CS 2210 Programming Project (Part III)

Semantic Analysis

In this phase of the pro[illegible]o create and manipulate symbol tables, and detect semantic errors as well as bring together the various issues of semantic analysis discussed in the class.

Due date

No due date for this project.

Project Summary

Your task is to write a static semantic analyzer that

- creates a symbol table for the source program;
- augments the abstract syntax tree (produced by the parser) with appropriate semantic information;
- checks that the source program abides by some of the semantic rules of the MINI-JAVA language, and reports any violations.

Scoping Rules

The scope of a name is the region of the program in which the name can be used. MINI-JAVA has just **three** levels of scoping. Variables (including formal parameters) declared within a method are local and can only be used within the declaring method. Variables and methods declared in a class are local to that class. They are directly accessible from the point of declaration to the end of the class, as well as from the bodies of methods (of that class) that do not redeclare the same name. The same variable name can be declared in other methods without a conflict. All methods declared within a class can reference the objects declared in the class, including methods.

Objects (including declarations and method names) declared in one class can be used by objects of other classes provided the proper class name is attached, e.g. **System.**println, **MyClass.**Sort etc.

The lifetime of object declarations within a method is limited to the execution of the method (i.e. the values are discarded upon exit), while objects declared in a class last for the lifetime of the program.

Static Semantics

The following static checks must be performed.

1. All names declared within the same block (i.e., method or class) must be unique.

2. All names used within the program must be declared such that using the scoping rule, there exists a corresponding declaration visible from each use of a name (i.e. there should be no use of undeclared names).

3. The whole program contains only one method with name “main”.

4. The number of ar[illegible]d component (i.e. USE) should match the number of dimensions declared fo[illegible]We will only have 1-dimension and 2-dimension arrays.

5. The number of ar[illegible]all must match the number of parameters in the declarations.

6. String constants [illegible] statements.

Symbol Table Creation

Both static semantic checking and code generation require semantic information about names defined by the user.

The **symbol table** is a data structure that stores the semantic information for each name declaration. A new entry is added to the symbol table as each declaration is encountered during semantic processing.

Since code will be generated in a separated phase (project 4) using the information stored in the symbol table (and possibly adding more information to the table), symbol table entries are never removed after being created.

If a given name **X** is defined in two different places in a program (say, in the class and then as a local variable within a method), the final symbol table will contain 2 separate entries for X with the appropriate semantic information.

Semantic attributes are added to the entry for a particular name at various times. Every entry has **NameAttr** and **TypeAttr** attributes which are entered in response to the name’s declaration and contain the unique index into the string table and (a pointer to) the appropriate type tree node, respectively.

Since MINI-JAVA has no input/output defined as part of the language itself, the input/output methods are accessible to all MINI-JAVA programs as a predefined class called **System**, that is *readln* and *println*.

The symbol table should be initialized with an entry for each of these methods. The body of these methods must be implemented by you in the next phase, i.e., you must generate appropriate code when input/output is needed.

Each entry in the symbol table should has a **LevelAttr** which is entered in response to the name’s declaration and is an integer indicating the level that the symbol was defined (nesting level).

You may want a boolean **IsFormalAttr** attribute to distinguish between formal arguments and local variables.

From the description of some of the attributes, it is clear that an attribute could have a value of integer, boolean, or **ILTree**.

In addition to the standard attributes, your semantic analyzer may add other attributes to entries as necessary for semantic processing and code generation such that the number of attributes of a given entry may vary as you want to store different additional information for different entries.

An acceptable way to design a symbol table entry is as a linked list of attributes values that can be of any type above. The order that attributes are added to the list for a given entry will be insignificant if you store an additional integer field with each attribute value. This field can indicate what attribute is stored there. For example, if the integer is 1, then the attribute value represents **TypeAttr**; if the integer is 2, then the attribute value represents **LevelAttr**, ... A symbol table entry could also be implemented as a row in the table.

There will be two flavors of look into the symbol table.

1. When processing a new declaration, you will want to perform a lookup operation restricted to the current block of declarations (i.e., the names declared locally within this block thus far) to determine whether the symbol is multi-declared.

2. When processing [illegible] will want to perform a lookup over all currently accessible names starting w[illegible]s, and if not found, continuing to the non-locals.

Thus two lookup pr[illegible]d the symbol table must be organized to include block information such that the lo[illegible]d first.

One way to approa[illegible] the symbol table as a stack that always contains the set of accessible declarations [illegible]ent block being processed during symbol table creation and syntax tree augmentation. The stack will grow and shrink by the number of local declarations as processing enters and leaves a block, respectively, and the latest instance of a variable declaration in the currently accessible blocks will always be the one closest to the top of the stack.

However, since the complete symbol table is needed for the code generation phase, the symbol table entries can not be deleted during this phase. In order to retain symbol table entries which are no longer active (i.e., accessible) with respect to the current block being processed, the symbol table stack can be maintained as a separate data structure with reference into the permanent symbol table structure.

For example, a stack of records can be maintained where each record contains a boolean marker, an id, and a pointer to the entry for that id in the symbol table. When the semantic analyzer begins to process the declarations of a new procedure definition, an **OpenBlock** routine pushes a record *(true, undefined, undefined)* onto the stack to indicate the beginning of a new block. Processing of subsequent declarations within that procedure includes pushing records of *(false,id,pointer)* onto the stack.

Note that the procedure name should be entered into the symbol table before calling **OpenBlock** to signal the start of a new block because the procedure name is actually defined where it is declared.

When the limited form of the lookup is called, it scans the stack from the top until it finds the desired id or the last true marker that was pushed onto the stack (indicating that the identifier is not yet declared in the current block).

After processing all the statements within a procedure definition, the **CloseBlock** routine pops all records since the last true marker, and then pops the true marker.

The following symbol table utility routines have been written for you and are available from class webpage.

STInit()

Called before any other symbol table routine to initialize the symbol table current nesting level, etc.

InsertEntry(ID:integer) return STIndex

Add an [illegible] symbol table for identifier ID (unique index into string table), [illegible] in symbol table. This entry has no attributes initially. ID shou[illegible]ed since last OpenBlock.

LookUP(ID:intege[illegible]

Return [illegible]stance of ID in symbol table, or NullST (i.e. 0) if none.

LoopUpHere(ID:i[illegible]

Return [illegible]stance of ID in symbol table since last OpenBlock. Return NullST (i.e. 0) if none.

LookUPField(ID:integer) return STIndex

Return the symbol table entry of record field ID. ST indicates where the search starts in the symbol table. Return NullST if none.

OpenBlock()

Start a new block of symbols in the symbol table.

CloseBlock()

Restore the symbol table to the set of STIndices prevailing before the last OpenBlock.

IsAttr(ST:STIndex; AttrNum:integer) return boolean

Return true iff ST has attribute numbered AttrNum. For a newly defined symbol, ST, it is false for all attributes.

GetAttr(ST:STIndex; AttrNum: integer) return integer, boolean or ILTree

Return value of attribute AttrNum of ST; value may be integer, boolean or ILTree.

SetAttr(ST:STIndex; AttrNum: integer; V: integer, boolean or ILTree

Makes GetAttr(..)=V and IsAttr(ST,AttrNum) true.

procedure STPrint()

Print all symbols and attributes in the symbol table.

Syntax Tree Augmentation

You are required to traverse the syntax tree you built in the second project and augment the syntax tree.

Your syntax trees have to be changed to incorporate the following:

1. Converting leaves of type **IDNode** which represent the int type to leaves **INTEGERTNode** (if this was not done already). You can modify to your tree manipulation routines of the second project for this change only.
2. Converting all other leaves of the type **IDNode** to leaves of type **STNode**.
3. Replacing the integer value stored in each **IDNode** by a unique pointer into the symbol table. Each occurrence of an id is replaced with a reference to the symbol table entry containing the necessary semantic information for that id. This may also require minor modifications to your tree to handle **STNodes**

Static Semantic Checking

Uniqueness of id declarations can be checked when adding new entries to the symbol table for each constant, variable, or procedure declaration. Undeclared id's can be found when modifying the IDNodes in statement subtrees (and subrange type subtrees) to point to the symbol table. When an undeclared id is encountered, it should be entered int[illegible]prevent repeat error messages. Method invocations with the incorrect number or ty[illegible]nvariable arguments passed as call by reference parameters can be checked when a[illegible]s name's IDNode leaf in the call with a pointer to the symbol table. Similarly, an inc[illegible]s in an indexed component can be detected when processing the array name's IDNo[illegible]nent.

Testing the Semanti[illegible]

Your semantic analyzer should output the complete symbol table, and the augmented syntax tree. For grading ease, your syntax tree print routine should adhere to the same specification as the second project with following modification. Since each **IDNode** has been replaced by an **STNode**, [ID:token value:lexeme] should be replaced by [STID:symbol table index, lexeme] You will need to change the tree print routine to handle this. Since different approaches to this assignment may lead to different orders of entries/attributes in the symbol table, the symbol table printer that is provided may have to be changed. The changed format should be a clear concise format including all attributes of every entry. All of the attributes should be clearly defined.

Error Handling

Your semantic analyzer should print descriptive error messages and recover from all detected static semantic errors and continue to find any additional semantic errors. Error messages should be as descriptive as possible, given available information at the time of error detection.

Assignment submission

No submission.