编译器构造实验(3): 语义分析器

Apr 22, 2021(updated 5/6/2021)

1. Semantic Analysis

In this phase of the project, you are required to create and manipulate symbol tables, and detect semantic errors.

2. Due date [截止时间]

The assignment is due May 20, 2021 23:59.

3. Project Summary [总体任务]

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 of proj2) with appropriate semantic information;
- checks that the source program follows some of the semantic rules of the MINI-JAVA language, and reports any violations.

4. Scoping Rules [作用域规则]

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:

1) Variables (including formal parameters) declared within a **method** are local and can only be used within the declaring method.
2) 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.
3) 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.

5. Static Semantics [静态语义]

The following two static checks must be performed.

1） All names declared within the same block (i.e., method or class) must be unique. [名称唯一性]

2） All names used within the program must be declared such that using the scoping rule, there exists a corresponding declarations visible from each use of a name (i.e., there should be no uses of undeclared names). [使用前先声明]

In addition do any three of the followings.

1） The number of array indices in an indexed component should match the number of dimensions declared for the array. We will only have 1-dimension and 2-dimension arrays. [数组维度一致]

2）The number of arguments in a method call must match the number of parameters in the declarations. [参数数目匹配]

3）The whole program contains only one method with name “main”. [入口函数唯一]

4）Fields accessed in objects or classes must be declared in the class. [类属性使用前先声明]

5）String constants cannot be used to initialize int variables. [类型一致]

6）Type check dot(.) and index([]) operators such that they are only done on objects/classes and arrays respectively. [类型检查]

Extra credit: For each additional check beyond the three, you will get 10 additional points (max 3*10).

6. Symbol Table Creation [符号表创建]

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 later phase 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 two separate entries for X with the appropriate semantic information.

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 symbol table initialization is already done for you in the function STInit() in proj3.c distributed with the project. The code serves as an example of what you need to do in your parser when other symbols are encountered. Note that you still need to add those strings (i.e., system/readln/println) to your string table for the code to work.

A **symbol table entry** is added to the table when a declaration of that symbol is encountered during parsing. Also various semantic attributes can be added to the entry depending on what kind of entry it is. All entries have a **KindAttr** to tell what kind of entry it is and along with it, the **NameAttr**, **NestAttr**, and **PredefinedAttr** attributes. The **NameAttr** is the unique index into the string table for that symbol. The **NestAttr** is an integer indicating the nesting level in which the symbol was defined. The **PredefinedAttr** is a boolean value indicating whether the symbol is predefined or not. All symbols other than **System**, *readln*, and *println* are not predefined.

Most entries, besides classes and procedures (methods with no return values), have a **TypeAttr** attribute which points to the type tree node for the declaration. The **TypeAttr** for a function is the type of its return value. Procedure and function entries have **ArgNumAttr** and **TreeAttr** attributes, which contain the number of formal parameters and the tree node for that method, respectively. At the code generation stage, the compiler will use the **TreeAttr** to traverse the tree to generate code for each method. Also, array entries have a **DimensionAttr** attribute that stores how many dimensions there are in the array. As you can see, the number of attributes for a given entry may vary as you want to store different additional information for different entries. All types of attributes and all types of entries are listed with descriptions in proj3.h distributed with the project.

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 **NestAttr**, ... A symbol table entry could also be implemented as a row in the table. There will be two flavors of lookups in 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. See the provided LookUpHere().
2) When processing the use of a name, you will want to perform a lookup over all currently accessible names starting with the local declarations, and if not found, continuing to the non-locals. See the provided LookUp().

Thus two lookup procedures are needed, and the symbol table must be organized to include block information such that the local declarations are found first.

One way to approach this is to maintain the symbol table as a stack that always contains the set of accessible declarations with respect to the current 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 cannot 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, i.e., LookUpHere(), 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:

STInit()

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

InsertEntry(ID:integer) return STIndex

Add an entry to current symbol table for identifier ID (unique index into string table), returning index of entry in symbol table. This entry has no attributes initially.
ID should not have been defined since last OpenBlock.

LookUP(ID:integer) return STIndex

Return most recently added instance of ID in symbol table, or NullST (i.e. 0) if none.

LoopUpHere(ID:integer) return STIndex

Return most recently added instance 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.

7. Syntax Tree Augmentation [语法树扩展]

You are required to traverse the syntax tree you built in the project 2 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 need to modify your tree manipulation routines of the project 2 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 **STNode**s.

8. Static Semantic Checking [静态语义检查]

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 into the symbol table to prevent repeat error messages.

Method invocations with the incorrect number or type of arguments and non-variable arguments passed as call by reference parameters can be checked when augmenting the method's name's IDNode leaf in the call with a pointer to the symbol table.

Similarly, an incorrect number of indices in an indexed component can be detected when processing the array name's IDNode in the indexed component.

9. Outputs [程序输出]

Output: 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 project 2 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.

10. Assignment submission [关于提交]

The submission should be a compressed file that contains your project source code (no executable please), with a *readme.txt*. More submission details will be announced later.