Stage 2: Symbol Table and AST

Due Apr 25 by 11:59pm **Points** 100

Before You Started...

Please change the value of variable *TEST_TO_RUN* at

```
src/test/java/CompilerStageTests.java, line 22
```

into

```
private final String[] TEST_TO_RUN = {"stage2"};
```

to use test cases for project 2 when you run *mvn test*. Or to use the following one to run both project 1 and project 2 test cases:

```
private final String[] TEST_TO_RUN = {"stage1", "stage2"};
```

Introduction

In this assignment you're going to build a symbol table and AST (Abstract Syntax Tree) from the parse tree you generated in last assignment.

Goal

Print the AST of the input Crux source code.

For example, given a Crux source code:

```
int foo(int a) {
  return a + 9487;
}
```

We have the AST:

ast.DeclarationList(1,4)

ast.FunctionDefinition(1,4)[Symbol(foo), [Symbol(a)]]

ast.StatementList(2,6)

ast.Return(2,6)

ast.OpExpr(2,15)[ADD]

ast.VarAccess(2,13)[a]

ast.LiteralInt(2,17)[9487]

The compiler template contains a pretty printer for the AST. When we run your compiler with the `--print-ast` option, it should print out:

```
foo(a) {
  return (a + 9487);
}
```

Project Guidance

The Parse Tree Visitor

You're going to build an AST based on the parse tree generated by ANTLR. However, we can't implement a virtual method, that all classes in the parse tree, that represent some sort of declaration, must implement. Hence, we need another way. One way to do so is to inspect the contents of every node and invoking the correct methods to process it's children. For example, for lowering a `DeclarationContext` into a `ast.Declaration` one could check, whether the declaration contains a `VariableDeclarationContext`, an `ArrayDeclarationContext` or a `FunctionDefinitionContext` to decide whether to invoke the method for creating a `ast.VariableDeclaration`, an `ast.ArrayDeclaration` or a `ast.FunctionDefinition`.

This is tedious however, and ANTLR also implements the visitor pattern on the parse tree to be able to implement virtual methods for certain classes in an extra class. This is achieved by overriding a virtual method `accept(CruxVisitor v)` on each node in the parse tree. The `CruxVisitor` implements for every class of the parse tree a method `visit`. For example, `VariableDeclarationContext` implements `accept` by calling the matching method on the `CruxVisitor` like this (simplified):

```
@Override
public void accept(CruxVisitor visitor) {
   visitor.visitVariableDeclaration(this);
}
```

Our implementation of `CruxVisitor` now can override the method `visitVariableDeclaration` to add its own behavior.

```
@Override
public void visitVariableDeclaration(VariableDeclarationContext ctx) {
    // Lower to VariableDeclaration
}
```

ANTLR provides further the possibility to add return values to every method. We make use of that in the template and implement a `CruxVisitor` each for `Declaration`, `Statement` and `Expression`.

To learn more about visitors, check out the Wikipedia article about the Visitor Pattern (http://en.wikipedia.org/wiki/Visitor_pattern) .

Hint

You can see the Java code generated by ANTLR by looking in the target/generated-sources/antlr directory in your repository.

Symbol and Symbol Table

We create a Symbol class, so that our compiler can model the variables and functions that Crux programmers declare in their programs. Every time the input source declares a variable or declares a function, the compiler creates a Symbol instance to represent it. A SymbolTable stores each of these Symbols for later retrieval, when the Crux program uses the variable or calls the function.

Modeling Scope

Semantically, the Crux has a layer of *scope* for each function. According to the Crux grammar, curly braces only occur in a statement-block. However, the statement-block occurs in one of two different places:

1. The body of code for a newly declared function.
2. The block of code for an if branch, else branch, or while loop.

Conceptually, each time the Parser encounters and statement-block it introduces a new scope to contain any newly declared Symbols (variables and functions). This rule is loosened somewhat because Crux's symbol semantics specify that parameters of a function are also scoped with the function body.

The Crux grammar allows curly braces to nest (an if-else inside a while loop inside a function). Therefore, our SymbolTable must model the nesting of scopes. Valid Crux source code, contains balanced curly braces; for every open brace there is a close brace. We can therefore treat the nested scopes as a *stack* of SymbolTables; one table for each scope. The parser models the entry of a scope by calling the enterScope() helper method. Similarly, when a scope closes (a closing curly brace), the parser calls exitScope().

Symbol Table Design Hints

Conceptually, the SymbolTable forms a list of `Map<String, Symbol>`. Each table has a pointer to a parent table, representing the outer scope. The `lookup(String name)` method *recursively* walked the list of tables, proceeding from the innermost scope to outermost scope. The lookup method returns the first Symbol matching name. Otherwise, it signals that no such symbol by the given name exists in any of the scopes.

Detectable Symbol Errors

Although we introduce Symbols in this Lab, we will not be detecting type-invalid usage until a later lab. For this lab, we only check that symbol names do not clash (Redefinition Error) and that they exist (Unknown Symbol Error). If some Crux source code declares and variable and later uses that name to call a function, we shall consider it valid because the name exists in the table when the function is parsed.

Resolve Symbol Errors

Symbols are not available outside of the scope (set of curly braces) where they are declared. If some Crux source code attempts to use a variable or function name that has not been declared in the current scope or any outer scope then the symbol table signals a `ResolveSymbolError`.

We implement this functionality through a helper method in the symbol table: Symbol `lookup(String name)`. When the Parser encounters an identifier use, it calls `lookup` to perform a lookup in the symbol table. If the lookup signals a failure, then `lookup` logs a `ResolveSymbolError`. Otherwise, lookup succeeds and returns the resulting Symbol so that the Parser may resume parsing.

Redefinition Errors

In some other languages, lookup uses both type and name to find the correct symbol. For example, Java allow function overloading and can distinguish with method is meant by inspecting the argument types at the call site. Crux is a simple language, and uses only the name to distinguish among possible symbols. Within each scope, all of the Symbols carry a unique name. If some Crux source code attempts to declare two variables, or two functions, or a function and a variable by the same name, in the same scope, then the symbol table signals a `DeclareSymbolError`.

We implement this functionality through a method in the symbol table: `Symbol add(String name)`. When the parser encounters a variable declaration or function definition, it calls `add` to perform an insertion in the symbol table. If the insertion signals a failure, then `add` logs a `DeclareSymbolError`. Otherwise, insertion succeeded and `add` returns the newly created Symbol so that the Parser may resume parsing.

Predefined Symbols

The Crux Specification contains a section describing certain predefined functions. We consider these functions to be built-in, because the Crux programmer doesn't have any way to implement them. However, programming in Crux would be might useless without the simple abilities represented by these functions. Before parsing, the symbol table should be pre-loaded with symbols representing these functions.

Design Goals for the AST

The AST that we create must faithfully represent the Crux program being compiled. Additionally, we seek to make the AST as clear and easy to use as possible. Because we will later perform traversals over the AST to check for semantic constraints, we consider all of the following issues in the design:

- **Concise**: We should like to clean up any unnecessary features that may be present in the Crux source. For example, the AST does not need to extra parentheses that may have been used in an expression.
- **Meaningful**: Nodes in the AST should carry some kind of semantic meaning. For example, we must track when and where variables and functions are declared or defined.
- **Instructive**: Nodes in the AST should represent an action (or instruction) that a computer might take. For example, we can have one node represent an `IfStatement`. It can have 3 children: `condition`, `thenBlock`, and `elseBlock`.
- **Organized**: Nodes in the AST should be categorically distinguishable. That is, we should be able to identify the difference between statements and expressions.

An AST is not the Parse Tree

In the first lab, we used ANTLR to produce a parse tree from the Crux source. That tree records how a Crux sentence (input source code) is broken down into syntactic pieces according to the rules of the Crux grammar. Just as its name implies the *Abstract* Syntax Tree, *abstracts* away some of the pieces that might be present in the parse tree.

The AST avoids carrying extra syntax.

A Crux sentence is allowed to carry extra information that does not necessarily change the semantics of the program. For example, according to the Crux grammar parentheses can be used to nest expressions arbitrarily. Consider the following code examples, their parse trees and the corresponding AST.

Examples:

- **Example 1:**

  Crux:

  void foo() { if true {
  return 5; } }

  Parse Tree:

program

declarationList declaration functionDefinition parameterList type statementBlock statementList statement ifStatement expression0 expression1 expression2 expression3 literal statementBlock statementList statement returnStatement expression0 expression1 expression2 expression3 literal

```
AST:
```

ast.DeclarationList(1,0)

ast.FunctionDefinition(1,0)[Symbol(foo), []]

ast.StatementList(2,8)

ast.IfElseBranch(2,8)

ast.LiteralBool(2,11)[TRUE]

ast.StatementList(3,12)

ast.Return(3,12)

ast.LiteralInt(3,19)[5]

ast.StatementList(2,8)

- **Example 2:**

  Crux:

  void foo() { if (((((true))))) {
  return 5; } }

  Parse tree:

program declarationList declaration functionDefinition parameterList type statementBlock statementList statement ifStatement expression0 expression1 expression2 expression3 expression0 expression1 expression2 expression3 expression0 expression1 expression2 expression3 expression0 expression1 expression2 expression3 expression0 expression1 expression2 expression3 literal statementBlock statementList statement returnStatement expression0 expression1 expression2 expression3 literal

```
AST:
```

ast.DeclarationList(1,0)

ast.FunctionDefinition(1,0)[Symbol(foo), []]

ast.StatementList(2,8)

ast.IfElseBranch(2,8)

ast.LiteralBool(2,16)[TRUE]

ast.StatementList(3,12)

ast.Return(3,12)

ast.LiteralInt(3,19)[5]

ast.StatementList(2,8)

The AST has correct operator association.

In the Crux grammar, the expression chain (expression0 -> expression1 -> expression2 -> expression3) contains only right-associative rules, which generate a right-associative parse tree. In spite of the parse tree generated, the operators and, or, add, sub, mul, and div are, semantically, all left-associative. The parse tree accurately capture precedence, but incorrectly represent operator associativity. Using right association for the grammar rules aids the construction of a left-factored LL(1) grammar, which in turn aids writing a recursive descent parser. However, we must now take care to ensure that the AST captures the left-associative semantics of these operators.

- **Example:**

  Crux:

  int foo() { return 3-1-1; // == 1 }

  Parse tree:

program declarationList declaration functionDefinition parameterList type statementBlock statementList statement returnStatement expression0 expression1 expression2 expression3 literal op1 expression2 expression3 literal op1 expression2 expression3 literal

```
AST:
```

ast.DeclarationList(1,0)

ast.FunctionDefinition(1,0)[Symbol(foo), []]

ast.StatementList(2,5)

ast.Return(2,5)

ast.OpExpr(2,15)[SUB]

ast.OpExpr(2,13)[SUB]

ast.LiteralInt(2,12)[3]

ast.LiteralInt(2,14)[1]

ast.LiteralInt(2,16)[1]

Nodes in the AST

The AST sits somewhere between a parse tree and a list of instructions for a machine to follow. It contains fewer nodes than the parse tree, and organizes those nodes into semantic categories. It contains higher-level information than a list of instructions, including variable declarations and function

definitions. We intend the AST to be an intermediate representation that bridges the gap between source code and machine code.

The Node Interface

As a tree data structure, the AST is composed of nodes which inherit the interface Node. Each node instance stores the line number and character position of the source code where it begins. Concrete subclasses store more specific information, to faithfully represent nodes that actually occur in Crux source code. We create a node class to record the actions a computer takes during execution of a Crux program. For example, Crux has nodes for declaring variables, looping, creating constants, evaluating arithmetic and logical expressions, indexing arrays, etc.

Categorizing the subclasses.

For each node in the Crux source code we associate a subclass of Node. Some nodes can only occur in certain parts of the Crux grammar. For example, FunctionDefinition can only occur as part of a DeclarationList and not inside a StatementList. In contrast, both ArrayDeclaration and VariableDeclaration can occur in either a DeclarationList or a StatementList. We use these observations to break down the nodes into 3 categories, each represented by an interface: Declaration, Statement, Expression.

Node	Category (Interface)	Description
`ArrayDeclaration`	`Declaration` `Statement`	The creation of an array.
`VariableDeclaration`	`Declaration` `Statement`	The creation of variable.
`FunctionDefinition`	`Declaration`	The creation of a function.
`LiteralBool`	`Expression`	An embedded boolean constant, either `true` or `false`.
`LiteralInt`	`Expression`	An embedded integer number.
`VarAccess`	`Expression`	he occurrence of an identifier as part of an expression (represents the address of that symbol).
`OpExpr`	`Expression`	Represents basic arithmetic, logical operation, or comparison of one or two other expressions.
`ArrayAccess`	`Expression`	An operator for indexing into an array. The amount to index are expressions.
`Call`	`Expression` `Statement`	A function call, including an `ExpressionList` of arguments.
`Assignment`	`Statement`	An assignment of a source expression to a destination designator.

Node	Category (Interface)	Description
`IfElseBranch`	`Statement`	Represents an conditional if-else branch. Includes the condition expression, and a `StatementList` for each of the then and else branches.
`For`	`Statement`	Represents a for loop, including a `StatementList` for the body.
`Return`	`Statement`	A way for functions to return a value (and exit early).
`Break`	`Statement`	A way to exit the loop.

Creating the AST

As the parser recursively descends through the parse tree of an input Crux source code, it constructs the AST incrementally. We modify the methods responsible for recursive descent traversal so that the each returns a branch of the final AST. For example, because the program method parses a list of declarations, it returns a ast.DeclarationList. Likewise, each method in the expression chain returns an Expression, being careful to implement correct associativity for the operations involved. By returning AST nodes from each method, the Parser can build up the final AST as it unwinds the recursive traversal.

Submission

Please follow the submission guides in the **Project Overview (https://canvas.eee.uci.edu/courses/54574/pages/project-overview#submission)** . Make sure you pass all the public test cases. **Don't forget to put the academy honesty statement into your `README`!**