Programming Assignment 5

Due 11pm, Nov 21

Overview

For this assignment you will write a type checker for harambe programs represented as abstract-syntax trees. Your main task will be to write type checking methods for the nodes of the AST. In addition you will need to:

1. Write a new main program, P5.java (an extension of P4.java).
2. Update the makefile used for program 4 to include any new rules needed for program 5.
3. Write two test inputs: typeErrors.ha and test.ha to test your new code.

Getting Started

You have a couple of options for completing p5:

Using Your Own Code

If you'd like to use your own code, you are free to do so. Copy everything over from your P4, change the name of your driver class to P5.java, and update your Makefile.

Correct implementations of ast.java and sym.java (and a helper class, Type.java) for program 4 are available at the links above, or you may use your own implementation.

Starting Fresh (Recommended)

If you don't want to use eclipse, you can use a fresh version of the code by downloading the tarball here. The Makefile assumes that you already have the CLASSPATH environment variable set. If you do not have it set then use configure.sh file from the previous assignments to set the CLASSPATH.

Specifications

- Type Checking
  - Preventing Cascading Errors
- Other Tasks
  - P5.java
  - Updating the Makefile
  - Writing test Inputs
- Some Advice

Type Checking

The type checker will determine the type of every expression represented in the abstract-syntax tree and will use that information to identify type errors. In the language we have the following types:

- int, bool, void (as function return types only), struct types, and function types.

A struct type includes the name of the struct (i.e., when it was declared/defined). A function type includes the types of the parameters and the return type.
The operators in the language are divided into the following categories:

- **logical**: not, and, or
- **arithmetic**: plus, minus, times, divide, unary minus
- **equality**: equals, not equals
- **relational**: less than (<), greater than (>), less then or equals (<=), greater than or equals (>=)
- **assignment**: assign

The type rules of the language are as follows:

- **logical operators and conditions**: Only boolean expressions can be used as operands of logical operators or in the condition of an if or while statement. The result of applying a logical operator to bool operands is bool.

- **arithmetic and relational operators**: Only integer expressions can be used as operands of these operators. The result of applying an arithmetic operator to int operand(s) is int. The result of applying a relational operator to int operands is bool.

- **equality operators**: Only integer or boolean expressions can be used as operands of these operators. Furthermore, the types of both operands must be the same. The result of applying an equality operator is bool.
  
  **Note**: You don't need to worry about equality operators between string literals. Either accepting it or declining it will be accepted in this assignment.

- **assignment operator**: Only integer or boolean expressions can be used as operands of an assignment operator. Furthermore, the types of the left-hand side and right-hand side must be the same. The type of the result of applying the assignment operator is the type of the right-hand side.

- **cout and cin**: Only an int or bool expression or a string literal can be printed by cout. Only an int or bool identifier can be read by cin. Note that the identifier can be a field of a struct type (accessed using .) as long as the field is an int or a bool.

- **function calls**: A function call can be made only using an identifier with function type (i.e., an identifier that is the name of a function). The number of actuals must match the number of formals. The type of each actual must match the type of the corresponding formal.

- **function returns**: A void function may not return a value. A non-void function may not have a return statement without a value. A function whose return type is int may only return an int; a function whose return type is bool may only return a bool.

  **Note**: some compilers give error messages for non-void functions that have paths from function start to function end with no return statement. For example, this code would cause such an error:

  ```
  int f() {
      cout << "hello";
  }
  ```

  However, finding such paths is beyond the capabilities of our compiler, so don't worry about this kind of error.

You must implement your type checker by writing appropriate member methods for the different subclasses of ASTnode. Your type checker should find all of the type errors described in the following table; it must report the specified position of the error, and it must give exactly the specified error message. (Each message should appear on a single line, rather than how it is formatted in the following table.)
<table>
<thead>
<tr>
<th>Type of Error</th>
<th>Error Message</th>
<th>Position to Report</th>
</tr>
</thead>
<tbody>
<tr>
<td>Writing a function; e.g., &quot;cout &lt;&lt; <code>f&quot;, where </code>f` is a function name.</td>
<td>Attempt to write a function</td>
<td>1st character of the function name.</td>
</tr>
<tr>
<td>Writing a struct name; e.g., &quot;cout &lt;&lt; <code>p&quot;, where </code>p` is the name of a struct type.</td>
<td>Attempt to write a struct name</td>
<td>1st character of the struct name.</td>
</tr>
<tr>
<td>Writing a struct variable; e.g., &quot;cout &lt;&lt; <code>p&quot;, where </code>p` is a variable declared to be of a struct type.</td>
<td>Attempt to write a struct variable</td>
<td>1st character of the struct variable.</td>
</tr>
<tr>
<td>Writing a void value (note: this can only happen if there is an attempt to write the return value from a void function); e.g., &quot;cout &lt;&lt; <code>f()&quot;, where </code>f` is a void function.</td>
<td>Attempt to write void</td>
<td>1st character of the function name.</td>
</tr>
<tr>
<td>Reading a function: e.g., &quot;cin &gt;&gt; <code>f&quot;, where </code>f` is a function name.</td>
<td>Attempt to read a function</td>
<td>1st character of the function name.</td>
</tr>
<tr>
<td>Reading a struct name; e.g., &quot;cin &gt;&gt; <code>p&quot;, where </code>p` is the name of a struct type.</td>
<td>Attempt to read a struct name</td>
<td>1st character of the struct name.</td>
</tr>
<tr>
<td>Reading a struct variable; e.g., &quot;cin &gt;&gt; <code>p&quot;, where </code>p` is a variable declared to be of a struct type.</td>
<td>Attempt to read a struct variable</td>
<td>1st character of the struct variable.</td>
</tr>
<tr>
<td>Calling something other than a function; e.g., &quot;x();&quot;, where <code>x</code> is not a function name. Note: In this case, you should not type-check the actual parameters.</td>
<td>Attempt to call a non-function</td>
<td>1st character of the variable name.</td>
</tr>
<tr>
<td>Calling a function with the wrong number of arguments. Note: In this case, you should not type-check the actual parameters.</td>
<td>Function call with wrong number of args</td>
<td>1st character of the function name.</td>
</tr>
<tr>
<td>Calling a function with an argument of the wrong type. Note: you should only check for this error if the number of arguments is correct. If there are several arguments with the wrong type, you must give an error message for each such argument.</td>
<td>Type of actual does not match type of formal</td>
<td>1st character of the first identifier or literal in the actual parameter.</td>
</tr>
<tr>
<td>Returning from a non-void function with a plain return statement (i.e., one that does not return a value).</td>
<td>Missing return value</td>
<td>0,0</td>
</tr>
<tr>
<td>Returning a value from a void function.</td>
<td>Return with a value in a void function</td>
<td>1st character of the first identifier or literal in the returned expression.</td>
</tr>
<tr>
<td>Returning a value of the wrong type from a non-void function.</td>
<td>Bad return value</td>
<td>1st character of the first identifier or literal in the returned expression.</td>
</tr>
<tr>
<td>Applying an arithmetic operator (+, -, *, /) to an operand with type other than int. Note: this includes the ++ and -- operators.</td>
<td>Arithmetic operator applied to non-numeric operand</td>
<td>1st character of the first identifier or literal in an operand that is an expression of the wrong type.</td>
</tr>
<tr>
<td></td>
<td>Relational</td>
<td>1st character of the first identifier or literal in an operand that is an expression of the wrong type.</td>
</tr>
<tr>
<td>Applying a relational operator (&lt;, &gt;, &lt;=, &gt;=) to an operand with type other than int.</td>
<td>operator applied to non-numeric operand</td>
<td>identifier or literal in an operand that is an expression of the wrong type.</td>
</tr>
<tr>
<td>Applying a logical operator (!, &amp;&amp;,</td>
<td></td>
<td>) to an operand with type other than bool.</td>
</tr>
<tr>
<td>Using a non-bool expression as the condition of an if.</td>
<td>Non-bool expression used as an if condition</td>
<td>1st character of the first identifier or literal in the condition.</td>
</tr>
<tr>
<td>Using a non-bool expression as the condition of a while.</td>
<td>Non-bool expression used as a while condition</td>
<td>1st character of the first identifier or literal in the condition.</td>
</tr>
<tr>
<td>Applying an equality operator (==, !=) to operands of two different types (e.g., &quot;j == true&quot;, where j is of type int), or assigning a value of one type to a variable of another type (e.g., &quot;j = true&quot;, where j is of type int).</td>
<td>Type mismatch</td>
<td>1st character of the first identifier or literal in the left-hand operand.</td>
</tr>
<tr>
<td>Applying an equality operator (==, !=) to void function operands (e.g., &quot;f() == g()&quot;, where f and g are functions whose return type is void).</td>
<td>Equality operator applied to void functions</td>
<td>1st character of the first function name.</td>
</tr>
<tr>
<td>Comparing two functions for equality, e.g., &quot;f == g&quot; or &quot;f != g&quot;, where f and g are function names.</td>
<td>Equality operator applied to functions</td>
<td>1st character of the first function name.</td>
</tr>
<tr>
<td>Comparing two struct names for equality, e.g., &quot;A == B&quot; or &quot;A != B&quot;, where A and B are the names of struct types.</td>
<td>Equality operator applied to struct names</td>
<td>1st character of the first struct name.</td>
</tr>
<tr>
<td>Comparing two struct variables for equality, e.g., &quot;a == b&quot; or &quot;a != b&quot;, where a and a are variables declared to be of struct types.</td>
<td>Equality operator applied to struct variables</td>
<td>1st character of the first struct variable.</td>
</tr>
<tr>
<td>Assigning a function to a function; e.g., &quot;f = g;&quot;, where f and g are function names.</td>
<td>Function assignment</td>
<td>1st character of the first function name.</td>
</tr>
<tr>
<td>Assigning a struct name to a struct name; e.g., &quot;A = B;&quot;, where A and B are the names of struct types.</td>
<td>Struct name assignment</td>
<td>1st character of the first struct name.</td>
</tr>
<tr>
<td>Assigning a struct variable to a struct variable; e.g., &quot;a = b;&quot;, where a and b are variables declared to be of struct types.</td>
<td>Struct variable assignment</td>
<td>1st character of the first struct variable.</td>
</tr>
</tbody>
</table>

**Preventing Cascading Errors**

A single type error in an expression or statement should not trigger multiple error messages. For example, assume that P is the name of a struct type, p is a variable declared to be of struct type P, and f is a function that has one integer parameter and returns a bool. Each of the following should cause only one error message:
cout << P + 1          // P + 1 is an error; the write is OK
(true + 3) * 4         // true + 3 is an error; the * is OK
true && (false || 3)   // false || 3 is an error; the && is OK
f("a" * 4);            // "a" * 4 is an error; the call is OK
1 + p();               // p() is an error; the + is OK
(true + 3) == x        // true + 3 is an error; the == is OK
                     // regardless of the type of x

One way to accomplish this is to use a special ErrorType for expressions that contain type errors. In the first example above, the type given to (true + 3) should be ErrorType, and the type-check method for the multiplication node should not report "Arithmetic operator applied to non-numeric operand" for the first operand. But note that the following should each cause two error messages (assuming the same declarations of f as above):

true + "hello"         // one error for each of the non-int operands of the +
1 + f(true)            // one for the bad arg type and one for the 2nd operand of the +
1 + f(1, 2)            // one for the wrong number of args and one for the 2nd operand of the +
return 3+true;         // in a void function: one error for the 2nd operand to +
                     // and one for returning a value

To provide some help with this issue, here is an example input file, along with the corresponding error messages. (Note: This is not meant to a complete test of the type checker; it is provided merely to help you understand some of the messages you need to report, and to help you find small typos in your error messages. If you run your program on the example file and put the output into a new file, you can use the Linux utility diff to compare your file of error messages with the one supplied here. This will help both to make sure that your code finds the errors it is supposed to find, and to uncover small typos you may have made in the error messages.)

Other Tasks

P5.java

The main program, P5.java, will be similar to P4.java, except that if it calls the name analyzer and there are no errors, it will then call the type checker.

Updating the Makefile

You will need to update the Makefile you used for program 4 so that typing "make" creates P5.class.

Writing Test Inputs

You will need to write two input files to test your code:

1. typeErrors.ha should contain code with errors detected by the type checker. For every type error listed in the table above, you should include an instance of that error for each of the relevant operators, and in each part of a program where the error can occur (e.g., in a top-level statement, in a statement inside a while loop, etc).

2. test.ha should contain code with no errors that exercises all of the type-check methods that you wrote for the different AST nodes. This means that it should include (good) examples of every kind of statement and expression.

Note that your typeErrors.ha should cause error messages to be output, so to know whether your type checker behaves correctly, you will need to know what output to expect.

Part of the grade depends on how thoroughly the input files you used, test the program. Make sure that you submit the input files you used to test your program.
Some Advice

Here are few words of advice about various issues that come up in the assignment:

- For this assignment you are free to make any changes you want to the code in `ast.java`. For example, you may find it helpful to make small changes to the class hierarchy, or to add new fields and/or methods to some classes.

- As for name analysis, think about which AST nodes need to have type-check methods. For example, for type checking, you do not need to visit nodes that represent declarations, only those that represent statements.