CS536 Programming Assignment 4

Due by 11:00 pm on March 30, 2020

Overview

For this assignment you will write a name analyzer for Wumbo programs represented as abstract-syntax trees. Your main task will be to write *name analysis* methods for the nodes of the AST. In addition you will need to:

1. Modify the `Sym` class (by including some new fields and methods and/or by defining some subclasses).
2. Modify the `IdNode` class in `ast.java` (by including a new `Sym` field and by modifying its `unparse` method).
3. Modify `P4.java` (an extension of `P3.java`).
4. Modify the `ErrMsg` class.
5. Write two test inputs: `nameErrors.wumbo` and `test.wumbo` to test your new code.

Specifications

Getting Started

The files are:

- `Sym.java`
- `SymTable.java`
- `DuplicateSymException.java`
- `EmptySymTableException.java`
- `ErrMsg.java`
- `Wumbo.cup`
- `Wumbo.jlex`
- `ast.java`
- `P4.java`
- `Makefile`
- `nameErrors.wumbo`
- `test.wumbo`

Here is `p4.zip` file. It is recommended to start your project with it.

Name Analysis

The name analyzer will perform the following tasks:

1. **Build symbol tables.** You will use the "list of hashtables" approach (using the `SymTable` class from program 1).

2. **Find multiply declared names, uses of undeclared names, bad `struct` accesses, and bad declarations.** Like C, the Wumbo language allows the same name to be declared in non-overlapping or nested scopes. The formal parameters of a function are considered to be in the same scope as the function body. All names must be declared before they are used. A *bad* `struct` access happens when either the left-hand side of the dot-access is not a name already declared to be of a `struct` type or the right-hand side of the dot-access is not the name of a field for the appropriate type of `struct`. A *bad* declaration is a declaration of anything other than a function to be of type `void` as well as the declaration of a variable to be of a *bad* `struct` type (the name of the `struct` type doesn't exist or is not a `struct` type).

3. **Add `IdNode` links**: For each `IdNode` in the abstract-syntax tree that represents a *use* of a name (not a declaration) add a "link" to the corresponding symbol-table entry. (As stated above, you will need to modify the `IdNode` class in `ast.java` to have a new field of type `Sym`. That is the field that your name analyzer will fill in with a link to the `Sym` returned by the symbol table's `globalLookup` method.)

You must implement your name analyzer by writing appropriate methods for the different subclasses of `ASTnode`. Exactly what methods you write is up to you (as long as they do name analysis as specified).

It may help to start by writing the name analysis method for `ProgramNode`, then work "top down", adding a method for `DeclListNode` (the child of a `ProgramNode`), then for each kind of `DeclNode` (except `StructDeclNode`), and so on (and then handle `StructDeclNode` and perhaps other `struct` related nodes at the end). Be sure to think about which nodes' methods need to add a new hashtable to the symbol table (i.e., when is a new scope being entered) and which methods need to remove a hashtable from the symbol table (i.e., when is a scope being exited).

Some of the methods will process the declarations in the program (checking for bad declarations and checking whether the names are multiply declared, and if not, adding appropriate symbol-table entries) and some will process the statements in the program (checking that every name used in a statement has been declared and adding links). Note that you should *not* add a link for an `IdNode` that represents a use of an undeclared name.

`struct` Handling Issues

Name analysis issues surrounding `struct`s come up in several situations:

- **Defining a `struct` type**: for example

  ```
  struct Point {
      int x;
      int y;
  };
  ```

  When defining a `struct`, the name of the `struct` type can't be a name that has already been declared. The fields of a `struct` must be unique to that particular `struct`; however, they can be a name that has been declared outside of the `struct` definition. For this reason, a recommended approach is to have a separate symbol table associated with each `struct` definition and to store this symbol table in the symbol for the name of the `struct` type.

- **Declaring a variable to be of a `struct` type**: for example

  ```
  struct Point pt;
  ```

  When declaring a variable of a `struct` type, in addition to determining if the variable name has been previously declared (and issuing a "multiply declared" error if it is), you should also check that the name of the `struct` type has been previously declared and is actually the name of a `struct` type.

- **Accessing the fields of a `struct`**: for example

```
pt.x = 7;
```

  When doing name analysis on something like *`LHS.RHS`*, you will need to check that *`LHS`* is the name of a variable that has previously been declared to be of a `struct` type and that *`RHS`* is the name of a field in the `struct` type associated with *`LHS`*.

Error Reporting

Your name analyzer should find all of the errors described in the table given below; it should report the specified position of the error, and it should give *exactly* the specified error message (each message should appear on a single line, rather than how it is formatted in the following table). Error messages should have the same format as in the scanner and parser (i.e., they should be issued using a call to `ErrMsg.fatal`).

If a declaration is both "bad" (e.g., a non-function declared `void`) and is a declaration of a name that has already been declared in the same scope, you should give *two* error messages (first the "bad" declaration error, then the "multiply declared" error).

Type of Error	Error Message	Position to Report
More than one declaration of an identifier in a given scope (note: includes identifier associated with a `struct` definition)	`Multiply declared identifier`	The first character of the ID in the duplicate declaration
Use of an undeclared identifier	`Undeclared identifier`	The first character of the undeclared identifier
Bad `struct` access (LHS of dot-access is not of a `struct` type)	`Dot-access of non-struct type`	The first character of the ID corresponding to the LHS of the dot-access.
Bad `struct` access (RHS of dot-access is not a field of the appropriate a `struct`)	`Invalid struct field name`	The first character of the ID corresponding to the RHS of the dot-access.
Bad declaration (variable or parameter of type `void`)	`Non-function declared void`	The first character of the ID in the bad declaration.
Bad declaration (attempt to declare variable of a bad `struct` type)	`Invalid name of struct type`	The first character of the ID corresponding to the `struct` type in the bad declaration.

Note that the names themselves should *not* be printed as part of the error messages.

During name analysis, if a function name is multiply declared you *should* still process the formals and the body of the function; don't add a new entry to the current symbol table for the function, but do add a new hashtable to the front of the `SymTable`'s list for the names declared in the body (i.e., the parameters and other local variables of the function).

If you find a bad variable declaration (a variable of type `void` or of a bad `struct` type), give an error message and add nothing to the symbol table.

Other Tasks

Extending the `Sym` Class

It is up to you how you store information in each symbol-table entry (each `Sym`). To implement the changes to the unparser described below you will need to know each name's type. For function names, this includes the return type and the number of parameters and their types. You can modify the `Sym` class by adding some new fields (e.g., a `kind` field) and/or by declaring some subclasses (e.g., a subclass for functions that has extra fields for the return type and the list of parameter types). You will probably also want to add new methods that return the values of the new fields and it may be helpful to change the `toString` method so that you can print the contents of a `Sym` for debugging purposes.

Modifying the `IdNode` Class

Two changes to the `IdNode` class are needed:

1. Adding a new field of type `Sym` (to link the node with the corresponding symbol-table entry), and

2. Changing the unparse method so that every use of an ID has its type (in parentheses) after its name. (The point of this is to help you to see whether your name analyzer is working correctly; i.e., does it correctly match each use of a name to the corresponding declaration, and does it correctly set the link from the `IdNode` to the information in the symbol table.) For names of functions, the information should be of the form: `param1Type, param2Type, ..., paramNType -> returnType`. For names of global variables, parameters, and local variables of a non-`struct` type , the information should be `int` or `bool`. For a global or local variable that is of a `struct` type, the information should be the name of the `struct` type. For example, given a program that contains this code:

```
struct Point {
    int x;
    int y;
};
int f(int x, bool b) { }
void g() {
    int a;
    bool b;
    struct Point p;
    p.x = a;
    b = a == 3;
    f(a + p.y*2, b);
    g();
}
```

   The unparser should print:

```
struct Point {
    int x;
    int y;
};
int f(int x, bool b) {
}
void g() {
    int a;
    bool b;
    struct Point p;
    p(Point).x(int) = a(int);
    b(bool) = (a(int) == 3);
    f(int,bool->int)((a(int) + (p(Point).y(int) * 2)), b(bool));
    g(->void)();
}
```

Summary

```
struct
```

- If a variable or a function with the same name has been declared in the same scope before, then do not add a SymTable entry for the struct. You don't have to process the variables of the struct in this case.
- A variable inside a struct with the same name as a variable or a function outside the struct is legal.
- A variable x inside a struct with the same name as another variable inside the struct is illegal. In this case, create SymTable for the struct and add all variables up to but excluding the second occurrence of x and then continue with the rest of the members.
- If a struct is used without declaration like a.b, then you can report two errors (undeclared ID and dot access of non-struct type) or you can just report undeclared ID.
- The name of the struct is in a scope that is one level outside the scope of the struct itself. Thus, a struct and one of its members can have the same name.

`function`

- A function with the same name as another function in the same scope is illegal. You must not add a new SymTable entry in the outer scope for this second occurrence. You should process the formals and the local variables for both the functions.
- A function with the same name as another variable in the same scope is illegal. In this case, do not create a SymTable entry for the function. However, continue processing the body of the function.
- If a function with formal parameter a also has a variable declared as a, then create the SymTable for the function and add the formal parameter but not the local variable and then continue with processing.
- If a function has 2 formal parameters or 2 local variables with the same name, then create the SymTable, add the first parameter/local variable, report the error and then continue with processing.
- The name of the function is in a scope that is one level outside the scope of the function itself. Thus, a function and one of its formals/local variables can have the same name.

`if/else/while`

- if/else and while statements have their own scope. So, names can be reused inside these statements.
- The if part and the else part have different scopes. So, the same name can be declared in both of them.

P4.java

The main program, `P4.java`, will be similar to `P3.java`, except that

- After parsing, if there are no syntax errors, it will call the name analyzer.
- After that, if there are no errors so far (either scanning, parsing, or name-analysis errors), it will call the unparser.

Calling the name analyzer means calling the appropriate method of the `ASTnode` that is the root of the tree built by the parser.

Modifying the `ErrMsg` Class

Your compiler should quit after the name analyzer has finished if any errors have been detected so far (either by the scanner/parser or the name analyzer). To accomplish this, you can add a static boolean field to the `ErrMsg` class that is initialized to `false` and is set to `true` if the `fatal` method is ever called (warnings should not change the value of this field). Your `main` program can check the value of this field and only call the unparser if it is `false`.

Writing Test Inputs

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

1. `nameErrors.wumbo` should contain code with errors detected by the name analyzer. This means that it should include bad and multiply declared names for all of the different kinds of names, and in all of the different places that declarations can appear. It should also include uses of undeclared names in all kinds of statements and expressions as well as bad `struct` accesses.
2. `test.wumbo` should contain code with no errors that exercises all of the name-analysis methods that you wrote for the different AST nodes. This means that it should include (good) declarations of all of the different kinds of names in all of the places that names can be declared and it should include (good) uses of names in all kinds of statements and expressions.

Note that your `nameErrors.wumbo` should cause error messages to be output, so to know whether your name analyzer behaves correctly, you will need to know what output to expect.

As usual, you will be graded in part on how thoroughly your input files test your code.

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`.
- The tree-traversal code you wrote to perform unparsing provides a good model for the traversal that you need to write to handle name analysis. However, you might not want to declare the name-analysis methods to be abstract methods of class `ASTnode` (as we did for `unparse`). This is because you will not need those methods for all nodes; e.g., you probably won't want a name-analysis method for all of the sub-classes of the `TypeNode` class.

  However, you will need to declare the name-analysis methods to be abstract methods of some of the classes that are lower down in the inheritance hierarchy; for example, you will need to declare an abstract name-analysis method for the `DeclNode` class, because the method for the `DeclListNode` class will call that method for each node in the list.
- If you are working with a partner, you will have to decide how to divide up the work. You might want to divide up some of the "incidental tasks" (like modifying the `ErrMsg`, `Sym`, and `IdNode` classes), then work together to get a small part of the name-analysis phase working (e.g., finding multiply declared global variables). Then you could split up the `ASTnode` subclasses and each implement the name-analysis methods for your subset of those classes (you might want to start by choosing just a few each, until you have a better idea which ones will require the most work).

  Don't forget to test your work as you go along, rather than waiting until everything is finished!

Handing in

Please read the following handing in instructions carefully. You will be needed to submit the the entire working folder as a compressed file as given below.

```
lastname.firstname.lastname.firstname.P4.zip
+---+ deps/
+---+ ast.java
+---+ Wumbo.cup
+---+ Wumbo.jlex
+---+ DuplicateSymException.java
+---+ EmptySymTableException.java
+---+ ErrMsg.java
+---+ Makefile
+---+ P4.java
+---+ Sym.java
+---+ SymTable.java
+---+ nameErrors.wumbo
+---+ test.wumbo
+---+ lastname.firstname.lastname.firstname.P4.pdf
```

Please ensure that you do not include any extra sub-directories. Do not turn in any `.class` files. If you accidentally turn in (or create) extra files or subdirectories, please remove them from your submission zip file.

If you are working in a pair, have only one member submit the program. Include both persons' name as given above. Also, mention the teammate's name as a comment while submitting the assignment on canvas.

Joining a Canvas group with your partner (also required for those who work alone)

To faciliate grading, before you submit the compressed file, please join a Canvas group with your partner (guide) so you can both receive grades for the same submission. Since you're working on P4, please join an empty group whose name is in the form of "P4-Pair #" (type "P4" in the search bar). Note that we have created the groups for you so you only need to join an empty one with your partner. If you work alone, please join an empty group as well.

Grading criteria

General information on program grading criteria can be found on the Assignments page.

For more advice on Java programming style, see Code Conventions for the Java Programming Language. See also the style and commenting standards used in CS 302 and CS 367.