General Usage

The plugin's usage is simple and straightforward, yet very flexible. Either add the provided set of annotations to your project and start using them, configure the plugin to use a custom set of annotations or simply use the UI configuration to make IntelliJ IDEA learn that e.g. the String argument of `Pattern.compile()` should be treated as a regular expression.

Using the annotations

IntelliLang makes use of three base-annotations: `@Language`, `@Pattern` and `@Subst`

- `@Language` is responsible for the Language injection feature
- `@Pattern` is used to validate Strings against a certain regular expression pattern
- `@Subst` is used to substitute non-compile time constant expressions with a fixed value. This allows you to validate patterns and build the prefix/suffix of an injected language (see below) even for non-constant expressions that are known to contain certain kinds of values during runtime.

The annotations supplied with IntelliLang are located in the file `annotations.jar` which can be found in `IDEA-CONFIG/plugins/IntelliLang/lib`.

@language

The `@Language` annotation can be used to annotate String fields, local variables, method parameters and methods returning Strings. This will cause String-literals that are assigned to a field/variable, passed as a parameter or are used as a method's return value to be interpreted as the specified language.

Additionally, there's the **Language Mismatch** inspection which checks for clashes between the *expected language* and the *actual language* when a field/variable is assigned or a value returned from a method.

The plugin supports *direct* and *indirect* annotations, i.e. you can either directly use the annotation like this:

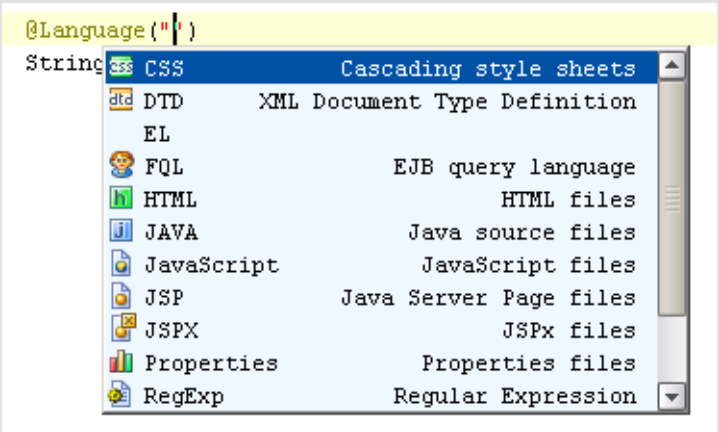
```
@Language("JavaScript")
    String code = "var x = 1 + 2";
```

or annotate another annotation class like this:

```
@Language("XPath")
    public @interface XPath { }
```

which can then simply be used to annotate elements as `@XPath`.

It's very easy to obtain the language-id that must be supplied as the annotation's value attribute: IntelliLang provides the list of available language via the regular code-completion action. Just select the appropriate language from the ctrl-space pop-up window:



@pattern

The `@Pattern` annotation is responsible for marking Strings that have to comply with a certain regular expression and can be used in just the same way as the `@Language` annotation. That means, it's possible to create derived annotations, such as an `@Number` annotation that requires a String to consist of one or more digits:

```
@Pattern("\\d+")
    public @interface Number { }
```

In fact, the predefined annotations already contain two of such derived annotations: The first one, `@PrintFormat`, matches the printf-like pattern used by `java.util.Formatter` and another one, `@Identifier`, describes a valid Java identifier. These are ready to use without having to specify any additional pattern.

@subst

The `@Subst` annotation is used to substitute references that are not compile-time constant which enables the plugin to do **Pattern Validation** based on the assumption that the substituted value is compatible with the values that are expected during runtime. The plugin will complain if the value does not match the expected pattern.

It also helps to build a valid context of prefix/suffix (see next section) for the Language Injection feature. Consider this example:

```
    @Subst("Tahoma")
    final String font = new JLabel().getFont().getName();

    @Language("HTML")
    String message = "<html><span style='font: " + font + "; font-size:smaller'>" + ... + "</span
```

Without substituting the value of the variable `font` with the value *Tahoma* (actually it could just be a single character here), the injected fragment would be syntactically incorrect, causing the error *a term expected* to be displayed after the "`font:`" instruction.

Supplying context: prefix and suffix

When annotating an element, it's possible to supply a prefix and a suffix that should be prepended/appended when the language fragment is being parsed. This can be used to supply context information, i.e. if the prefix for a JavaScript injection is "var someContextVariable;", IntelliJ IDEA will know that the variable *someContextVariable* is declared and will not warn about it being undeclared when it's used.

Apart from the manual ability to supply a prefix and suffix, IntelliLang dynamically determines those values from the context a String literal is being used in:

```
@Language(value =JavaScript, prefix = "function doSomethingElse(a){ }")
    String code = "function doSomething() {\n" +
    "  var x = 1;\n" +
    "  doSomethingElse(x);\n" +
    "}";
```

In this example, the JavaScript language will be injected into each of the three String literals, and each one's prefix and suffix will be calculated from the preceding and following expressions so that the resulting text that's being parsed is valid JavaScript syntax:

- no "missing '}'" error will be displayed: The closing brace is part of the first literal's suffix.
- the variable `x` that is used in `doSomethingElse(a);` will be declared: Its declaration is part of the second literal's prefix
- the function `doSomethingElse()` will be known as well: It's defined in the statically supplied prefix

There are some issues with the IntelliJ IDEA Language Injection API that impose certain restrictions on the prefix/suffix of an injected language fragment. For instance, it's not allowed for a token of the language to span across the prefix/suffix of an element. This could e.g. happen if the prefix ends with a whitespace character and the fragment starts with whitespace. The plugin deals with this special situation in the way that it trims the prefix/suffix and inserts exactly one space character as a separator. However, this doesn't work if a space character is no token separator, which e.g. applies to JavaScript string literals. Such cases cannot be automatically dealt with and IntelliJ IDEA core will produce an assertion.

Even though the dynamic prefix/suffix calculation provides a proper context for the language fragment, some things may not work as expected. Most notably this are the refactoring (rename) and navigation functions. See also the known issues below.