ANTLR AND AST

Introduction

This lab will guide you in completing the ANTLR grammar for your coursework, and give you some prompts in designing the AST to use for your IR.

ANTLR grammars

We saw the basics of making an ANTLR grammar last week. This section covers a few more features of ANTLR that are worth knowing and shows how to generate and use a parser.

Repeated rules

We saw that rules in the lexical grammar can use symbols like ?, *, and + to stand for 0-1, 0-many, or 1-many respectively. The same symbols can be used with nonterminals in the syntax:

```
// traditional recursive definition
statementList: statement statementList
             | /* empty */
             ;

// convenient shorthand definition
statementList: statement*;
```

The generated parser will process 0 to many statement nonterminals as part of the statement list and record them all in the parse tree.

Named productions

By default, ANTLR will generate one node in the parse tree for each nonterminal in the derivation. If a nonterminal has several alternatives, they will all have the same node type and the programmer must distinguish between them manually.

ANTLR allows the programmer to ‘name’ the productions of a nonterminal; in this case, the parse tree will have a different type of node for each production. This can make it easier later to quickly identify which rule was parsed.

```
sexpr: atom   # atomSexpr
     | list   # listSexpr
     ;
```

Note that **all** alternatives must have a name, or **none** of them. You cannot name some and leave others unnamed. Note also that several alternatives can share a name.

Labelled components

The third feature to introduce is the idea of labelling specific parts of a production. This tells ANTLR to generate members in its parse tree nodes with specific names, which will make it quicker to refer to specific children of the node later:

```
salutation: ‘dear’ firstname=Name lastname=Name ‘,’;
```

The above rule uses the terminal `Name` twice. By default, the salutation node would contain a list of `Name`s, and the programmer would need to remember which was which. With labels, the node will have two separate members called `firstname` and `lastname` which makes the resulting code more readable.

Creating and using ANTLR parsers

When the ANTLR tool is run on a grammar file (or when ANTLR is run automatically as part of a Gradle build) it will generate several classes for you:

- A lexer (i.e. a scanner) to tokenise the input,
- A parser to process tokens according to the grammar,
- A ‘tree listener’ which performs actions for different parse tree nodes during a tree walk, and optionally
- A ‘tree visitor’ which visits the nodes of a parse tree and returns a value.

In a Gradle build in IntelliJ, the generated classes are placed under `build/generated-src/antlr/main/<package-name>`. You can look at these if you like, but don’t bother modifying them because they’ll get reset the next time you build your project.

Listeners and visitors are used for generating IRs, type checking, and other tasks. We’ll look at those in the next lab. For now, we will look at constructing and using the generated lexer and parser classes. For a grammar called X, ANTLR will have generated classes called `XLexer` and `XParser`.

Parser construction

The code for constructing a parser is quite boilerplate. The basic pattern is as follows:

```
XLexer lexer = new XLexer(CharStreams.fromString(s));
TokenStream tokens = new CommonTokenStream(lexer);
XParser parser = new XParser(tokens);
```

The first line constructs the lexer. The argument to the constructor is an instance of ANTLR’s `CharStream` interface that represents the input. There are several different static factory methods for constructing char streams from different sources; constructing from a string is quite common.

The second line constructs a `TokenStream` which is responsible for calling the lexer and producing tokens on demand. This is then provided as an argument to the parser constructor. You now have a usable parser object!

Parser use

Using the parser is simple. For every nonterminal in your grammar, the parser will have a method with the same name. When that method is called, it will attempt to parse a single instance of the specified nonterminal using the token stream provided when it was constructed. If it fails, it returns null; if it succeeds, it returns an object of type `XContext` where X is the name of the nonterminal.

For example, our Lisp grammar has a nonterminal called sexpr, we would attempt to parse a single s-expression like this:

```
// LispParser parser
SexprContext ctx = parser.sexpr();
```

Task: Finish implementing the s-expression parser from the last lab. Set up a main function that defines a simple s-expression in a string, then constructs a parser to parse it. Explore what methods exist on the context object (the simplest way of doing this is to use the method autocomplete in IntelliJ to list them).

Task: Implement the grammar for Nottran as far as you intend to go. It’s up to you whether you want to enter it all at once or start small and build it up as you progress.

Designing ASTs

Designing an AST for a language is relatively straightforward. You will typically require one node type for each meaningful nonterminal in your language (i.e. the nonterminals that directly correspond to language constructs, not the extra ones needed to avoid left recursion, etc.). Each node will need some way of recording its children, and optionally its parent.

An example set of AST classes for Lisp might look like this:

```
interface Ast {}
interface Atom extends Ast {}
record Number(int value) implements Atom {}
record Symbol(String value) implements Atom {}
record LispList(List<Ast> elems) implements Ast {}
```

Notice how we use empty interface to group types together, so that anywhere that expects an atom could receive either a number or a symbol as an argument.

Question: Would there be a benefit to making these interfaces sealed?

Task: Work out the AST classes required to represent a Nottran program. Do you need a similar hierarchy of interfaces to represent everything?