ANTLR TREE WALKING

Introduction

In this lab you will implement your AST generation for your coursework. You will also be given pointers on how to implement your symbol table.

Setup

Download the provided code from Moodle, unzip it, and import it to IntelliJ. The code demonstrates a complete working example of our s-expression grammar including a couple of tree walkers.

TreeListeners and TreeVisitors

When ANTLR generates your lexer and parser, it also generates a couple of helper classes for walking parse trees. For a language called X:

- `XListener` is an interface that defines methods for 'entering' and 'exiting' nodes of a parse tree, which can perform a computation but do not return anything;
- `XBaseListener` is an abstract class that provides empty implementations of all the `XListener` methods, so you only need to implement the ones you want;
- `XVisitor<T>` is an interface that defines methods for 'visiting' a parse tree node, performing some computation, and returning a value of type `T`; and
- `XBaseVisitor<T>` is an abstract class that provides empty implementations of all the `XVisitor<T>` methods.

TreeListener methods

For each nonterminal defined in your grammar, the listener defines two methods. For a nonterminal called Thing, the methods are

- `void enterThing(ThingContext ctx)`, and
- `void exitThing(ThingContext ctx)`

The 'enter' method is called when 'entering' a node during a tree walk (i.e. when the node is first encountered) and the 'exit' method is called when leaving the node (i.e. when the node has been fully processed, including its children). The actions you carry out in these methods are entirely up to you and depends on the purpose of the specific listener you are implementing.

The example implementation online prints out the structure of an s-expression as it enters and exits each node. Change the input expression and see how the output changes.

TreeVisitor methods

The visitor defines the same kinds of methods as the listener. For each nonterminal, a method of type `T visitThing(ThingContext ctx)` is generated. There are two key differences between the visitor and the listener:

1. The visitor only defines a single `visitX` method for each nonterminal, whereas the listener has separate `enterX` and `exitX` methods, and
2. The visitor methods all return a value of type `T`, whereas the listener methods are void.

The visitor also has a general visit() method which take an arbitrary context object and defers to the appropriate specific method depending on the concrete type of the context. In the example code, the visitor recursively calls visit to build up an AST from the subtrees of a parse tree.

The default implementations of the visit methods simply visit all the child nodes of the given node and return their result. If the programmer overrides a visit method, this does not happen and the programmer is responsible for manually specifying how to visit any child nodes and what order to do this in. In the example code, notice that we don't need to override `visitAtomExpr` or `visitListExpr` because those will automatically visit the relevant child nodes.

Constructing ASTs

The `AstBuilder` class in the demo code is a good model for how to use an ANTLR tree visitor to build an AST. The class extends `LispBaseVisitor` and uses our pre-defined `Ast` interface as the generic type. This is important because all visit methods must return the same type. Each visit method constructs an instance of the relevant concrete class by recursively visiting child nodes when necessary, casting them to the appropriate types, and returning the constructed AST class.

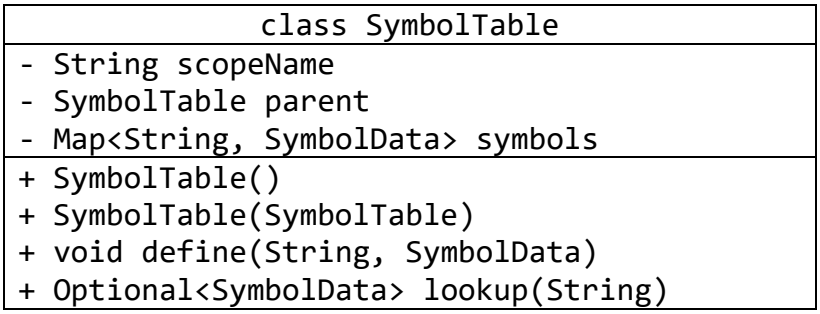
Task: Define your own version of AstBuilder for your coursework and implement a couple of simple conversions. Does this structure suggest any changes you might want to make to your AST classes?

Symbol Tables

As mentioned in the lecture, symbol tables are used to communicate information about symbols between the different phases of the compiler. This means that your AST builder needs a reference to the symbol table so it can record any symbols it finds.

In the example code, we do not track any real information about symbols, just their names. We use a simple set as the symbol 'table' and update it whenever we visit a symbol atom.

A more sophisticated symbol table might be structured around the class diagram below. This is only a guide; feel free to modify it if you have better ideas!



This class represents a single node in a hierarchy of scoped symbol tables, arranged as a tree. The first constructor creates a symbol table with no defined parent (assigning the parent member to null) while the second assigns a parent to the newly created node.

This data structure would need to be augmented with a method for finding an individual symbol table based on the name of its scope. Ideally, this would be an O(1) operation as it is something your compiler will do repeatedly.

Task: Implement a symbol table class based on the class diagram and your own ideas.

Task: Work out how to search for a table with a given scope. You could create a wrapper class that maps scope names to symbol tables or implement a tree search to find a scope from the root.

Task: Modify your coursework AST builder to receive a reference to your symbol table and think about how you could begin to populate the symbol table during AST generation.