ANTLR

jGuru

Lexical Analysis with ANTLR

A *lexer* (often called a scanner) breaks up an input stream of characters into vocabulary symbols for a parser, which applies a grammatical structure to that symbol stream. Because ANTLR employs the same recognition mechanism for lexing, parsing, and tree parsing, ANTLR-generated lexers are much stronger than DFA-based lexers such as those generated by DLG (from PCCTS 1.33) and lex.

The increase in lexing power comes at the cost of some inconvenience in lexer specification and indeed requires a serious shift your thoughts about lexical analysis. See a comparison of LL(k) and DFA-based lexical analysis.

ANTLR generates predicated-LL(k) lexers, which means that you can have semantic and syntactic predicates and use k>1 lookahead. The other advantages are:

- You can actually read and debug the output as its very similar to what you would build by hand.
- The syntax for specifying lexical structure is the same for lexers, parsers, and tree parsers.
- You can have actions executed during the recognition of a single token.
- You can recognize complicated tokens such as HTML tags or "executable" comments like the javadoc `@`-tags inside `/** ... */` comments. The lexer has a stack, unlike a DFA, so you can match nested structures such as nested comments.

The overall structure of a lexer is:

```
class MyLexer extends Lexer;
options {
  some options
}
{
  lexer class members
}
lexical rules
```

Lexical Rules

Rules defined within a lexer grammar must have a name beginning with an uppercase letter. These rules implicitly match characters on the input stream instead of tokens on the token stream. Referenced grammar elements include token references (implicit lexer rule references), characters, and strings. Lexer rules are processed in the exact same manner as parser rules and, hence, may specify arguments and return values; further, lexer rules can also have local variables and use recursion. The following rule defines a rule called `ID` that is available as a token type in the parser.

```
ID : ( 'a'..'z' )+
   ;
```

This rule would become part of the resulting lexer and would appear as a method called `mID()`.

Lexer rules allow your parser to match *context-free* structures on the input character stream as opposed to the much weaker *regular* structures (using a DFA--deterministic finite automaton). For example, consider that matching nested curly braces with a DFA must be done using a counter whereas nested curlies are trivially matched with a context-free grammar:

```
ACTION
    :   '{' ( ACTION | ~'}' )* '}'
    ;
```

The recursion, of course, is the dead giveaway that this is not an ordinary lexer rule.

Because the same algorithms are used to analyze lexer and parser rules, lexer rules may use more than a single symbol of lookahead, can use semantic predicates, and can specify syntactic predicates to look arbitrarily ahead, thus, providing recognition capabilities beyond the LL(k) languages into the *context-sensitive*. Here is a simple example that requires k>1 lookahead:

```
ESCAPE_CHAR
    :   '\\' 't' // two char of lookahead needed,
    |   '\\' 'n' // due to common left-prefix
    ;
```

To illustrate the use of syntactic predicates for lexer rules, consider the problem of distinguishing between floating point numbers and ranges in Pascal. Input `3..4` must be broken up into 3 tokens: `INT`, `RANGE`, followed by `INT`. Input `3.4`, on the other hand, must be sent to the parser as a `REAL`. The trouble is that the series of digits before the first `'.'` can be arbitrarily long. The scanner then must consume the first `'.'` to see if the next character is a `'.'`, which would imply that it must back up and consider the first series of digits an integer. Using a non-backtracking lexer makes this task essentially impossible. However, a syntactic predicate can be used to specify what arbitrary lookahead is necessary:

```
class Pascal;

prog:   INT
        (   RANGE INT
            { System.out.println("INT .. INT"); }
        |   EOF
            { System.out.println("plain old INT"); }
        )
    |   REAL { System.out.println("token REAL"); }
    ;

lexclass LexPascal;

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

protected
INT :   ('0'..'9')+
    ;

protected
REAL:   INT '.' INT
    ;

RANGE
    :   ".."
    ;

RANGE_OR_INT
    :   ( INT ".." ) => INT  { $setType(INT); }
    |   ( INT '.' )  => REAL { $setType(REAL); }
    |   INT                  { $setType(INT); }
    ;
```

ANTLR lexer rules are even able to handle FORTRAN assignments and other difficult lexical constructs. Consider the following `DO` loop:

```
DO 100 I = 1,10
```

If the comma were replaced with a period, the loop would become an assignment to a weird variable called "`DO100I`":

```
DO 100 I = 1.10
```

The following rules correctly differentiate the two cases:

```
DO_OR_VAR
    :   (DO_HEADER)=> "DO" { $setType(DO); }
    |   VARIABLE { $setType(VARIABLE); }
    ;

protected
```

```
DO_HEADER
options { ignore=WS; }
    :   "DO" INT VARIABLE '=' EXPR ','
    ;

protected INT : ('0'..'9')+;

protected WS : ' ';

protected
VARIABLE
    :   'A'..'Z'
        ('A'..'Z' | ' ' | '0'..'9')*
        { /* strip space from end */ }
    ;

// just an int or float
protected EXPR
    :   INT ( '.' (INT)? )?
    ;
```

Return values

All rules return a token object (conceptually) automatically, which contains the text matched for the rule and its token type at least. To specify a user-defined return value, define a return value and set it in an action:

```
protected
INT returns [int v]
    :   (‘0’..’9’)+ { v=Integer.valueOf($getText); }
    ;
```

Non-protected rules cannot have a return type as the parser cannot access the return value leading to confusion.

Skipping characters

To have the characters matched by a rule ignored, set the token type to `Token.SKIP`. For example,

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

Distinguishing between lexer rules

ANTLR generates a rule called `nextToken` which has an alternative containing a lexer rule reference, one alternative for each non-`protected` lexer rule. The first few characters of the token are used to route the lexer to the appropriate lexical rule. The alternatives of nextToken are analyzed for determinism; i.e., with k characters of lookahead can the lexer determine which lexer rule to attempt.

The ANTLR input:

```
INT : ('0'..'9')+;
WS : ' ' | '\t' | '\r' | '\n';
```

produces a switch statement in nextToken:

```
switch (LA(1)) {
  case '0': case '1': case '2': case '3':
  case '4': case '5': case '6': case '7':
  case '8': case '9':
    mINT(); break;
  case '\t': case '\n': case '\r': case ' ':
    mWS(); break;
  default: // error
}
```

Definition order and lexical ambiguities

ANTLR 2.0 does not follow the common lexer rule of "first definition wins" (the alternatives within a rule, however, still follow this rule). Instead, sufficient power is given to handle the two most common cases of ambiguity, namely "keywords vs. identifiers", and "common prefixes"; and for especially nasty cases you can use syntactic predicates.

Keywords and literals

Many languages have a general "identifier" lexical rule, and keywords that are special cases of the identifier pattern. A typical identifier token is defined as:

```
ID : LETTER (LETTER | DIGIT)*;
```

This is often in conflict with keywords. ANTLR 2.0 solves this problem by letting you put fixed keywords into a literals table. The literals table (which is usally implemented as a hash table in the lexer) is checked after each token is matched, so that the literals effectively override the more general identifier pattern. Literals are created in one of two ways. First, any double-quoted string used in a parser is automatically entered into the literals table of the associated lexer. Second, literals may be specified in the lexer grammar by means of the literal option. In addition, the testLiterals option gives you fine-grained control over the generation of literal-testing code.

Common prefixes

Fixed-length common prefixes in lexer rules are best handled by increasing the lookahead depth of the lexer. For example, some operators from Java:

```
class MyLexer extends Lexer;
options {
  k=4;
}
GT : ">";
GE : ">=";
RSHIFT : ">>";
RSHIFT_ASSIGN : ">>=";
UNSIGNED_RSHIFT : ">>>";
UNSIGNED_RSHIFT_ASSIGN : ">>>=";
```

Token definition files

Token definitions can be transferred from one grammar to another by way of token definition files. This is accomplished using the importVocab and exportVocab options.

Character classes

Use the ~ operator to invert a character or set of characters. For example, to match any character other than newline, the following rule references ~'\n'.

```
SL_COMMENT: "//" (~'\n')* '\n';
```

The ~ operator also inverts a character set:

```
NOT_WS: ~(' ' | '\t' | '\n' | '\r');
```

The range operator can be used to create sequential character sets:

```
DIGIT : '0'..'9' ;
```

Token Attributes

See the next section.

Lexical lookahead and the end-of-token symbol

A unique situation occurs when analyzing lexical grammars, one which is similar to the end-of-file condition when analyzing regular grammars. Consider

how you would compute lookahead sets for the ('b' |) subrule in following rule B:

```
class L extends Lexer;

A       :       B 'b'
        ;

protected  // only called from another lex rule
B       :       'x' ('b' | )
        ;
```

The lookahead for the first alternative of the subrule is clearly 'b'. The second alternative is empty and the lookahead set is the set of all characters that can follow references to the subrule, which is the follow set for rule B. In this case, the 'b' character follows the reference to B and is therefore the lookahead set for the empty alt indirectly. Because 'b' begins both alternatives, the parsing decision for the subrule is nondeterminism or ambiguous as we sometimes say. ANTLR will justly generate a warning for this subrule (unless you use the `warnWhenFollowAmbig` option).

Now, consider what would make sense for the lookahead if rule A did not exist and rule B was not protected (it was a complete token rather than a "subtoken"):

```
B       :       'x' ('b' | )
        ;
```

In this case, the empty alternative finds only the end of the rule as the lookahead with no other rules referencing it. In the worst case, **any** character could follow this rule (i.e., start the next token or error sequence). So, should not the lookahead for the empty alternative be the entire character vocabulary? And should not this result in a nondeterminism warning as it must conflict with the 'b' alternative? Conceptually, yes to both questions. From a practical standpoint, however, you are clearly saying "heh, match a 'b' on the end of token B if you find one." I argue that no warning should be generated and ANTLR's policy of matching elements as soon as possible makes sense here as well.

Another reason not to represent the lookahead as the entire vocabulary is that when we eventually add UNICODE support, a vocabulary of '\u0000'..'\uFFFF' is too large to store (one set is 2^16 / 32 long words of memory!). Any alternative with '<end-of-token>' in its lookahead set will be pushed to the ELSE or DEFAULT clause by the code generator so that huge bitsets can be avoided.

The summary is that lookahead purely derived from hitting the end of a lexical rule (unreferenced by other rules) cannot be the cause of a nondeterminism. The following table summarizes a bunch of cases that will help you figure out when ANTLR will complain and when it will not.

`X : 'q' ('a')? ('a')? ;`	The first subrule is nondeterministic as 'a' from second subrule (and end-of-token) are in the lookahead for exit branch of (...)?
`X : 'q' ('a')? ('c')? ;`	No nondeterminism.

`Y : 'y' X 'b' ;`
 	
`protected`
`X : 'b'`
`|`
`;`
	Nondeterminism in rule X.

`X : 'x' ('a'|'c'|'d')+`
`| 'z' ('a')+`
`;`
	No nondeterminism as exit branch of

	loops see lookahead computed purely from end-of-token.
`Y : 'y' ('a')+ ('a')? ;`	Nondeterminism between 'a' of (...)+ and exit branch as the exit can see the 'a' of the optional subrule. This would be a problem even if ('a')? were simply 'a'. A (...)* loop would report the same problem.
`X : 'y' ('a' 'b')+ 'a' 'c' ;`	At k=1, this is a nondeterminism for the (...)? since 'a' predicts staying in and exiting the loop. At k=2, no nondeterminism.
`Q : 'q' ('a' | )? ;`	Here, there is an empty alternative inside an optional subrule. A nondeterminism is reported as two paths predict end-of-token.

You might be wondering why the first subrule below is ambiguous:

```
('a')? ('a')?
```

The answer is that the NFA to DFA conversion would result in a DFA with the 'a' transitions merged into a single state transition! This is ok for a DFA where you cannot have actions anywhere except after a complete match. Remember that ANTLR lets you do the following:

```
('a' {do-this})? ('a' {do-that})?
```

One other thing is important to know. Recall that alternatives in lexical rules are reordered according to their lookahead requirements, from highest to lowest.

```
A : 'a'
  | 'a' 'b'
  ;
```

At k=2, ANTLR can see 'a' followed by '<end-of-token>' for the first alternative and 'a' followed by 'b' in the second. The lookahead at depth 2 for the first alternative being '<end-of-token>' suppressing a warning that depth two can match any character for the first alternative. To behave naturally and to generate good code when no warning is generated, ANTLR reorders the alternatives so that the code generated is similar to:

```
A() {
        if ( LA(1)=='a' && LA(2)=='b' ) { // alt 2
```

```
                match('a'); match('b');
        }
        else if ( LA(1)=='a' ) { // alt 1
                match('a')
        }
        else {error;}
}
```

Note the lack of lookahead test for depth 2 for alternative 1. When an empty alternative is present, ANTLR moves it to the end. For example,

```
A       :       'a'
        |
        |       'a' 'b'
        ;
```

results in code like this:

```
A() {
        if ( LA(1)=='a' && LA(2)=='b' ) { // alt 2
                match('a'); match('b');
        }
        else if ( LA(1)=='a' ) { // alt 1
                match('a')
        }
        else {
        }
}
```

Note that there is no way for a lexing error to occur here (which makes sense because the rule is optional--though this rule only makes sense when `protected`).

Semantic predicates get moved along with their associated alternatives when the alternatives are sorted by lookahead depth. It would be weird if the addition of a {true}? predicate (which implicitly exists for each alternative) changed what the lexer recognized! The following rule is reorder so that alternative 2 is tested for first.

```
B       :       {true}? 'a'
        |       'a' 'b'
        ;
```

Syntactic predicates are **not** reordered. Mentioning the predicate after the rule it conflicts with results in an ambiguity such as is in this rule:

```
F       :       'c'
        |       ('c')=> 'c'
        ;
```

Other alternatives are, however, reordered with respect to the syntactic predicates even when a switch is generated for the LL(1) components and the syntactic predicates are pushed the default case. The following rule illustrates the point.

```
F       :       'b'
        |       {/* empty-path */}
        |       ('c')=> 'c'
        |       'c'
        |       'd'
        |       'e'
        ;
```

Rule F's decision is generated as follows:

```
        switch ( la_1) {
        case 'b':
        {
            match('b');
            break;
        }
        case 'd':
        {
            match('d');
            break;
        }
        case 'e':
```

```
        {
            match('e');
            break;
        }
        default:
            boolean synPredMatched15 = false;
            if (((la_1=='c'))) {
                int _m15 = mark();
                synPredMatched15 = true;
                guessing++;
                try {
                    match('c');
                }
                catch (RecognitionException pe) {
                    synPredMatched15 = false;
                }
                rewind(_m15);
                guessing--;
            }
            if ( synPredMatched15 ) {
                match('c');
            }
            else if ((la_1=='c')) {
                match('c');
            }
            else {
                if ( guessing==0 ) {
                    /* empty-path */
                }
            }
        }
```

Notice how the empty path got moved after the test for the 'c' alternative.

Scanning Binary Files

[Prior to 2.2.3, ANTLR was unable to parse binary files.] Character literals are not limited to printable ASCII characters (though at the moment UNICODE characters above 0xFF do not work). To demonstrate the concept, imagine that you want to parse a binary file that contains strings and short integers. To distinguish between them, marker bytes are used according to the following format:

format	description
'\0' *highbyte lowbyte*	Short integer
'\1' *string of non-'\2' chars* '\2'	String

Sample input (274 followed by "a test") might look like the following in hex (output from UNIX **od -h** command):

```
0000000000    00 01 12 01 61 20 74 65 73 74 02
```

or as viewed as characters:

```
0000000000    \0 001 022 001 a      t  e  s  t 002
```

The parser is trivially just a (...)+ around the two types of input tokens:

```
class DataParser extends Parser;

file:   (   sh:SHORT
            {System.out.println(sh.getText());}
        |   st:STRING
            {System.out.println("\""+
               st.getText()+"\"");}
        )+
    ;
```

All of the interesting stuff happens in the lexer. First, define the class and set the vocabulary to be all 8 bit binary values:

```
class DataLexer extends Lexer;
options {
    charVocabulary = '\u0000'..'\u00FF';
}
```

Then, define the two tokens according to the specifications, with markers around the string and a single marker byte in front of the short:

```
SHORT
    :   // match the marker followed by any 2 bytes
        '\0' high:. lo:.
        {
        // pack the bytes into a two-byte short
        int v = (((int)high)<<8) + lo;
        // make a string out of the value
        $setText(""+v);
        }
    ;

STRING
    :   '\1'!   // begin string (discard)
        ( ~'\2' )*
        '\2'!   // end string (discard)
    ;
```

To invoke the parser, use something like the following:

```
import java.io.*;

class Main {
    public static void main(String[] args) {
        try {
            // use DataInputStream to grab bytes
            DataLexer lexer =
              new DataLexer(
                new DataInputStream(System.in)
              );
            DataParser parser =
                new DataParser(lexer);
            parser.file();
        } catch(Exception e) {
            System.err.println("exception: "+e);
        }
    }
}
```

Manipulating Token Text and Objects

Once you have specified what to match in a lexical rule, you may ask "what can I discover about what will be matched for each rule element?" ANTLR allows you to label the various elements and, at parse-time, access the text matched for the element. You can even specify the token object to return from the rule and, hence, from the lexer to the parser. This section describes the text and token object handling characteristics of ANTLR.

Manipulating the Text of a Lexical Rule

There are times when you want to look at the text matched for the current rule, alter it, or set the text of a rule to a new string. The most common case is when you want to simply discard the text associated with a few of the elements that are matched for a rule such as quotes.

ANTLR provides the '!' operator that lets you indicate certain elements should not contribute to the text for a token being recognized. The '!' operator is used just like when building trees in the parser. For example, if you are matching the HTML tags and you do not want the '<' and '>' characters returned as part of the token text, you could manually remove them from the token's text before they are returned, but a better way is to suffix the unwanted characters with '!'. For example, the
 tag might be recognized as follows:

```
BR  :  '<'! "br" '>'! ; // discard < and >
```

Suffixing a lexical rule reference with '!' forces the text matched by the invoked rule to be discarded (it will not appear in the text for the invoking rule). For example, if you do not care about the mantissa of a floating point number, you can suffix the rule that matches it with a '!':

```
FLOAT : INT ('.'! INT!)? ; // keep only first INT
```

As a shorthand notation, you may suffix an alternative or rule with '!' to indicate the alternative or rule should not pass any text back to the invoking rule or parser (if nonprotected):

```
// ! on rule: nothing is auto added to text of rule.
rule! : ... ;

// ! on alt: nothing is auto added to text for alt
rule : ... |! ...;
```

Item suffixed with '!'	Effect
char or string literal	Do not add text for this atom to current rule's text.
rule reference	Do not add text for matched while recognizing this rule to current rule's text.
alternative	Nothing that is matched by alternative is added to current rule's text; the enclosing rule contributes nothing to any invoking rule's text. For nonprotected rules, the text for the token returned to parser is blank.
rule definition	Nothing that is matched by **any** alternative is added to current rule's text; the rule contributes nothing to any invoking rule's text. For nonprotected rules, the text for the token returned to parser is blank.

While the '!' implies that the text is not added to the text for the current rule, you can label an element to access the text (via the token if the element is a rule reference).

In terms of implementation, the characters are always added to the current text buffer, but are carved out when necessary (as this will be the exception rather than the rule, making the normal case efficient).

The '!' operator is great for discarding certain characters or groups of characters, but what about the case where you want to insert characters or totally reset the text for a rule or token? ANTLR provides a series of special methods to do this (we prefix the methods with '$' because Java does not have a macro facility and ANTLR must recognize the special methods in your actions). The following table summarizes.

Method	Description/Translation
`$append(x)`	Append x to the text of the surrounding rule. Translation: `text.append(x)`
`$setText(x)`	Set the text of the surrounding rule to x. Translation: `text.setLength(_begin); text.append(x)`
`$getText`	Return a String of the text for the surrounding rule. Translation; `new String(text.getBuffer(), _begin,text.length()-_begin)`
`$setToken(x)`	Set the token object that this rule is to return. See the section on Token Object Creation. Translation: `_token = x`
`$setType(x)`	Set the token type of the surrounding rule. Translation: `_ttype = x`
`setText(x)`	Set the text for the entire token being recognized regardless of what rule the action is in. No translation.
`getText()`	Get the text for the entire token being recognized regardless of what rule the action is in. No translation.

One of the great things about an ANTLR generated lexer is that the text of a token can be modified incrementally as the token is recognized (an impossible

task for a DFA-based lexer):

```
STRING: '"' ( ESCAPE | ~('"'|'\\') )* '"' ;

protected
ESCAPE
    :    '\\'
         ( 'n' { $setText("\n"); }
         | 'r' { $setText("\r"); }
         | 't' { $setText("\t"); }
         | '"' { $setText("\""); }
         )
    ;
```

Token Object Creation

Because lexical rules can call other rules just like in the parser, you sometimes want to know what text was matched for that portion of the token being matched. To support this, ANTLR allows you to label lexical rules and obtain a `Token` object representing the text, token type, line number, etc... matched for that rule reference. This ability corresponds to be able to access the text matched for a lexical state in a DFA-based lexer. For example, here is a simple rule that prints out the text matched for a rule reference, INT.

```
INDEX   :       '[' i:INT ']'
                {System.out.println(i.getText());}
        ;

INT     :       ('0'..'9')+ ;
```

If you moved the labeled reference and action to a parser, it would the same thing (match an integer and print it out).

All lexical rules *conceptually* return a `Token` object, but in practice this would be inefficient. ANTLR generates methods so that a token object is created only if any invoking reference is labeled (indicating they want the token object). Imagine another rule that calls INT without a label.

```
FLOAT   :       INT ('.' INT)? ;
```

In this case, no token object is created for either reference to INT. You will notice a boolean argument to every lexical rule that tells it whether or not a token object should be created and returned (via a member variable). All nonprotected rules (those that are "exposed" to the parser) must always generate tokens, which are passed back to the parser.

Heterogeneous Token Object Streams

While token creation is normally handled automatically, you can also manually specify the token object to be returned from a lexical rule. The advantage is that you can pass heterogeneous token objects back to the parser, which is extremely useful for parsing languagues with complicated tokens such as HTML (the `<img>` and `<table>` tokens, for example, can have lots of attributes). Here is a rule for the <img> tag that returns a token object of type ImageToken:

```
IMAGE
{
  Attributes attrs;
}
  :  "<img " attrs=ATTRIBUTES '>'
     {
     ImageToken t = new ImageToken(IMAGE,$getText);
     t.setAttributes(attrs);
     $setToken(t);
     }
  ;
ATTRIBUTES returns [Attributes a]
  :  ...
  ;
```

The `$setToken` function specifies that its argument is to be returned when the rule exits. The parser will receive this specific object instead of a `CommonToken` or whatever else you may have specified with the

`Lexer.setTokenObjectClass` method. The action in rule `IMAGE` references a token type, `IMAGE`, and a lexical rule references, `ATTRIBUTES`, which matches all of the attributes of an image tag and returns them in a data structure called `Attributes`.

What would it mean for rule `IMAGE` to be protected (i.e., referenced only from other lexical rules rather than from `nextToken`)? Any invoking labeled rule reference would receive the object (not the parser) and could examine it, or manipulate it, or pass it on to the invoker of that rule. For example, if `IMAGE` were called from `TAGS` rather than being nonprotected, rule `TAGS` would have to pass the token object back to the parser for it.

```
TAGS : IMG:IMAGE
       {$setToken(img);} // pass to parser
     | PARAGRAPH // probably has no special token
     | ...
     ;
```

Setting the token object for a nonprotected rule invoked without a label has no effect other than to waste time creating an object that will not be used.

We use a `CharScanner` member `_returnToken` to do the return in order to not conflict with return values used by the grammar developer. For example,

```
PTAG: "<p>" {$setToken(new ParagraphToken($$));} ;
```

which would be translated to something like:

```
protected final void mPTAG()
  throws RecognitionException, CharStreamException,
         TokenStreamException {
    Token _token = null;
    match("<p>");
    _returnToken =
      new ParagraphToken(text-of-current-rule);
}
```

Filtering Input Streams

You often want to perform an action upon seeing a pattern or two in a complicated input stream, such as pulling out links in an HTML file. One solution is to take the HTML grammar and just put actions where you want. Using a complete grammar is overkill and you may not have a complete grammar to start with.

ANTLR provides a mechanism similar to AWK that lets you say "here are the patterns I'm interested in--ignore everything else." Naturally, AWK is limited to regular expressions whereas ANTLR accepts context-free grammars (Uber-AWK?). For example, consider pulling out the <p> and
 tags from an arbitrary HTML file. Using the filter option, this is easy:

```
class T extends Lexer;
options {
    k=2;
    filter=true;
}

P : "<p>" ;
BR: "<br>" ;
```

In this "mode", there is no possibility of a syntax error. Either the pattern is matched exactly or it is filtered out.

This works very well for many cases, but is not sophisticated enough to handle the situation where you want "almost matches" to be reported as errors. Consider the addition of the <table...> tag to the previous grammar:

```
class T extends Lexer;
options {
    k=2;
    filter = true;
}
```

```
P : "<p>" ;
BR: "<br>" ;
TABLE : "<table" (WS)? (ATTRIBUTE)* (WS)? '>' ;
WS : ' ' | '\t' | '\n' ;
ATTRIBUTE : ... ;
```

Now, consider input "<table 8 = width ;>" (a bogus table definition). As is, the lexer would simply scarf past this input without "noticing" the invalid table. What if you want to indicate that a bad table definition was found as opposed to ignoring it? Call method

```
setCommitToPath(boolean commit)
```

in your TABLE rule to indicate that you want the lexer to commit to recognizing the table tag:

```
TABLE
    :   "<table" (WS)?
        {setCommitToPath(true);}
        (ATTRIBUTE)* (WS)? '>'
    ;
```

Input "<table 8 = width ;>" would result in a syntax error. Note the placement after the whitespace recognition; you do not want <tabletop> reported as a bad table (you want to ignore it).

One further complication in filtering: What if the "skip language" (the stuff in between valid tokens or tokens of interest) cannot be correctly handled by simply consuming a character and trying again for a valid token? You may want to ignore comments or strings or whatever. In that case, you can specify a rule that scarfs anything between tokens of interest by using option `filter=`*`RULE`*. For example, the grammar below filters for <p> and
 tags as before, but also prints out any other tag (<...>) encountered.

```
class T extends Lexer;
options {
    k=2;
    filter=IGNORE;
    charVocabulary = '\3'..'\177';
}

P : "<p>" ;
BR: "<br>" ;

protected
IGNORE
    :   '<' (~'>')* '>'
        {System.out.println("bad tag:"+$getText);}
    |   ( "\r\n" | '\r' | '\n' ) {newline();}
    |   .
    ;
```

Notice that the filter rule must track newlines in the general case where the lexer might emit error messages so that the line number is not stuck at 0.

The filter rule is invoked either when the lookahead (in nextToken) predicts none of the nonprotected lexical rules or when one of those rules fails. In the latter case, the input is rolled back before attempting the filter rule. Option `filter=true` is like having a filter rule such as:

```
IGNORE : . ;
```

Actions in regular lexical rules are executed even if the rule fails and the filter rule is called. To do otherwise would require every valid token to be matched twice (once to match and once to do the actions like a syntactic predicate)! Plus, there are few actions in lexer rules (usually they are at the end at which point an error cannot occur).

Is the filter rule called when commit-to-path is true and an error is found in a lexer rule? No, an error is reported as with filter=true.

What happens if there is a syntax error in the filter rule? Well, you can either put an exception handler on the filter rule or accept the default behavior, which is to consume a character and begin looking for another valid token.

In summary, the filter option allows you to:

1. Filter like awk (only perfect matches reported--no such thing as syntax error)
2. Filter like awk + catch poorly-formed matches (that is, "almost matches" like <table 8=3;> result in an error)
3. Filter but specify the skip language

ANTLR Masquerading as SED

To make ANTLR generate lexers that behave like the UNIX utility sed (copy standard in to standard out except as specified by the replace patterns), use a filter rule that does the input to output copying:

```
class T extends Lexer;
options {
  k=2;
  filter=IGNORE;
  charVocabulary = '\3'..'\177';
}

P  : "<p>" {System.out.print("<P>");};
BR : "<br>" {System.out.print("<BR>");};

protected
IGNORE
  :  ( "\r\n" | '\r' | '\n' )
     {newline(); System.out.println("");}
  |  c:. {System.out.print(c);}
  ;
```

This example dumps anything other than <p> and
 tags to standard out and pushes lowercase <p> and
 to uppercase. Works great.

Nongreedy Subrules

Quick: What does the following match?

```
BLOCK : '{' (.)* '}';
```

Your first reaction is that it matches any set of characters inside of curly quotes. In reality, it matches '{' followed by every single character left on the input stream! Why? Well, because ANTLR loops are *greedy*--they consume as much input as they can match. Since the wildcard matches any character, it consumes the '}' and beyond. This is a pain for matching strings, comments and so on.

Why can't we switch it around so that it consumes only until it sees something on the input stream that matches what **follows** the loop, such as the '}'? That is, why can't we make loops *nongreedy*? The answer is we can, but sometimes you want greedy and sometimes you want nongreedy (PERL has both kinds of closure loops now too). Unfortunately, parsers usually want greedy and lexers usually want nongreedy loops. Rather than make the same syntax behave differently in the various situations, Terence decided to leave the semantics of loops as they are (greedy) and make a subrule option to make loops nongreedy.

Greedy Subrules

I have yet to see a case when building a parser grammar where I did not want a subrule to match as much input as possible. For example, the solution to the classic if-then-else clause ambiguity is to match the "else" as soon as possible:

```
stat : "if" expr "then" stat ("else" stat)?
     | ...
     ;
```

This ambiguity (which statement should the "else" be attached to) results in a parser nondeterminism. ANTLR warns you about the `(...)?` subrule as follows:

```
warning: line 3: nondeterminism upon
        k==1:"else"
        between alts 1 and 2 of block
```

If, on the other hand, you make it clear to ANTLR that you want the subrule to match greedily (i.e., assume the default behavior), ANTLR will not generate the warning. Use the `greedy` subrule option to tell ANTLR what you want:

```
stat : "if" expr "then" stat
       ( options {greedy=true;} : "else" stat)?
     | ID
     ;
```

You are not altering the behavior really, since ANTLR was going to choose to match the "else" anyway, but you have avoided a warning message.

There is no such thing as a nongreedy `(...)?` subrule because telling an optional subrule not to match anything is the same as not specifying the subrule in the first place. If you make the subrule nongreedy, you will see:

```
warning in greedy.g: line(4),
        Being nongreedy only makes sense
        for (...)+ and (...)*
warning: line 4: nondeterminism upon
        k==1:"else"
        between alts 1 and 2 of block
```

Greedy subrules are very useful in the lexer also. If you want to grab any whitespace on the end of a token definition, you can try (WS)? for some whitespace rule WS:

```
ID : ('a'..'z')+ (WS)? ;
```

However, if you want to match ID in a loop in another rule that could also match whitespace, you will run into a nondeterminism warning. Here is a contrived loop that conflicts with the (WS)? in ID:

```
LOOP : (  ID
       |  WS
       )+
     ;
```

The whitespace on the end of the ID could be matched in ID or in LOOP now. ANTLR chooses to match the WS immediately, in ID. To shut off the warning, simply tell ANTLR that you mean for it do be greedy, it's default behavior:

```
ID : ('a'..'z')+ (options {greedy=true;}:WS)? ;
```

Nongreedy Lexer Subrules

ANTLR's default behavior of matching as much as possible in loops and optional subrules is sometimes not what you want in lexer grammars. Most loops that match "a bunch of characters" in between markers, like curly braces or quotes, should be nongreedy loops. For example, to match a nonnested block of characters between curly braces, you want to say:

```
CURLY_BLOCK_SCARF
    :   '{' (.)* '}'
    ;
```

Unfortunately, this does not work--it will consume everything after the '{' until the end of the input. The wildcard matches anything including '}' and so the loop merrily consumes past the ending curly brace.

To force ANTLR to break out of the loop when it sees a lookahead sequence consistent with what follows the loop, use the greedy subrule option:

```
CURLY_BLOCK_SCARF
    :   '{'
        (
            options {
                greedy=false;
            }
        :   .
```

```
        )*
        '}'
    ;
```

To properly take care of newlines inside the block, you should really use the following version that "traps" newlines and bumps up the line counter:

```
CURLY_BLOCK_SCARF
    :   '{'
        (
            options {
                greedy=false;
            }
        :   '\r' ('\n')? {newline();}
        |   '\n'         {newline();}
        |   .
        )*
        '}'
    ;
```

Limitations of Nongreedy Subrules

What happens when what follows a nongreedy subrule is not as simple as a single "marker" character like a right curly brace (i.e., what about when you need k>1 to break out of a loop)? ANTLR will either "do the right thing" or warn you that it might not.

First, consider the matching C comments:

```
CMT : "/*" (.)* "*/" ;
```

As with the curly brace matching, this rule will not stop at the end marker because the wildcard matches the "*/" end marker as well. You must tell ANTLR to make the loop nongreedy:

```
CMT : "/*" (options {greedy=false;} :.)* "*/" ;
```

You will not get an error and ANTLR will generate an exit branch

```
do {
    // nongreedy exit test
    if ((LA(1)=='*')) break _loop3;
    ...
```

Ooops. k=1, which is not enough lookahead. ANTLR did not generate a warning because it assumes you are providing enough lookahead for all nongreedy subrules. ANTLR cannot determine how much lookahead to use or how much is enough because, by definition, the decision is ambiguous--it simply generates a decision using the maximum lookahead.

You must provide enough lookahead to let ANTLR see the full end marker:

```
class L extends Lexer;
options {
        k=2;
}

CMT : "/*" (options {greedy=false;} :.)* "*/" ;
```

Now, ANTLR will generate an exit branch using k=2.

```
do {
    // nongreedy exit test
    if ((LA(1)=='*') && (LA(2)=='/'))
        break _loop3;
    ...
```

If you increase k to 3, ANTLR will generate an exit branch using k=3 instead of 2, even though 2 is sufficient. We know that k=2 is ok, but ANTLR is faced with a nondeterminism at it will use as much information as it has to yield a deterministic parser.

There is one more issue that you should be aware of. Because ANTLR generates linear approximate decisions instead of full LL(k) decisions,

complicated "end markers" can confuse ANTLR. Fortunately, ANTLR knows when it is confused and will let you know.

Consider a simple contrived example where a loop matches either ab or cd:

```
R : (   options {greedy=false;}
    :   ("ab"|"cd")
    )+
    ("ad"|"cb")
  ;
```

Following the loop, the grammar can match ad or cb. These exact sequences are not a problem for a full LL(k) decision, but due to the extreme compression of the linear approximate decision, ANTLR will generate an inaccurate exit branch. In other words, the loop will exit, for example, on ab even though that sequence cannot be matched following the loop. The exit condition is as follows:

```
// nongreedy exit test
if ( _cnt10>=1 && (LA(1)=='a'||LA(1)=='c') &&
     (LA(2)=='b'||LA(2)=='d')) break _loop10;
```

where the `_cnt10` term ensures the loop goes around at least once (but has nothing to do with the nongreedy exit branch condition really). Note that ANTLR has compressed all characters that can possibly be matched at a lookahead depth into a single set, thus, destroying the sequence information. The decision matches the cross product of the sets, including the spurious lookahead sequences such as ab.

Fortunately, ANTLR knows when a decision falls between its approximate decision and a full LL(k) decision--it warns you as follows:

```
warning in greedy.g: line(3),
    nongreedy block may exit incorrectly due
    to limitations of linear approximate lookahead
    (first k-1 sets in lookahead not singleton).
```

The parenthetical remark gives you a hint that some k>1 lookahead sequences are correctly predictable even with the linear approximate lookahead compression. The idea is that if all sets for depths 1..(k-1) are singleton sets (exactly one lookahead sequence for first k-1 characters) then linear approximate lookahead compression does not weaken your parser. So, the following variant does not yield a warning since the exit branch is linear approximate as well as full LL(k):

```
R : (   options {greedy=false;}
    :   .
    )+
    ("ad"|"ae")
  ;
```

The exit branch decision now tests lookahead as follows:

```
   (LA(1)=='a') && (LA(2)=='d'||LA(2)=='e')
```

which accurately predicts when to exit.

Lexical States

ANTLR has the ability to switch between multiple lexers using a token stream multiplexor. Please see the discussion in streams.

The End Of File Condition

A method is available for reacting to the end of file condition as if it were an event; e.g., you might want to pop the lexer state at the end of an include file. This method, `CharScanner.uponEOF()`, is called from `nextToken()` right before the scanner returns an `EOF_TYPE` token object to parser:

```
public void uponEOF()|
    throws TokenStreamException, CharStreamException;
```

This event is not generated during a syntactic predicate evaluation (i.e., when the parser is guessing) nor in the middle of the recognition of a lexical rule (that would be an IO exception). This event is generated only after the complete evaluation of the last token and upon the next request from the parser for a token.

You can throw exceptions from this method like "Heh, premature eof" or a retry stream exception. See the includeFile/P.g for an example usage.

Case sensitivity

You may use option `caseSensitive=false` in the lexer to indicate that you do not want case to be significant when matching characters against the input stream. For example, you want element `'d'` to match either upper or lowercase D, however, you do not want to change the case of the input stream. We have implemented this feature by having the lexer's `LA()` lookahead method return lowercase versions of the characters. Method `consume()` still adds the original characters to the string buffer associated with a token. We make the following notes:

- The lowercasing is done by a method `toLower()` in the lexer. This can be overridden to get more specific case processing. using option caseSensitive calls method `CharScanner.setCaseSensitive(...)`, which you can also call before (or during I suppose) the parse.
- ANTLR issues a warning when `caseSensitive=false` and uppercase ASCII characters are used in character or string literals.

Case sensitivity for literals is handled separately. That is, set lexer option `caseSensitiveLiterals` to false when you want the literals testing to be case-insensitive. Implementing this required changes to the literals table. Instead of adding a String, it adds an ANTLRHashString that implements a case-insensitive or case-sensitive hashing as desired.

Note: ANTLR checks the characters of a lexer string to make sure they are lowercase, but does not process escapes correctly--put that one on the "to do" list.

Ignoring whitespace in the lexer

One of the great things about ANTLR is that it generates full predicated-LL(k) lexers rather than the weaker (albeit sometimes easier-to-specify) DFA-based lexers of DLG. With such power, you are tempted (and encouraged) to do real parsing in the lexer. A great example of this is HTML parsing, which begs for a two-level parse: the lexer parsers all the attributes and so on within a tag, but the parser does overall document structure and ordering of the tags etc... The problem with parsing within a lexer is that you encounter the usual "ignore whitespace" issue as you do with regular parsing.

For example, consider matching the `<table>` tag of HTML, which has many attributes that can be specified within the tag. A first attempt might yield:

```
OTABLE   :      "<table" (ATTR)* '>'
         ;
```

Unfortunately, input "`<table border=1>`" does not parse because of the blank character after the `table` identifier. The solution is not to simply have the lexer ignore whitespace as it is read in because the lookahead computations must see the whitespace characters that will be found in the input stream. Further, defining whitespace as a rudimentary set of things to ignore does not handle all cases, particularly difficult ones, such as comments inside tags like

```
<table <!--wow...a comment--> border=1>
```

The correct solution is to specify a rule that is called after each lexical element (character, string literal, or lexical rule reference). We provide the lexer rule option `ignore` to let you specify the rule to use as whitespace. The solution to our HTML whitespace problem is therefore:

```
TABLE
options { ignore=WS; }
       :        "<table" (ATTR)* '>'
       ;

// can be protected or non-protected rule
WS     :        ' ' | '\n' | COMMENT | ...
       ;
```

We think this is cool and we hope it encourages you to do more and more interesting things in the lexer!

Oh, almost forgot. There is a **bug** in that an extra whitespace reference is inserted after the end of a lexer alternative if the last element is an action. The effect is to include any whitespace following that token in that token's text.

Tracking Line Information

Each lexer object has a `line` member that can be incremented by calling `newline()` or by simply changing its value (e.g., when processing `#line` directives in C).

```
SL_COMMENT : "//" (~'\n')* '\n' {newline();} ;
```

Do not forget to split out '`\n`' recognition when using the not operator to read until a stopping character such as:

```
BLOCK: '('
           ( '\n' { newline(); }
           | ~( '\n' | ')' )
           )*
       ')'
     ;
```

Another way to track line information is to override the `consume()` method:

Tracking Column Information

To track column information, you must currently do a bit of work. First, track column in lexer by adding members to save the column number and override consume() to set the column number that a token starts in:

```
class L extends Lexer; {
  protected int tokColumn = 1;
  protected int column = 1;
  public void consume() {
    if ( inputState.guessing==0 ) {
      if (text.length()==0) {
        // remember token start column
        tokColumn = column;
      }
      if (LA(1)==`\n') { column = 1; }
      else { column++; }
    }
    super.consume();
  }
}
```

Second, override makeToken() to set the token column start value for the created tokens:

```
class L extends Lexer;
{
  ...
  protected Token makeToken(int t) {
    Token tok = super.makeToken(t);
    tok.setColumn(tokColumn);
    return tok;
  }
}
```

Third, define a new `Token` object with column information:

```
public class MyToken
extends antlr.CommonToken {
```

```
  protected int column;
  public int getColumn() { return column; }
  public void setColumn(int c) { column=c; }
}
```

Finally, tell the lexer to create your new token objects:

```
L lexer = new L(System.in);
lexer.setTokenObjectClass("MyToken");
```

Naturally, you can set the `column` member in your lexical rules.

Using Explicit Lookahead

On rare occasions, you may find it useful to explicitly test the lexer lookahead in say a semantic predicate to help direct the parse. For example, /*...*/ comments have a two character stopping symbol. The following example demonstrates how to use the second symbol of lookahead to distinguish between a single '/' and a "*/":

```
ML_COMMENT
    :    "/*"
         (  { LA(2)!='/' }? '*'
         | '\n' { newline(); }
         | ~('*'|'\n')
         )*
         "*/"
    ;
```

The same effect might be possible via a syntactic predicate, but would be much slower than a semantic predicate. A DFA-based lexer handles this with no problem because they use a bunch of (what amount to) gotos whereas we're stuck with structured elements like while-loops.

A Surprising Use of A Lexer: Parsing

The following set of rules match arithmetical expressions in a lexer **not** a parser (whitespace between elements is not allowed in this example but can easily be handled by specifying rule option `ignore` for each rule):

```
EXPR
{ int val; }
    :   val=ADDEXPR
        { System.out.println(val); }
    ;

protected
ADDEXPR returns [int val]
{ int tmp; }
    :   val=MULTEXPR
        ( '+' tmp=MULTEXPR { val += tmp; }
        | '-' tmp=MULTEXPR { val -= tmp; }
        )*
    ;

protected
MULTEXPR returns [int val]
{ int tmp; }
    :   val=ATOM
        (   '*' tmp=ATOM { val *= tmp; }
        |   '/' tmp=ATOM { val /= tmp; }
        )*
    ;

protected
ATOM returns [int val]
    :   val=INT
    |   '(' val=ADDEXPR ')'
    ;

protected
INT returns [int val]
    :   ('0'..'9')+
        {val=Integer.valueOf($getText);}
    ;
```

But...We've Always Used Automata For Lexical Analysis!

Lexical analyzers were all built by hand in the early days of compilers until DFAs took over as the scanner implementation of choice. DFAs have several advantages over hand-built scanners:

- DFAs can easily be built from terse regular expressions.
- DFAs do automatic left-factoring of common (possibly infinite) left-prefixes. In a hand-built scanner, you have to find and factor out all common prefixes. For example, consider writing a lexer to match integers and floats. The regular expressions are straightforward:

  ```
  integer : "[0-9]+" ;
  real    : "[0-9]+{.[0-9]*}|.[0-9]+" ;
  ```

  Building a scanner for this would require factoring out the common `[0-9]+`. For example, a scanner might look like:

  ```
  Token nextToken() {
    if ( Character.isDigit(c) ) {
      match an integer
      if ( c=='.' ) {
        match another integer
        return new Token(REAL);
      }
      else {
        return new Token(INT);
      }
    }
    else if ( c=='.' ) {
      match a float starting with .
      return new Token(REAL);
    }
    else ...
  }
  ```

Conversely, hand-built scanners have the following advantages over DFA implementations:

- Hand-built scanners are not limited to the regular class of languages. They may use semantic information and method calls during recognition whereas a DFA has no stack and is typically not semantically predicated.
- Unicode (16 bit values) is handled for free whereas DFAs typically have fits about anything but 8 bit characters.
- DFAs are tables of integers and are, consequently, very hard to debug and examine.
- A tuned hand-built scanner can be faster than a DFA. For example, simulating the DFA to match `[0-9]+` requires *n* DFA state transitions where *n* is the length of the integer in characters.

  Tom Pennello of Metaware back in 1986 ("Very Fast LR Parsing") generated LR-based parsers in machine code that used the program counter to do state transitions rather than simulating the PDA. He got a huge speed up in parse time. We can extrapolate from this experiment that avoiding a state machine simulator in favor of raw code results in a speed up.

So, what approach does ANTLR 2.xx take? Neither! ANTLR 2.xx allows you to specify lexical items with expressions, but generates a lexer for you that mimics what you would generate by hand. The only drawback is that you still have to do the left-factoring for some token definitions (but at least it is done with expressions and not code). This hybrid approach allows you to build lexers that are much stronger and faster than DFA-based lexers while avoiding much of the overhead of writing the lexer yourself.

In summary, specifying regular expressions is simpler and shorter than writing a hand-built lexer, but hand-built lexers are faster, stronger, able to handle unicode, and easy to debug. This analysis has led many programmers to write hand-built lexers even when DFA-generation tools such as `lex` and `dlg` are commonly-available. ANTLR 1.xx made a parallel argument concerning PDA-based LR parsers and recursive-descent LL-based parsers. As a final justification, we note that writing lexers is trivial compared to building parsers;

also, once you build a lexer you will reuse it with small modifications in the future.

Version: $Id: //depot/code/org.antlr/release/antlr-2.7.0/doc/lexer.html#3 $