The Definitive ANTLR Reference

Building Domain-Specific Languages

Chapter 10

Error Reporting and Recovery

The quality of a language application's error messages and recovery strategy often makes the difference between a professional application and an amateurish application. Error recovery is the process of recovering from a syntax error by altering the input stream or consuming symbols until the parser can restart in a known state. Many hand-built and many non-*LL*-based recognizers emit less than optimal error messages, whereas ANTLR-generated recognizers automatically emit very good error messages and recover intelligently, as shown in this chapter. ANTLR's error handling facility is even useful during development.

During the development cycle, you will have a lot of mistakes in your grammar. The resulting parser will not recognize all valid sentences until you finish and debug your grammar. In the meantime, informative error messages help you track down grammar problems. Once you have a correct grammar, you then have to deal with ungrammatical sentences entered by users or even ungrammatical sentences generated by other programs gone awry.

In both situations, the manner in which your parser responds to ungrammatical input is an important productivity consideration. In other words, a parser that responds with "Eh?" and bails out upon the first syntax error is not very useful during development or for the people who have to use the resulting parser. For example, some SQL engines can only tell you the general vicinity where an error occurred rather than exactly what is wrong and where, making query development a trial-and-error process.

Developers using ANTLR get a good error reporting facility and a sophisticated error recovery strategy for free. ANTLR automatically generates recognizers that emit rich error messages upon syntax error and successfully resynchronize much of the time. The recognizers even avoid generating more than a single error message for each syntax error.

New in v3.

This chapter describes the automatic error reporting and recovery strategy used by ANTLR-generated recognizers and shows how to alter the default mechanism to suit your particular needs.

10.1 A Parade of Errors

The best way to describe ANTLR's error recovery strategy is to show you how ANTLR-generated recognizers respond to the most common syntax errors: mismatched token, no viable alternative, and early exit from an EBNF (. . .)+ subrule loop. Consider the following grammar for simple statements and expressions, which we'll use as the core for the examples in this section and the remainder of the chapter:

Download errors/E.g

```
grammar E;

prog:   stat+ ;

stat:   expr ';'
        {System.out.println("found expr: "+$stat.text);}
    |   ID '=' expr ';'
        {System.out.println("found assign: "+$stat.text);}
    ;

expr:   multExpr (('+'|'-') multExpr)*
    ;

multExpr
    :   atom ('*' atom)*
    ;

atom:   INT
    |   '(' expr ')'
    ;

ID  :   ('a'..'z'|'A'..'Z')+ ;
INT :   '0'..'9'+ ;
WS  :   (' '|'\t'|'\n'|'\r')+ {skip();} ;
```

And here is the usual test rig that invokes rule **prog**:

Download errors/TestE.java

```
import org.antlr.runtime.*;

public class TestE {
    public static void main(String[] args) throws Exception {
        ANTLRInputStream input = new ANTLRInputStream(System.in);
        ELexer lexer = new ELexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        EParser parser = new EParser(tokens);
        parser.prog();
    }
}
```

First run some valid input into the parser to figure out what the normal behavior is:

```
⇐ $ java TestE
⇐ (3);
⇐ EOF
⇒ found expr: (3);
  $
```

Upon either expression statement or assignment statement, the translator prints a message indicating the text matched for rule **stat**. In this case, (3); is an expression, not an assignment, as shown in the output.

Now, leaving off the final), the parser detects a mismatched token because rule **atom** was looking for the right parenthesis to match the left parenthesis:

```
⇐ $ java TestE
⇐ (3;
⇐ EOF
⇒ line 1:2 mismatched input ';' expecting ')'
  found expr: (3;
  $
```

The line 1:2 component of the error message indicates that the error occurred on the first line and at the third character position in that line (indexed from zero, hence, index 2).

Generating that error message is straightforward, but how does the parser successfully match the ; and execute the print action in the first alternative of rule **stat** after getting a syntax error all the way down in **atom**? How did the parser successfully recover from that mismatched token to continue as if nothing happened? This error recovery feature is called *single token insertion* because the parser pretends to insert the missing) and keep going. We'll examine the mechanism in Section 10.7, *Recovery by Single Symbol Insertion*, on page 260. Notice that with multiple expressions, the parser successfully continues and matches the second alternative without error:

```
⇐ $ java TestE
⇐ (3;
⇐ 1+2;
⇐ EOF
⇒ line 1:2 mismatched input ';' expecting ')'
  found expr: (3;
  found expr: 1+2;
  $
```

ANTLR also avoids generating cascading error messages if possible. That is, recognizers try to emit a single error message for each syntax error. In the following sample run, the first expression has two errors: the missing) and the missing ;. The parser normally emits only the first error message, suppressing the second message that has the [SPURIOUS] prefix:

```
⇐ $ java TestE
⇐ (3
⇐ a=1;
⇐ EOF
⇒ line 2:0 mismatched input 'a' expecting ')'
  [SPURIOUS] line 2:0 mismatched input 'a' expecting ';'
  found expr: (3
  found assign: a=1;
  $
```

Another common syntax error occurs when the parser is at a decision point and the current lookahead is not consistent with any of the alternatives of that rule or subrule. For example, the decision in rule **atom** must choose between an integer and the start of a parenthesized expression. Input 1+; is missing the second operand, and rule **atom** would see ; instead, causing a "no viable alternative exception:"

```
⇐ $ java TestE
⇐ 1+;
⇐ EOF
⇒ line 1:2 no viable alternative at input ';'
  found expr: 1+;
  $
```

The parser successfully recovers by scanning ahead to look for a symbol that can follow a reference to **atom** or a rule that has invoked **atom**. In this case, the ; is a viable symbol following a reference to **atom** (and therefore **expr**). The parser consumes no tokens and simply exits from **atom** knowing that it is probably correctly resynchronized.

Back When You Could Almost Parse C++

In the early 1990s, I was consulting at NeXT and was helping Sumana Srinivasan build a C++ code browser using ANTLR v1 (ANTLR is still used in NeXTStep, er, Mac OS X today). The manager, Steve Naroff, insisted that the ANTLR-generated parser provide the same high-quality error messages as his hand-built C parser did. Because of this, I introduced the notion of parser exception handling (the analog of the familiar programming exception handling) and created a simple single-token deletion mechanism. Ironically, the ANTLR-generated C++ recognizer emitted better messages in some circumstances than the hand-built parser because ANTLR never got tired of computing token sets and generating error recovery code—humans, on the other hand, often get sick of this tedious job.

You will also run into early subrule exit exceptions where a one-or-more (...)+ subrule matched no input. For example, if you send in an empty input stream, the parser has nothing to match in the stat+ loop:

```
⇐ $ java TestE
⇐ EOF
⇒ line 0:-1 required (..)+ loop did not match anything at input '<EOF>'
  $
```

The line and character position information for EOF is meaningless; hence, you see the odd 0:-1 position information.

This section gave you a taste of ANTLR's default error reporting and recovery capabilities (see Section 10.7, *Automatic Error Recovery Strategy*, on page 258 for details about the automatic error recovery mechanism). The next few sections describe how you can alter the standard error messages to help with grammar debugging and to provide better messages for your users.

10.2 Enriching Error Messages during Debugging

By default, recognizers emit error messages that are most useful to users of your software. The messages include information only about what was found and what was expected such as in the following:

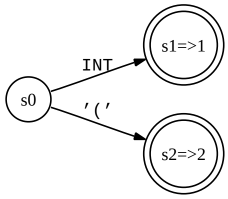
```
line 10:22 mismatched input INT expecting ID
```

Unfortunately, your grammar has probably 200 references to token **ID**. Where in the grammar was the parser when it found the **INT** instead of the **ID**? You can use the debugger in ANTLRWorks to set a breakpoint upon exception and then just look to see where in the grammar the parser is. Sometimes, though, sending text error messages to the console can be more convenient because you do not have to start the debugger.

With a little bit of work, you can override the standard error reporting mechanism to include information about the rule invocation stack. The invocation stack is the nested list of rules entered by the parser at any given moment, that is, the stack trace. You can also add more information about the mismatched token. For no viable alternative errors, you can do even more. For example, the following run illustrates a rich, no viable alternative error message that is much more useful for debugging grammars than the default:

```
⇐ $ java TestE2
⇐ 1+;
⇐ EOF
⇒ line 1:2 [prog, stat, expr, multExpr, atom] no viable alternative,
  token=[@2,2:2=';',<7>,1:2] (decision=5 state 0)
  decision=<<35:1: atom : ( INT | '(' expr ')' );>>
  found expr: 1+;
  $
```

The message includes a rule invocation stack trace where the last rule mentioned is the rule the parser was in when it encountered the syntax error. The error includes a detailed report on the token itself that includes the token index, a character index range into the input stream, the token type, and the line and character position within the line. Finally, for no viable alternative exceptions such as this, the message includes information about the decision in a grammar: the decision number, the state within the decision's lookahead DFA, and a chunk of the grammar that describes the decision. To use the decision and state information, turn on ANTLR option -dfa, which will generate DOT (graphviz) descriptions you can display. Filenames are encoded with the grammar name and the decision number, so, for example, the DOT file for decision 5 of grammar **E** is E_dec-5.dot and looks like the following:



The state 0 mentioned in the error message is s0 in the diagram. In this case, the parser had a lookahead of ; that clearly does not match either alternative emanating from s0; hence, you have the no viable alternative exception.

To get these rich error messages, override two methods from BaseRecognizer, displayRecognitionError() and getTokenErrorDisplay(), where the grammar itself stays the same:

Download errors/E2.g

```
grammar E2;

@members {
public String getErrorMessage(RecognitionException e,
                              String[] tokenNames)
{
    List stack = getRuleInvocationStack(e, this.getClass().getName());
    String msg = null;
    if ( e instanceof NoViableAltException ) {
       NoViableAltException nvae = (NoViableAltException)e;
       msg = " no viable alt; token="+e.token+
          " (decision="+nvae.decisionNumber+
          " state "+nvae.stateNumber+")"+
          " decision=<<"+nvae.grammarDecisionDescription+">>";
    }
    else {
       msg = super.getErrorMessage(e, tokenNames);
    }
    return stack+" "+msg;
}
public String getTokenErrorDisplay(Token t) {
    return t.toString();
}
}
```

The next section describes how to improve error messages for your users rather than for yourself during debugging.

10.3 Altering Recognizer Error Messages

This section describes the information available to you when generating error messages and provides an example that illustrates how to enrich error messages with context information from the grammar.For each problem that can occur during sentence recognition, the recognizer creates an exception object derived from RecognitionException.

RecognitionException

The superclass of all exceptions thrown by an ANTLR-generated recognizer. It tracks the input stream; the index of the symbol (character, token, or tree node) the recognizer was looking at when the error occurred; the erroneous symbol pointer (int, Token, or Object); the line; and the character position within that line.

MismatchedTokenException

Indicates that the parser was looking for a particular symbol that it did not find at the current input position. In addition to the usual fields, this object tracks the expected token type (or character code).

MismatchedTreeNodeException

Indicates that the tree parser was looking for a node with a particular token type and did not find it. This is the analog of a mismatched token exception for a token stream parser. It tracks the expected token type.

NoViableAltException

The recognizer came to a decision point, but the lookahead was not consistent with any of the alternatives. It tracks the decision number and state number within the lookahead DFA where the problem occurred and also stores a chunk of the grammar from which ANTLR generated the decision.

EarlyExitException

The recognizer came to a (..)+ EBNF subrule that must match an alternative at least once, but the subrule did not match anything. It tracks the decision number but not the state number because it is obviously not in the middle of the lookahead DFA; the whole thing was skipped.

FailedPredicateException

A validating semantic predicates evaluated to false. It tracks the name of the rule in which the predicate failed as well as the text of the predicate itself from your grammar.

MismatchedRangeException

The recognizer tried to match a range of symbols, usually characters, but could not. It tracks the minimum and maximum element in the range.

MismatchedSetException

The recognizer attempted to match a set of symbols but could not. It tracks the set of elements in which the recognizer was interested.

MismatchedNotSetException

The recognizer attempted to match the inverse of a set (using the ~ operator) but could not.

Figure 10.1: ANTLR recognition exceptions

These exception objects contain information about what was found on the input stream, what was expected, and sometimes information about the location in the grammar associated with the erroneous parser state.

To avoid forcing English-only error messages and to generally make things as flexible as possible, the recognizer does not create exception objects with string messages. Instead, it tracks the information necessary to generate an error.

Then the various reporting methods in BaseRecognizer generate a localized error message, or you can override them. Do not expect the exception getMessage() methods to return anything. The table in Figure 10.1, on the preceding page, summarizes the exception classes and the information they contain.

Improved in v3.

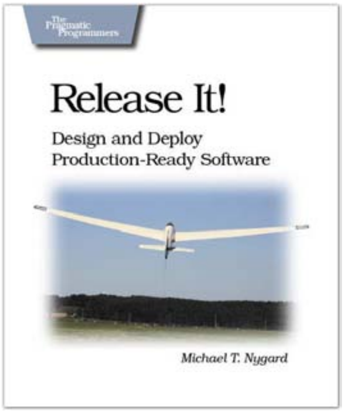
Beyond the information in these exception objects, you can collect any useful information you want via actions in the grammar and then use it to provide better error messages for your users.

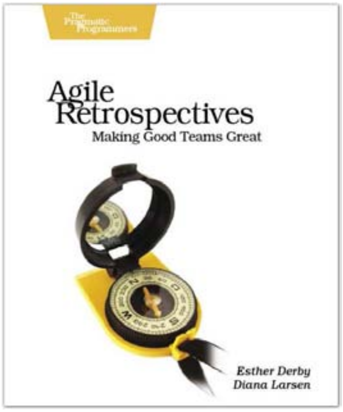
One of the most useful enhancements to error messages is to include information about the kind of abstract construct the parser was recognizing when it encountered an error. For example, instead of just saying "missing **ID**," it is better to say "missing **ID** in expression." You could use the literal rule name such as "multExpr," but that is usually meaningless to users.

You can think of this as a *paraphrase* mechanism because you are representing a collection of grammar rules with a short description. What you want is a map from all rules associated with a particular abstract language construct (that is, declarations, statements, and expressions) to a user-friendly string such as "expression."

In v2, there was a ***paraphrase*** *option that automated this.*

The easiest way to implement a paraphrase mechanism is to push a string onto a stack when you enter a rule that represents an abstract construct in a language and then pop the value off when leaving the rule. Do not push a paraphrase string for all rules. Just push a paraphrase for the top-level rule such as **expr**, but not **multExpr** or **atom**.


The Pragmatic Programmers
Release It!
Design and Deploy Production-Ready Software
Michael T. Nygard


The Pragmatic Programmers
Agile Retrospectives
Making Good Teams Great
Esther Derby
Diana Larsen