Extracting Parameters

Table of Contents

There are several effective ways of extracting parameters in PDSL, the best choice depending on your use case.

Simple use cases: Lexer tokens

The simplest way to extract text out of a sentence is to just grab the token directly.

This can be seen in this simple parameter lexer, which has a *NUMBER* token that will only grab digits in a sentence:

```
lexer grammar SimpleParameterLexer;

fragment WS: (EOF | [\n\r\t ]+);
NUMBER: [0-9]+;
SENTENCE_PARAMETER: [\t ]* ('Given ' | 'And ') 'I have the number "';
END: '"' WS;
```

The parser rule you make will let you grab any token it is composed of. In this case, *parameterExample* will allow the user to grab the text of *SENTENCE_PARAMETER*, *NUMBER* and *END*.

```
parser grammar SimpleParameterParser;

options {tokenVocab=SimpleParameterLexer;}

// The context object will allow us to grab NUMBER directly (as well as the other
tokens)
parameterExample: SENTENCE_PARAMETER NUMBER END;
multiExample: parameterExample+;
```

You actually *use* this when you implement a listener for your parser. Calling *getText()* on any token will return a string of whatever that token matched.

```
private static class SimpleListener extends SimpleParameterParserBaseListener {

    @Override
    public void enterParameterExample(SimpleParameterParser.ParameterExampleContext
ctx) {
        // You can directly reference tokens in a parser rule
        System.out.println(ctx.NUMBER().getText());
    }
}
```

If you have a lexer that defines a token inside another token you will not be able to grab the nested token, only the parent.

```
FOO: 'foo';
BAR: 'bar';
// If you use this in the parser you can only get 'foobar' from NESTED_TOKEN.
// You cannot access FOO or BAR directly.
NESTED_TOKENS: FOO BAR;
```

Avoiding Ambiguity: Island Grammars

Sometimes you can have phrases that are similar enough to each other that it can cause a problem. Consider the example below:

```
lexer grammar BadLexer;

/* Contrast this lexer with PowerfulParameterLexer

Despite the fact that it has the same tokens, the fact that it is not split into
different 'modes' causes a problem.
A docstring and quoted string are so similar it causes ambiguity:
"Some phrase"

"""
some phrase
"""

This ambiguity causes it to produce an error node. These can be very difficult to
debug (and it is a good idea to make
sure you always throw an exception with the visitErrorNode in any listener you produce
for this reason).

The solution is to make your lexer ignore all of the other similar tokens when it is
parsing a specific type. This
is done by making the 'Island Grammars' seen in PowerfulParameterLexer.g4
*/
fragment WS:[\n\r\t ]+ EOF?;
GIVEN_THE_PARAMETER: WS? ('Given ' | 'And ') 'the parameter';
OPEN_QUOTE: ' "' ;
DOCSTRING_START: ':' WS '"""' ;

CLOSE_QUOTE: '"' WS? -> mode(DEFAULT_MODE);
BODY: ~'"'+;

END_DOCSTRING: '"""' WS -> mode(DEFAULT_MODE);
DOCSTRING: .+?;
```

The above lexer attempts to match either strings in double quotes such as `"Fizz buzz"` as well as docstrings such as:

```
"""


Fizz
buzz

""""
```

However the fact that they are both enclosed in double quotes creates ambiguity. This can create difficult to troubleshoot issues.

It is a best practice to always override visitErrorNode in your listeners and have it throw an exception. If your grammar ever produces an error node it can fail silently and take you too much time to figure out what went wrong. In the event you get an error node the likely cause is that your grammar is ambiguous. If the cause is not obvious to you consider doing a binary search by removing half the tokens/rules in your grammar until you can find it.

This problem can be avoided cleanly with the use of *Island Grammars*. Island grammars allow your lexer to switch to a different 'mode' in which it will only consider a subset of tokens until it switches to another.

This is accomplished with the syntax `→ mode(SOME_MODE_YOU_MADE)` following some lexeme you defined. An example of that is below:

```
lexer grammar PowerfulParameterLexer;

fragment WS:[\n\r\t ]+ EOF?;
GIVEN_THE_PARAMETER: WS? ('Given ' | 'And ') 'the parameter';
OPEN_QUOTE: ' "' -> mode(QUOTED_MODE);
DOCSTRING_START: ':' WS '"""' -> mode(DOCSTRING_MODE);

mode QUOTED_MODE;
    CLOSE_QUOTE: '"' WS? -> mode(DEFAULT_MODE);
    BODY: ~'"'+;

mode DOCSTRING_MODE;
    END_DOCSTRING: '"""' WS -> mode(DEFAULT_MODE);
    DOCSTRING: .+?;
```

Note that when an open quote is seen we go into **QUOTED_MODE**. While in quoted mode we can only see 3 tokens: `CLOSE_QUOTE`, `BODY` and `WS`. (The `WS` is only visible in this mode because we started using it in the `CLOSE_QUOTE` token and thus pulled it in from the parent context).

An important consequence of this is other tokens like `DOCSTRING_START` or `END_DOCSTRING` aren’t considered. They might as well not exist when we are in **QUOTED_MODE**. Because of that there is no ambiguity to worry about!

We exit **QUOTED_MODE** when we see the `CLOSE_QUOTE` token. After that we go back to **DEFAULT_MODE** which is ANTLR’s standard mode (the one you start in and stay in unless you explicitly kick into a different mode).

The parser doesn’t really change at all when you make island grammars. Simply make sure you are telling to look for the proper tokens you would expect to find as you change modes:

```
parser grammar PowerfulParameterParser;
options {tokenVocab=PowerfulParameterLexer;}
quotedParameter: GIVEN_THE_PARAMETER OPEN_QUOTE BODY CLOSE_QUOTE;
docstringParameter: GIVEN_THE_PARAMETER DOCSTRING_START DOCSTRING+ END_DOCSTRING;



polymorphicDslAllRules: (quotedParameter | docstringParameter)+;
```

(Discouraged) Manual parsing

You are always free to avoid manually splitting up your tokens and extract information from the text manually.

```
// Calling getText() on this could return something like
// 'Given the parameter "5"\n'
GIVEN_THE_PARAMETER: 'Given the parameter "' [0-9]+ '"' [\r\n]+;
```

It's better if you avoid this. One of the fundamental purposes of a parser and lexer is to do this work *for* you (even if you aren't using it for testing).

- Adding regular expressions, manually splitting strings and the like will make it harder to write and maintain your primary logic.
- It is also more likely to be error-prone, especially after a refactor.
- You will also lose the ability to detect these errors at compile time and only find them at run time.

Real world example

You can see a running example for the code discussed here at ParametersTest.java

The implementations of the *ParseTreeListeners* in particular show you how to grab and use the parameters.