Assignment 4: Static Semantic Analysis

CSI4104 Compiler Design, Fall 2021
Document Version 1.6

Due date

- Please **do not submit** Assignment 4 before Sunday, Dec 12, 2021! The grace period of Assignment 3 ends on the previous day, and any submission before the end of the grace period will overwrite your Assignment 3 submission!

- The due date for Assignment 4 is Tuesday, Dec 14, 2021, 23:00.

- **Please note:** there is no grace period for Assignment 4. (To schedule Assignment 4 before Assignment 5 and the semester end.)

1 Introduction

In this assignment we will perform static semantic analysis of the ASTs that we generated in Assignment 3. Specifically, we check whether a program conforms to the static semantic specification of the MiniC language.

To keep the workload manageable at this stage of the semester, you are provided with a Visitor for the complete MiniC language, where you will have to fill in your code at specific places. Semantic Analysis of arrays will only be done as a bonus assignment. The assignment walks you through the complete semantic analysis process, to ensure your understanding of this part of a compiler. Emphasis is on comprehension, the coding effort on your side will be less than 100 lines of Java code.

2 Specification

MiniC static semantic analysis ensures two types of constraints, namely (1) scope rules and (2) type rules. Therefore static semantic analysis consists of two subphases:

1) *Identification:* Here we apply the scope rules of MiniC (discussed in the lecture) to relate each applied occurrence of an identifier to its declaration.

2) *Type checking*: we apply the type rules of MiniC to determine the type of each construct. We compare that the type we have found is compatible to the type that is expected in that context.

This assignment involves developing a visitor class in file `SemanticAnalysis.java` that implements the visitor methods from the visitor interface in the file `MiniC.AstGen.Visitor.java`. Our semantic analyzer is a visitor object that performs type checking and identification while it visits the AST in a depth-first traversal. In case of ill-typed constructs, appropriate error messages must be issued. As before, if no lexical,

syntactic or semantic error is found, the compiler should print "`Compilation was successful.`", otherwise it should print "`Compilation was unsuccessful.`".

3 Identification

A scope stack has already been implemented for you. Identification relates each applied occurrence of an identifier to its declaration, by applying the scope rules of MiniC. We use the scope stack to associate identifiers with their declaration. In the MiniC compiler, the identifier's declaration is represented as a pointer to the AST subtree (`FunDecl`, `VarDecl`, or `FormalParamDecl`) that represents that declaration. This attribute is represented by the instance variable *declAST* in `MiniC.AstGen.ID.java`:

```
public class ID extends Terminal {

    public AST declAST;

    public ID (String Lexeme, SourcePos pos) {
     super (pos);
     this.Lexeme = Lexeme;
     declAST = null;
    }

    public void accept(Visitor v) {
     v.visit(this);
    }

}
```

During identification, we use the scope stack to store identifiers and their declarations for later lookup. (We perform look-ups when we encounter applied occurrences of identifiers.) The scope stack consists of two classes.

1) `MiniC.SemanticAnalysis.IdEntry.java` is the internal data structure for stack entries of the scope stack.
2) `MiniC.SemanticAnalysis.ScopeStack.java` defines all methods that the scope stack supports.

The scope stack is used as follows during identification:
▪ Whenever the semantic analyzer visits an AST corresponding to a declaration (`FunDecl`, `VarDecl` and `FormalParmDecl` ASTs), it will call `enter()` to enter the identifier of the declaration (i.e., the defining occurrence of that identifier) into the scope stack. MiniC does not allow declarations of the **same** identifier within a single block. To detect duplicates (i.e., re-declarations), the scope stack's `enter()` method will return `false`

if an identifier of that name is already present in the current scope. Your code needs to check this return value to report re-declarations.

Example 1:
```
{
int f;
int f; // Error: re-declaration of f in the current block.
}
```
Another slightly more complicated example addresses the re-declaration of a formal parameter as a local variable:

Example 2:
```
void foo (int f) {
int f; // Error: re-declaration of f.
}
```

- Whenever the semantic analyzer visits an applied occurrence of an identifier I, it will call `retrieve(I)` to retrieve the declaration of I from the scope stack. The scope stack will return the pointer to the AST subtree that corresponds to I's declaration. This pointer will be stored with the ID node for I. If no declaration is found, the scope stack will return a `null` pointer. This indicates the use of an identifier that has not been declared. Of course this is a MiniC semantic error that has to be reported.
- Whenever the semantic analyzer visits a compound statement `{...}`, it calls `scopeStack.openScope()` at the start of the block to open a new scope. At the end of the block it will call `scopeStack.closeScope()` to close this scope again.

The scope stack is already pre-loaded with the entries for the Standard Environment of MiniC (many programming languages provide the programmer with a set of pre-defined types, constants and functions that the programmer can use without having to declare them). The MiniC standard environment contains 9 built-in functions and several primitive types. The declarations of these constructs do not appear in the printout of your ASTs, but you can use the MiniC compiler's `-envast` option to print the AST of the standard environment. In order to make the use of built-in functions such as `putInt()` possible from within MiniC programs, the ASTs for those functions and primitive types are constructed in the constructor of package `MiniC.StdEnvironment`. Before semantic analysis, the semantic analyzer initializes the scope stack with the identifiers for those functions and primitive types.

Identifier	Scope Stack Level	Attribute
int	1	ptr to the int type AST
bool	1	ptr to the bool type AST
float	1	ptr to the float type AST
void	1	ptr to the void type AST
getInt	1	ptr to the getInt AST
putInt	1	ptr to the putInt AST
getBool	1	ptr to the getBool AST

stack growth direction...

`putBool`	1	ptr to the putBool AST
`getFloat`	1	ptr to the getFloat AST
`putFloat`	1	ptr to the putFloat AST
`getString`	1	ptr to the getString AST
`putString`	1	ptr to the putString AST
`putLn`	1	ptr to the putLn AST
here the identifiers from the source code of the program follow...		

It is recommended that you read `MiniC.SemanticAnalysis.IdEntry.java`, `MiniC.SemanticAnalysis.ScopeStack.java`, `MiniC.StdEnvironment.java` and the constructor of `MiniC.SemanticAnalysis.SemanticAnalysis.java` to understand the details of the MiniC scope stack mechanism.

4 Error Messages

Semantic errors must be reported to the user. The file `SemanticAnalysis.java` contains predefined error messages that you should use.

private String errMsg[] = {
"#0: main function missing",
"#1: return type of main must be int",
//defining occurrences of identifiers,
//for local, global variables and for formal parameters:
"#2: identifier redeclared",
"#3: identifier declared void",
"#4: identifier declared void[]",
//applied occurrences of identifiers:
"#5: undeclared identifier",
//assignment statements:
"#6: incompatible types for =",
"#7: invalid lvalue in assignment",
//expression types:
"#8: incompatible type for return statement",
"#9: incompatible types for binary operator",
"#10: incompatible type for unary operator",
//scalars:

"#11: attempt to use a function as a scalar",
//arrays:
"#12: attempt to use scalar/function as an array",
"#13: wrong type for element in array initializer",
"#14: invalid initializer: array initializer for scaler",
"#15: invalid initializer: scalar initializer for array",
"#16: too many elements in array initializer",
"#17: array subscript is not an integer",
~~"#18: array size missing",~~ (not used)
//functions:
"#19: attempt to reference a scalar/array as a function",
//conditional expressions:
"#20: "if" conditional is not of type boolean",
"#21: "for" conditional is not of type boolean",
"#22: "while" conditional is not of type boolean",
//parameters:
"#23: too many actual parameters",
"#24: too few actual parameters",
"#25: wrong type for actual parameter"
};

If a semantic error is detected, you can report it using the ErrorReporter as follows:
`reporter.reportError(errMsg[index] + "what else you want to say", "", sourcepos);` See the class `ErrorReporter.java` for further details. You are not allowed to change the error messages in array `errMsg[]`, otherwise automated marking won't work anymore!

5 Writing Your Semantic Analyzer

We provide you with an updated version of the skeleton compiler that you should extend to create your MiniC semantic analyzer. You will find the new skeleton on the server in the archive file /opt/ccugrad/Assignment4/CSI4104_Assignment_4.tgz. Please set up your programming environment as specified in Assignment 1.

In the following, you find a description of the skeleton-files that have been added or changed compared to Assignment 3. All directories are relative to the top-level directory `CSI4104_Assignment_4`:

Package MiniC (directory MiniC)

`MiniC.java`: the new version of the compiler driver. The TreeDrawer incorporates type information into the AST. This can be helpful for debugging a semantic analyzer. `StdEnvironment.java`: The standard environment of the MiniC language.

Package Scanner (directory MiniC/Scanner)

Scanner.java: empty scanner implementation which you should **replace** by **your own scanner** from **Assignment 1**. In case your scanner from Assignment 1 does not work properly, you can use scanner classfiles provided with this assignment instead (see Section ``Provided Classfiles'' below).

Package Parser (directory MiniC/Parser)

`Parser.java:` empty Parser implementation which you should replace by your own parser from Assignment 3. In case your parser from Assignment 3 does not work properly, you can use scanner and parser classfiles provided with this assignment instead (see Section "Provided Classfiles" below).

Package SemanticAnalysis (directory MiniC/SemanticAnalysis)

`SemanticAnalysis.java`: this is the skeleton of a visitor to perform semantic analysis on MiniC ASTs. You will add your code at specific places in this file. Those places are already marked (see below).

`IdEntry.java`, `ScopeStack.java`: the files implementing the scope stack (see the section on identification).

Subdirectory tst/base/testcases: contains MiniC testcases. Several testcases are semantically invalid! **Note that testcases with a number higher than c26*.mc belong to the bonus assignment (see section below).**

Subdirectory tst/base/solutions: contains the output that your compiler should generate for each of the testcases. Note: for marking, we will only `grep` for specific "ERROR:..." messages. Whether your compiler outputs additional ERROR messages or whether the line numbers in the error messages are correct will not be checked (see section on "Marking" below).

The decorated images for the AST testcases are provided onine, like with Assignment 3, at http://elc.yonsei.ac.kr/courses/csi4104/csi4104_assignment4.html

Package TreeDrawer (directory MiniC/TreeDrawer)

Has been updated to include type information with AST nodes, e.g. <int> for an integer type, <?> for an unknown type, and <error> for an error type.

Package AstGen (directory MiniC/AstGen)

A few classes had to be extended for semantic analysis.

The following list contains typical static semantic constraints that we have to check:
 (1.)Identifiers must be declared before they are used.
 (2.)An Identifier cannot be declared more than once in a block.
 (3.)No identifier can be declared to have type `void` or type `void[]`.
 (4.)Operands in expressions must be compatible with operators.
 (5.)Left-hand and right-hand sides of assignments must be type-compatible.
 (6.)A function must be called with the correct number of arguments. The type of an actual
      parameter must be assignment-compatible with the corresponding formal parameter.
 (7.)The type of a return value must be assignment-compatible with the result type of the
      corresponding function.
 (8.)Conditions in `if/for/while` statements must be of type `bool.` For example:

<center>`while(1) {} //ERROR #22.`</center>

For an expression, the analyzer will determine the type of the expression and store it in the corresponding expression node (that's "decorating the AST"). An expression may actually consist of several nodes which form an AST subtree. The MiniC *type* attribute is evaluated bottom up in this expression subtree (i.e., type information is synthesized). For each node in the expression subtree, this synthesized type attribute is stored in the instance variable **type** that is defined in the abstract class MiniC.AstGen.Expr.java and inherited in all expression classes. For every expression tree node, you compute this type information from the type information of the children (this requires of course that children have already been visited, which is ensured by our depth-first traversal).

During semantic analysis, we often need to check whether two types are equal or whether one type can be assigned to the other type. For this purpose AST classes `IntType`, `FloatType`, `BoolType`, `StringType` and `ErrorType` contain methods `Tequal()` and `AssignableTo()`.

Let e1Type and e2Type be the types of two expressions.
- `e1Type.Tequal(e2Type)` returns true iff e1Type and e2Type are identical. Example: If both e1Type and e2Type are of type int, `Tequal()` will return true. If one type is int and the other type is float, `Tequal()` will return false.
- `e1Type.AssignableTo(e2Type)` will return true if e1Type is assignment-compatible with e2Type (for a definition of assignment-compatibility, see below).

In addition, both tests return true if e1Type or e2Type is the `errorType`. This avoids generating too many error messages with large expressions (there is no use complaining at each AST expression node up the tree after we have reported the initial type error, see also our lecture slides on "Semantic Analysis").

To use the above methods, the StdEnvironment already provides the six predefined types

`StdEnvironment.intType`
`StdEnvironment.boolType`

<center>7</center>

```
StdEnvironment.floatType
StdEnvironment.stringType
StdEnvironment.voidType
StdEnvironment.errorType
```

You find those types in file StdEnvironment.java.
Example: if you want to find out whether some type e1Type is int, you would use

```
if (e1Type.Tequal(StdEnvironment.intType))....
```

6 Type Coercions and Assignment Compatibility

An expression is a *mixed-mode* expression if it has operands of different types, e.g.,

```
1 + 2.0  //type int and type float
```

In MiniC, like in C, C++ and Java, certain kinds of mixed-mode expressions are permitted. When an operator has operands of different types, they will be automatically converted by the compiler to a common type, if this mixed-mode expression is legal according to the language semantic rules. Such an automatic (also called "implicit") type conversion is called *type coercion*. This is different from an explicit type conversion or *type cast* like

```
2.0 + (float) 1
// the int literal is type-cast to a float literal 1.0
```

Type casts do not exist in MiniC, but they exist in C, C++ and Java. In MiniC, the following operators allow **int** or **float** operands:
+   -   *   /   <   <=   >   >=   ==   !=
If one operand is of type **float**, the compiler will perform coercion to convert the other operand to **float**. Therefore, if at least one operand of a binary operator is of type **float**, then the operation is a floating-point operation (and not an integer operation). If both operands are of type **int**, then the operation is an integer operation. We must implement these semantic rules in our semantic analyzer. Specifically, the above rules are implemented in the visit() routine for BinaryExpr nodes.

Assignment coercions can occur when the value of an expression is assigned to a variable. In that case the value of the right-hand side expression is converted to the type of the left-hand side of the assignment statement, e.g.,

```
float a; int b; a = b;  // coercion of b from int to float.
```

The following rules apply to assignment coercions in MiniC:
(R1) If the type of the left-hand side variable is **int**, the type of the right-hand side must be **int.**

8

(R2) If the type of the left-hand side variable is **float,** the type of the right-hand side must be **int** or **float**. In case of **int,** it is coerced to **float** by the compiler.

(R3) If the type of the left-hand side variable is **bool,** then the right-hand side must be of type **bool.**

If the type of a right-hand side is compatible according to the above rules, then we say the right-hand side expression is *assignment-compatible* to the left-hand side. As already mentioned, the MiniC types provide the method `AssignableTo()` to check for assignment compatibility. Note that if `AssignableTo()` returns true, it may require you to insert an **int**-to-**float** coercion if you are facing Case (R2) above.

Because an argument of a function call is an expression, assignment coercions may also be required when arguments are passed to functions. Our semantic analyzer must check the actual arguments of a function-call versus the formal parameters of the called function. The same holds for return statements that return a value.

We achieve **int**-to-**float** type coercions by introducing extra conversion nodes into the program's AST. This is done in the `visit()` routines for `AssignStmt, CallExpr, BinaryExpr` and `ReturnStmt`. We use a special unary operator `i2f` to convert an int value to a float value. SemanticAnalysis.java already provides a routine that does exactly that:

```
private Expr i2f (Expr e) {
    Operator op = new Operator ("i2f", new SourcePos());
    op.type = StdEnvironment.intType;
    UnaryExpr eAST = new UnaryExpr (op, e, new SourcePos());
    eAST.type = StdEnvironment.floatType;
    return eAST;
}
```

It generates an operator "`i2f`" and sets its type to **int.** It generates a UnaryExpr node with two children, the "`i2f`" operator and the argument expression e. The (return) type of the new UnaryExpr is **float**.

Example:
     Expr AST for e <int>
      =>
        UnaryExpr <float>
          | \
          | \
       i2f<int> Expr AST for e <int>

AST node annotations <...> denote the value of the *type-attribute* of a node.

7 Overloaded Operators

Several MiniC operators, e.g., * and /, can be applied to either a pair of floating-point numbers, or a pair of integer numbers. Such operators are said to be overloaded. Our semantic analyzer must replace each overloaded operator by the appropriate operator required in the specific situation (e.g., *<int> for integer multiplication with expression 2 * 3).

In the Java Virtual Machine (JVM), the Boolean values true and false are represented by integer values. Therefore all MiniC operators acting on bool values must be set to type **int!** The MiniC operators that can act on Boolean values are &&, ||, !, ==, and !=. Our semantic analyzer sets the type of an operator by decorating the type attribute of an operator AST node (see e.g., i2f in the example above).

Resolving overloaded operators is straight-forward:
(1.)**&&, ||** and **!**: These operators can only act on bool values. Because of the above JVM requirement their type is therefore int.
(2.)**+ - * / < <= > >=** (where +, - are both unary and binary): These must be resolved from the operands. An operator is of type float only if at least one of the operands is of type float.
(3.)**==** and **!=**: These operators can be applied to either a pair of bool values, integers or floating-point numbers. Because of the above JVM requirement their type is again int.

```
// Example type coercion:
void foo() {
  float f;
  int i;
  f = i + 2;
}
```
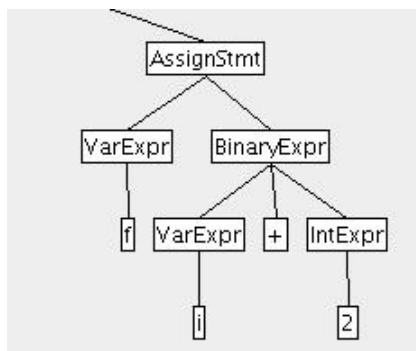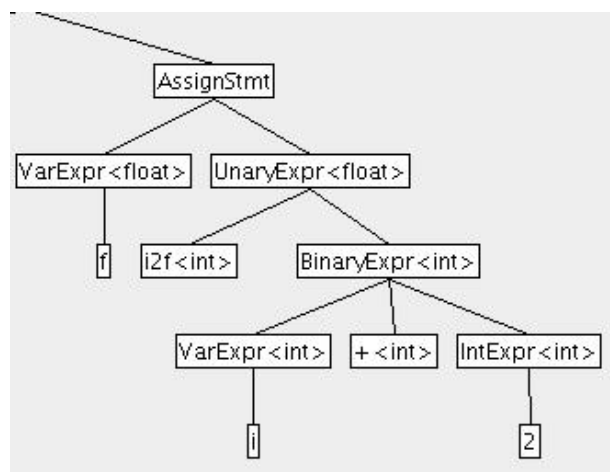

Figure 1: AST after coercion and overload resolution.

Figure 2: AST before Coercion

10

8 Your Task:

File `SemanticAnalysis.java` contains a visitor to perform static semantic analysis of MiniC ASTs. This analyzer is not complete yet---it has to be completed by you. At specific places in this code you find instructions on what you should do to complete this semantic analyzer. Those instructions are easy to spot, they all start with a comment as follows:

```
// STEP [123]
```

Your task can be divided into three subtasks:

- **STEP 1:** instructions given under STEP 1 tell you how to complete the identification of our semantic analyzer (i.e., scope stack handling).
- **STEP 2:** instructions given under STEP 2 tell you where you have to perform type checking, and what to do there. In STEP 2 you'll compute type information for MiniC expressions and pass them up the AST.
- **STEP 3:** this step involves several additional MiniC static semantic checks, such as to ensure that a MiniC file contains a `main()` routine.

It is strongly recommended to proceed in the order STEP1→STEP2→STEP3.

Whenever during semantic analysis you detect an error, you must report an error message from the `errMsg[]` array, using the `reporter` object.

One word of caution: the `SemanticAnalysis.java` code skeleton compiles as is. It can already produce un-decorated ASTs for some of the test-cases. However, since no type-information is contained in the AST yet, the type tags <..> contain only question marks or error types. Because type information is missing in the AST, some test-cases may make the semantic analyzer skeleton crash (Some code already provided to you assumes the presence of type information that you'll have to provide yet; Especially `BinExpr` where the left-hand side is a literal are likely to cause problems).

A second word of caution: the code skeletons contain many `assert()` statements to ensure that preconditions are satisfied. That way it is usually easy to find run-time errors right at the place they occur. You are strongly advised to (1) use this programming style for your own code, and (2) to run java programs with the `-ea` flag, i.e., java `-ea` ..., to enable checking of `assert()` statements at run-time.

Provided Classfiles

In case your scanner and/or parser do not work properly, you can use the classfiles provided in the ``resources/'' directory-tree of the CSI4104_Assignment_4.tgz archive. The following two Gradle tasks are provided:

1) `jarNoScanner`: uses the classfiles in resources/scanner_only to provide a working scanner implementation. (The parser must be provided by the course participant.)
2) `jarNoScannerNoParser`: uses the classfiles in resources/scanner_and_parser to provide a working scanner and parser implementation.

Building, Running and Testing Your Semantic Analyzer

Like with the previous assignments, we use the Gradle build automation system to compile our MiniC compiler. The build tasks ``jar'' and ``jarNoScanner'' are the same as with Assignment 2. Build task "jarNoScannerNoParser" has been added (see Section ``Provided Classfiles'').

The MiniC compiler now accepts a set of command-line switches:

- ▪ -ast1 to display the AST from the parser on-screen.
- ▪ -ast2 to display the AST from semantic analysis on-screen.
- ▪ -envast to display the AST of the MiniC standard environment on-screen.
- ▪ -t <file> prints the AST from the parser in file <file>, using the TreePrinter.
- ▪ -u <file> unparses the AST from the parser into file <file>.

Using option -ast2, you can compare the annotations produced by your semantic analyzer to the annotations provided on the CSI4104 website at
http://elc.yonsei.ac.kr/courses/csi4104/csi4104_assignment4.html.

> The most efficient way to test your semantic analyzer is to automatically run it on the provided testcases and ``diff'' the output with the solutions provided in directory MiniC/SemanticAnalysis/tst/base/solutions. You may adapt the test scripts provided with Assignment 1 and 2 for this purpose.
> You are allowed to share test scripts on LearnUs. The most useful test scripts will be awarded a participation point. (Please do **not** share any other code on LearnUs!)

9 Bonus Assignment

Semantic analysis of initializing expressions and arrays. This is done as a bonus assignment that will give you 10% extra if you solve the testcases c27*.mc and above. The places in the SemanticAnalysis.java code skeleton that have to be extended for the bonus assignment are marked as "STEP4".

MiniC arrays have been derived and simplified from C arrays. MiniC permits only one-dimensional arrays. Array subscripts start at 0. If an array has n elements, then valid subscripts are from 0..n-1. With this assignment, checking of valid subscripts is however not attempted in the compiler! A subscript can be any expression of type int. The element type of an array can only be **bool**, **int** or **float**.
In MiniC it is mandatory to specify the array range with formal parameters. (E.g., `void foo(int a[]){}` is not allowed, but `void foo(int a[2]){}` is.)

An array variable itself without square brackets [] cannot appear in any MiniC expression, except as an argument in a function call. MiniC requires that a formal array parameter exactly matches the argument provided at the function call-site:

```
void foo(int x[22]) {}

int main(){
 int a[21];
 int b[22];
 int c;
 foo(a); // ERROR (array range does not match)
 foo(b); // semantically correct
 c = a;  // ERROR
}
```

Arrays can be initialized with an initializer list between curly braces:

```
int a, b[3] = {1+a, 2, 3};     // ok
int c[2] = {1.0};              // type error
```

Every element in the list must be of the element type of the array. Type coercions of initializing expressions from int to float apply as with assignment statements. It is permissible that the list of initializing expressions contains fewer elements than the array's range specifies. It is however not allowed to have more expressions.

Non-array variables can be initialized with a single expression, e.g.,

```
int a, b = 22*a;
```

It is not permitted in MiniC to initialize an array-variable without curly braces, and it is not permitted to initialize a non-array variable with curly braces, e.g.,

```
int a = {1};         // ERROR
int b[1] = 1;        // ERROR
```

The provided testcases and the error messages in file SemanticAnalysis.java provide additional information for arrays and initializing expressions.

10 Marking

Please note that all assignments are *individual* assignments. You are most welcome to discuss your ideas with your colleagues, but 'code-sharing' and 'code-reuse' are not allowed. Assignments are designed to facilitate understanding of the course contents. Assignments are also a part of the preparation for the exams.

Your semantic analyzer will be assessed by how well it handles various semantically legal and illegal MiniC programs. Only *syntactically* legal programs will be used. Your type checker will not be marked on how well it recovers from errors or whether it outputs spurious error messages.

We will use small test-cases, so that each test-case checks for one semantic error. We will use the UNIX grep command to search your output for the "ERROR..." message specific to that semantic error. All other output will not be considered. Consider the following example:

```
int main() {
  int i;
  i = 1.0+true;
}
```

Our semantic analyzer reports

```
********** MiniC Compiler **********
Syntax Analysis ...
Semantic Analysis ...
ERROR: #9: incompatible types for binary operator 7..14,
          line 3.
Compilation was unsuccessful.
```

Error #9 is the one that is required in this case. If semantic analysis is continued in the presence of errors, then the further spurious error message is possible:

```
ERROR: #6: incompatible types for =
```

Whether your semantic analyzer reports spurious error messages or not will not be assessed.

Positional information (line numbers and column numbers) will not be assessed.

11 Deliverables and Submission

Please do **not** submit Assignment 4 prematurely! See Section "Due date" at the beginning of this document for the time-period submission is allowed for Assignment 4.

You should extend the semantic analyzer in file SemanticAnalysis.java. Note that for this assignment it is only necessary to modify file SemanticAnalysis.java. Submission will therefore be restricted to this single file. It is *not allowed to modify other files, e.g., the AST class hierarchy, Token.java aso, otherwise automated testing of your parser might no longer work. For the same reason, you are not allowed to modify the error messages in the errMsg[] array.*

14

To solve this assignment, it is not necessary nor allowed to use packages and libraries other than the ones already imported in file SemanticAnalysis.java.

To submit your file SemanticAnalysis.java, please log in on our server, create a submission directory, e.g., ~/myassignment4, and copy your modified SemanticAnalysis.java to this directory.

After copying the file to your submission directory, you must cd to the submission directory and type ``submit''. This will collect your file and copy it to our repository.

Please refer to the submission instructions from Assignment 1 for further details, in particular on how to make sure that your submission was received correctly.

If you have questions, don't hesitate to contact us. ☺

All the best!