

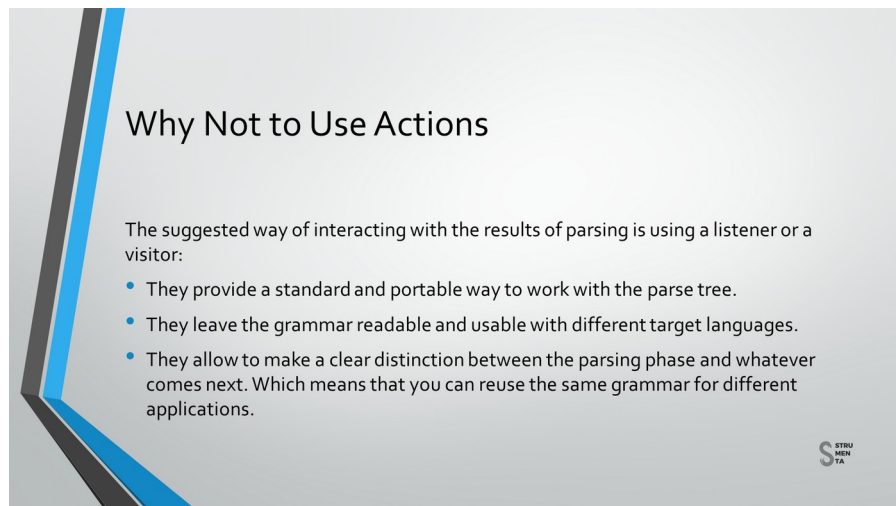


Lesson 07

Actions



In this lesson, we are going to learn what actions are and how to use them in ANTLR grammars. We are going to learn their advantages and drawbacks. We are going to see that they are powerful, but you should use them with caution.



Why Not to Use Actions

The suggested way of interacting with the results of parsing is using a listener or a visitor:

- They provide a standard and portable way to work with the parse tree.
- They leave the grammar readable and usable with different target languages.
- They allow to make a clear distinction between the parsing phase and whatever comes next. Which means that you can reuse the same grammar for different applications.

STRUMENTA

The suggested way of interacting with the result of parsing is using a listener or a visitor. They provide a standard and portable way to work with the parse tree. They leave their grammars readable and usable with different target languages. They allow to make a clear distinction between the parsing phase and whatever comes next, which means that you can reuse the same grammar for different applications.

These are all desirable things and if you can get away without using action, you should.

However, sometimes you cannot avoid them.



Why to Use Actions

There are mainly two reasons to use actions:

- Efficiency and performance
- To parse context-sensitive languages

There are situation in which you need to use the power of actions.

There are mainly two reasons to use them:

- efficiency and performance,
- to parse context-sensitive languages.

Efficiency and Performance

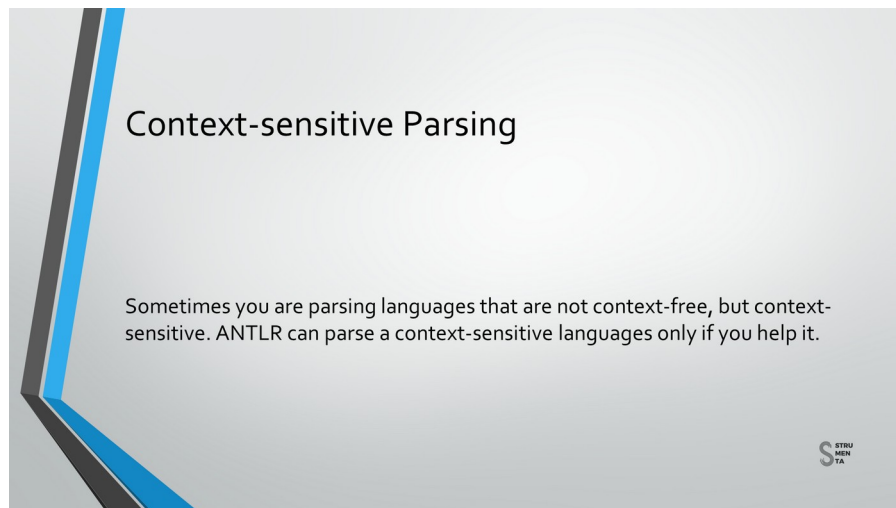
- There is no need to build a parse tree after parsing. So, you can get the exact result you need ready to use, after the parsing ends, saving memory and time.
- You can stop parsing as soon as there is an error. This saves time.



If you need to get the best efficiency and performance possible, you can alter the parsing process or perform operation while ANTLR is parsing.

If you use actions, the first benefit is that there is no need to build a parse tree after parsing. So, you can get the exact result you need ready to use, after the parsing ends, saving memory and time.

Another advantage is that you can stop parsing as soon as there is an error. This saves time.



The second reason is the most important one.

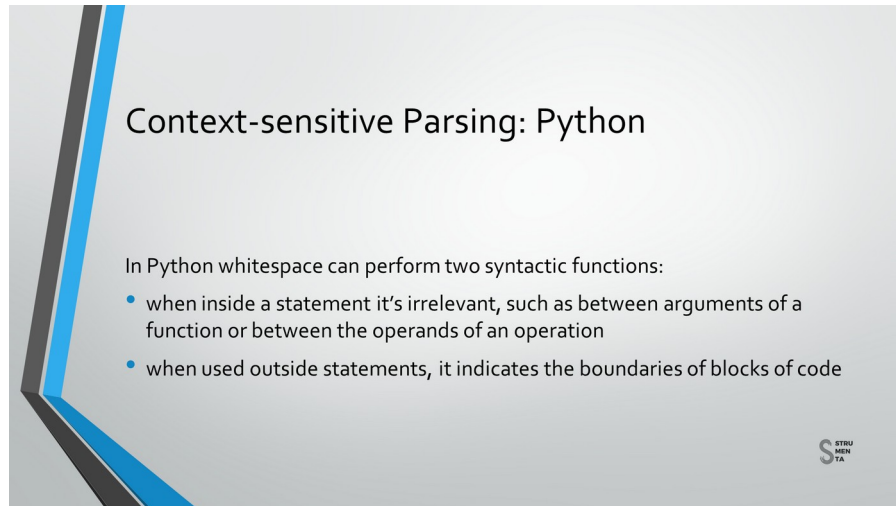
Sometimes, you are parsing languages that are not context-free but context-sensitive.

In our first lesson, we already used the term "context-free language", saying that this is the kind of languages that ANTLR can parse.

Now, actually ANTLR can also parse a context-sensitive language, if you help it. You can help it with actions.

The easiest way to do that is with a specific type of action called a semantic predicate. Semantic predicates are the object of the next lesson, so here we only mention them.





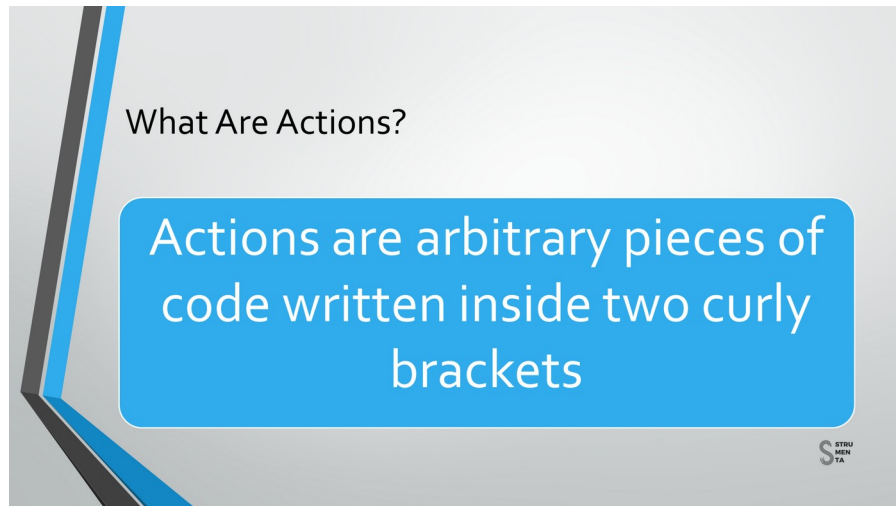
An example of a context-sensitive language is Python. In Python, whitespace can perform two syntactic functions:

- 1) when inside a statement, it's irrelevant, such as between arguments of a function or between the operands of an operation;
- 2) when used outside statements, it indicates the boundaries of blocks of code.

This means that the role of whitespace is context-sensitive. It depends on the elements surrounding it.

ANTLR can deal with this situation only with your help. Now, what are exactly actions?





They are arbitrary pieces of code written inside two curly brackets. Usually, these pieces of code deal with parsing but in theory you can do whatever you want with it.

You could do something completely arbitrary with them, like collecting statistics so the times have always match.

From the ANTLR point of view, it's just code that will be put inside the generated parser or lexer. The only requirement is that the code must be valid code for the target language.

This means both the syntax must be correct and that you cannot reference functions that are not available from the parser.

So, if you use an external library, you must also include it properly to make the resulting parser works.

You may ask yourself, "How do I include properly a library? With magic?"

No, actually there are three places where to put code for actions.



Where to Put Action Code

- in the header position, so that they will be put before the parser class. And this is where you can do things like importing modules in Python.
- in the members position, which will output them at the beginning of the parser class
- next to each rule, or sub-rule, which will make ANTLR output them in the corresponding lexer or parser rule



- In the header position, so that they will be put before the parser class, and this is where you can do things like importing modules in Python;
- In the members position, which will output them at the beginning of the parser class;
- Next to each rule or sub-rule which will make ANTLR output them in the corresponding lexer or parser rule.

```
grammar Color;
1
2
3
4 WHITESPACE : (' ' | '\t') -> skip;
5 NEWLINE : ('\n'? '\n' | '\r');
6
7 fragment LOWERCASE : [a-z] ;
8 fragment UPPERCASE : [A-Z] ;
9 fragment HEX LETTER : [a-f] ;
10 fragment DIGIT : [0-9] ;
11
12 HEX : (HEX_LETTER | DIGIT)+ ;
13
14 # command : color NEWLINE
15 | NEWLINE ;
16
17
18 # color
19
```



What we will build is an application that reads the name of a color from common line and outputs the corresponding exhibition of our presentation.

In addition to that, we can also add custom colors with their own hex code to the list of colors.

So, we are going to see how to use actions and how to create an interactive ANTLR software.

Let's see some code.

We create a grammar with the name color, and we put few basic rules. We skip whitespace but we keep newlines.

We have one rule called color. It matches a string representing a color and an assignment to other custom color.

Then we have another rule called command that matches a color with a newline or a newline alone.

We add a header element to the simple grammar. We use it to import this CSS color module.

Then we add a member element for the parser.

The Python code itself is very simple. There are two functions:

- In setColor, we add a new color to the custom colors dictionary;
- In getCode, we check if the provided name corresponds either to a custom color or to a standard color.

If this is the case, we write on the proper code. Otherwise, we answer with an error.

Since this is Python code, whitespace is relevant, so pay attention to not add unnecessary whitespace to the start of each line. This is to everyone: you cannot add whitespace neither in the members or headers section nor in the actions.

The color rule is where we get things moving.

First, we have to add returns and the name of the variable next to the name of the rule. We could actually return more than one variable if we needed to. That's because it's just a field inside a color context subject.



This is Python so we don't need a type for the variable, but if your language requires it, you will have to add it here.

For instance, in C#, you would write `string answer` between square brackets to return a string variable.

Next to the alternative name of `color`, we add an action to assign to `answer` the result of the function `getCode`. Note that we assign the value to `answer`, we do not return the tab ourselves. Do not put return statement inside an action. If you use that, you're going to break the parser.

To access the variables defined by the parser itself, we prepend that with the dollar sign. In this case we can access `answer`. We can also access the variables that contains the object mentioned by the rule. So, we can access `NAME`, or `HEX`.

The fields of these objects are called attributes in ANTLR parlance.

In this example, we access the attribute `tests` of `name` `index`. The attributes available are pretty much the same that are present in a token object when you use them in a visitor or a listener, so you can get the text, typeline and so forth.

For parser object, you may be interested in the `texts` and `context` object. The `context` object is accessible with the attribute `CTX`. The action corresponding to the assignment of a color is quite similar. The difference is that `setColor` does not return anything. So, we have to set the `answer` directly. We make it a string that reports the number of custom colors present.

All that remains to do in this grammar is adding an action to print the answer. We add this action next to the parser alternative or the command rule.

Now we can take care of our one main Python program. After adding the necessary input, we do something different from the usual.

We create a parser instance first without giving it any input.

We also switch off the generation of the parse tree since we don't need it.

Then we start reading from the start that input one line at a time.



For each line, we set up the lexer making sure that the line and column value are properly set. You want to that, get the correct location in case of errors in parsing.

For instance, try to input an invalid character like the semicolon.

Without setting the line value, you would always get this position of that input, line one, in the error message. In this case, setting the column field is superfluous since we always start from zero. But if you are reading parts of a line, you have to do that.

Once we have a working lexer, we use the method `setInputStream` to add the streams of tokens to the parser. Finally we launch the method for the command rule that will try to match the input with a rule command.

If you launch the Python program, you can keep adding colors until you've written end of five token, which is the combination of Control + Z or Control + D, depending on your platform.


Now, before moving on, we can look at the file `colorParser.py` generated by ANTLR.

Inside the `color` method, you will find the code of the action we define. Actually, ANTLR is a bit smarter than that: if you use an attribute, you will see that ANTLR ensures that it has a valid value. For example, when you use `setColor` with an attribute `text` of the token name, ANTLR checks first that the text exist. Otherwise, it uses the default value of `none`.



Actions in Lexer Rules

- It's rarer but actions can also be used in lexer rules.
- You cannot access attributes inside lexer actions.
- There is fundamentally one reason to use actions in lexer rule: alter the token.



We have seen an example of actions in parser rules. This is where they are used more commonly. However, they can also be used in the lexer, but you can do less with them. You cannot access attributes inside lexer actions.

That is fundamentally one reason to use actions in lexer rule: alter the token. You may want to do that to simplify the token or for some special need.

TEXT Token Example

```
TEXT      : '[' ~[\]]+ '[' {self.text = self.text[1:-1]};
```

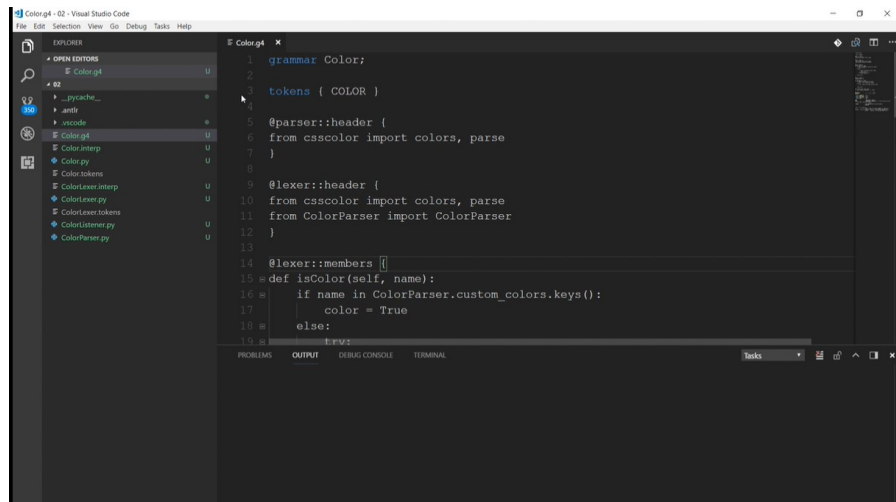
This rule can be erroneously be matched in some paratermized type.

```
// the '>>' is matched as BITSHIFT token  
List<Dictionary<string, int>> x;
```



To see an example of the first case, we can go back to the TEXT token of our chat grammar. You will recall that we were forced to add to the token the beginning and ending square brackets. We could very easily remove them with an action. All that we would need to do is adding an action that will set the text field of the token object with a new value.

You can see how we can set the text in this slide. Notice that different runtimes may have slightly different naming conventions. Obviously, there is not field self in Java, but you may also need to use a setter and a getter instead of using a field.



```
Color.g4
1 grammar Color;
2
3 tokens { COLOR }
4
5 @parser::header {
6   from csscolor import colors, parse
7 }
8
9 @lexer::header {
10  from csscolor import colors, parse
11  from ColorParser import ColorParser
12 }
13
14 @lexer::members {
15  def isColor(self, name):
16      if name in ColorParser.custom_colors.keys():
17          color = True
18      else:
19          color = False
20  }
21
22 EOF;
```

An example of a case when you need to use action in a lexer is dynamic keywords. Let's see an example grammar in which you need to do just that.

We are going to use as a starting grammar, the one we used in the previous example. The difference is that now, we are going to add a color token.

This color token is dynamically defined, so that it's matched only when the string is a valid color. If you use an invalid string, there is no match and we get a parsing error.

Let's start with adding a tokens option with a color value at the start of the grammar. This is not strictly required, but it's useful for clarity. Since we are not going to add the color token with a



normal rule, we can use this option to inform ANTLR that we are going to define our color token in some way.

Now we must have two header declarations: one for the lexer, and one for the parser.

The parser one is the same as the previous one, we import the CSS color module.

In the lexer one, we do the same thing but we also import the color parser because we will access it from the Lexer.

We also add the lexer number section in which we add its color function. This function is quite similar to the getCode function. The difference is that we just write true or false, depending on whatever the string is, a valid color or not.

We could also modify the getCode function inside parser member since now we are sure that we will only get valid colors. However, this is not necessary.

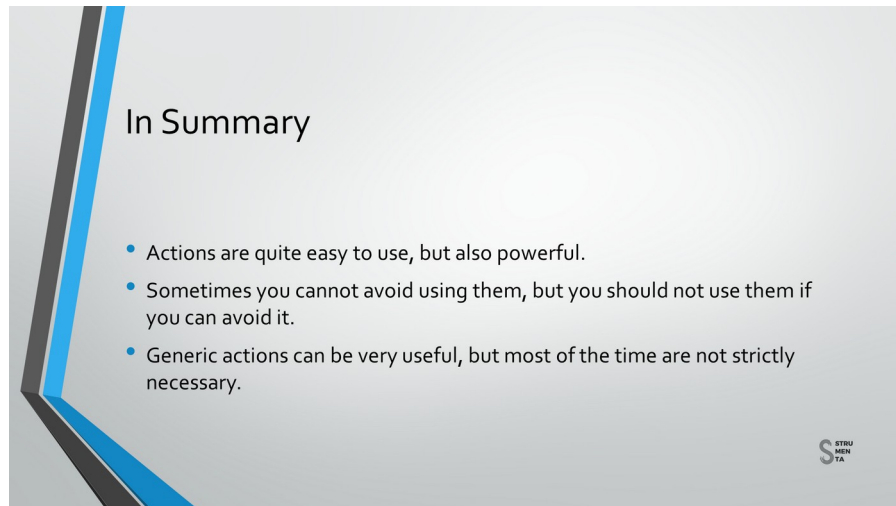
The only thing left to do is adding a lexer action for the name rule. All we have to do, is to check whatever the text of the token is a valid color. If that's true, we change its type and we're up.

Now we have a newly mint Color token.

Now we can test it.

If we try an input like red, everything works fine. If you try something like cool, we get an error. That's because it's considered a Name token. So, it is expecting an equal sign, because that's only valid rule for a Name token. But if we create a custom color with the name "cool", we now have a Color token.





In Summary

- Actions are quite easy to use, but also powerful.
- Sometimes you cannot avoid using them, but you should not use them if you can avoid it.
- Generic actions can be very useful, but most of the time are not strictly necessary.

STRUMENTA

As you can see, actions are easy to use, but quite powerful. Sometimes you cannot avoid using them, but you should not use them if you can.

Generic action can be very useful to make the life of the user of the grammar easier. However, most of the time, they are not strictly necessary.

Generally, there is only one type of action that you may be forced to use to get a correct parsing, a semantic predicate. We are going to see that in the next lesson.

In this next lesson, we've seen what actions are and we have learned about their advantages and drawbacks.

Thank you for watching.

