

## Espresso Abstract Syntax Trees

This document describes how to use the code for the Abstract Syntax Trees (ASTs) for the Espresso programming language. The code described here is in the files `ast.h` and `ast.cc`.

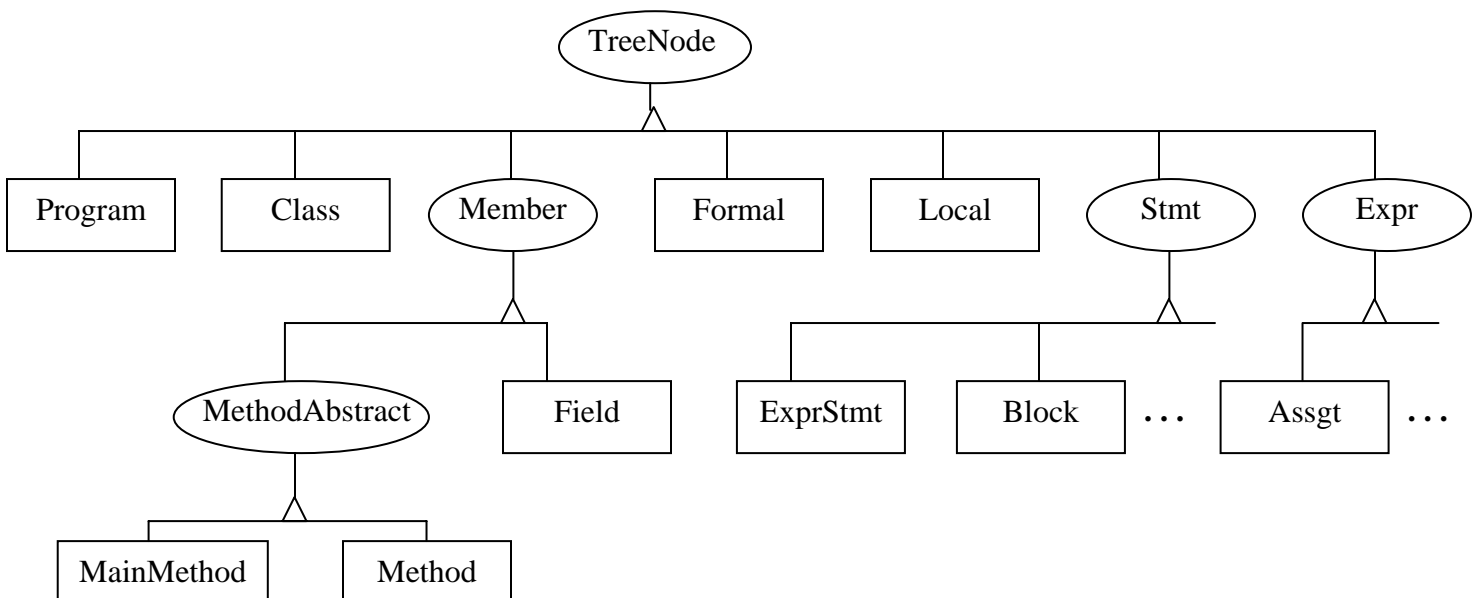
ASTs for compilers generally need many different node types, because, for the various programming language constructs, you will need to store different kinds of data and differing numbers of subtrees. For example, to represent a call to `println`, you need one subtree for the expression we are printing, but for an `if-else` statement, you will need three subtrees: one for the condition (an expression), and the other two for the statements that comprise the two branches. Yet, because `if-else` and `println` statements can be used in the *same* place in the language, for example, either one can appear in the list of statements in a method, it would be useful if they were derived from a common statement class. This way, we can use a variable of type `Stmt*` anywhere a statement can go, no matter what kind of statement it is. Using inheritance this way, we can also write polymorphic functions to traverse the tree to perform some action on all the nodes of the tree, that has a common interface, but different implementations. For example, we can call `dump` on a `Stmt`, to print that subtree, without having to know what kind of `Stmt` it is exactly. Each `Stmt` subclass has its own `dump` method. (There's more about `dump` and polymorphic tree traversals in the last few sections of this document.)

Note: Because these are trees, and because we're using inheritance in C++, all of these objects are allocated dynamically, and are referenced through pointer types (e.g., `Stmt*`).

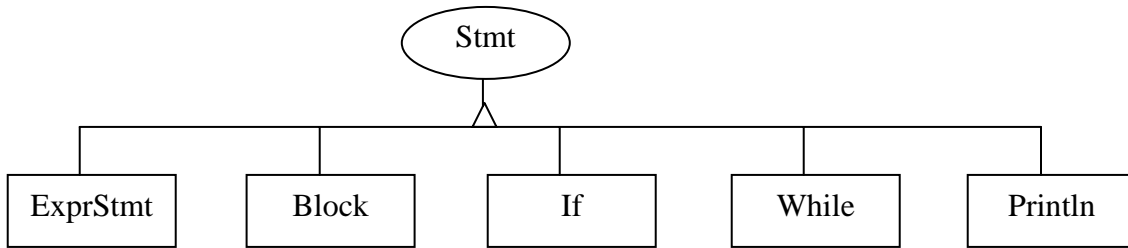
There are two major kinds of nodes in the tree, the ones inherited from `TreeNode`, which have a fixed number of subtrees, and those that are instantiations of STL vectors, which are for representing lists of subtrees. We'll describe each of these separately.

## Non-list classes

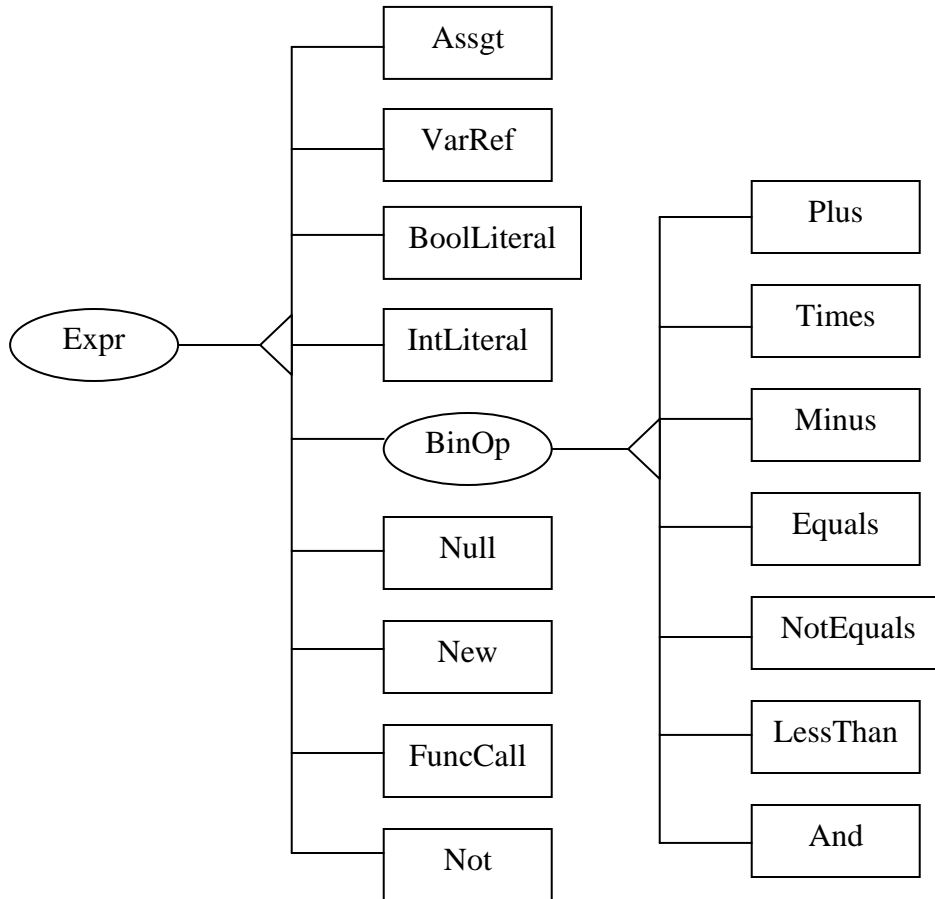
The base class of the hierarchy is `TreeNode`. Here is the inheritance hierarchy for the non-list classes, except for the subclasses of `Stmt` and `Expr` (to be shown in the next diagrams). For these diagrams we're using a variant of UML class diagrams such that we'll show abstract base classes (ABCs) as ovals, and concrete classes as rectangles. Recall that ABCs cannot be instantiated.



### Statement Classes



### Expression Classes



## List ASTs

When an AST node can have an arbitrary number of children, such as the actual parameters that are the children of a method-call node, we use what we call here a *list AST*. A list AST is represented as the appropriate instantiation of the STL vector template class. To make the code easier to read and write, we have provided typedefs for each of these template instantiations (see `parser.h`). For example, for a list of `Stmts`, `StmtList` is defined as follows:

```
typedef vector<Stmt *> StmtList;
```

Here are all the list AST types:

- **ClassList**
- **MemberList**           for the list of members (Field or Method) of a class
- **FormalList**
- **LocalList**           for local variable declarations in a Method
- **StmtList**
- **ExprList**

They all follow the same naming scheme, such that the element type for a list AST type is a pointer to the type that appears in the list AST type name (e.g., `MemberList` has elements of type `Member*`). Here is an example of how to build and print one of these lists (the list operations are shown in bold to just for emphasis):

```
StmtList *sl = new StmtList(); // create an empty statement list
Stmt *s;
Symbol *id = . . .;
                // create some kind of statement
s = new ExprStmt(new Assgt(id,
                          new IntLiteral(100)));

sl->push_back(s);           // add a statement to the end of list
sl->push_back(new Println(NULL));
                          // add another statement to the list

dump(cout, sl, 0);        // print out elements of the the list
                          // one per line, indenting them by 0 spaces
```

Note: The `dump` function for list ASTs is a non-member function since we can't add member functions to STL vectors.

Note that we are going to carry around *pointers* to list ASTs, just like we do with other AST nodes. For example, the constructor for a `Block` (a kind of `Stmt`) takes a `StmtList*` as its parameter (this constructor is shown in the next section of this document). So, all list ASTs should be created dynamically, as shown in the example above.

## Class Constructors

All nodes will be dynamically allocated, so you will always be calling `new` to create a node. Here we just show the constructor name (i.e., class name) and the formal parameters for that constructor for each concrete node type.

The only time you will pass a `NULL` argument to any of these constructors is for optional parts. They will be noted again below, but are only allowed for the optional `else` in an `if` statement, and for the optional argument to a `println` statement. Any constructor parameters that are `List` types may never be `NULL`: empty lists are not represented as `NULL` pointers; you create an empty list by using `new` with the appropriate default list constructor (see previous section for details of constructing lists).

The purpose of the formal parameters in the following should be self-explanatory from their names.

- Derived from **TreeNode**

**Program**(Class \*mainClass, ClassList \*classList)

**Class**(Symbol \*name, MemberList \*memberList)

Note: There is no special type for the Espresso class containing `main`. However, for this “main class”, we’ll create it such that its `memberList` will contain just one object that will be of the specialized type `MainMethod` (see constructor for `MainMethod` a few lines down).

**Formal**(Symbol \*type, Symbol \*name)

**Local**(Symbol \*type, Symbol \*name)

- Derived from **Member** (subclass of **TreeNode**)

**Field**(Symbol \*type, Symbol \*name)

- Derived from **MethodAbstract** (subclass of **Member**)

**MainMethod**(FormalList \*formalList,  
                  LocalList \*localList,  
                  StmtList \*stmtList)

**Method**(Symbol \*name,  
          FormalList \*formalList,  
          Symbol \*returnType,  
          LocalList \*localList,  
          StmtList \*stmtList,  
          Expr \*returnExpr)

Note that since `return` can only appear in a restricted place in an Espresso program, it is not a separate `Stmt` subclass, but rather, its expression is just stored as part of a `Method`.

- Derived from **Stmt** (subclass of **TreeNode**)
  - ExprStmt**(Expr \*expr)
    - ExprStmt is for an expression used as a statement. Only certain kinds of expressions can be used this way (see grammar in Espresso Language Reference).
  - Block**(StmtList \*stmtList)
  - If**(Expr \*cond, Stmt \*thenPart, Stmt \*elsePart)
    - The elsePart may be NULL, for an if without an else.
  - While**(Expr \*cond, Stmt \*body)
  - Println**(Expr \*arg)
    - The arg may be NULL, for an println without an argument.
  
- Derived from **Expr** (subclass of **TreeNode**)
  - Assgt**(Symbol \*id, Expr \*expr)
  - VarRef**(Symbol \*id)
    - VarRef is for using a variable in an expression. (E.g., on the right-hand-side of an assignment expression.)
  - BoolLiteral**(bool value)
  - IntLiteral**(int value)
  - Null**()
  - New**(Symbol \*objectType)
  - FuncCall**(Expr \*objExpr, Symbol \*funcName, ExprList \*args)
    - objExpr is for the expression left of the dot.
  - Not**(Expr \*expr)
  
- Derived from **BinOp** (subclass of **Expr**)
  - Plus**(Expr \*left, Expr \*right)
  - Minus**(Expr \*left, Expr \*right)
  - Times**(Expr \*left, Expr \*right)
  - Equals**(Expr \*left, Expr \*right)
  - NotEquals**(Expr \*left, Expr \*right)
  - LessThan**(Expr \*left, Expr \*right)
  - And**(Expr \*left, Expr \*right)

## How you will be using the ASTs in your parser

For the parser, the actions in your bison file will call the appropriate constructor(s) (using `new` to create nodes), to create the AST, bottom-up, for the Espresso program parsed. For lists, you will be calling constructors with `new` to create empty lists and `push_back` to add elements to lists. After parsing is done the given `testparser.cc` calls `dump` on the tree to show the results of parsing. The dump routines are described more in the next section.

## The dump routines

There are dump routines provided for each class, so that working together, they print out the whole tree in pre-order, using indentation to indicate subtrees. For most AST classes `dump` is a member function, but, as shown earlier, the list ASTs use a non-member `dump` function.

One of the things printed by the dump functions for each node is the line number that was saved with the tree node when it was created. This initialization was done “behind your back” directly from the global variable (see code for class `TreeNode`). Also, it will show the line number for the *last* line of the Espresso construct because it’s only after the parser has seen all the tokens that make up the construct that the root of this piece of the tree gets built in bottom-up parsing.

For each `Expr` node it also prints a field called `type`, for storing the type of the expression. During the parsing phase, all of the `Expr` nodes will show `no_type` for the type. You will be updating the `type` field as part of the semantic analysis phase; you don’t have to worry about it in the parser.

### **How you will be using the ASTs in the later phases of compilation**

For the semantic analyzer and code generator you will be adding to the tree data structure (e.g., adding the type info, as mentioned above), but also, you will need to modify the tree code itself to add any necessary fields and/or member functions. Much of this will code be in the form of polymorphic tree-traversal code. E.g., for type-checking (part of semantic analysis), to `typeCheck` a `Plus` node, you need to `typeCheck` its subtrees in turn, which could be any `Expr` node, so it’s useful to have a polymorphic `typeCheck` function that works on any kind of node – no `switch` statements involved. A good example of polymorphic tree-traversal code is the set of `dump` routines that we wrote for you. We have tried to design the inheritance hierarchy so that you can take advantage of commonalities between classes by putting some of your code in superclasses, but made subclasses for where you need data or functions specific to a single Espresso construct.

When you modify the code for the last phases you will only be adding things, not removing them or moving them around. Also *do not change* the structure of the class hierarchy (i.e., which classes are defined or which ones are inherited from which other ones), because that will necessitate rewriting your parser and the dump routines. There is more documentation related to adding functionality to the AST classes in `ast.h`.