

# CS126: Introduction to Compilers

## Assignment 5: SYN2

**Out: 10/3/03**

**Due: 10/10/03**

### 1.0 Introduction

In the last two assignments you built, tested and expanded your parser for DECAF. This time you are going to extend that parser so that it produces abstract syntax trees.

### 2.0 Abstract Syntax Trees

We are providing you with the definition and implementation of the abstract syntax trees you are going to need for DECAF. The top-level definition for an abstract syntax tree node is in the file `DecafAst.java`. The actual node definitions can be found in the files `DecafSymbol.java`, `DecafScope.java`, `DecafType.java` and `ast/Ast.java`:

```
ast/Ast.java      All actual AST nodes (expressions, statements, etc.)
DecafSymbol.java Abstract AST node for symbols
DecafScope.java  Abstract AST node for scopes
DecafType.java   Abstract AST node for types
```

Most of the AST nodes you can generate by calling the appropriate constructor for the node type that is relevant. However, type nodes are special in that there should be a unique instance of each type node that is shared by all uses of the type. These nodes are created by calling the appropriate static method of `DecafType`. Also, most of the list nodes provide an ‘add’ method to add a child to the end of the current list.

You will also need to use some of the types defined elsewhere in Decaf when building the AST nodes. In particular, you will need to use the types `SymbolKind`, `Operator` from `DecafConstants`, and the type `DecafModifier`.

We went over what AST should be generated for each DECAF construct in class. The following table summarizes that discussion:

<b>Production</b>	<b>AST</b>	<b>Notes</b>
Start -> Class+	ClassList ( Class ... )	
Class -> <b>class</b> <i>id</i> Super? { Member* }	Class ( SuperClass MemberList )	Atom id;
Super -> <b>extends</b> <i>id</i>	MemberList ( Member ... )	
Member -> Field   Method   Ctor	Super (Type)	use Object

<b>Production</b>	<b>AST</b>	<b>Notes</b>
Field -> Modifier* Type VarDeclList	FieldDecl ( Field ...) Field ( Initializer)	int modifiers Type type;
Method -> Modifier* Type <b>id</b> FormalArgs Block	Method(Type,MethodBody) MethodBody(FormalList StatementList)	Atom id; Atom id; int modifiers;
Ctor -> Modifier* <b>id</b> FormalArgs Block	Constructor(Type,MethodBoty)	Atom id; int modifiers;
Modifer -> ...		int
FormalArgs -> ( FormalArgList? )	FormalList ( Formal ...)	
FormalArgList -> FormalArg   FormalArg , FormalArgList		
FormalArg -> Type VarDeclId	Formal ( Type )	Atom id;
Type -> PrimitiveType		
Type -> <b>id</b>	ClassType	Atom id;
Type -> Type [ ]	ArrayType	
PrimitiveType -> <b>boolean</b>   <b>char</b>   <b>int</b>   <b>void</b>	Boolean, Char, Int, Void (also Null, Init, Meta)	
VarDeclList -> VarDecl { , VarDecl}* VarDecl -> VarDeclId [ = Expression] VarDeclId -> <b>id</b> { [ ] }		int count;
Block -> { Statement* }	StatementList ( Statement ...)	
Statement -> ;	EmptyStatement	
Statement -> Type VarDeclList ;	DeclStatement (Local ...)	Atom id; int count;
	Local(Expression)	init expr
Statement -> <b>if</b> ( Expression ) Statement [ <b>else</b> Statement ]	IfStatement(Expression,Statement) IfStatement(Expression,Statement,Statement)	
Statement -> Expression ;	ExpressionStatement(Expression)	
Statement-> <b>while</b> ( Expression ) Statement	WhileStatement(Expression,Statement)	
Statement-> <b>return</b> [Expression] ;	ReturnStatement ReturnStatement(Expression)	
Statement-> <b>continue</b> ;	ContinueStatement()	
Statement -> <b>break</b> ;	BreakStatement()	
Statement ->Block	BlockStatement(Block(Statement ...))	
Statement -> <b>super</b> actual_args ;	SuperStatement(Call(Name(Name[super],[<init>],ExpressionList))	
Expression -> Expression BinaryOp Expression	OpExpression(Expression,Expression)	OP op;
Expression-> UnaryOp Expression	OpExpression(Expression)	Op op;
Expression -> Primary		
Primary -> NewArrayExpr		
Primary ->NonNewArrayExpr		
Primary -> <b>id</b>	Name	Atom id;

Production	AST	Notes
NewArrayExpr -> <b>new</b> <i>id</i> Dimension+   new PrimitiveType Dimension+ Dimension -> [ Expression ]	NewArray(Type,Expression...)	
NonNewArrayExpr -> Literal		
NonNewArrayExpr -> <b>this</b>	Name	
NonNewArrayExpr -> ( Expression )		
NonNewArrayExpr -> <b>new</b> <i>id</i> ActualArgs	Call(Name(New(Type),[CTOR]),ExpressionList)	
NonNewArrayExpr -> <i>id</i> ActualArgs	Call(Name(Name[CLASS],[id]) ExpressionList)	
NonNewArrayExpr -> Primary . <i>id</i> ActualArgs	Call(Name(Expression,[id]),ExpressionList)	
NonNewArrayExpr -> <b>super</b> . <i>id</i> ActualArgs	Call(Name(Name[super],[id]),ExpressionList)	
NonNewArrayExpr -> ArrayExpr		
NonNewArrayExpr -> FieldExpr		
FieldExpr -> Primary . <i>id</i>	Name(Expression,[id])	
FieldExpr -> <b>super</b> . <i>id</i>	Name(Name[super],[id])	
ArrayExpr -> id Dimension	ArrayRef(Name[id],Expression)	
ArrayExpr -> NonNewArrayExpr Dimension	ArrayRef(Expression,Expression)	
Literal -> <b>null</b>   <b>true</b>   <b>false</b>   intLiteral   charLiteral   stringLiteral	LiteralNull(), LiteralBoolean(), LiteralInt(), LiteralChar(), LiteralString()	value
ActualArgs -> ( ExprList? )	ExpressionList(Expression...)	
ExprList -> Expression { , Expression }		

## 3.0 Assignment

You should modify your parser to generate abstract syntax trees by adding appropriate actions to your JavaCC specification. If necessary, you can add helper routines in your Parser class.

### 3.1 Locations

As noted in class, the abstract syntax trees track the locations at which they were created. To facilitate this, they do two things. First, they have a method

```
void setLocation(Location)
```

that allows the location to be set explicitly. Second, the constructors for the various AST nodes will all take a Location as the first parameter. Note that the underlying AST support code will ensure that an AST node has a location that is the minimum of its location and any subnodes.

## 3.2 Handling Multiple Files

If you are allowing multiple files to be parsed in a single run, you should ensure that these share a `ClassList` abstract syntax tree. This should probably be managed by your `Parser` class implementation. Note that when your parser is first called, this part of the abstract syntax tree will already have been setup. (The tree will also include all the predefined classes and methods.) You can get the root of the syntax tree by calling `DecafScope.getOuterScope()`.

## 3.3 Printing the AST

To debug your parser, you can run DECAF with the `-parse_only` option, which will cause it to print the AST in a machine readable form.