3. Which type of error you are not able to identify using the program inspection?

There are certain types of errors that may not be easily caught through inspection, including:

- Concurrency issues: Errors like race conditions or deadlocks might not be immediately visible in sequential code inspections without a deep understanding of how threads or asynchronous calls interact.
- Performance-related issues: Certain performance bottlenecks or inefficient algorithms are not always clear from inspection alone. Profiling and performance testing are required to detect such problems.
- Semantic errors: A logic flaw that arises from misunderstanding the problem requirements or applying the wrong algorithm (even if the code is syntactically correct) would not be evident from basic program inspection.

4. Is the program inspection technique worth applying?

Yes, program inspection is definitely worth applying, especially during the earlier stages of development. It can catch many syntactic and logical errors before they become costly or time-consuming bugs to fix.

Benefits include:

- **Identifying common issues like uninitialized variables or incorrect data references.**
- **Helping ensure the program aligns with design specifications by catching logic errors early.**
- **Enhancing the reliability of the program by detecting subtle errors that automated testing might miss.**

II. CODE DEBUGGING: Debugging is the process of localizing, analyzing, and removing suspected errors in the code (Java code given in the .zip file)

➔ **Armstrong.txt:**

1. Data Reference/Initialization Errors:

- In the `while` loop, the line `remainder = num / 10;` should actually extract the last digit. The correct operation should be `num % 10` to get the remainder (last digit), not `num / 10`, which gives the quotient.

2. Computation Errors:

- The computation for `remainder = num / 10;` is incorrect. It should be `remainder = num % 10;` since we need the last digit of the number.

3. Control Flow/Comparison Errors:

- There doesn't seem to be an issue with the control flow, but the comparison of `check == n` is correct.

4. I/O Errors:

- Input and output seem correct; it prints whether the number is an Armstrong number or not based on the result.

➔ GCD and LCM:

Data Reference/Initialization Errors:

- In the GCD calculation, the variables a and b are initialized based on incorrect conditions. The lines:
- Control Flow/Comparison Errors:
  - The logic in the LCM calculation has a control flow issue where the conditions to find the LCM are incorrectly compared, resulting in an infinite loop or incorrect output.
- I/O Errors:
  - There are no input/output errors. The program correctly takes input from the user and outputs the results for GCD and LCM.

➔ **Knapsack**

● There is one error in the program, as identified above.

● To fix this error, you would need one breakpoint at the line: int option1 = opt[n][w]; to ensure n and w are correctly used without unintended increments.

➔ **Magic Number**

● There are two errors in the program, as identified above.

● To fix these errors, you would need one breakpoint at the beginning of the inner while loop to verify the execution of the loop. You can also use breakpoints to check the values of num and s during execution

➔ **Merge Sort**

● There are multiple errors in the program, as identified above.

● To fix these errors, you would need to set breakpoints to examine the values of left, right, and array during execution. You can also use breakpoints to check the values of i1 and i2 inside the merge method.

➔ **Multiply Matrices**

● There are multiple errors in the program, as identified above.

● To fix these errors, you would need to set breakpoints to examine the values of c, d, k, and sum during execution. You should pay particular attention to the nested loops where the matrix multiplication occurs.

➔ **Quadratic Probing**

● There are three errors in the program, as identified above.

● To fix these errors, you would need to set breakpoints and step through the code while examining variables like i, h, tmp1, and tmp2. You should pay attention to the logic of the insert, remove, and get methods

➔ **Sorting Array**

● There are two errors in the program as identified above.

● To fix these errors, you need to set breakpoints and step through the code. You should focus on the class name, the loop conditions, and the unnecessary semicolon.

➔ Stack Implementation

● There are three errors in the program, as identified above.

● To fix these errors, you would need to set breakpoints and step through the code, focusing on the push, pop, and display methods. Correct the push and display methods and add the missing pop method to provide a complete stack implementation.

➔ **Tower of Hanoi**

● There is one error in the program, as identified above.

● To fix this error, you need to replace the line: doTowers(topN ++, inter--, from+1, to+1);

● with the correct version: doTowers(topN - 1, inter, from, to);