What's In A Word?
Lexical Analysis For PL/SQL And SQL

Charles Wetherell
Oracle Corporation

21 April 2016

Abstract

A friend wrote a PL/SQL source code analyzer. He was surprised when his tool worked on `return 'Y';` but failed on `return'Y';`. Is the blank-free version legal PL/SQL? Does his tool have a bug? The answer lies in the lexical structure of PL/SQL and SQL. This note explains lexical analysis and provides an answer to our friend's question.

1 Words

Ye Highlands and ye Lowlands
Oh, where hae ye been?
They hae slain
  the Earl o'Moray
And laid him on the green.

The Bonnie Earl o'Moray

A friend builds PL/SQL programming tools as a business. While testing a source code analyzer, he was puzzled by an anomaly. He had written the statement

```
return'Y';
```

in a PL/SQL program (probably the missing space after `return` was a typo). He passed the program to his source code analysis tool and the tool failed! When he rewrote the statement as

```
return 'Y';
```

the tool worked exactly as expected. By contrast, the PL/SQL compiler happily accepted both forms. Our friend wondered if he had discovered a bug in the PL/SQL compiler. Or perhaps he just didn't understand the rules of PL/SQL completely.

For any language, each new utterance must be broken into meaningful pieces. Each language has its own rules and regulations. If these are unknown or unclear, the analysis can be a challenge. Mistakes are the basis of puns and ludicrous misunderstandings. The writer Sylvia Wright loved the song *The Bonnie Earl o'Moray* as a child but only realized when she was an adult that the last line of the verse shown above was *And laid him on the green*, not *And Lady Mondegreen*. Natural language errors like this are now known as *mondegreens*.

A PL/SQL program or a SQL statement is an utterance, commonly a text preserved in a file. For these texts, just as for English or Hindi or Mandarin, the first step is to find the words. How is this done? What are the specific rules for PL/SQL and SQL? Most programmers never worry about the rules; they write sensible programs and don't notice the odd cases. But for folks who write tools to create, analyze, or manage PL/SQL and SQL, the details do matter. Mondegreens are to be avoided. The

knowledge may also help those who set coding standards, write elegant code, or are curious about everything to do with the two languages.

2 Lexical Principles

> Linguistic theories generally regard human languages as consisting of two parts: a lexicon, essentially a catalogue of a language's words (its wordstock); and a grammar, a system of rules which allow for the combination of those words into meaningful sentences.
>
> Wikipedia

Artificial languages have an advantage over natural languages: they are specifically designed for easy decoding. PL/SQL and SQL are no exceptions. These languages share their rules for finding words. Even better, they share several principles for the word finding rules.

Before going further, it is important to note that there is no meaning ascribed to words at this stage. The job of finding words begins and ends with the words themselves. Whether they are arranged properly, whether they make sense when combined, whether they are useful in any way is outside consideration. Only the words themselves matter.

Programming languages use a term of art for their words: the *token*. Language tools break text expressed as a sequence of characters into a sequence of tokens so no two tokens overlap and so every character is accounted for. Here are the lexical principles for PL/SQL and SQL; other programming languages are similar.

Category Each token has a *category*. For example, a token may be an *identifier*, a *numeric literal*, a *single character operator*, *white space*, and so on. Natural languages commonly only have the categories *word*, *punctuation*, and *white space* although they might include a few more like *number*.

Start Pattern The lexicon defines a family of *start patterns*. For example, an identifier must start with an alphabetic character. A *string literal* may start with several different patterns, but those patterns all include the single quote character `'`. Each start pattern identifies a particular category; several distinct patterns may all identify the same category.

Listing 3.1: The Lexical Algorithm

```
1  -- Input is a file with text
2  -- Output is an array of tokens
3
4  function LexIt(text)
5
6    TheTokens = []
7
8    while text is not empty
9      T := find start of token category
10     error if T is null
11     Token = find end of category T
12     error if Token is null
13     push Token on end of TheTokens
14
15   return TheTokens
16
17 end
```

Ordering The start patterns are ordered so that the first that might apply is always taken. This ensures that `/*` is seen as the token that starts a comment and not as the pair of a divide operator `/` followed by a multiply operator `*`.

Greedy Each token category is *greedy*. Once a start pattern has been found, the token continues until no more characters belonging to its *continuation* can be found. Identĩfiers continue so long as letters, digits, and a few special characters are seen. A string literal that starts with a single quote `'` continues until a second single quote is seen. A single character operator has, by definition, no continuation and terminates as soon as its start pattern is found.

Using *identifier* as a token category makes some PL/SQL and SQL purists anxious (myself included when I am in maximum purity mode) because there is another notion of *identifier* used in the semantic analysis of these languages. Any anxiety will be relieved by a discussion later on.

3 The Lexical Algorithm

The principles are fine, but how are they applied in practice? Two ingredients are needed: a description of each token category along with its start and continuation information and an algorithm to do the *lexical analysis*. Listing 3.1 provides the algorithm in some imaginary programming language.

Imagine a demon standing on the first brick of a long road, each brick inscribed with a character. The demon looks down at the brick under his feet and then may have to look ahead as far as two more bricks. Once he has seen these starting bricks, the demon determines the category of the token begining underfoot. There may be no

Listing 3.2: Example PL/SQL And Token String

```
1 begin
2   HumptyDumpty('abc' || to_char(1.0));
3 end;
```

Category	Text
Identifier	`begin`
Whitespace	`CR`
Whitespace	␣
Whitespace	␣
Identifier	`HumptyDumpty`
SingleCharOperator	`(`
StringLiteral	`'abc'`
Whitespace	␣
DoubleCharOperator	`||`
Whitespace	␣
Identifier	`to_char`
SingleCharOperator	`(`
NumericLiteral	`1.0`
SingleCharOperator	`)`
SingleCharOperator	`)`
SingleCharOperator	`;`
Whitespace	`CR`
Identifier	`end`
SingleCharOperator	`;`
Whitespace	`EOF`

Figure 1: Tokens From The PL/SQL Block

legal category; in that case, the demon announces an error. Now the demon walks until the next brick in front of him can no longer continue the token under construction. At that moment, he announces that the characters from where he started to where he is standing form a token. After the demon takes one step forward onto the next character, he starts the process again. It is also possible that the continuation does not end properly and, once again, the demon may announce an error. When the demon comes to the end of the road and the end of a token at the same time, the analysis is complete and the demon retires. Readers who know something about *finite state machines* will recognize this as an informal description of such a machine.

How might this algorithm work on a small PL/SQL program? Consider the example of Listing 3.2 and the token list from Figure 1. The first surprise is that the word `begin` is an identifier. Isn't it supposed to a *reserved word* or *keyword*? When parsing and semantic analysis come along, they may give identifiers more detailed roles, but for the purpose of breaking text into tokens, if it looks like an identifier, it is an identifier.

Secondly, every token has a text associated. That text is what the demon found when walking along. Surprisingly, the demon found three instances of whitespace in a row: `CR` (that is, *carriage return*), a blank ␣, and another ␣. Why weren't those amalgamated into one token? Because there is no need; the analysis is simpler if each whitespace item is just a character by itself. The same reasoning applies to the *end of file* character `EOF` that ends the token list.

The token list observes the lexical principles.

- The first token starts at the beginning of the text.
- The last token ends at the end of the text.
- Every text character appears in a token.
- No character appears in more than one token.
- The tokens appear in text order.

4 Categories

What are the categories of tokens and how are they recognized? On line 9 of Listing 3.1, there is a magical command `find start of token category`. This command knows how all the SQL and PL/SQL tokens start. Figure 2 encapsulates this knowledge.

There are some restrictions on the way the rules are applied.

- If the character under observation is not in the table, a lexical error has occurred.
- The rules must be applied in the order they are listed. The first item that matches determines the category. For example, `nq'` is the beginning of a string literal. It is not an identifier `nq` followed by the beginning of a simple string literal `'`.
- The characters of a category start must appear exactly as they are in the table. There can be no gaps. This is why white space in its several flavors is also a lexical category.
- The notation `digit` is shorthand for the set of characters that include all the decimal digits. The category which starts `.digit` stands for `.0`, `.1`, `.2`, and so on.
- Similarly, the notation `alpha` stands for all the alphabetic characters.
- Because PL/SQL may be found in a `WITH` clause of a SQL statement, lexical analysis of both languages must be prepared to deal with all of these categories.
- The SQL Reference Manual syntax diagram for numeric literals allows a leading `+` or `-` as part of the literal. This is an error. A sign character in that position is actually a single character operator that applies to the literal value that follows it. Because a negative integer literal can be used in some places where the original SQL developers did not want to specify full expressions, this tiny "hack" was added to the specification of numeric literals. It is easier to understand lexical analysis if the optional sign is treated on its own as an operator.

Start	Token Category
␣	Whitespace
TAB	Whitespace
CR	Whitespace
LF	Whitespace
EOF	Whitespace
"	QuotedIdentifier
'	StringLiteral
n'	StringLiteral
N'	StringLiteral
q'	StringLiteral
Q'	StringLiteral
nq'	StringLiteral
nQ'	StringLiteral
Nq'	StringLiteral
NQ'	StringLiteral
/*+	CommentHint
/*	Comment
--+	EOLHint
--	EOLComment
!=, <>	DoubleOperator
~=, ^=	DoubleOperator
<=, >=	DoubleOperator
>=	DoubleOperator
:=	DoubleOperator
«, »	DoubleOperator
||	DoubleOperator
=>	DoubleOperator
..	DoubleOperator
**	DoubleOperator
*/	EndHint
.digit	NumericLiteral
digit	NumericLiteral
+, -, *, /, :, ., ,, (,)	SingleOperator
@, =, <, >, ;, [,], {, }	SingleOperator
^, $, ?, |, ', \	SingleOperator
alphabetic	Identifier

Figure 2: Token Start Combinations
This table is reasonably complete but should not be regarded as definitive.

- Notice that `*/` is a category start. But it is only used to end a hint, not a comment. When it appears as the end of the comment, it is part of the continuation for the comment and not a token on its own.
- A comma `,` is a single operator token and so a comma is a token start. The two commas in the table are in two different type fonts exactly to make this point.

The token starts are in hand and each category needs a continuation. For white space, double operators, and single operators, the category start is also just the token so there is no need to go further. The other tokens are more or less complicated.

Identifier An identifier continues from its start character so long as there are alphabetic characters, decimal digits, and the special characters `$`, `_`, and `#`. The first character that is not one of these ends the identifier.

Quoted Identifier A quoted identifier runs from its start to the next double quote character `"`. Neither a null character nor an end of line may appear in a quoted identifier. This means that (almost) any character sequence can form a quoted identifier *except* one which includes a double quote.

Numeric Literal The SQL Reference Manual provides an elaborate and correct diagram except for the beginning `+` or `-`. Once again, only the characters from a numeric literal are allowed; anything else is an error including extraneous blanks.

Simple String Literal A string literal that begins with a single quote `'`, `n'`, or `N'` continues until the next single quote is found.

Q String Literal A string literal that begins with one of the `Q` or `NQ` sequences has a complicated continuation. The first character (call it c) after the single quote `'` is used as a new terminator. The string literal runs until the sequence c`'` is seen. This allows these literals to include the single quote as part of the literal.

Comment A comment continues until the sequence `*/` is seen. Nothing inside the comment matters except these characters. This means comments which start this way cannot be nested.

EOL Comment An *EOL comment* is one terminated by an end of line character. That character is not part of the comment itself. An end of file also ends this kind of comment as it is regarded as an implicit end of line. Notice that any other lexical item may be embedded in an EOL comment and will be ignored whether well formed or in error.

The two hint categories are more complicated because the material inside the token has structure itself. Exactly the same lexical analysis that is needed for the original text is needed within the hint. There are two ways to think about how to analyze a hint.

- Find the entire text that is the token for the hint. In other words, treat the hint as a comment of the same sort as the hint. Once the text of the comment has been discovered, break it into three parts: the hint start token, the hint end token, and the text of the interior. Restart an independent lexical analysis on the interior text and "patch" those tokens between the hint start and hint end tokens.
- Once a begin hint token has been seen, change the lexical category rules slightly. In particular, disallow any comment start tokens inside a hint. This has the same effect as the first scheme but it creates the tokens as they are seen; it does not require two passes over the interior of the hint.

Hints appear in SQL but because SQL statements may be embedded in PL/SQL programs, both languages must be prepared to analyze them. Also, SQL purportedly bars some constructs in hints, but probably it is best to think of analyzing the hint text into tokens without any restrictions and to let later processing stages decide whether the hint was legal. In practice, the lexical analyzer is likely to combine features of these two techniques depending on how the tokens are consumed after they are discovered.

5 The Answer

After all of this, the answer to the original question is clear. The statement

```
return 'Y';
```

is converted into the tokens

```
Identifier = return
  Whitespace = ␣
StringLiteral = 'Y'
 SingleOperator = ;
```

The blank between `return` and `'Y'` stops the continuation of the identifier `return`. In

```
return'Y';
```

it is the single quote `'` that stops the identifier continuation. Because the later phases of the PL/SQL compiler essentially ignore whitespace, this sequence is also legal

```
Identifier = return
StringLiteral = 'Y'
 SingleOperator = ;
```

6 Practice

In any language, one of the first steps is to list the vocabulary. For SQL and PL/SQL, the vocabulary is the list of tokens. Unfortunately, there does not seem to be any

complete token listing in the Oracle reference manuals. I have culled these descriptions from several manual sections.

Operators The PL/SQL manual has a reasonably complete list in a section titled *Delimiters*. The SQL reference manual does not seem to have such a list and so the various operators need to be inferred from the syntax diagrams. Notice that what we are calling an *operator* might be called *punctuation* in a natural language. Identifiers (like `PRIOR`) that function as operators do not fall in this category.

Whitespace Neither SQL nor PL/SQL provides a very clear account of white space although probably an experienced programmer can understand what is intended.

String Literals The SQL manual has a reasonable syntax diagram for all forms of string literals.

Numeric Literals The SQL manual has a reasonable syntax diagram for numeric literals. As noted before, the leading `+` or `-` sign is a mistake. The signs should be interpreted as *unary operators* before the literal.

Identifiers The SQL manual does not have an obvious definition of an identifier. The PL/SQL manual has a section *Identifiers* that describes the lexical categories `Identifier` and `QuotedIdentifier` accurately enough.

Comments The PL/SQL reference manual has a reasonable definition of both kinds of comments; the SQL manual also has a definition.

Hints The SQL reference manual describes the internal syntax allowed in hints, but it does not make clear that the interior of a hint is a kind of recursive environment so far as finding tokens goes. It specifically does not address the presence of comments inside hints.

Reserved Words And Keywords An identifier that has special syntactic or operational meaning is called a *reserved word* or *keyword* depending on its exact use. The SQL reference manual has an appendix that lists the reserved words; these words may not be used as unquoted identifiers. So, for example, a SQL table may not be named using the unquoted identifier `TABLE` but it may be named using the quoted identifier `"TABLE"`. There are about 100 reserved words. Beyond these, there are around 2,000 identifiers that have some special meaning to PL/SQL or SQL, for example `LOCAL`, `BINARY_DOUBLE`, `CUBE`, `GROUPING`, `C`, or `ROLLUP`. A keyword may be used as an ordinary identifier, but such a use may run afoul of its intended use as part of the PL/SQL or SQL languages. The PL/SQL compiler issues warnings for keywords that are used as identifiers.

Practicing programmers probably do not have too much trouble with the informality of the lexical descriptions. Those who are creating machine generated code or who are trying to decipher particularly obscure texts may wish for more clarity.

It is not to hard to write a practical lexical analyzer once token definitions are known. There are at least three common ways to do so. The quick and dirty way is a handwritten lexical anlayzer. Listing 6.1 shows the skeleton of such an analyzer. It isn't hard to write predicates like `isWhitespace`. The arms of the `CASE` statement need to be organized so that the choices are made in the proper order; for example `nq'abca'` must be seen as a complete string literal, not as the identifier `nq` immediately followed by the literal `'abca'`. There are other practical problems to solve (notably input

Listing 6.1: Skeleton Of A Hand Coded Lexical Analyzer

```
case
  when isWhiteSpace(nextChar)     then -- do white space
  when isHintStart(nextChar)      then -- grab hint and recurse
  when isCommentStart(nextChar)   then -- process comment
  when isStringStart(nextChar)    then -- process string
  when isNumberStart(nextChar)    then -- process number
  when isOperatorStart(nextChar)  then -- process operator
  when isAlphabetic(nextChar)     then -- process identifier
  else
    -- FAIL
end case;
```

buffer management and subscanning of hints), but nothing beyond the capabilities of a reasonably competent programmer.

Another approach uses *regular expression* to do the pattern matching. Theoretically, programming languages define their tokens so that they are all well described by regular expressions. In practice, this approach can be hard to use because of constructs like `NQ'` string literals, hints, and comments.

A third approach uses an existing tool that constructs a lexical analyzer from a specification of the tokens. The original `Unix` tool `lex` is a good example; there are probably several dozen follow-ons available. The `lex` tool defines a *domain specific language* for lexical analysis. The user writes a complete description of the tokens (a "program") and the tool generates code that does the analysis on demand. The generated lexer is usually available as a callable subroutine and is often linked with a parser built by a tool like the *parser generator* `yacc`. For a project of any size or commercial importance, the use of a generated lexical analyzer is probably the best choice because it will draw on considerable theory and practical experience by experts and thus be much less susceptible to bugs and other problems. The PL/SQL lexical analyzer is generated precisely for these reasons. The SQL lexical analyzer is hand written, probably because it was built at a time when tools were less widely available.

7 Final Thoughts

The use of `Identifier` and `QuotedIdentifier` as lexical categories may seen confusing. PL/SQL and SQL use these terms (along with *unquoted identifier*) in a different way when discussing the semantics of texts. The languages also contain *reserved words* and *keywords* which have the same form as identifiers and may sometimes be used as identifiers. What is going on here?

Lexical analyzers are myopic. Their categories are simple and rigid. They don't care about what happens to the tokens they produce. Lumping `select`, `cube`, and

`My_Long_ID` into one group on the basis of simple textual structure makes sense to a lexical analyzer. Let the client receiving the tokens sort out the distinctions. I used *identifier* for tokens that start with alphabetics because I could not think of a better term. Once the next tool receives of the token stream, it may recategorize the tokens however it pleases.

In practice, public tools like *lex* and the lexer used by PL/SQL do the classification of `Identifier` tokens in a separate step after each token is recognized. The language designer provides lists of reserved words and keywords to the tool; typically, each entry has a unique integer to encode its specific use. As an `Identifer` token is discovered, it is filtered by these lists and, if found there, it is revised to fall in the appropriate category with the encoding attached. This allows later stages (notably the parser) to distinguish those "words" which carry special meaning. Anything that falls through the filter is an *unquoted identifier* to PL/SQL and SQL. Along with the *quoted identifiers*, they form the set of legal *identifiers* in those languages.

This note concentrates on the mechanics of lexical analysis. No attention is paid to whether the token stream is remotely legal. Experience shows that language analysis is best done in stages. Assigning clear responsibilities to each stage makes for better and less buggy tools. Lexical analysis has a bright, clear definition. Better it does its job and other tools do theirs than to build a confused mash-up.

Our friend who asked the original question probably did not expect a dissertation of the theory and practice of lexical analysis. But because the PL/SQL and SQL manuals do not provide a complete account of lexical structure in one place and because a little theory always helps when understanding practical problems, my discussion erred on the side of too much information. If our friend wants to build a new tool either to analyze or to generate Oracle languages, I hope he has a better understanding of tool requirements.