Assignment 4: Static Semantics (or Contextual Analysis)

COMP3131/9102: Programming Languages and Compilers

Term 1, 2020

Worth: 30/100 marks

Due: 11:59pm Monday 13 Aprial 2020

Revision Log

1. Specification

You are to implement a semantic or contextual analyser that checks that the program conforms to the source language's context-sensitive constraints (i.e., static semantics) according to the VC Language Definition. This part of the compilation process is referred to as the *semantic or contextual analysis*.

There are two types of context-sensitive constraints:

- *Scope rules:* These are the rules governing declarations (*defined occurrences* of identifiers) and *applied occurrences* of identifiers.
- *Type rules:* These are the rules that allow us to infer the types of language constructs and to decide whether each construct has a valid type.

Therefore, the semantic analysis consists of two subphases:

- *Identification:* applying the scope rules to relate each applied occurrence of an identifier to its declaration, if any.
- *Type checking:* applying the type rules to infer the type of each construct and comparing that type with the expected type in the context.

This assignment involves developing a visitor class (named `Checker`) that implements the set of visitor methods in the interface `VC.ASTs.Visitor.java`. Your semantic analyser will be a *visitor object* that performs both identification and type checking in one pass by visiting the AST for the program being compiled in the depth-first traversal.

In the case of ill-typed constructs, appropriate error messages as specified below must be reported.

As before, if no lexical, syntactic or semantic error is found, your compiler should announce success by printing:

```
Compilation was successful.
```

Otherwise, the following message should be printed:

```
Compilation was unsuccessful.
```

2. Identification

This subphase of the semantic analyser has been implemented for you. Identification relates each applied occurrence of an identifier to its declaration, if any, by applying the VC's scope rules. The standard method of implementing this subphase is to employ a symbol table that associates identifiers with their attributes. In the VC compiler, the attribute for an identifier is represented by a pointer (an inherited attribute) to the subtree that represents the declaration (GlobalVarDecl, LocalVarDecl or FuncDecl) of the identifier. This attribute is represented by the instance variable `decl` in VC.ASTs.Ident.java:

```
package VC.ASTs;
import VC.Scanner.SourcePosition;
public class Ident extends Terminal {
  public AST decl;
  public Ident(String value , SourcePosition position) {
    super (value, position);
    decl = null;
  }
  public Object visit(Visitor v, Object o) {
    return v.visitIdent(this, o);
  }
}
```

There is only one symbol table organised as a stack for storing the identifiers in all scopes. Two classes are used:

- `VC.Checker.IdEntry.java`: defining what a symbol table entry looks like.
- `VC.Checker.SymbolTable.java`: defining all methods required for symbol table management.

The symbol table methods are called at the following visitor methods of the class `Checker`:

- `visitGlobalVarDecl`, `visitLocalVarDecl` and and `visitFuncDecl`: whenever the semantic analyser visits the declaration at a subtree, it will call `insert` to enter the identifier, its scope level and a pointer to the subtree into the symbol table.

  To detect duplicate declarations using the same identifier, you call the method `retrieveOneLevel`. This method returns a pointer to the identifier entry if the identifier was declared before in the current scope and null otherwise.
- `visitIdent:` whenever the semantic analyser visits an applied occurrence of an identifier `I`, it will call `retrieve` with the identifier `I` and thus retrieves the pointer to the subtree representing its declaration. It will then decorate the identifier node for `I` by establishing a link to this declaration. This link is null if no declaration is found. This fact will be used by you to detect undeclared variables.
- `visitCompoundStmt:` whenever the semantic analyser visits a block, it calls `openScope` at the start of the block to open a new scope and `closeScope` at the end of the block to close the current scope.

The symbol table is not empty before the semantic analysis for the program begins. Many languages contain a standard collection of pre-defined constants, variables, types and functions that the programmer can use without having to introduce themselves. The VC standard environment includes only 11 built-in functions and a few primitive types. The "declarations" of these functions do not appear in the AST for the program being compiled! In order to make it possible for a link from a call, say, `putIntLn`, to be established with its "declaration", the following AST for the function is constructed by the VC compiler:

The name of the parameter is insignificant and is thus set to "".

The ASTs for the other eight built-in functions are similarly constructed.

Before analysing the program, the semantic analyser initialises the symbol table with the identifiers for the 11 functions as follows:

Ident	Level	Attr
getInt	1	ptr to the getInt AST
putInt	1	ptr to the putInt AST
putIntLn	1	ptr to the putIntLn AST
getFloat	1	ptr to the getFloat AST
putFloat	1	ptr to the putFloat AST
putFloatLn	1	ptr to the putFloatLn AST
putBool	1	ptr to the putBool AST
putBoolLn	1	ptr to the putBoolLn AST
putString	1	ptr to the putString AST
putStringLn	1	ptr to the putStringLn AST
putLn	1	ptr to the putLn AST
the identifiers in the program		

stack

You are required to read

- `VC.Checker.IdEntry.java`,
- `VC.Checker.SymbolTable.java`,
- `VC.StdEnvironment.java`, and
- the method `establishEnvironemnt` in `AST.Checker.Checker.java`

to ensure your understanding of the identification subphase.

3. Error Messages

On detecting some semantic errors, your checker must print some error messages.

Your error messages must be taken from the following array that is already defined for you in `Checker.java`:

```
private String errMesg[] = {
  "*0: main function is missing",
  "*1: return type of main is not int",

  // defined occurrences of identifiers
  // for global, local and parameters
  "*2: identifier redeclared",
  "*3: identifier declared void",
  "*4: identifier declared void[]",

  // applied occurrences of identifiers
  "*5: identifier undeclared",

  // assignments
  "*6: incompatible type for =",
  "*7: invalid lvalue in assignment",

   // types for expressions
  "*8: incompatible type for return",
  "*9: incompatible type for this binary operator",
  "*10: incompatible type for this unary operator",

   // scalars
   "*11: attempt to use an array/fuction as a scalar",

   // arrays
   "*12: attempt to use a scalar/function as an array",
   "*13: wrong type for element in array initialiser",
   "*14: invalid initialiser: array initialiser for scalar",
   "*15: invalid initialiser: scalar initialiser for array",
   "*16: excess elements in array initialiser",
   "*17: array subscript is not an integer",
   "*18: array size missing",


   // functions
   "*19: attempt to reference a scalar/array as a function",

   // conditional expressions in if, for and while
  "*20: if conditional is not boolean",
  "*21: for conditional is not boolean",
  "*22: while conditional is not boolean",

  // break and continue
  "*23: break must be in a while/for",
  "*24: continue must be in a while/for",

  // parameters
  "*25: too many actual parameters",
  "*26: too few actual parameters",
  "*27: wrong type for actual parameter",

  // reserved for errors that I may have missed
  "*28: misc 1",
  "*29: misc 2",

  // the following are not required
  "*30: statement(s) not reached",
  "*31: missing return statement",
};
```

The error messages 28 -- 29 are reserved for some errors that I might have missed. They can be added in these slots later.

If there there is a type error detected at the subtree rooted at "ast", you can report an error message as follows:

```
 reporter.reportError("errMesg[index] +. blah blah ", "", ast.position);
```

See `ErrorReporter.java` regarding how the position information will be printed.

It is also possible to pass a non-empty string as the 2nd argument so that it is printed in the position marked by % as follows:

```
 reporter.reportError(errMesg[index] + ": %", "blah blah", ast.position);
```

Again you can read `ErrorReporter.java` to find out how this works.

You are not allowed to modify the error message array `errMesg`. On detecting a semantic error, the error message your checker reports must contain one of the error strings defined in `errMesg` as a substring. As demonstrated in the supplied solution files, `t1.sol and t2.sol`, you can certainly add more "words" in an *official* error message to make it more informative.

To avoid printing a cascade of spurious error messages, you are advised to use the simple error recovery technique explained in Solution 2 to Question 2 in Week 9 Tutorial. Essentially, the compiler assigns `StdEnvironment.errorType` to every ill-typed expression and prints an error message. However, the compiler will refrain from printing any error messages for an expression if any of its subexpressions has the type `StdEnvironment.errorType`.

Personally, the Java compiler's error handling is very good. It is helpful to run Java on similar test cases to examine how various semantic errors are detected and reported.

4. Writing Your Type Checker

Set up your compiling environment as specified in Assignment 1 spec.

Download and install the supporting classes for this assignment as follows:

```
1. Copy ~cs3131/VC/Checker/Checker.zip into your VC directory
2. Set your current working directory as VC.
3. Extract the bundled files in the zip file as follows:
   unzip Checker.zip
```

The files bundled in this zip archive are listed below. If you have trouble in handling Checker.zip, you can also download the supporting classes individually **all** from `~cs3131/VC/Checker` and install them into the respective directories (i.e., packages) as specified below:

```
The Checker package:
===================

Checker.java:        semantic analyser skeleton
IdEntry.java:        symbol table entry
SymbolTable.java:    symbol table management

Test Files:           t1.vc and t2.vc
Solution Files:       t1.sol and t2.sol

The VC package:
==============

StdEnvironment.java: The VC language environment
vc.java:             main compiler module (different from that in Assignment 3)
```

Your static analyser will use `ErrorReporter.java` you installed in your VC directory in Assignment 1. If you have not done so or have lost the file, copy it from ~cs3131/VC.

You need to read the VC Language Definition to find out all context-sensitive constraints that should be enforced. Here is a list of typical checks:

1. All identifiers must be declared before used.
2. An identifier cannot be declared more than once in a block.
3. No identifier can be declared to have the type void or void [].
4. Operands must be type compatible with operators.
5. Assignment must be type compatible.
6. A function must be called with the right number of arguments, and in addition, the type of an actual parameter must be assignment compatible with the type of the corresponding formal parameter.
7. The type of a returned value must be assignment compatible with the result type of the corresponding function.
8. The "conditional expression" in a **for/if/while** statement must evaluate to a boolean value. Therefore, the following program

   ```
   while (1)
      // do something
   ```

   should cause the error message numbered 21 to be printed.
9. **break** and **continue** must be contained in a while/for. By introducing an instance variable to record the nesting depth of a while statement, both checks can be done in a few lines.
10. A array name itself can only be used as an argument in a function call:

    ```
    void f(int x[]) { }
    int main() {
      f(x);  //  OK
      x + 1; //  ERROR
      return 0;
    }
    ```

11. An array variable can only be indexed by integer expressions.
12. Array initialisers must be used for arrays only, by following the rules given in the VC language specification.

In addition, if a return statement is immediately followed by a statement (other than ";"), you may report an error. But this is optional.

It is also optional to detect if a function returning a non-void type always contains a return statement in every possible execution path. Run the Java compiler on some test cases to see how well this is done.

In the case of expressions, your type checker will infer the type of an expression and store the type (as a synthesised attribute) in the corresponding expression node. This synthesised attribute is represented by the instance variable `type` defined in the abstract class `VC.ASTs.Expr.java` and inherited in all its concrete expression classes and the instance variable `type` defined in the abstract class `VC.ASTs.Var.java` and inherited in its concrete class `VC.ASTs.SimpleVar.java`.

The synthesised attribute type for variables and expressions will be evaluated bottom-up. You can pass the attribute bottom-up by letting all corresponding visitor methods return `ast.type`.

Note that `type` is a synthesised attribute, which is computed in the depth-first traversal of the AST.

All the following six AST classes:

```
IntType  FloatType  BooleanType StringType  VoidType  ErrorType
```

contain the methods `equals` and `assignable`, which will be used to compare if two types are identical and if two types are assignment compatible, respectively. Let *e1Type* and *e2Type* be the types of two expressions. Then

- *e1Type.equals(e2Type)* returns true iff both types are identical.
- *e1Type.assignable(e2Type)* returns true iff *e2Type* is assignment compatible to *e1Type* .

In addition, both tests return true if *e1Type* or *e2Type* is errorType. This tactic avoids generating too many spurious errors.

In the case when an array name is passed as an argument in a function call, don't rely on the method *assignable*! You need to handle this as a special case.

Accordingly, the standard environment contains the six pre-defined types:

```
StdEnvironment.intType
StdEnvironment.floatType
StdEnvironment.booleanType
StdEnvironment.stringType
StdEnvironment.voidType
StdEnvironment.errorType
```

They are declared in `VC.StdEnvironment.java` and initialised in the method `establishEnvironment` of the class `Checker`. The first five are already used in the partially finished class `Checker`.

You are given only two test files, which covers all semantic errors defined in the error message table given in Section 3.

`Checker.java` does not compile. The Java compiler will complain its being an abstract class unless have have implemented all the missing visitor methods.

You need to add roughly 500 lines of code to obtain a static analyser that works beautifully for the VC language. You are free to modify the supplied visitor methods in Checker.java, although most of them should work already. However, it is not necessary to modify the constructor and the method establishEnvironment.

5. Decorating ASTs

The results of semantic analysis is recorded by *decorating* the AST as explained above. In summary, the following decorations are used:

- Each Ident node is decorated by establishing a link to its declaration if any and to null otherwise.
- Each SimpleVar node or any of expression nodes is decorated by setting its type field to the type of the expression.

6. Type Coercions

In addition to performing the identification and type checking, the semantic analyser also handles type coercions to facilitate the final code generation.

In the language, type coercions will go only from **int** to **float**. Let *x*:**T** denote the fact that the variable or expression *x* is of type **T**. In the following four cases, the expression *e* must be converted to **float:**

- an assignment v:**float** = e:**int.**
- a mixed-mode binary expression e_1:**int** <op> e_2:**float** (or e_1:**float** <op> e_2:**int.**)
- a call expression f(... e:**int** ...), where the corresponding parameter declaration is `void/int/float foo(... float f ...) { ... }`.
- an expression in a return statement **return** e:**int**, where the return type for the corresponding function is **float**.

Type coercions are achieved by introducing extra nodes into the program's AST in the three visitor methods `visitAssignExpr`, `visitBinaryExpr`, `visitArg` and `visitReturnStmt`. A special unary operator *i2f* is used to convert an integer value to a floating-point value. For example, assignment coercions can be realised by including the following code in `visitBinaryExpr`:

```
        Operator op = new Operator("i2f", dummyPos);
        UnaryExpr eAST = new UnaryExpr(op, ast.E, dummyPos);
        eAST.type = StdEnvironment.floatType;
        ast.E = eAST;
```

This will change the original BinaryExpr AST from

```
                BinaryExpr
                  / \
                 /   \
               v   ExprNode for e
```

to

```
                BinaryExpr
                  / \
                 /   \
               v    UnaryExpr
                       /  \
                      /    \
                 i2f   ExprNode for e
```

The other three kinds of coercions are implemented using similar code.

In addition to performing the type coercions discussed above, your static analyser is required to replace each overloaded operator with an appropriate non-overloaded operator to indicate whether the intended operation is an integer or a floating-point operation.

Some operators such as + and - are overloaded in the sense that they can be applied to either a pair of integers or a pair of floating-point numbers. Every operator *<op>* is associated with two non-overloaded operators: *i<op>* for integer operations and *f<op>* for floating-point operations. For example, the two non-overloaded operators for + are i+ and f+, the two non-overloaded operators for <= are i<= and f<=, and so on.

In JVM, the boolean values are represented by integer values. Therefore, an operator *<op>* acting on boolean values is represented using *i<op>*. The operators that can act on boolean values are &&, ||, !, == and !=.

It is straightforward to resolve the overloaded operators:

1. **&&, || and !**
   These operators can act only on boolean values and will always be replaced by i&&, i|| and i!, respectively. The code required is:

   ```
   ast.O.spelling = "i" + ast.O.spelling;
   ```

2. **+ - * / < <= > >= (where + and - are both unary and binary)**
   These operators must be replaced with appropriate non-overloaded operators. A given expression is evaluated using the floating-point operation as long as one of the operands is of type **float**. The code required is:

   ```
   ast.O.spelling = "f" + ast.O.spelling;
   or
   ast.O.spelling = "i" + ast.O.spelling;
   ```

   whichever is appropriate.
3. **== and !=**
   These two operators can be applied to either a pair of boolean values or a pair of integers or a pair of floating-point numbers. In the first case, the integer operations should be used ((after type coercion, if required, as explained in Section 7.1 of the VC Spec). The other two cases are handled similarly as in Case 2.

Type coercions will not be assessed for this assignment. But it will be accessed in Assignment 5 -- your code generator would not work properly if type coercions are incorrect.

The total number of lines required for type coercions is about 25 lines, with the same five lines repeated a few times.

The following example is used to illustrate type coercions for assignment statements. Here are the ASTs before and after type coercions are performed:

```
int main() {
  float f;
  int i;
  f = i + 1;
  return 0;
}
```

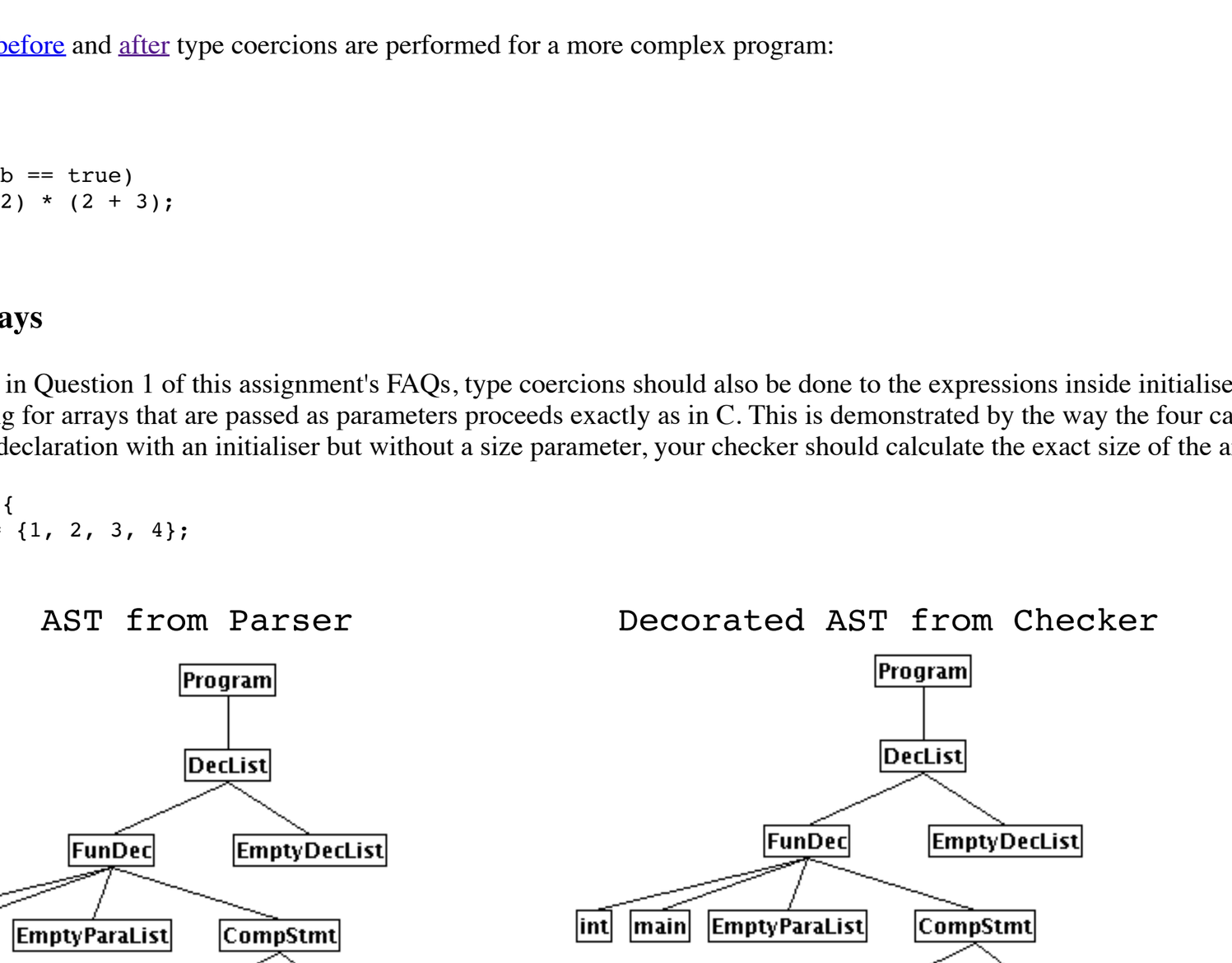
Here are the ASTs before and after type coercions are performed for a more complex program:

```
int main() {
  float x;
  boolean b;
  if (x != 0 && b == true)
    x = (+1.0 + 2) * (2 + 3);
  return 0;
}
```

7. More on Arrays

- As explained in Question 1 of this assignment's FAQs, type coercions should also be done to the expressions inside initialisers.
- Type checking for arrays that are passed as parameters proceeds exactly as in C. This is demonstrated by the way the four calls in t2.vc are type-checked.
- For an array declaration with an initialiser but without a size parameter, your checker should calculate the exact size of the array from the initialiser and then modify the the AST from the parser to fill the missing size parameter. This is demonstrated below by an example:

  ```
  int main() {
    int x[] = {1, 2, 3, 4};
  }
  ```



Note that EmptyExpr for ArrType has been replaced by IntExpr --> 4, where 4 is the size of the array x.

8. The Parser

If you want to use our parser in case yours does not work properly, copy `~cs3131/VC/Parser/Parser-Sol.zip` to your Parser directory and type:

```
unzip Parser-Sol.zip
```

This installs the class `Parser.class` under package `VC.Parser`.

It is not necessary to understand how this parser works. Your type checker will only work on the AST constructed for the program by the parser.

9. Testing Your Type Checker

For this assignment, there does not seem to be a need to produce by default both a linearised AST and a reconstructed VC program every time when the compiler is run. The compiler options have been changed slightly as follows:

```
[jxue@daniel Checker]$ java VC.vc
======= The VC compiler =======

[# vc #]: no input file

Usage: java VC.vc [-options] filename

where options include:
        -d [1234]           display the AST (without SourcePosition)
                            1:  the AST from the parser (without SourcePosition)
                            2:  the AST from the parser (with SourcePosition)
                            3:  the AST from the checker (without SourcePosition)
                            4:  the AST from the checker (with SourcePosition)
        -t [file]           print the (non-annotated) AST into
                            (or filename + "t" if  is unspecified)
        -u [file]           unparse the (non-annotated) AST into
                            (or filename + "u" if  is unspecified)
```

10. Marking Criteria

Your type checker will be assessed only by examining how it handles various semantically legal and illegal programs. Only syntactically legal programs will be used.

Your type checker will not be marked up or down for how well it recovers from semantic errors and how well it avoids spurious error messages.

Small test cases will be designed so that, in general, each test case has only one semantic error. In the case of multiple semantic errors in a test case, these errors are designed to be independent of each other, as exemplified by the supplied test cases t1.vc and t2.vc. If necessary, the error messages from your type checker will also be examined manually.

Therefore, we will have to use "fgrep" rather than "diff" to mark this assignment. As an example, on the following program:

```
int main() {
  int i;
  float j;
  i = j = 1.0 + true;
  return 0;
}
```

the output from our checker is:

```
% java VC.vcchecker y.vc
======= The VC compiler =======

Pass 1: Lexical and syntactic Analysis
Pass 2: Semantic Analysis
ERROR: 5(9)..5(18): *9: incompatible type for this binary operator: +
Compilation was unsuccessful.
```

Two more error messages are possible:

- j = 1.0 + true: incompatible type for =
- i = j: incompatible type for =

Our checker regards these as spurious errors and has refrained from reporting them. Whether your checker chooses to report them or not will not be marked. In this particular case, error message number 9 must be reported.

The positional information will not be assessed.

As before, there are no subjective marks.

11. Submitting Your Checker

```
give cs3131 checker Checker.java (and vc.java if you have modified it)
```

12. Late Penalties

This assignment is worth 30 marks (out of 100). You are strongly advised to start early and do not wait until the last minute. You will lose 6 marks for each day the assignment is late.

Extensions will not be granted unless you have legitimate reasons and have let the LIC know ASAP, preferably one week before its due date.

13. Plagiarism

As you should be aware, UNSW has a commitment to detecting plagiarism in assignments. In this particular course, we run a special program that detects similarity between assignment submissions of different students, and then manually inspect those with high similarity to guarantee that the suspected plagiarism is apparent.

If you receive a written letter relating to suspected plagiarism, please contact the LIC with the specified deadline. While those students can collect their assignments, their marks will only be finalised after we have reviewed the explanation regarding their suspected plagiarism.

This year, CSE will adopt a uniform set of penalties for the programming assignments in all CSE courses. There will be a range of penalties, ranging from "0 marks for the assessment item", "negative marks for the value of the assessment item" to "failure of course with 0FL."

Here is a statement of UNSW on plagiarism:

Plagiarism is the presentation of the thoughts or work of another as one's own.*

Examples include:

- direct duplication of the thoughts or work of another, including by copying material, ideas or concepts from a book, article, report or other written document (whether published or unpublished), composition, artwork, design, drawing, circuitry, computer program or software, web site, Internet, other electronic resource, or another person's assignment without appropriate acknowledgement;
- paraphrasing another person's work with very minor changes keeping the meaning, form and/or progression of ideas of the original;
- piecing together sections of the work of others into a new whole;
- presenting an assessment item as independent work when it has been produced in whole or part in collusion with other people, for example, another student or a tutor; and,
- claiming credit for a proportion a work contributed to a group assessment item that is greater than that actually contributed.†

Submitting an assessment item that has already been submitted for academic credit elsewhere may also be considered plagiarism. Knowingly permitting your work to be copied by another student may also be considered to be plagiarism. An assessment item produced in oral, not written form, or involving live presentation, may similarly contain plagiarised material.

The inclusion of the thoughts or work of another with attribution appropriate to the academic discipline does *not* amount to plagiarism.

Students are reminded of their Rights and Responsibilities in respect of plagiarism, as set out in the University Undergraduate and Postgraduate Handbooks, and are encouraged to seek advice from academic staff whenever necessary to ensure they avoid plagiarism in all its forms.

The Learning Centre website is the central University online resource for staff and student information on plagiarism and academic honesty. It can be located at:

www.lc.unsw.edu.au/plagiarism

The Learning Centre also provides substantial educational written materials, workshops, and tutorials to aid students, for example, in:

- correct referencing practices;
- paraphrasing, summarising, essay writing, and time management;
- appropriate use of, and attribution for, a range of materials including text, images, formulae and concepts.

Individual assistance is available on request from The Learning Centre.
Students are also reminded that careful time management is an important part of study and one of the identified causes of plagiarism is poor time management. Students should allow sufficient time for research, drafting, and the proper referencing of sources in preparing all assessment items.

* Based on that proposed to the University of Newcastle by the St James Ethics Centre. Used with kind permission from the University of Newcastle.
† Adapted with kind permission from the University of Melbourne.

Have fun!

Jingling Xue
Last updated 04/05/2020 10:03:22