ANTLR: A Compiler Development Tool

Stephen A. Edwards

ANTLR comprises the functionality of a lexer generator (e.g., supplanting lex or flex), a parser generator (e.g., supplanting YACC or Bison), and an AST walker. It is highly customizable, has been under development for many years, and generates code in Java, C++, or Sather formats. I have found it to be very flexible and embody many best compiler design practices.

To illustrate ANTLR, I show how it can be used to build a grammar for a simple calculator-like language that parses and interprets "programs" that look like

```
foo = 3 + 4;
bar = 2 * (5 + 4);
print "The answer is";
print foo + bar;
if foo+bar then print "non zero";
```

1 The Lexer

Figure 1 shows the lexer for the simple langauge. Running this through ANTLR produces a Java class called "SimpLexer" that produces tokens from the input stream. Figure 4 shows how a lexer is created, connected to the standard input stream (this could also have been a file), and passed to the parser.

Lexer rules begin with a capital letter and contain grammar-like specifications for the text of a token.

Unlike Lex or Flex, ANTLR generates predictive lexers that behave much like recursive-descent parsers. The first k characters of a token must be enough to distinguish which non-protected rule is in force. For this example $k = 2$ (set in the options section) because the lexer needs to look two characters ahead to decide when a string constant terminates.

Protected rules in the lexer can be used by other rules, but do not return tokens by themselves. In this example, LETTER and DIGIT are used in the rules for ID and NUMBER; the parser will never see LETTER or DIGIT tokens by itself.

ANTLR handles keywords differently than automata-based scanners. Rather than having a separate rule for each keyword and construct a complicated automata that can identify them, ANTLR assumes keywords are caught by other rules (typically identifiers, as in this lexer, although any rule can be used in this way: the rule for PARENS is used in this way in this example). After each token is recognized, the scanner checks its literals table to see if the text for the token matches one of the literals (e.g., a keyword). If the text matches, the scanner returns the type of the literal token, not the rule that matched it.

An advantage of this approach is that the parser has a simple mechanism for passing keywords ("if," "then," "else," "(," and ")" in this example) to the lexer. Literal strings—in double quotes—in the parser are entered in the lexer's literal table and matched.

This technique of matching token text can have pitfalls. We do not want a string such as "if" being parsed as a keyword. In this example, I avoided this problem by turning literal matching off for all but the ID and PARENS rules; the ANTLR-generated lexer will not check its literal table when it encounters a string.

```
class SimpLexer extends Lexer;
options { testLiterals = false; k = 2; }

PLUS   : '+' ;
MINUS  : '-' ;
TIMES  : '*' ;
DIV    : '/' ;
ASSIGN : '=' ;
PARENS options { testLiterals = true; }
       : '(' | ')' ;
SEMI   : ';' ;

protected LETTER : ('a'..'z' | 'A'..'Z') ;
protected DIGIT : '0'..'9' ;

ID options { testLiterals = true; }
  : LETTER (LETTER | DIGIT | '_')* ;
NUMBER : (DIGIT)+;

STRING : '"'! ('"' '"'! | ~('"'))* '"'! ;

WS : ( ' ' | '\t' | '\n' { newline(); } | '\r' )
     { $setType(Token.SKIP); } ;
```

Figure 1: The Lexer for the simple langauge.

Actions can be included in the lexer by enclosing Java code in braces. I've used an action to discard whitespace: by setting the type of the token to "SKIP," the lexer discards the token and goes on to the next. A call to "newline" in the rule for whitespace counts the number of lines.

Like all of ANTLR, lexer specifications use an extended BNF form in rules. In this example, I've used the Kleene star operator to represent zero or more instances of letters, digits, or underlines in the rule for ID.

The rule for STRING is rather cryptic, but very convenient because of ANTLR's ability to modify the text of a token as it is being scanned (impossible with an automata-based scanner). The chosen syntax for string constants is a sequence of characters enclosed in double quotes `"`. A double quote is included in the string by doubling double quotes, i.e., `""`. The lexer rule modifies the string as it is being scanned so the text of the string token is as desired: the `!`s after the first and last double quotes discard them when they are encountered, removing the surrounding quotes, and the `!` in the body discards the second of two double quotes in a row. Thus, only the first double quote in a pair is copied into the text, exactly what the escape sequence implies.

2 The Parser

Figure 2 shows the parser for the simple langauge. ANTLR builds LL(k) recursive-descent parsers. The advantage of such an approach is that actions can be inserted anywhere within the

```
class SimpParser extends Parser;
options { buildAST = true; k = 2; }

file : (expr SEMI!)+ EOF!;

expr
  : "if"^ expr "then"! expr
    (options {greedy=true;} : "else"! expr)?
  | "print"^ (STRING | expr)
  | ID ASSIGN^ expr
  | expr1
  ;

expr1 : expr2 ( (PLUS^ | MINUS^) expr2 )* ;
expr2 : expr3 ( (TIMES^ | DIV^) expr3 )* ;
expr3
  : ID
  | "("! expr ")"!
  | NUMBER
  | MINUS^ expr3
  ;
```

Figure 2: The parser for the simple langauge.

grammar, including very early in a rule, and that the operation of the parser is easy-to-understand because it is essentially a direct translation of the grammar. The disadvantage of this approach is that LL(k) langauges are more restricted than the LALR(1) grammars YACC or Bison parse. This example has $k = 2$ because it is necessary to look two tokens ahead to recognize the assignment operation (both assignment and expr1 can start with an ID, but only assignment may have an "=" following it).

Rules in the parser begin with lowercase letters and contain EBNF expressions (include Kleene closure `()*`, zero-or-one `()?`, one-or-more `()+` in addition to sequencing, choice, and grouping) describing the grammar for each rule.

Unlike YACC or Bison, an ANTLR grammar needs operator precedence and associativity stated implicitly. E.g., in this example, the "if" "print" and assignment operators are at the lowest precedence, $+$ and $-$ are next, then $*$ and $/$, and finally atoms. This is perhaps the most awkward aspect of specifying ANTLR grammars compared to other parser generators.

Many grammars are inherently ambiguous. A typical example is the "dangling else" problem: there is confusion over which "if" owns an "else" clause when if-then-else statements nest. The usual solution is to attach the "else" to the nearest "if," and this parser is no exception. This is specified by setting the "greedy" option in the optional "else" clause part of the "if" rule. Normally, the generated code for parsing an optional clause needs to decide whether to try to parse the optional clause or continue one. It does this by looking at tokens that start the optional clause as well as those that may follow it. An "else" token appears in both sets because the "expr" being parsed could be the one after a "then", which may have an "else" following it. The greedy option makes the parser prefer the optional clause over skipping it when such ambiguity arises.

ANTLR parsers can automatically generate ASTs; this example takes advantage of this facility. By setting the `buildAST` option true, by default every token becomes a sibling of the AST being constructed, but annotations can refine this behavior. A `!` following a token supresses the generation of an AST node (e.g., for punctuation such as parentheses and semicolons). A `^` following a token makes it the root of a new subtree. This is used in virtually every rule to impose structure on the AST.

By default, ANTLR builds homogeneous ASTs (i.e., every node

```
class SimpWalker extends TreeParser;
{ java.util.Hashtable dict = new java.util.Hashtable(); }

file { int a; } : (a=expr)+ ;

expr returns [ int r ]
  { int a, b, c; r = 0; }
  : #("if" a=expr b=expr {c = 0;} (c=expr)?
      { if (a != 0) r=b; else r=c; } )
  | #("print"
       ( s:STRING { System.out.println(#s.getText()); }
       | a=expr { System.out.println(a); } ) )
  | #(ASSIGN ID a=expr
      { dict.put(#ID.getText(), new Integer(a)); } )
  | #(PLUS a=expr b=expr { r = a + b; } )
  | #(MINUS a=expr b=expr { r = a - b; } )
  | #(TIMES a=expr b=expr { r = a * b; } )
  | #(DIV a=expr b=expr { r = a / b; } )
  | ID { if ( !(dict.containsKey(#ID.getText())))
  System.err.println("unrecognized: "+#ID.getText());
         r = ((Integer) dict.get(#ID.getText())).intValue();
       }
  | NUMBER {
     r = Integer.parseInt(#NUMBER.getText(), 10);
   }
  ;
```

Figure 3: The AST walker for the simple language.

is a member of the same class), but it has extensive facilities for automatically building more complicated ASTs.

3 The AST Walker

Once the parser has built the AST, it's useless unless you do something with it, such as traverse it to check static semantics, transform it into a lower-level representation, or, in this case, execute it in an interpretive style.

Figure 3 shows ANTLR rules for an AST walker that interprets the AST. The rules look much like those for a parser, but they operate on trees. The syntax `#(PLUS expr expr)` means "match a tree whose root is a PLUS token with two children that match the expr rule." No lookahead is used: the root of each tree must be enough to disambiguate among multiple rules. Note also that no precedence rules are necessary: the structure of the AST already embodies them from parsing.

The rule for expr is the only interesting one. The rule definition says it returns an integer—the value of the expression—and the generated method contains local variables a, b, and c. The actions simply evaluate any child expressions then compute and return the value of the expression. Assignment enters the value of its expression in a hash table indexed by the name of the variable, and the rule for ID attempts to retrieve such an entry by searching on the text of the token—the identifier itself. The number rule converts the text of its token—the actual number—into a base-10 integer before returning it.

```
import antlr.CommonAST;
class Simp {
  public static void main(String[] args) {
    try {
      SimpLexer l = new SimpLexer(System.in);
      SimpParser p = new SimpParser(l);
      p.file();
      SimpWalker w = new SimpWalker();
      w.file((CommonAST) p.getAST());
    } catch (Exception e) {
      System.err.println(e);
}}}
```

Figure 4: The driver for the simple langauge.