Quick ANTLR Introduction

ANTLR\(^1\), ANother Tool for Language Recognition, provides support for a number of useful features that allow for the rapid development of a simple compiler front-end. Industrial-strength front-ends can also be created, but they are beyond the scope of this discussion. Of specific interest for our purposes are lexing, parsing, and tree parsing. These topics will be discussed further below. Note, however, that this document is not intended to fully document ANTLR (that is already done on the ANTLR website). This document is a rapid introduction to the features of ANTLR and an attempt to highlight some common stumbling points\(^2\).

1 Lexical Analysis

Lexical analysis (or lexing/scanning) is the process of breaking an input into legal tokens. A lexer is defined in ANTLR, as will be seen, in a syntax similar to that used for the later parts of the front-end. A lexer definition (in a .g file\(^3\)) begins with a line like the following.

```
class MyLexer extends Lexer;
```

All lines that follow, until either end-of-file or the definition of another ANTLR component, are part of the lexer definition. As you might expect (and can readily verify), this definition will be turned into a class. Each rule (see later) will become a method in that class.

1.1 Options

The next part of the definition, when needed, is an options section that sets some options for the lexer. For example,

```java
options {
    k=2; // set lookahead
    charVocabulary=' '..'\u007F'; // allow ascii
}
```

There are other options, but they are not of immediate interest.

1.2 Rules

Though not immediately the next section, the section of most immediate interest is the definition of the lexing rules. These rules define the behavior of the lexer. Each rule must have a name beginning with an uppercase letter. Some simple examples are:

```
EQUAL : "=" ;
LE : "\<" ;
TIMES : "\*" ;
```

\(^1\)ANTLR was written by Terence Parr and can be found at http://www.antlr.org.

\(^2\)Additional advice on common problems will be added as more students use this tool.

\(^3\)Though one can break the lexer, parser, and tree parser into different files, this document assumes that all are in the same file.
1.2.1 Regular Expression Operators

One can also use regular expression operators to denote repetition (*, +), optional (?), negation (~), and inclusive range (..). For example,

ID
   options {testLiterals=true;}
   : ('a'..'z' | 'A'..'Z') ('a'..'z' | 'A'..'Z' | '0'..'9' | '_')* ;

This rule specifies that an ID can be an alphabetical character (upper- or lowercase) followed by zero or more alphanumeric characters plus the underscore. Since most keywords, if not reserved, would be valid IDs, the testLiterals option instructs the lexer to compare each ID against a set of literals specified by the programmer in the tokens section (see section 1.3).

1.2.2 Actions

One can also specify actions that will take place after a match. These actions are just snippets of code that will be executed as the lexer breaks the input into tokens. For example, this next rule defines the whitespace tokens and upon matching instructs the lexer to skip the token.

WS
   : ( ','
       | '\t'
       | '\f'
       // handle newlines
       | ( options {generateAmbigWarnings=false;}
         : "\r\n" // Evil DOS
         | '\r' // Macintosh
         | '\n' // Unix (the right way)
         )
         { newline(); /* update line number */ }
       )+
       { $setType(Token.SKIP); }
   ;

1.2.3 Syntactic Predicates

Occasionally a set of rules will match in their prefix. Such rules can be combined, but the lexer needs additional hints to determine which rule to use to complete processing of a token. This is where a syntactic predicate comes in handy. Consider the following rules.\footnote{Note that declaring a rule protected simply means that only other rules can invoke it. You can think of it as a protected rule.}

\protected
   DIGIT : '0'..'9'  ;
\protected
   INT : ( DIGIT )+  ;
\protected
   REAL : INT , INT  ;
\protected
   NUMBER : ( INT '.' ) => REAL { $setType(REAL); }
           | INT { $setType(INT); }
   ;

These rules together differentiate between an integer token and a floating point (real) token. The action code sets the token type. Few of your lexer rules will be this complicated.
1.3 Tokens
As mentioned previously, some lexer rules may define a set of tokens that includes what are commonly considered “keywords”. Unfortunately, such rules do not distinguish between tokens that belong to the set, and tokens that would normally belong to the set but that should be considered separately. To differentiate between such token types, one can define a tokens section.

```java
tokens{
    FUN="fun";
    IF="if";
    THEN="then";
}
```

This section defines a set of tokens and their corresponding token types. This allows one to use FUN in the parser instead of using ID and then checking for the actual string “fun” at a later point.

1.4 Literal Code
Just before your rules, you can specify some code (methods and declarations) that will be literally included in your generated lexer class. These methods and fields can be accessed within the actions for your rules. This code is specified between a pair of curly braces. For example,

```java
{
    private static final String SOMETHING = "SOMETHING";
    private boolean flag = false;

    protected void methodOne()
    {
        ...
    }

    public void uponEOF()
        throws TokenStreamException, CharStreamException
    {
        ...
    }
}
```

The uponEOF method will be automatically invoked by the lexer once it scans an end-of-file.

1.5 Summary
The following is a high-level view of the structure of a lexer definition:

```java
... imports ...
class LexerName extends Lexer;
... options ...
... tokens ...
... literal code ...
... rules ...
```

The header and imports sections were not discussed previously. The header section (labeled with header) is useful for placing the generated files in a Java packaged. The imports section works much like the literal code section and is useful for imports in Java or for includes in C++.
2 Syntactic Analysis

Syntactic analysis (or parsing) is the process of matching tokens against the specified grammar. As previously noted, a parser is defined in ANTLR in a manner very similar to that for the lexer. In fact, the same sections can be defined for the parser as for the lexer and they serve the same general purposes.

The definition of a parser begins with a class declaration of the following form.

class MyParser extends Parser;

2.1 Options

The options section contains option settings specific to the parser.

options
{
  k=1; // set lookahead
  buildAST=true; // let ANTLR build the AST
  defaultErrorHandler=false; // don’t use ANTLR’s error handling
}

As before, the k value determines the number of lookahead tokens. A higher value can be used to eliminate (in some cases) ambiguity, but also typically results in slower execution time.

The buildAST flag instructs ANTLR to construct an abstract syntax tree during parsing. This is an extremely useful feature that will be discussed further in section 2.4 and exploited by the tree parser discussed in section 3.

The defaultErrorHandler flag can be set to false to allow the programmer greater control over error handling.

As before, additional options can be found in the documentation found on the ANTLR website.

2.2 Grammar Rules

The parser’s grammar rules are defined much like the rules for the lexer. Moreover, the grammar rules are very similar to those of EBNF which makes encoding an EBNF grammar very simple.

Each rule begins with a symbol (non-terminal) denoting the name of the rule (this name is used as the name for the generated method). The symbol is followed by a colon (:) and then any number of alternatives separated by pipes (|). Furthermore, parentheses can be used for grouping within a rule to allow for nested alternatives (useful for operators). Each rule must be terminated by a semicolon (;).

Some examples,

expr_or_decl :
  (declaration | expression)? SEMI

declaration :
  VAL ID EQUAL expression
  | FUN ID formals EQUAL expression

simple :
  term ((PLUS | MINUS) term)*

The first rule demonstrates the use of a nested alternative as well as an optional part of the rule (denoted by a ?). The second rule has two simple straightline alternatives. Notice that those portions of the rule in all capitalized letters are tokens (terminals) defined by the lexer. The third rule demonstrates repetition using Kleene closure (recall that + can also be used to require at least one instance).
2.3 Tokens

The parser’s tokens section is used to introduce “imaginary” tokens. These tokens can be used (as discussed in the next section) in the construction of ASTs, but they don’t necessarily correspond directly to a token in the parsed language. They are often useful in distinguishing between alternatives of the same grammar rule and for constructing a tree that is more easily parsed by the tree parser.

As a simple example, consider the following.

```
tokens
{
    FORMALS;
    BINDINGS;
    EXPR;
    DECL;
    BODY;
    LIST;
    NULL;
    APP;
    TYPE;
    TUPLE;
}
```

Alone, these tokens are not of any obvious use. Some will be used in the next section.

2.4 Tree Building

If the buildAST flag is set in the options section, then ANTLR will attempt to build an abstract syntax tree as it parses the input. The grammar rules above, however, would lead to poorly formed trees that are of little help to the tree parser. To address this problem, ANTLR provides annotations for normal tree building and support for explicit tree building.

2.4.1 Annotations

There are two annotations that may be used.

- carat (^): This annotation is used to specify the root of a subtree.
- bang (!): This annotation is used to exclude a portion of the grammar from the subtree.

Consider the following grammar rule and modifications made to it (for each example, the parsed input is the “string” `a b c`).

```
start : ID ID ID ;
```

The tree constructed by the rule without annotations is:

```
X-AST
```

This is obviously not a very useful tree as it does not contain information about the other two IDs. Adding a root annotation will change the manner in which the tree is constructed. Though a root isn’t required within a specific rule, it is necessary for multiple nodes at the same “level” to be rooted, or some will be lost.

A first example of a root annotation follows.

```
start : ID^ ID ID ;
```
The tree constructed by this rule is rooted on the first ID.

Placing the root annotation on the middle ID changes the structure of the tree. All other nodes are now children of the middle ID.

\[
\text{start : ID ID^ ID ;}
\]

Similarly, the next example demonstrates moving the root annotation to the last ID.

\[
\text{start : ID ID ID^ ;}
\]

Finally, multiple root annotations are allowed. In this example, you can see that the first ID (which will have already been parsed) becomes a subtree of the second (the first root). This constructed tree becomes a subtree of the next root (at the final ID).

\[
\text{start : ID ID^ ID^ ;}
\]

Finally, as a simple demonstration, the bang operator can be used to exclude a subtree. This is useful in excluding keywords (such as \texttt{else}) and symbols (such as \texttt{(}) that are of no use to the tree parser.
2.4.2 Explicit Tree Construction

At times it is useful to explicitly construct your abstract syntax trees. ANTLR provides mechanisms to do just that. When constructing your own trees, imaginary tokens (discussed in section 2.3) are useful as the roots of subtrees.

There are two main reasons to construct your own trees. The first is to distinguish between different alternatives. The second is to construct a tree that simplifies analysis in the tree parser.

First consider the following rule.

```
expr_or_decl : (declaration | expression) ? SEMI! ;
```

As written, the tree returned from this rule will be either that returned from declaration or that returned from expression (which could have any of a number of root nodes). It is convenient in writing the tree parser if the number of root nodes is limited to a small set.

Consider this alternate definition.

```
expr_or_decl : (declaration { #expr_or_decl = #([DECL, "DECL"], #expr_or_decl); } | expression { #expr_or_decl = #([EXPR, "EXPR"], #expr_or_decl); } )? SEMI! ;
```

The action code (between the {}) constructs a new node for each alternative. Let’s consider each part in turn.

- **assignment target**: In this assignment, the tree bound to the rule itself is being changed. This is denoted by assigning to the name of the rule prefixed by a # (to specify tree access).
- **new tree**: The new tree is constructed by #(), where the contents between the ()’s become the nodes of the tree. The first element is the root. All subsequent elements (separated by commas) become subtrees of that root.
- **new root**: The new root is constructed using []’s. The first label in the []’s is the imaginary token. The string that follows is not strictly necessary, but is used as the node’s label in the AST frame.
- **old tree**: The old tree is that which was automatically generated by ANTLR. One can access this in the same manner as the assignment target.

All explicit tree construction will work in a manner similar to the example above. Consider another more complex example.
declaration

: VAL^ ID EQUAL! expression
|! FUN i:ID f:formals EQUAL e:expression
  { #declaration = #(FUN, #i, #f, #([BODY, "BODY"], #e)); }

This rule actually mixes default tree construction with explicit tree construction. The AST for the first alternative (VAL) is constructed entirely by ANTLR. The construction of the AST in the second alternative is entirely explicit. Note the bang (!) immediately following the pipe (|) in the second rule. This annotations instructs ANTLR to *not* construct a default tree for this alternative. Instead the tree will be constructed in the action code.

Note also that portions of the second alternative are labeled. This is for convenience of reference in the action code. As before, a new tree is being assigned for the rule (assigned to #declaration). This tree is constructed with FUN (which is an actual token in this grammar\(^5\)) at the root. Its subtrees are the ID (denoted by #i), the formal arguments (denoted by #f), itself a tree, and a subtree that is being constructed in place. This subtree is itself rooted at the imaginary token BODY. This construction (with the appropriate rules for the other non-terminals) results in the following tree.

As a final example, the following demonstrates one way to build a tree for the right-associate operator that simplifies handling in the tree parser. This required splitting the original grammar rule into two. The original rule was

\[
\text{listterm} \rightarrow \text{simple} \{ :: \text{simple}\}^*
\]

\(^5\)Note carefully that since FUN is *not* an imaginary token, the construction here differs slightly from before. Specifically, the []'s are excluded.
In the following ANTLR rules, the token :: is denoted by CONS.

listterm!
  : s:simple c:cons_extra
    {
      #listterm = #c.equals(#(#[NULL,"NULL"])) ?
      #s :
      #(#[CONS, "CONS"], #s, #c);
    }
  ;
cons_extra!
  : (CONS! s:simple) rest:cons_extra
    {
      #cons_extra = #rest.equals(#(#[NULL,"NULL"])) ?
      #s :
      #(#[CONS, "CONS"], #s, #rest);
    }
  | { #cons_extra = #(#[NULL, "NULL"]); }
  ;

Consider cons_extra first. It begins with a bang (!) annotation before the colon (:). This instructs ANTLR to suppress automatic tree construction for the entire rule (instead of a specific alternative). In the first alternative, if a :: operator is found, then a simple is parsed, and an attempt is made to continue parsing. Once complete (i.e., when the subsequent call to cons_extra is finished), the abstract syntax tree for this alternative is constructed. If nothing was found after the simple call (as signified by a programmer-defined NULL tree), then the tree returned by simple (labeled #s) is returned directly (it was the last simple). Otherwise, a CONS node is constructed combining #s with #rest. Similar logic is used in the listterm rule. The result of this construction is the following on input 1 :: 2 :: 3 :: a.

You may find it illustrative to trace through this example to see how this tree is constructed.

2.5 Customized AST Nodes

ANTLR its own node type to construct the abstract syntax tree. This is often sufficient, but it is 0 useful to direct ANTLR to use a class of your own design instead. For example, one might want to annotate certain nodes
with additional information.

The following example demonstrates a simple extension to store line numbers in (most) AST nodes. This example begins with a simple class that will store line numbers and provide later access to them.

```java
import antlr.*;

public class LineNumberAST
    extends CommonAST
{
    private int _line;

    public LineNumberAST()
    {
        super();
    }

    public LineNumberAST(Token tok)
    {
        super(tok);
        _line = tok.getLine();
    }

    public void initialize(Token tok)
    {
        super.initialize(tok);
        _line = tok.getLine();
    }

    public void setLine(int line)
    {
        _line = line;
    }

    public int getLine()
    {
        return _line;
    }
}
```

To direct ANTLR to use this class, after instantiating the parser, but before invoking it, call the `setASTNodeClass` method with the name of the class (e.g., `parser.setASTNodeClass("LineNumberAST")`). Invocation of this method results in the specified class being used when nodes are constructed.

The `setASTNodeClass` method has a global effect. It is also useful to change a specific node type. This can be done with an annotation in the parser definition. For example, the following rule specifies that a instance of `AnnotatedFunctionAST` should be used when constructing the subtree for `FUN`.

```java
function
    ::=  FUN<AST=AnnotatedFunctionAST>
        id:ID
        p:parameters
        r:return_type
        LBRACE
        d:declarations
        s:statement_list
        RBRACE
```
Though you may not need to use these features, do keep them in mind as they can be used to great effect.

2.6 Summary

The following is a high-level view of the structure of a parser definition:

```java
... imports ...
class ParserName extends Parser;
... options ...
... tokens ...
... literal code ...
... rules ...
```

3 Tree Parser

The final aspect of ANTLR discussed in this document is a tree parser. ANTLR’s support for tree parsing is really just a set of features supporting node matching and tree traversal. The tree parser is, in a sense, a realization of the visitor pattern. In defining the tree parser rules, you are actually defining the actions to take at each node. Those actions then trigger invocations of additional rules. For this reason, a tree parser is often called a tree walker.

While walking the tree, the tree parser can carry and compute information about the program (e.g., type information or evaluation information). In addition, nodes of the tree can be stored and retrieved for later parsing. This can be done using either the ANTLR syntax or the ANTLR AST API.

As before, a tree parser definition begins with a declaration.

```java
class MyTreeParser extends TreeParser;
```

3.1 Options

The only option of immediate interest (which may be required depending on how you setup your files) is `importVocab`. This option specifies to the tree parser where the token definitions are stored. The value of the option will depend on where you have defined your tokens, but will likely be either the name of the lexer or the parser.

```java
options {
    importVocab=MyLexer;
}
```

3.2 Literal Code

More so than in the lexer or parser, it is often convenient to include additional helper methods as part of the definition of a tree parser. These methods can be invoked from the action code in a rule to perform additional...
computation. Furthermore, as mentioned previously, it is occasionally convenient to use such methods to make calls back into the tree parser.

Consider the following example. This method takes an AST node as an argument and then makes a call back into the parsing code. The actual argument is expected to be an actual subtree passed from one of the rules. This method then (typically after some additional logic) invokes a tree parser rule and passes the subtree as an argument.

```java
private void foo(AST node)
{
    // run tree parser on node
    this.rule(node);
}
```

### 3.3 Rules

Tree parser rules look very similar to the rules used in the lexer and parser. The difference, however, is that a tree parser attempts to match the rules against the shape of a tree, whereas the parser matched against a token stream and the lexer matched against a character stream.

Consider the following abbreviated example (which just happens to be recursive). This rule attempts to match the current AST node against one of the four alternatives. If the current node has any of the four tokens `PLUS`, `MINUS`, `STAR`, or `DIV` as its root and two `expr` as its subtrees, then this rule will successfully parse the tree. If the tree is a leaf with root ID\(^6\), then no further parsing takes place.

```plaintext
expr
 : #(PLUS expr expr)
 | #(MINUS expr expr)
 | #(STAR expr expr)
 | #(DIV expr expr)
 | INT
```

When parsing, only a “sufficient” match is done, not an exact match. This means that if the current node matches an alternative of a rule up to the end of the rule, then that rule will match even if the tree node has additional children. For example, a node of the form `#(PLUS INT INT INT INT)` will still match the rule above.

### 3.3.1 Return

A rule can be modified to return a value. For example, the `expr` rule, representing expressions, can be modified to return the result of the expression. This is done with a `returns` clause that specifies the return type, a variable name for the return value, and, typically, an initial value. Values are then “returned” by assigning to the variable specified.

To access the returned values from other rules, a pattern can include assignments to local variables. Local variables can be defined in a literal block between the rule symbol and the colon (`:`). Local variables are then assigned to (using `=`) as part of the parse. Note that variables used in this manner must be declared, whereas labels (such as that on INT in the following example) are not declared.

The following example illustrates a simple calculator. The tree parser walks the tree computing the values of each expression as it goes (the values are propagated from the bottom up).

```plaintext
expr returns [int r=0]
{
    int a,b; }
 : #(PLUS a=expr b=expr) {r = a + b;}
 | #(MINUS a=expr b=expr) {r = a - b;}
```

\(^6\)Note that a match that does not specify children to match is not enclosed in `#()`.
| #(STAR a=expr b=expr) {r = a * b;} |
| #(DIV a=expr b=expr) {r = a / b;} |
| i:INT {r = (int)Integer.parseInt(#i.getText());} |

Note that $i$ is a label on INT. This results in the tree at that point being bound to $i$ (implicitly declared as an AST reference). Variables $a$ and $b$ are declared local to the rule and take on the values returned by the calls to $\text{expr}$.

### 3.3.2 Delaying Parsing

There are times when it is useful to delay parsing for part of a tree. Instead of immediately parsing the tree, it can be saved and parsed at a later time. A label can be used to get a reference to an actual AST node, but the label itself does not prevent parsing. Consider the following bizarre example.

```plaintext
expr returns [int r=0]
{ int a,b; }
  : #(PLUS a=expr b=expr) {r = a + b;}
  | #(MINUS a=expr b=expr) {r = a - b;}
  | #(STAR a=expr b=c:expr) {r = a * b; System.out.println(c.toStringTree());}
  | #(DIV a=expr b=expr) {r = a / b;}
  | i:INT {r = (int)Integer.parseInt(#i.getText());}
;
```

Though $c$ does get bound to the subtree representing the second operand of STAR, that subtree is still parsed. To prevent parsing, the invocation of $\text{expr}$ (which is what that portion of the rule is translated into) must be removed. This can be done using a . wildcard\(^7\). The above example can be modified as follows.

```plaintext
expr returns [int r=0]
{ int a,b; }
  : #(PLUS a=expr b=expr) {r = a + b;}
  | #(MINUS a=expr b=expr) {r = a - b;}
  | #(STAR a=expr c:.) {r = a * expr(c); System.out.println(c.toStringTree());}
  | #(DIV a=expr b=expr) {r = a / b;}
  | i:INT {r = (int)Integer.parseInt(#i.getText());}
;
```

In the new version, the second child is bound to $c$. The value of that subtree is computed in the action code by explicitly invoking the parsing rule for $\text{expr}$ and passing $c$ as its argument. Of course, this example is very simplistic. Realistically, $c$ could have been passed to another method (perhaps defined in the literal code block) or stored in a collection.

### 3.3.3 Parameters and Exceptions

In addition to returning values, parsing rules can take parameters and throw exceptions. Parameters are specified in a comma-separated list enclosed in []'s just after the symbol name. A throws clause can be specified immediately following a returns clause. Thus, all together, the definition of a rule closely matches that of a method in Java, C++, and C# (though the returns clause is in a different place).

So, in general, a rule may begin with the following form (all before the colon (:)).

```plaintext
symbol [type\(_1\), parameter\(_1\), ..., type\(_n\), parameter\(_n\)] returns [type name=initial value]
  throws exception\(_1\), ..., exception\(_n\)
```

The following example illustrates the use of a parameter, a return, and an exception. This example extends the calculator to support identifiers (though the code to assign a value to an identifier is not shown).

\(^7\)One could also take advantage of the “sufficient” match feature and use the ANTLR AST API to explicitly access different parts of the tree.
expr [HashMap bindings] returns [int r=0]
   throws InvalidIdException
{ int a,b; }
   : #(PLUS a=expr[bindings] b=expr[bindings]) {r = a+b;}
   | #(MINUS a=expr[bindings] b=expr[bindings]) {r = a-b;}
   | #(STAR a=expr[bindings] b=expr[bindings]) {r = a*b;}
   | #(DIV a=expr[bindings] b=expr[bindings]) {r = a/b;}
   | id:ID
   { Integer val = (Integer)bindings.get(id.getText());
      if (val == null)
      { throw new InvalidIdException("unbound identifier: " + id.getText());
      }
      r = val.intValue();
   }
   | i:INT {r = (int)Integer.parseInt(#i.getText());}
;

3.4 Lists

Some abstract syntax trees will have a list of children each with the same root node. One can specify tree parses using the repetition (*) and optional (?) operators just as with a token parser. For example, consider a tree constructed according to the following rule.

foo
   : (ID)* EOL! { #foo = #([LIST, "LIST"], #foo); }
   ;

This rule will construct a tree of the following form.

![AST Diagram]

The following tree parser rules will visit each child and print its ID value. Note well that the action code is inside of the repetition. If it were placed after the repetition, then only the last node ID would be printed.

foo
   : #(LIST bar)
   ;
bar
   : (id:ID { System.out.println(id); })*
   ;

One can combine the above rules into a single rule, but this can be cumbersome. For example,

foo
   : #(LIST (id:ID { System.out.println(id); }))*
   ;
3.5 Summary

The following is a high-level view of the structure of a tree parser definition:

```
... imports ...
class TreeParserName extends TreeParser;
... options ...
... literal code ...
... rules ...
```

4 Putting it All Together

Once the lexer, parser, and tree parser are defined, they can be used by instantiating and then invoking the appropriate methods corresponding to the start rules. Consider the following snippet of (Java) code (note that this code reads from standard in, but could easily be changed to read from a file).

```java
MyLexer lexer = new MyLexer(System.in);
MyParser parser = new MyParser(lexer);

// invoke the parser
parser.start_rule();

// if a tree was built, get it
AST t = parser.getAST();

// incredibly useful for debugging, displays AST in a frame
if (wantGUI)
{
    ASTFrame frame = new ASTFrame("AST", t);
    frame.setVisible(true);
}

// invoke tree parser
MyTreeParser treeparser = new MyTreeParser();
treeparser.start_rule(t, ... any necessary arguments for rule ...);
```