CSE-304 Programming Assignment #3

Abstract Syntax Tree

Due: Mon., Nov 9, 2009

Description

In this assignment, you will build the abstract syntax tree for the `Decaf` compiler. You will add actions to the given `Bison` parser. You are also provided with the code for AST data structures themselves (see the description below). At the end, the AST for the entire program is to be stored in a global variable called `toplevel`. You are also provided with a driver program that invokes the Bison-generated parser and prints the AST.

AST Structure

The AST data structures consist of four main parts:

- **Entities:** Information about program symbols such as classes, methods, fields, variables, etc. This information is stored in a global symbol table. For this assignment, you will *not* write code to add information to the symbol table: the code for this is already given in the template Bison file. But you will need to access the symbol table (to refer to these entities from elsewhere in the AST). The symbol table structure and API are described below.
- **Statements:** Information about statements in the program are stored in these structures.

  Expressions: Information about statements in the program are stored in these structures.

  Types: Information about types in the program are stored in this structure.

Each of these parts is described in detail below.

Entities

The symbol table stores information about each identifier in the program. Each entry in the symbol table is called an "entity". For AST purposes, I have subclassed "Entities", based on its kind, into the following classes, each of which contain more detailed information about entities of that kind:

VariableEntity: This class contains information about variables in the input program. Each instance of this class has the following information:

- *name:* Name of the variable entity (inherited from `Entity`).
- *type:* A pointer to the type of this variable.
- *dimensions:* An integer, denoting the number of dimensions of this variable. (Scalar variables have dimension 0; Array variables have dimension > 0).

FieldEntity: This class contains information about the fields in the input program. Each instance of this class has the following information:

- *name:* Name of the field entity (inherited from `Entity`).
- *visibility_flag:* A boolean, indicating whether the field is public or not. I.e. *visibility_flag* is `true` is the field is public, and `false` otherwise.

  Note that the default visibility is public.

- *static_flag:* A boolean, indicating whether the field is a static field or not. I.e. *static_flag* is `true` is the field is static, and `false` otherwise.

  Note that a field is non-static by default.
- *type:* A pointer to the type of this field.
- *dimensions:* An integer, denoting the number of dimensions of this field. (Scalar field have dimension 0; Array fields have dimension > 0).

MethodEntity: This class contains information about the methods in the input program. Each instance of this class has the following information:

- *name:* Name of the method entity (inherited from `Entity`).
- *visibility_flag:* A boolean, indicating whether the method is public or not. I.e. *visibility_flag* is `true` is the method is public, and `false` otherwise.

  Note that the default visibility is public.
- *static_flag:* A boolean, indicating whether the method is a static field or not. I.e. *static_flag* is `true` is the method is static, and `false` otherwise.

  Note that a method is non-static by default.
- *return_type:* A pointer to the return type of this method.
- *formal_params:* A list of entity pointers representing the formal parameters of this method. Note that each formal parameter is an instance of *VariableEntity*.
- *method_body:* A pointer to this method's body statement.

ConstructorEntity: This class contains information about the constructors in the input program. Each instance of this class has the following information:

- *name:* Name of the constructor entity (inherited from `Entity`).
- *visibility_flag:* A boolean, indicating whether the constructor is public or not. I.e. *visibility_flag* is `true` is the constructor is public, and `false` otherwise.

  Note that the default visibility is public.
- *formal_params:* A list of entity pointers representing the formal parameters of this constructor. Note that each formal parameter is an instance of *VariableEntity*.
- *constructor_body:* A pointer to this constructor's body statement.

ClassEntity: This class contains information about the class symbols in the input program. Each instance of this class has the following information:

- *name:* Name of the class entity (inherited from `Entity`)
- *superclass:* Superclass (if any) of this entity. This field points to the superclass entity. If the entity has no super class, then this field is NULL.
- *class_members:* A list of entities representing the members (i.e. fields, methods and constructors) of this class entity.

The fields themselves are private and cannot be directly accessed; use the accessor methods of the same name to get a field's value. For instance, the type of a variable entity `ve` can be accessed as `ve->type()`.

Some fields may be modifiable after the entity is created; the modification methods are denoted by their `set_` prefix. For instance, to change the type of a variable entity `ve`, to a different type, say `newtype`, you can use `ve->set_type(newtype)`.

In addition, each of these classes have a `print` method which writes out the information to `cout`.

For the purposes of this assignment, the only symbol table method that you will need to invoke is:

`Entity* find_entity(char *name, Kind kind, bool *current):` searches for an entity of a given name and kind in the symbol table, and returns the entity reference (NULL if not found).

Name of an entity is a string, represented by a character pointer. Kind refers to the kind of entity: e.g. class entity, method entity etc. It is an enumerated type; see `include/Symbols.hh` for the definition of Kind.

The third parameter "`current`" is used to return whether the entity was found in the current block or in an enclosing block; this is used to check if there are duplicate definitions. For the purposes of this assignment (mpre precisely, in the code you will write) you will ignore the value of `current`.

The entity classes are defined in file `include/AstSymbols.hh` Take a look at that header file for a fuller understanding of the API.

Statements:

Statements are a set of classes implementing the following tree grammar:

```
statement:
      IfStatement(expression, statement, statement)
    | WhileStatement(expression, statement)
    | ForStatement(statement, expression, statement, statement)
    | ExprStatement(expression)
    | ReturnStatement(expression)
    | BlockStatement(list of statement)
    | DeclStatement(list of entities)
    | BreakStatement
    | ContinueStatement
    | SkipStatement
```

More concretely, `Statement` is an abstract class ("pure virtual") from which classes representing the different kinds of statement are derived. For instance, `IfStatement` is a derived class (i.e. subclass) of `Statement` and has three fields: `expr`, a pointer to an instance of `Expression` class (see "Expressions" below) that denotes the condition in the if-statement; `thenpart`, a pointer to an instance of the `Statement` class that denotes the "then part" of the if-statement; and `elsepart`, a pointer to an instance of `Statement` class that denotes the "else part" of the if-statement. These classes are declared in file `Ast.hh`. As you can see, most of these class definitions are straightforward and the meaning of the various fields are clear from the context. The less obvious statement classes are described below:

- `ForStatement:` There are four parts to a `for` statement. For instance, consider the following code fragment:

  ```
  for(i = 0; i< n; i++)
     sum = sum+a[i];
  ```

  In the above example, `i=0` is called the loop initializer. This is a statement (actually, always a specific form statement called `ExprStatement`). The `init` field of `ForStatement` is a Statement pointer, and represents this component of for-statement. The fragment `i<n` is called the loop guard, and is represented by the `guard` field (an `Expression` pointer). The fragment `i++` updates the loop counter; this part is also in general an ExprStatement, and is represented by `update` field, which is a Statement pointer. Finally, the body of the loop (`sum = sum+a[i];`) is represented by `body` field which is also a Statement pointer.

- `BlockStatement:` When a sequence of statements is enclosed in braces (i.e. between "{" and "}") it forms a block statement. Thus BlockStatement has a single field, `stmt_list` which is a (pointer to) a list of Statement pointers. The order of statements in this list is the same as the order in which they appear in the program text. This list is implemented by the STL container `list`.

- `DeclStatement:` Represents variable declarations. The set of variables declared (i.e. pointers to VariableEntities) is represented by the field `var_list`.
- `ExprStatement:` Represents "statement-like" expressions such as assignments, method invocations, etc. This class has a single field `expr`, which is a pointer to the Expression instance.
- `SkipStatement:` Represents a "NOP" (do-nothing) statement. For instance, an if-statement with no else part is represented in the AST with an `IfStatement` object whose `elsepart` field points to a skip statement.

Expressions:

Expressions are a set of classes implementing the following tree grammar:

```
expression:
    AssignExpression(expression, expression)
    | MethodInvocation(expression, name, sequence of expressions)
    | BinaryExpression(binary_operator, expression, expression)
    | UnaryExpression(unary_operator, expression)
    | AutoExpression(auto_operator, expression)
    | ArrayAccess(expression, expression)
    | FieldAccess(expression, name)
    | NewInstance(name, sequence of expressions)
    | NewArrayInstance(type, name, sequence of expressions)
    | ThisExpression
    | SuperExpression
    | IdExpression(entity)
    | NullExpression
    | IntegerConstant(int)
    | FloatConstant(float)
    | BooleanConstant(bool)
    | StringConstant(string)
```

This tree grammar is implemented as a C++ datastructure in the same way as done for Statements: Expression is an abstract class, and for each kind of expression, there is a derived class. The definition of these classes is also in `Ast.hh`. Again, the fields of these data structures are clear from the context most of the time. The less obvious ones are described below:

- `MethodInvocation:` The first field is a "path" to the method, the object on which this method operates. For instance, in the expression `a.b.f(2)` is a method expression where the prefix `a.b` corresponds to the object for this method, the method name is `f` and its argument is a singleton list `2`. If a method is invoked without a "path" prefix, then it is assumed to operate on `this` object.
- `AutoExpression:` Expressions that do auto increment and decrement (e.g. `i++`) are represented as auto expressions.
- `FieldAccess:` Expressions of the form `a.b.c` are field access expressions. The prefix path (`a.b` in the above example) is the first field (`base`), and the field name (e.g. `c`) is in `name`.
- `NewInstance:` These expressions are used to create new instances of classes (e.g. `new List()`). The class (e.g. `List`, and the arguments to pass to the class constructor are the fields of `NewInstance`.

- `NewArrayInstance:` These expressions are used to create new instances of arrays (e.g. `new int[10][20][]`). The element type (e.g. `int`), dimensions (3 in this example) and the sequence of expressions representing array bounds (10,20 in this example) are the fields of `NewArrayInstance`.

Note that in the Decaf grammar, local variables are also parsed as field access expressions (those with empty `base`, due to the following grammar rule:

$$\begin{array}{rcl} \textit{field\_access} & ::= & \textit{primary} \;.\; \textit{ID} \\ & | & \textit{ID} \end{array}$$

So we need a way to distinguish between actual field access expressions and simple local variable access. For instance, when we see `x`, we need to resolve whether this is a field access (short for `this.x`) or a reference to a local variable `x`.

For field access with empty prefix path (e.g. expression `x` parsed using the second rule above), we will do this resolution as follows. First we will check if the name matches a variable entity accessible from the current scope[1]. If so, `x` will be resolved as a simple local variable, and we will generate an `IdExpression` instead of a `FieldAccess` expression. If there is no variable called `x`, we look for a field entity of the same name; if one is found, then `x` is an `IdExpression` that refers to this field entity. If there is no field entity that matches, then we look for a class entity with the same name, and generate an `IdExpression` that refers to this class entity.

Note that if there is no entity that matches a given name, we cannot signal error yet. For instance, that name may refer to a field inherited from the super class (and hence not in the same lexical scope). To handle, this, instead of throwing an error, we postpone the name resolution to a later stage by generating a `FieldExpression` with `this` as the `base` (i.e. treat `x` as `this.x`).

Types:

Types are a set of classes implementing the following tree grammar:

```
type:
     IntType
   | FloatType
   | BooleanType
   | StringType
   | VoidType
   | ClassType(entity)
   | ErrorType
```

This tree grammar is implemented in C++ class in the usual way.

Project Path

You will be adding actions to the `decaf.yy` file given to you. Perhaps the most complex part of AST building is creating relevant entities in the symbol table. The good news is, this part is done for you: the code for this is already in `decaf.yy`. For this assignment, you need to focus only on the AST for statements and expressions.

Apart from the actions themselves, you need to decide what type of attributes you'll maintain for the different grammar symbols in order to successfully build the AST. This is specified in two parts. First, you will define a set of fields in YYSTYPE that can be used to store the different attribute values. Second,

[1] Use the `find_entity` method of the symbol table to do this. The global symbol table is in variable `global_symtab` in the template Bison file given to you

you will write specifications that associate the attribute of a grammar symbol to the corresponding field of YYSTYPE. This is done using the "%type" directives. See `decaf.yy` for examples of such directives.

Doing this project top-down may make it easier to make progress in small steps. For instance, start with reading `decaf.yy` to understand how AST for classes is built (ignoring fields, methods and constructors inside them). Then look at how field declarations are processed, followed by method and constructor declarations. By now, you will have a good idea of how to proceed with constructing the AST for statements and expressions. Once you familiarize yourselves with parts of the AST API, you can start by first adding in fields, and then methods and constructors.

As the second step, try to add action to each rule of "Stmt", where the action consists of simply creating a `SkipStatement` and assigning it as the `.ast` attribute; and uncomment the actions for variable declaration statement and error statement. When you do this, you will see that Bison will need you to specify a "%type" directive for nonterminal `stmt`; this may also involve adding to the "%union" declaration. As the last part of the second step, add actions to "StmtStar" nonterminal to create a list of statements (actually a list of pointers to `Statement` objects). When this step is successfully completed, you will see that you can successfully print the AST at the top level (the body of methods may still be blank since all statements are represented as skip statements!)

Now you can move into generating AST for statements but ignoring expressions (by treating them as, say, NullExpression). The statement part is particularly simple, so you may not need to break things down any further. Once all the statements have been worked out, you can move on to expressions.

In principle, you must be able to complete the AST construction by modifying only `decaf.yy`. If you need any functions in addition to what the AST API offers, you should write your own. You can stick this code at the end of `decaf.yy`, or in a separate file.

Available Material

The following material is available in the accompanying `.tgz` archive:

- A `src` directory containing source files for the driver, lexical analyzer, symbol table, and the AST data structures.
- A `include` directory containing header files for the symbol table and AST data structures.
- A `tests` directory containing sample input files `in*` and the respective output files `out*`.
- Decaf grammar specification in Bison (`src/decaf.yy`).

Deliverables

If you implemented AST construction by modifying `decaf.yy` alone, you can simply submit the new `decaf.yy`. If you wrote code in other files, then archive the entire directory structure (include and src directories) along with the modified Makefile using tar and submit that archive. Again: if you change any file other than `decaf.yy`, please submit them all, including those you did not change! If you touched only `decaf.yy`, submitting that alone is sufficient.

Please use the assignment form on Blackboard to upload your submission.