— AnnoFlex —

An annotation-based code generator for lexical scanners

Written in Java, For Java

Version 0.9 (2017-10-21)

Stefan Czaska

We can only see a short distance ahead,
but we can see plenty there that needs to be done.

— Alan M. Turing [9]

Contents

1 Preparations

1.1 Licensing

AnnoFlex is provided to you under the terms and conditions of the 3-Clause BSD license. The full text of this license can be found in the root directory of the installation package of AnnoFlex. You must read and accept this license before you may install and use AnnoFlex.

1.2 System Requirements

AnnoFlex has the following system requirements:

- The code generator requires Java 7 or higher.
- The generated scanner code is compatible to Java 1.1.
- CPU and memory requirements are moderate. The code generation for a typical scanner takes less than a second and does not require more than 32 MB of RAM.

For an overview about the CPU and memory requirements of various scanners have a look at the performance section in chapter 5.3.

1.3 Installation

In order to run AnnoFlex it is not mandatory necessary to install it in your system. Nevertheless, for a comfortable usage it is recommended to install it. This ensures that it can be used everywhere in your system. To do this extract the downloaded installation package and perform the following two environment variables adjustments:

1. Create a new environment variable called `ANNOFLEX_HOME` which points to the root directory of the extracted AnnoFlex package.
2. Add an entry to your `PATH` environment variable which points relative to the bin directory of AnnoFlex. For example: `%ANNOFLEX_HOME%\bin`.

The extracted installation package should have the following structure:

File	Description
`\bin`	Contains scripts to start AnnoFlex.
`\examples`	Contains code examples.
`\lib`	Contains the compiled source code of AnnoFlex.
`\source`	Contains the source code of AnnoFlex.
`changelog.txt`	A list of all changes and improvements of all versions of AnnoFlex.
`license.txt`	The terms and conditions under which it is allowed to use and redistribute AnnoFlex.
`manual.pdf`	The manual of AnnoFlex.
`readme.txt`	Describes the most important things you need to know about AnnoFlex.

Table 1: Content of the installation package.

1.4 IDE Integration

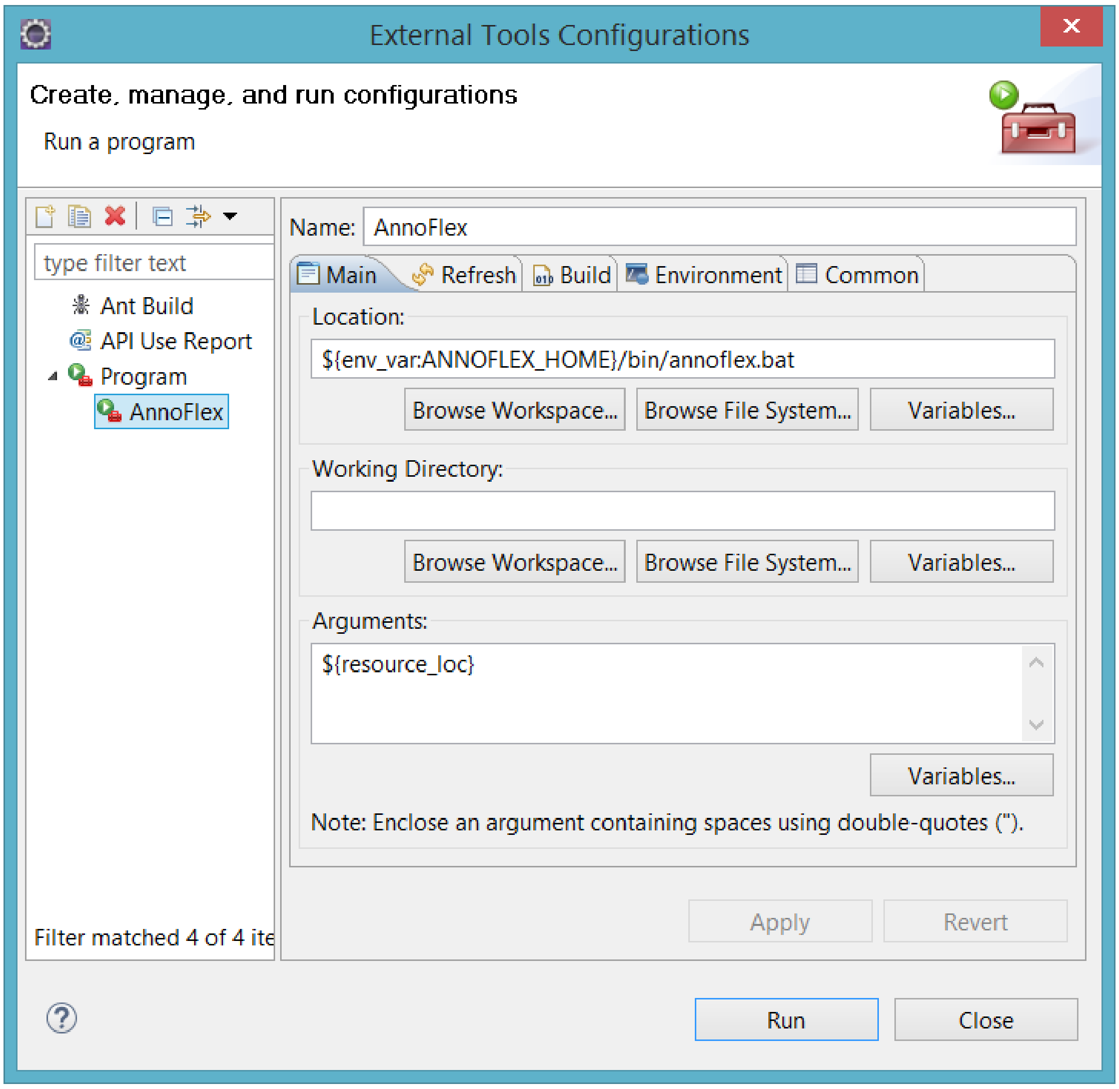
In order to be able to update the code of your scanner definitions within your IDE with just a few clicks it is recommended to add AnnoFlex as an external tool. The following two sections describe how this can be done for Eclipse and IntelliJ IDEA. Furthermore it is recommended to use a keyboard shortcut to launch the external tool. This makes the code update of your scanner definition very comfortable especially during the main development phase where typically a lot of changes are made.

1.4.1 Eclipse

Eclipse provides an *External Tools Configurations* dialog in which launch configurations for external tools can be created. In order to add AnnoFlex as such a tool perform the following steps:

1. Open the *External Tools Configurations* dialog.
2. Click on the *New launch configuration* button or double click the *Program* tree node in order to create a new configuration.
3. Set the *Name* of the configuration to *AnnoFlex*.
4. In the *Main* tab use the following settings:
   a) Set *Location* to `${env_var:ANNOFLEX_HOME}/bin/annoflex.bat`
   b) Set *Arguments* to `${resource_loc}`
5. In the *Refresh* tab use the following settings:
   a) Enable *Refresh resources upon completion*.

The following screenshot shows how the *Main* tab should look after these changes have been made:



Name: AnnoFlex

Location: ${env_var:ANNOFLEX_HOME}/bin/annoflex.bat

Working Directory:

Arguments: ${resource_loc}

Note: Enclose an argument containing spaces using double-quotes (").

Figure 1: The External Tools Configuration dialog in Eclipse with a configuration for AnnoFlex.

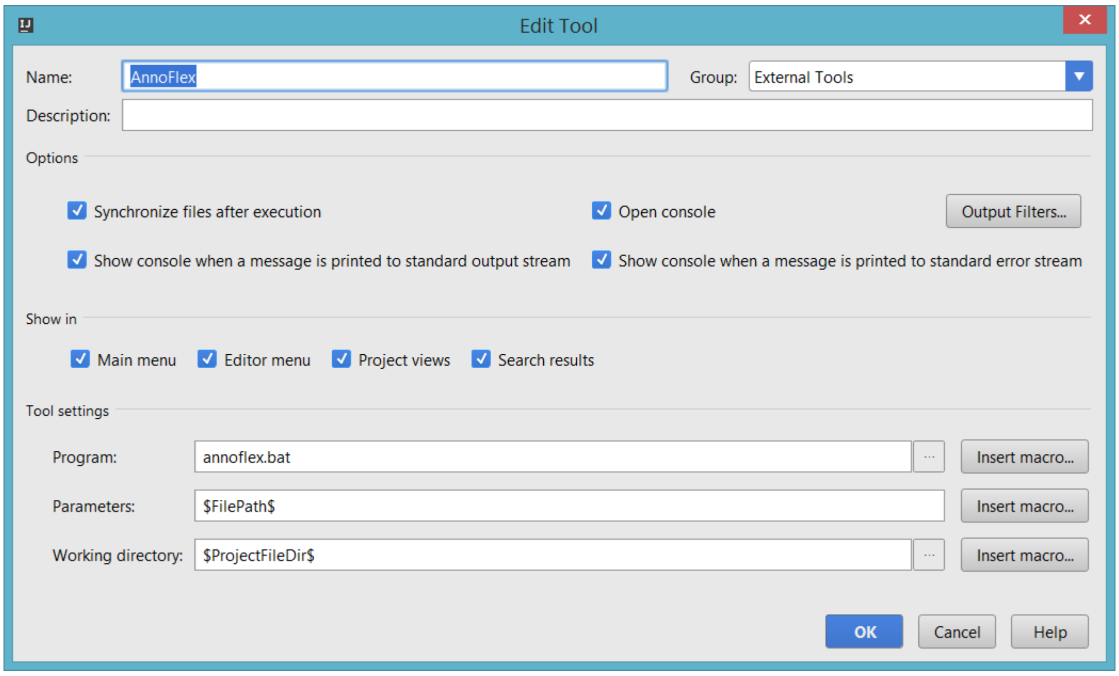
1.4.2 IntelliJ IDEA

Intellij IDEA provides support for external tools via the *External Tools* property page of the *Settings* dialog. In this dialog launch configurations for external tools can be created. In order to add AnnoFlex as such a tool perform the following steps:

1. Open the *Settings* dialog.
2. Click on *Tools* and then on *External Tools*.
3. Click on the *Add* button in order to create a new *Tool* entry.
4. Set the *Name* of the configuration to *AnnoFlex*.

5. In the *Options* section use the following settings:
   a) Enable *Show console when a message is printed to standard output stream.*
   b) Enable *Show console when a message is printed to standard error stream.*
6. In the *Tool settings* section use the following values:
   a) Set *Program* to `annoflex.bat`
   b) Set *Parameters* to `$FilePath$`
   c) Set *Working directory* to `$ProjectFileDir$`

The following screenshot shows how the *Edit Tool* dialog should look after these changes have been made:



Edit Tool

Name: AnnoFlex   Group: External Tools
Description:

Options
☑ Synchronize files after execution   ☑ Open console
☑ Show console when a message is printed to standard output stream   ☑ Show console when a message is printed to standard error stream

Show in
☑ Main menu   ☑ Editor menu   ☑ Project views   ☑ Search results

Tool settings
Program: annoflex.bat
Parameters: $FilePath$
Working directory: $ProjectFileDir$

Figure 2: The Edit Tool dialog in IntelliJ IDEA with a configuration for AnnoFlex.

2 Scanner Definition

Generated scanners are usually defined via lexical rules. A lexical rule assigns a code fragment to a regular expression. The regular expression is detected by a scanning algorithm and the code fragment is called during this algorithm every time the regular expression is found in the input. AnnoFlex uses Javadoc-tag-based annotations for the definition of lexical rules. No common Annotations are used. The main reason for this design choice is that inside Java comments no double escaping is necessary in order to express escaped expressions. This simplifies the syntax in contrast to expressions inside Java strings. The following sections describe the syntax of expressions, explains the concept of lexical states and describe how macros can be used to reduce the size of a scanner definition.

2.1 The `@expr` Tag

Lexical rules can be defined in AnnoFlex with the `@expr` tag. In order to do that the tag must be placed in a Javadoc comment of a non-static class method with no parameters but any type of return value. The content of the tag must be a regular expression. It forms in conjunction with the annotated method a lexical rule. The syntax of the `@expr` tag can be summarized as follows:

```java
/**
 * @expr <regex>
 */
void createToken() {
    // Place token creation code here.
    // Use return values to return specific tokens.
}
```

Each method may only have one `@expr` tag. If multiple expressions are necessary then this can be achieved with the union operator (`|`, see section 2.2.3) and possibly multiple lines if the expression is long. The annotated method may not have parameters. The return type may only be a primitive type (e.g. `int` or `short`) or a simple reference type (e.g. `String`). Array types (e.g. `int[]`) and parameterized types (e.g. `List<String>`) are currently not supported. The return type of all methods of lexical rules must be equal. Sole exception are `void`-methods which can always be used in addition to non-`void` methods. They are called on matches of regular expressions in the same way as non-`void` methods, but after the call the lexical analysis is continued instead to stop it and to return to the caller. If all methods have `void` as their return type then `boolean` is used as the return type of the lexical analysis. In that case, `true` is used to indicate a match and `false` in all other cases.

2.2 Base Expressions

This section describes the most important types of regular expressions in AnnoFlex. All other expressions which do not fall into this category can be considered as specialized expressions and are explained in separate sections after this one. An overview of all available base expressions including a short description gives the following table:

Expression	Description
Single Character	Matches s single character.
Concatenation	Concatenates two or more expressions.
Union	Unions two or more expressions.
String Sequence	A character sequence with a reduced number of meta characters.
Quantifier	Specifies how often an expression may occur.
Modifier	Transforms a simple expression into a complex expression.
Grouping/Nesting	Ensures atomicity and overrides operator precedence.
Character Class	Defines a set of characters.
Lookahead	Looks characters ahead but excludes them from the final match.

Table 2: Expression types.

2.2.1 Single Character

A regular expression that matches a single character can be expressed via the character itself. This is possible for all characters except meta characters. Meta characters are characters that are used to describe the structure of a regular expression. They must be escaped with a backslash if they should be part of the words that are described by the regular expression. Following a few examples of single character expressions:

```
@expr a    # matches the letter "a"
@expr 1    # matches the digit "1"
@expr \\   # matches the backslash character "\"
```

2.2.2 Concatenation

A regular expression that straight follows another expression represents a concatenation. A concatenation matches the text of the first expression followed by the text of the second expression.

```
@expr ab         # matches "ab"
@expr AnnoFlex   # matches "AnnoFlex"
```

2.2.3 Union

The union of two regular expressions is an expression which matches the text of the first expression or the text of the second expression. Unions can be expressed via the union operator `|`.

```
@expr a|b             # matches "a" or "b"
@expr Hello | World   # matches "Hello" or "World"
```

2.2.4 String Sequence

A string sequence is a special version of the concatenation. It only allows single characters and escape sequences and has a reduced set of meta characters in order to simplify the creation of keyword-like expressions. The only meta characters in string sequences are the backslash character \ and the quote character `"`. The backslash can be used to express escape sequences and the quote character can be used to end the string sequence.

```
@expr "Hello World"   # matches "Hello World"
@expr "+*?"           # matches "+*?"
```

2.2.5 Quantifier

Quantifiers specify how often an expression may occur. Possible specifications are either fixed ranges or ranges with an unbounded upper limit. The following table lists all types of quantifiers:

Quantifier	Name	Description
?	Zero-Or-One	Once or not at all.
*	Zero-Or-More	Zero or more times.
+	One-Or-More	One or more times.
{n}	Exactly	Exactly n times.
{n,}	At-Least	At least n times.
{,n}	Up-To	Up to n times.
{n,m}	From-To	At least n times but not more than m times.

Table 3: Syntax and description of quantifiers.

```
@expr a?      # one single "a" or nothing
@expr ab+     # "a" followed by one or more "b"
@expr a{2,4} # two, three or four "a" characters
```

2.2.6 Modifier

Modifiers can be used to create complex expressions based on the combination of a simple expression and a transformation algorithm. A modified expression consist of a modification character followed an arbitrary expression. The modification character specifies the transformation algorithm and the expression after the modification character is the expression which is transformed into the complex expression. AnnoFlex supports the following modifiers:

Modifier	Name	Description
!	Not	Creates the complement of the specified expression. The complement of an expression accepts everything that is not accepted by the expression.
~	Until	Creates an expression that matches everything up to and including the specified expression. For a given regular expression `r` the `Until` modifier is an abbreviation for `!([^]*r[^]*)r`.

Table 4: Syntax and description of modifiers.

```
@expr !a  # matches everything but an "a"
@expr ~a  # matches everything up to and including an "a"
```

2.2.7 Grouping/Nesting

The order in which operators of regular expressions are evaluated depends on the priority of the operators. This priority is fix and can not be changed which results in a default evaluation order which always applies. If an order other than this default order is necessary then this can be achieved by using round brackets. A group of expressions that is surrounded by round brackets represents a single expression exactly as if the group would consist of one single expression without the brackets. This way it is possible to apply operators to any group of expressions independent of the operator priorities inside this group. The following examples demonstrate this:

```
@expr (a|b)(c|d)   # equals [ab][cd]
@expr (ab)+        # one or more "ab"
```

2.2.8 Character Class

Character classes can be used to create any type of character sets. For the creation of these sets multiple components are available. On the top-most level of the character class there are sequences, sequence operators and a negation operator. Sequences are evaluated from left to right and between each pair of two sequences there must be an operator which defines how the right sequence is merged into the left sequence. Inside a sequence there are single characters, character ranges and nested character classes. They are written one after the other without an explicit operator between them. They are merged with an implicit union operator into a character set. The negation operator at the top-most level of the character class negates the entire character class after all sequences have been processed. The following table lists all syntactic possibilities inside character classes:

Type	Name	Associativity	Priority
`[a]`	Single character	n/a	1
`[a-z]`	Character range	n/a	1
`[[a-z]]`	Grouping/Nesting	n/a	1
`[abc0-9]`	Union (implicit)	n/a	2
`[abc||0-9]`	Union (explicit)	left-to-right	3
`[a-z&&opq]`	Intersection	left-to-right	3
`[a-z--opq]`	Set difference	left-to-right	3
`[a-z~~b-y]`	Symmetric difference	left-to-right	3
`[^a]`	Negation	n/a	4

Table 5: Syntax and description of character class components.

Following some examples with special cases that show how character classes are resolved:

```
@expr [^]          # all characters
@expr [ ]          # space character
@expr [+-]         # sign character
@expr [[^]---+]    # equals [[^]--[-+]]
@expr [a-z----x]   # INVALID, two consecutive operators
@expr []           # INVALID, empty set
@expr [a--a]       # INVALID, empty set
```

POSIX Character Classes

In POSIX there are multiple predefined character classes for important topics like letters and digits. AnnoFlex supports these character classes but only in their ASCII version. All corresponding Unicode characters above code point 127 are not part of these character classes.

Character Class	Description
[[:alnum:]]	Letters and digits [a-zA-Z0-9]
[[:alpha:]]	Letters [a-zA-Z]
[[:ascii:]]	All ASCII characters [\x00-\x7F]
[[:blank:]]	Space or tab [\x09\x20]
[[:cntrl:]]	Control characters [\x00-\x1F\x7F]
[[:digit:]]	Decimal digits [0-9]
[[:graph:]]	Printing characters, excluding space [\x21-\x7E]
[[:lower:]]	Lowercase letters [a-z]
[[:print:]]	Printing characters, including space [\x20-\x7E]
[[:punct:]]	Printing characters, excluding letters, digits and space !"#$%&'()*+,-./:;<=>?@[\]^_`{|}~
[[:space:]]	Whitespace [\x09-\x0d\x20]
[[:upper:]]	Uppercase letters [A-Z]
[[:word:]]	Word characters [a-zA-Z0-9_]
[[:xdigit:]]	Hexadecimal digits [a-fA-F0-9]

Table 6: POSIX character classes.

POSIX character classes are only evaluated inside normal character classes. Outside of character classes they are not resolved and represent a normal character class which has the characters of the POSIX class name and a colon in their character set. An example of this case can be found in the following list of examples:

```
@expr [[:xdigit:]]+              # hex numbers
@expr [[:alpha:]][[:word:]]*     # simple identifiers
@expr [[:^letter:]--[:blank:]]   # no letters and no spaces
@expr [:digit:]                  # VALID but equals [:digt]
```

Full Stop Character Class (., \N, \V)

A dot (.) and the escape sequences \N and \V represent a full stop character class which stands for all characters which are not a Unicode newline character:

```
[^\x0a-\x0d\x85\u2028\u2029]
```

The dot has this meaning only outside of character classes. Inside a character class it simply represents a single dot character. The escape sequences \N and \V do not have this restriction. They can be used inside and outside of character classes.

```
@expr //.*       # end-of-line comment
@expr [\V--\t]   # full stop also at tabs
```

2.2.9 Lookahead

Lookaheads can be used to exclude a specific part at the end of an expression from the final match result. A lookahead expression consists of two parts, a content part and a condition part. The content part is the part of the expression which should actually be matched and the condition part is an expression which must follow the content expression but without being part of the final match. Lookaheads can be read as "x if followed by y". Lookaheads are expressed via the slash (/) operator. It must be placed between the content expression and the condition expression. Lookahead expressions may also occur multiple times in an expression but they must be placed at the top-most level of the expression and surrounded by round brackets and separated via the union operator. An example for that case can be found in the following list of examples:

```
@expr a / b               # "a" if followed by "b"
@expr a | b / c | d       # equals [ab]/[cd]
@expr a | (b / c) | d     # equals [a]|(b/c)|[d]
@expr (a / b) | (c / d)   # two separate lookaheads
```

Lookahead expressions should be used with care as they are slower than expressions without a lookahead. This results from the fact that the condition part is always thrown away after the match has been determined. If the next run of the scanner starts immediately after the content expression, which is usually the case, then the condition characters must be read again. This is especially slow if the condition part is very long. The worst case is a condition expression of variable length which always reaches up to the end of the input. Such a condition can lead to a scanner runtime of $\mathcal{O}(n^2)$. Because of this it makes sense to use lookahead expressions only if real necessary and if so then they should have whenever possible a condition length which is fix and as short as possible.

2.3 Escape Sequences

AnnoFlex supports three different types of escape sequences. One for characters, one for character classes and one for character sequences. Each escape sequence type can be used at all places where the corresponding expression type is allowed. Escape sequences are always started with a backslash followed by at least one character. The exact syntax varies from sequence to sequence and is explained in the next sections.

2.3.1 Escaped Characters

Characters can be escaped in many different ways. AnnoFlex supports the syntax for single letters, numerical values and Unicode names. The following table lists all possibilities in detail:

Sequence	Description
`\a`	The alarm bell character `0x07`
`\b`	The backspace character `0x08` (only inside a character class)
`\cX`	The control character corresponding to `X`
`\e`	The escape character `0x1b`
`\f`	The form feed character `0x0c`
`\n`	The line feed character `0x0a`
`\N{name}`	The Unicode character with the specified name
`\o{O..O}`	The character with octal value `O..O` of arbitrary length
`\r`	The carriage return character `0x0d`
`\t`	The tab character `0x09`
`\uHHHH`	The character with the specified 16-bit hexadecimal value `0xHHHH`
`\u{H..H}`	The character with hexadecimal value `0xH..H` of arbitrary length
`\UHHHHHH`	The character with the specified 24-bit hexadecimal value `0xHHHHHH`
`\xHH`	The character with the specified 8-bit hexadecimal value `0xHH`
`\x{H..H}`	The character with hexadecimal value `0xH..H` of arbitrary length
`\O`	The character with the specified 8-bit single digit octal value `"O"` (0 to 7)
`\OO`	The character with the specified 8-bit double digit octal value `"OO"` (00 to 77)
`\OOO`	The character with the specified 8-bit triple digit octal value `"OOO"` (000 to 377)
`\CHAR`	The specified character if it is not a letter or digit

Table 7: Escape sequences for single characters.

Following some examples of escaped characters:

```
@expr \a  # alarm bell character
@expr \b  # backspace character
```

2.3.2 Escaped Character Classes

Escape sequences can also be used to express character classes. AnnoFlex supports all common sequences for letters, digits and whitespace. The general purpose sequence for Unicode character properties is also supported. Further details about this topic can be found in section 2.4. The following table lists all supported types of escape sequences for character classes:

Sequence	Description
`\d`	An ASCII digit `[0-9]`
`\D`	An ASCII non-digit `[^\d]`
`\h`	A Unicode horizontal white space character `[\x09\x20\xa0 \u1680\u180e\u2000-\u200a\u202f\u205f\u3000]`
`\H`	A Unicode non-horizontal whitespace character `[^\h]`
`\N`	A Unicode non-vertical whitespace character `[^\v]`
`\p{spec}`	A Unicode character with the specified property
`\P{spec}`	A Unicode character without the specified property
`\s`	An ASCII whitespace character `[ \t\n\x0B\f\r]`
`\S`	An ASCII non-whitespace character `[^\s]`
`\v`	A Unicode vertical whitespace character `[\x0a-\x0d\x85\u2028\u2029]`
`\V`	A Unicode non-vertical whitespace character `[^\v]`
`\w`	An ASCII word character `[a-zA-Z0-9_]`
`\W`	An ASCII non-word character `[^\w]`

Table 8: Escape sequences for character classes.

Following some examples of escaped character classes:

```
@expr \d  # ASCII digit
@expr \h  # horizontal whitespace character
```

2.3.3 Escaped Character Sequences

For character sequences there exists two escape sequences. The first one stands for all Unicode possibilities to express a line terminator and the second one is a general purpose sequence which escapes everything until a special end sequence occurs. The following table describes both sequences in detail:

Sequence	Description
`\R`	Unicode newline sequence: `\x0d\x0a | [\x0a-\x0d\x85 \u2028\u2029]`
`\Q\E`	Quote sequence: Starts quoting with `\Q` and ends it with `\E`. All characters are interpreted as simple characters and lose their special meaning if they have one. This also includes escape sequences. Only \E is detected as a valid escape sequence and ends the quote sequence.

Table 9: Escape sequences for character sequences.

Following some examples of escaped character sequences:

```
@expr \R      # newline sequence
@expr \Q\R\E  # matches "\R"
```

2.4 Unicode Character Properties

In Unicode there are a large number of predefined character properties. They assign property-specific values to all Unicode characters. These values can be used in two principle ways. On the one hand it is possible to query the value for a single character, for example in order to check whether a specific character is a digit or not and on the other hand it is possible to determine for a property all characters which have a specific value, for example in order to determine the set of all characters which are digits. The second use-case is particularly interesting for the definition of character classes in regular expressions. In order to express a character class via Unicode character properties the escape sequences `\p` and `\P` can be used. They combine a property name, a property value and an assignment operator in order to express a character class. The syntax is as follows:

```
UnicodeProperty := PositiveSpec | NegativeSpec

PositiveSpec := "\p{" Spec "}"
NegativeSpec := "\P{" Spec "}"

Spec := BinaryPropertyName
      | ScriptValue
      | GeneralCategoryValue
      | PropertyName ( ":" | "=" | "!=" ) PropertyValue
```

The positive specification (p, lowercased) uses the specified character set as is. The negative specification (P, capitalized) uses the complement of the specified character set which means that all character which are not part of the specified set are part of the final set. The inner specification consists of four different cases. The last one (name, operator and value) is a generic selector. Depending on the used operator it selects all characters with a specified property value or all characters that do not have a specified value. The first three cases (binary, script and general category) are shortcuts for commonly occurring properties. Unicode guarantees that these properties will never have intersections in their names and values. This makes it possible to omit the value (binary) or name (script and general category) and the operator and to specify only the remaining part as a single value.

2.4.1 Binary Properties

A binary property is a property which only knows two values. One for binary true and one for binary false. A character for which a binary property is true has the specified Unicode property and if the binary property is false then it does not have the specified Unicode property. The digit property can be used as a simple example. A character is either a digit or not. If it is a digit then the value of the binary property is true otherwise it is false. In order to make the definition of binary properties more comfortable there are multiple values available to express both cases. Besides a long version, which is *yes* and *no*, there is also a short version, which is *y* and *n*. Additionally there exists alternative aliases for each of these values which makes in total eight different values which are summarized in the following table:

Long	**Short**	**Additional Aliases**
Yes	Y	True, T
No	N	False, F

Table 10: Value aliases of binary properties.

The following list of examples demonstrates how binary properties and their value aliases are used:

```
@expr \p{whitespace=yes}  # whitespace characters
@expr \P{whitespace=no}   # also whitespace characters
@expr \p{whitespace}      # whitespace via shortcut
@expr \p{uppercase:true}  # uppercase characters
@expr \p{lowercase!=y}    # all but lowercase characters
@expr \p{hexdigit:f}      # all but hexadecimal characters
```

Unicode defines a large number of binary properties. The most of them are supported by AnnoFlex. Only a small subset is currently unsupported. The following list shows which properties are supported:

Alnum	IdStart
Alphabetic	IdContinue
Any	Ideographic
ASCII	IDSBinaryOperator
ASCIIHexDigit	IDSTrinaryOperator
Assigned	JoinControl
BidiControl	LogicalOrderException
BidiMirrored	Lowercase
Blank	Math
CaseIgnorable	NoncharacterCodePoint
Cased	OtherAlphabetic
ChangesWhenCasefolded	OtherDefaultIgnorableCodePoint
ChangesWhenCasemapped	OtherGraphemeExtend
ChangesWhenLowercased	OtherIdContinue
ChangesWhenNFKCCasefolded	OtherIdStart
ChangesWhenTitlecased	OtherLowercase
ChangesWhenUppercased	OtherMath
CNTRL	OtherUppercase
CompositionExclusion	PatternSyntax
Dash	PatternWhitespace
DefaultIgnorableCodePoint	PrependedConcatenationMark
Deprecated	Print
Diacritic	Punct
Digit	QuotationMark
ExpandsOnNFC	Radical
ExpandsOnNFD	SentenceTerminal
ExpandsOnNFKC	SoftDotted
ExpandsOnNFKD	TerminalPunctuation
Extender	UnifiedIdeograph
FullCompositionExclusion	Uppercase
Graph	VariationSelector
GraphemeBase	Whitespace
GraphemeExtend	Word
GraphemeLink	XDigit
HexDigit	XIdContinue
Hyphen	XIdStart

Additional to the predefined properties of Unicode there are some AnnoFlex-specific properties. The most of them are also available via other syntax constructs (escape sequence and character class) and are redefined as a character property for the sake of completeness. The following table lists all AnnoFlex-specific properties:

Name	**Description**
HWhitespace	Horizontal whitespace characters `\h`
VWhitespace	Vertical whitespace characters `\v`
PosixAlnum	Letters and digits `[a-zA-Z0-9]`
PosixAlpha	Letters `[a-zA-Z]`
PosixASCII	All ASCII characters `[\x00-\x7F]`
PosixBlank	Space or tab `[\x09\x20]`
PosixCntrl	Control characters `[\x00-\x1F\x7F]`
PosixDigit	Decimal digits `[0-9]`
PosixGraph	Printing characters, excluding space `[\x21-\x7E]`
PosixLower	Lowercase letters `[a-z]`
PosixPrint	Printing characters, including space `[\x20-\x7E]`
PosixPunct	Printing characters, excluding letters, digits and space `!"#$%&'()*+,-./:;<=>?@[\]^_`{|}~`
PosixSpace	Whitespace `[\x09-\x0d\x20]`
PosixUpper	Uppercase letters `[A-Z]`
PosixWord	Word characters `[a-zA-Z0-9_]`
PosixXDigit	Hexadecimal digits `[a-fA-F0-9]`
JavaIdentifierStart	Corresponds to `Character.isJavaIdentifierStart()`
JavaIdentifierPart	Corresponds to `Character.isJavaIdentifierPart()`

Table 11: AnnoFlex-specific binary properties.

2.4.2 Script, Block and Age

The *Script*, *Block* and *Age* properties are according to Unicode *catalog* properties. Each property has its own property-specific list of values. The *Script* property specifies to which script a character belongs, the *Block* property specifies to which Unicode code block a character belongs and the *Age* property specifies in which Unicode version a character has been introduced. As the list of available values for these properties is long the values are not listed here. Only a few examples are shown in order to demonstrate how these properties are typically used.

```
@expr \p{latin}                   # latin characters
@expr \p{script=greek}            # greek characters
@expr \p{block=Basic Latin}       # basic latin characters
@expr \p{block=ASCII}             # ASCII characters
@expr \p{age=9.0}                 # all Unicode 9.0 chars
@expr [\p{age=9.0}--\p{age=8.0}]  # chars since Unicode 9.0
```

2.4.3 General Category

The *General Category* property is a default categorization of all characters. It differentiates between *letters*, *marks*, *numbers*, *punctuation*, *symbols*, *separators* and *special characters*. Each category has a long and an abbreviated name. Both names and a short description of all available values are listed in the following table:

Abbr	**Long**	**Description**
Lu	UppercaseLetter	An uppercase letter
Ll	LowercaseLetter	A lowercase letter
Lt	TitlecaseLetter	A digraphic character, with first part uppercase
LC	CasedLetter	Lu | Ll | Lt
Lm	ModifierLetter	A modifier letter
Lo	OtherLetter	Other letters, including syllables and ideographs
L	Letter	Lu | Ll | Lt | Lm | Lo
Mn	NonspacingMark	A nonspacing combining mark (zero advance width)
Mc	SpacingMark	A spacing combining mark (positive advance width)
Me	EnclosingMark	An enclosing combining mark
M	Mark	Mn | Mc | Me
Nd	DecimalNumber	A decimal digit
Nl	LetterNumber	A letterlike numeric character
No	OtherNumber	A numeric character of other type
N	Number	Nd | Nl | No
Pc	ConnectorPunctuation	A connecting punctuation mark, like a tie
Pd	DashPunctuation	A dash or hyphen punctuation mark
Ps	OpenPunctuation	An opening punctuation mark (of a pair)
Pe	ClosePunctuation	A closing punctuation mark (of a pair)
Pi	InitialPunctuation	An initial quotation mark
Pf	FinalPunctuation	A final quotation mark
Po	OtherPunctuation	A punctuation mark of other type
P	Punctuation	Pc | Pd | Ps | Pe | Pi | Pf | Po
Sm	MathSymbol	A symbol of mathematical use
Sc	CurrencySymbol	A currency sign
Sk	ModifierSymbol	A non-letterlike modifier symbol
So	OtherSymbol	A symbol of other type
S	Symbol	Sm | Sc | Sk | So
Zs	SpaceSeparator	A space character (of various non-zero widths)
Zl	LineSeparator	U+2028 LINE SEPARATOR only
Zp	ParagraphSeparator	U+2029 PARAGRAPH SEPARATOR only
Z	Separator	Zs | Zl | Zp
Cc	Control	A C0 or C1 control code
Cf	Format	A format control character
Cs	Surrogate	A surrogate code point
Co	PrivateUse	A private-use character
Cn	Unassigned	A reserved unassigned code point or a noncharacter
C	Other	Cc | Cf | Cs | Co | Cn

Table 12: Values of the General Category property.

Similar to the script property and the binary properties the general category property supports shortcuts which means that the property name and the operator can be omitted. The following examples show how the general category property is typically used:

```
@expr \p{gc=letter}  # all letter characters
@expr \p{letter}     # letter characters via shortcut
@expr \P{N}          # all but number characters
```

2.4.4 Character Names

All Unicode characters can be addressed via unique character names. For this purpose there exists two character properties, the *name* and the *nameAlias* property. The *name* property defines the main name of a character and the *nameAlias* property defines additional character names which often occurs in practice. If a character should be addressed via one of its names the general purpose escape sequences `\p` and `\P` can be used. The *name* property has special logic which also accepts name aliases if the specified name is not a valid character name. The *nameAlias* property does not have this logic and acts as a raw property which only accepts valid name aliases. Additionally there exists the escape sequence `\N`. Similar to the *name* property it combines character names and name aliases. It has already been introduced in section 2.3.1 as a character escape sequence. The following list of examples shows all possibilities to address characters via character names:

```
# Note: space is a name and tab is a name alias

@expr \p{name=space}      # space via name property
@expr \p{nameAlias=tab}   # tab via nameAlias property
@expr \p{name=tab}        # VALID, name accepts aliases
@expr \p{nameAlias=space} # INVALID, space is not an alias
@expr \P{name=space}      # all except a space character
@expr \N{space}           # space via shortcut
@expr \N{tab}             # tab via shortcut
```

2.5 Lexical States

Lexical states can be used to switch between sets of regular expressions during the scanning algorithm of the generated scanner. Each set of expressions represents a single sub-scanner which can be activated by switching into the corresponding lexical state. The default state is called the initial state. Further lexical states can be defined via start conditions in regular expressions.

2.5.1 Start Conditions

Start conditions can be used to define new lexical states. They must be specified via angle brackets at the start of a regular expression or at the start of a sub expression inside a top level union (see examples below). Inside the angle brackets there must be specified a comma separated list of condition names. The specified names are the names of the lexical states. The expression that follows the list of start conditions is part of each sub-scanner represented by the corresponding names. The angle brackets may also contain a single asterisk which adds the expression to all lexical states. The following listing shows all syntactic possibilities for the definition of lexical states. Of particular interest here is the operator priority of start conditions and the union operator.

```
@expr <Condition1>a    # "a" if lexical state is Condition1
@expr <C1,C2>b         # "b" if lexical state is C1 or C2
@expr c                # "c" if lexical state is Initial
@expr <Initial>c       # "c" if lexical state is Initial
@expr <*>d             # "d" in all lexical states
@expr <C>a|b           # "a or b" if lexical state is C
@expr (<A>a)|(<B>b)    # "a" if lexical state is A or
                       # "b" if lexical state is B
@expr <C>(<A>a)|(<B>b) # "a" if lexical state is A or C or
                       # "b" if lexical state is B or C
```

The names of start conditions are simple ASCII identifiers that start with a letter optional followed by letters, digits and underscores. As they are used for the generation of static final integer constants they are also case insensitive and insensitive against a camel case to underscore normalization which works as follows. If the name of a start condition contains at least one lowercase character then the name is considered unnormalized. In order to normalize it, underscores are inserted at all camel case boundaries. A camel case boundary is a position at which a transition from lowercase to uppercase occurs or a position at which a digit/non-digit change occurs. Following some examples that demonstrate the normalization rules:

```
stateOne -> STATE_ONE # lowercase/uppercase transition
state1   -> STATE_1   # digit/non-digit change
state1a  -> STATE_1_A # two digit/non-digit changes
STATE1   -> STATE1    # already normalized
STATE_1A -> STATE_1A  # already normalized
```

2.5.2 Condition Areas

Condition areas can be used to simplify the assignment of start conditions to a large number of expressions. If all expressions of a longer list of lexical rules have the same start conditions then it is easier to define the start and end of a start condition range instead to assign the start condition to each expression separately. The condition area is defined via two end-of-line comments. The first defines the start of the condition area and the second the end. Both comments must contain a special marker text which is detected by AnnoFlex as a processing instruction. The text of the first comment must additionally contain a list of start conditions. The syntax is the same as in expressions. The following example shows a condition area in action:

```
//%%LEX-CONDITION-START%%C1,C2%%

/** @expr a */ void createA() {}
/** @expr b */ void createB() {}
/** @expr <C3>c */ void createC() {}
...

//%%LEX-CONDITION-END%%
```

All expressions inside a condition area inherit the start conditions of that area. Additional it is possible to define further start conditions per expression.

2.5.3 Lexical State Stacks

If a scanner uses lexical states then it can be necessary to organize the state changes in the form of an ordered sequence. If this is the case then it is useful to put all previous states on state changes onto a stack. This makes it possible to restore them in reversed order. Such a stack is called a lexical state stack. The simplest implementation in AnnoFlex uses an `ArrayList` and two methods to enter and leave a lexical state. The following code requires knowledge about the code generation and the customization of the generated code which are described later. If it should be unclear what the code below exactly does then have a look at section 3.6.12, 3.7.3 and 4.2.13 first.

```java
/**
 * This is the stack of lexical states.
 */
private ArrayList<Integer> stateStack = new ArrayList<>();

/**
 * The stack initially contains the "initial" state.
 */
public void setString(String string) {
    setStringInternal(string);

    stateStack.clear();
    stateStack.add(LEXICAL_STATE_INITIAL);
}

/**
 * Pushes a state onto the stack and enters it.
 */
private void enterLexicalState(int lexicalState) {
    stateStack.add(lexicalState);
    setLexicalState(lexicalState);
}

/**
 * Leaves the current state and enters the previous one.
 */
private void leaveLexicalState() {
    stateStack.remove(stateStack.size()-1);
    setLexicalState(stateStack.get(stateStack.size()-1));
}
```

2.6 Macros

Macros are named expressions which can be reused in other expressions in order to make them shorter and more readable. They can also be used to remove duplicates from expressions which makes them more maintainable.

2.6.1 The `@macro` Tag

For the definition of macros there exists a separate Javadoc tag, the *macro* tag. It assigns an expression to an identifier. The assigned expression may contain any type of expression except start conditions. The identifier to which the expression is assigned must be a simple ASCII identifier that starts with a letter optional followed by letters, digits and underscores. The syntax of a macro definition can be summarized as follows:

```
/**
 * @macro <name> = <regex>
 */
```

Macros are referenced inside expressions by their name. The name must be surrounded by curly brackets. Macros can also be nested which means that they can also be used inside other macros. Cycles are invalid and reported as an error. Examples for the definition and reference of macros can be found in the examples sub-section of this section.

2.6.2 Global and Local Macros

AnnoFlex distinguishes between global and local macros. Global macros are macros which are defined in the class comment of a scanner. Their scope is the entire scanner definition which means that their name must be unique across all global and local macros. Local macros are macros which are defined in a lexical rule. Their scope is limited to this rule and their name must be unique across all macros in this rule and all global macros. Local macros can be used to avoid naming conflicts if a large number of macros are present. They can also be used to simplify the usage of macros because it is easier to edit a macro definition and its reference if both are part of the same comment.

2.6.3 Examples

The following sample code shows how macros are used. It combines global and local macros and shows that the name of a local macro may also appear in other local macros without to produce a naming conflict.

```
/**
 * This is a global macro.
 *
 * @macro NewLine = \r | \n | \r\n
 */
public class Scanner {

    /**
     * This is a local macro.
     *
     * @macro Content = (\\\" | [^\r\n"])*
     * @expr \" {Content} \"? | (\" {Content} / {NewLine})
     **/
    String createDoubleQuote() {
        return "DoubleQuote";
    }

    /**
     * This is also a local macro.
     *
     * @macro Content = (\\' | [^\r\n'])*
     * @expr ' {Content} '? | (' {Content} / {NewLine})
     **/
    String createSingleQuote() {
        return "SingleQuote";
    }
}
```

2.7 Syntax Summary

The following grammar summarizes the syntax of regular expressions in AnnoFlex:

```
RootExpr := [Conditions] LookaheadExpr
          | [Conditions] TopLevelExpr { '|' TopLevelExpr }

TopLevelExpr := BaseExpr
              | '(' Conditions BaseExpr ')'
              | '(' [ Conditions ] LookaroundExpr ')'

Conditions := '<' Condition { ',' Condition } '>'

LookaheadExpr := BaseExpr '/' BaseExpr | Macro

BaseExpr := BaseLiteralChar | EscapedChar
          | '!' Expr | '~' Expr
          | BaseExpr Quantifier
          | BaseExpr BaseExpr
          | BaseExpr '|' BaseExpr
          | CharClass | '.'
          | '"' StringElement { StringElement } '"'
          | EscapedCharSequence
          | Macro | '(' BaseExpr ')'

Quantifier := '*' | '+' | '?' | '{' Number '}' |
            | '{' Number ',}' | '{,' Number '}' |
            | '{' Number ',' Number '}'

CharClass := '[' ['^'] CCSequence { CCOp CCSequence } ']'
           | EscapedCharClass

CCOp := '||' | '&&' | '--' | '~~'

CCSequence := CCChar | CCChar '-' CCChar
            | CharClass | EscapedCharSequence
            | Macro

CCChar := CCLiteralChar | EscapedChar

StringElement := StringLiteralChar | EscapeSequence

Macro := '{' Identifier '}'
```

Meta Characters

Meta characters are characters that are used to describe the structure of a regular expression. They can not be used as a normal text character. If the character which they represent should be part of the text then they must be escaped with a backslash. There are three different sets of meta characters which are used depending on context:

Context	Meta Characters
String Sequence	`\ "`
Character Class	`\ [ ] ^ { }`
All others	`\ [ ] ^ { } | ( ) < > " . ? * + / ~ ! $`

Table 13: Meta characters in regular expressions.

Operator Precedence

AnnoFlex has the following operator precedence in regular expressions:

Name	Operator	Associativity	Priority
Grouping/Nesting	(expr)	n/a	1
Modifier	! ~	right-to-left	2
Quantifier	* + ? {n} {n,} {,n} {n,m}	left-to-right	3
Concatenation	expr1 expr2	n/a	4
Union	expr1 | expr2	n/a	5
Lookahead	expr1 / expr2	n/a	6
Condition	<A>expr	n/a	7

Table 14: Operator precedence of regular expressions.

Whitespace Handling

As whitespace is a good aid to increase the readability of regular expressions it is ignored in AnnoFlex by default. The only contexts in which whitespace is handled are string sequences and character classes. The following table summarizes this:

Context	Whitespace Handling
String Sequence	Sensitive, considers whitespace
Character Class	Sensitive, considers whitespace
All others	Insensitive, ignores whitespace

Table 15: Whitespace handling in regular expressions.

3 Code Generation

The previous chapter has shown how a scanner definition is structured and which types of regular expressions can be used to define the scanner language. Now it's time to see how the code for a such a scanner definition can be generated and which features are covered by this code.

3.1 The Code Area

The primary design goal of AnnoFlex is to keep all parts of a lexical scanner in one single Java source code file. This includes, of course, the generated code. As there are many possibilities for the location of that code AnnoFlex provides a mechanism to specify the location. This mechanism is called the *code area*. It consists of two end-of-line comments. The first defines the start of the code area and the second the end. Both comments must contain a special marker text which is detected by AnnoFlex as a processing instruction. The following example shows how both markers look:

```
//%%LEX-MAIN-START%%
//%%LEX-MAIN-END%%
```

The source code of the generated scanner is always placed between these two comments. Any content in between is fully replaced by the new code. Manual performed modifications are always discarded. No partial update is performed at any time. This ensures that the generated code does not depend on previous code generations.

3.2 Running the Code Generator

The command line API of the code generator is as follows:

```
annoflex [options] <file>
```

wheras `<file>` is a scanner class whose code area should be updated and `[options]` are command line specific settings which specify how the current run of the generator should behave. All available options and their meanings are summarized in the following table:

Option	Description
-b --bom	Specifies whether a Byte Order Mark should be generated at the start of the scanner file. Requires one of the following mode constants: `ON`, `OFF`, `AUTO` (default)
-c --charset	Specifies the character encoding for reading and writing the scanner file. Requires a valid character set name as for example `US-ASCII` or `UTF-8` (default).
-h --help	Shows the command line help text.
-l --linesep	Sets which line separators should be used for the scanner file. Requires one of the following line separator mode constants: `LF`, `CR`, `CRLF`, `SYSTEM` or `AUTO` (default).
-v --version	Shows the version of AnnoFlex, Java and the operating system.

Table 16: Parameters of the command line API.

Byte Order Mark constants:

- `ON`: Enables the Byte Order Mark.
- `OFF`: Disables the Byte Order Mark.
- `AUTO (default):` Enables the Byte Order Mark if it is present on load and disables it if it is not present on load.

Character Set name: The specified name must be a valid character set name which is supported by the used Java VM. Valid values could be for example `US-ASCII` or `ISO-8859-1`. If no character set has been specified then `UTF-8` is uses as default on all platforms.

Line Separator mode:

- `LF`: Uses the line feed character `0x0a` as the line separator for the whole file. All existing line separators are converted to `0x0a`.
- `CR`: Uses the carriage return character `0x0d` as the line separator for the whole file. All existing line separators are converted to `0x0d`.
- `CRLF`: Uses the sequence `0x0d0x0a` as the line separator for the whole file. All existing line separators are converted to `0x0d0x0a`.
- `SYSTEM`: Uses the system line separator as the line separator for the whole file. All existing line separators are converted to the system line separator.
- `AUTO (default):` Uses the line separator which is present in the file. If the file contains multiple different line separators then the primary line separator is used. The primary line separator is the one which occurs most often in the file and if there are multiple possibilities then the first of these is used. If the file does not contain a line separator then the system line separator is used. All existing line separators are converted to the chosen line separator.

Following some examples of the command line API:

```
annoflex TutorialPart1.java
annoflex -c utf-8 TutorialPart2.java
annoflex -b on -l crlf TutorialPart3.java
```

The exit status of the command line API is 0 on success and 1 on every type of error which especially includes errors in the scanner definition. Warnings are not handled like errors and lead to a successful run of the generator.

```
D:\>annoflex TutorialPart1.java
[Info] Loading file "TutorialPart1.java"
[Info] Computing scanner
[Info] Saving file "TutorialPart1.java"
[Info] 0 Errors, 0 Warnings

D:\>echo %errorlevel%
0
```

If the code generator detects AnnoFlex-specific errors in the scanner class then the Java file is not updated. The only action which is performed in that case is the display of the corresponding errors in the console. If the scanner class has Java-specific compile errors then it is allowed to run the code generator. This especially applies to errors inside the code area and inside methods that use generated code as these areas are not evaluated for the code generation. If an update of the code area will help to fix some of these errors then feel free to run the code generator.

3.3 Generation Results

After a successful run of the generator the code area contains the code of the scanner. It provides all means which are necessary to perform a lexical analysis based on the specified lexical rules. The following topics are covered by the generated scanner:

- Character Input
- Start position of next scan
- Match results
- Scanning method
- Lexical states

Details about each of these topics are described in the next sections of this chapter. Additional information about these topics can be found in chapter 4 which handles the customization of the scanner. The generated code is divided into *code sections*. Each code section covers a certain topic of the generated scanner as for example “all fields related to the character input” or “all methods in conjunction with match results”. A code section usually contains class members in the form of fields and methods. There are only two exceptions, the *logo* section and the *statistics* section which only contain a single comment. All code sections which do have class members also act as a *member group* whereby the name of the code section is used as the name of the member group. Member groups are used by certain options (described in chapter 4). They apply properties to all members of a group. The *all* member group references all members of all member groups.

3.4 Scanner Input

The generated scanner requires a sequence of characters in order to be able to perform its work. This character input can be specified in AnnoFlex in two different ways. Either by a *java.lang.String* or by a *java.io.Reader*. The first possibility is called *string mode* and the second one is called *reader mode*. The next sections describe the difference between both modes.

3.4.1 String Mode

The *string mode* is based on a *java.lang.String* instance. By default the entire string is used for the lexical analysis but it is also possible to restrict the analysis to a specific region. This eliminates the need to create a substring of the input if only a part of it should be scanned. All characters which lie outside the specified region are from the scanner's point of view non-existent and never used. There are no character access restrictions in string mode. A random access to all characters is possible at any time. Also the start position of the next scan can be changed at any time. This makes it possible to create tokens in any possible way, even backwards. However, the scan direction is in all cases forwards which means that the scanning algorithm iterates the characters from the current start position to the end of the input. A real backward scanning is only possible if a reversed version of the string is used.

3.4.2 Reader Mode

The *reader mode* is based on a *java.io.Reader* instance. All input characters are read one after the other and stored in a buffer. The access to the characters in this buffer depends on the used buffering strategy. There is the *all characters* strategy and the *current match* strategy. The difference between both strategies is whether previously read characters are overwritten by new characters or not. The following figure visualized the difference between both strategies:

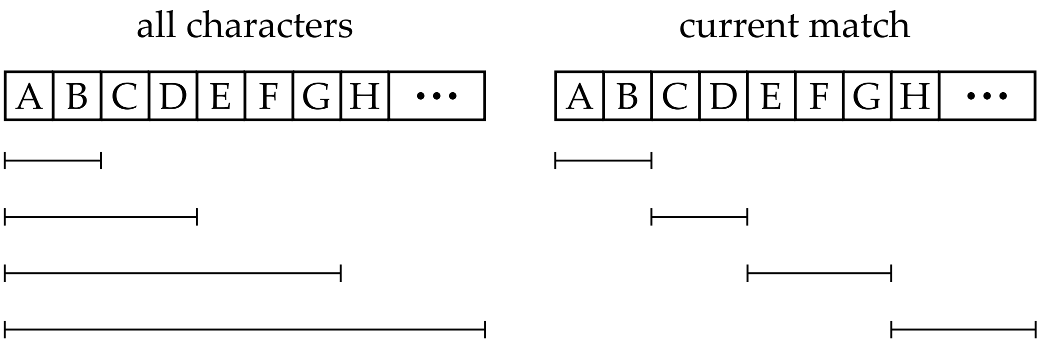


Figure 3: Buffering strategies in *reader* mode.

The *all characters* buffering strategy stores all characters in the buffer and does never discard previously read characters. This mode is comparable to the string mode with the exception that characters are first available when they have been read. A random access to all characters in the buffer is possible at any time. Also the start position can be moved backwards at any time which allows you to rescan specific parts of the input if necessary. The *current match* strategy stores only those characters in the buffer which are necessary to determine the current match. Every time a new scan starts it is possible that characters of previous matches are discarded and overwritten by new characters. Whether this actually happens or not depends on the length of previous matches and the length of the current match. This makes it in general impossible to predict it. Because of that only the characters of the current match should be accessed in reader mode as otherwise unpredictable errors can happen at any time.

3.5 The Scanning Algorithm

The main function of the generated scanner is the detection of regular expressions and the regions to which they apply. These regions are called *matches* and the process of finding these regions is called *pattern matching.* Each run of this process requires a start position. In AnnoFlex this start position is called the *dot.* This name is derived from the fact that positions in character sequences are sometimes visualized as a dot. The result of a pattern match is a character range. If this range is empty then no matching regular expression could be found. If this range is non-empty then a matching regular expression has been found. The method of the corresponding lexical rule is called in that case. It has now the possibility to handle the match. A typical action is the creation of a token but it is also possible to change only some internal states or to do nothing. There are no limitations what such a method may do or not. Methods of lexical rules can do whatever the scanner requires at this point of time. The entire process which combines the dot, pattern matching, match ranges and lexical rule methods is called the *scanning algorithm* or the *scanning method.* A snapshot of its properties is shown in the following figure:

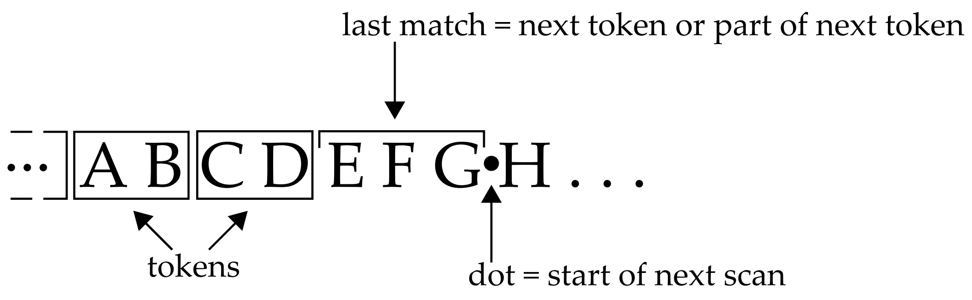

last match = next token or part of next token
... A B C D E F G • H ...
tokens
dot = start of next scan

Figure 4: Properties of the scanning algorithm.

There are two types of lexical rule methods. Methods with a return type and methods without a return type. If a method has a return type then it is assumed that it creates a token and if it does not have a return type then it is assumed that it does not create a token. The return value of a method with a return type is always passed through to the caller of the scanning method even if the value is null (or another value depending on the return type). The lexical analysis is suspended for such methods and first resumed if the scanning method is called again. Methods without a return type usually only change internal states, as for example the lexical state of the scanner. They only prepare the creation of a token and delegate this task to a method call in the future. When a method without a return type has been called by the scanning method the lexical analysis is immediately continued by starting a new pattern matching. This procedure repeats until a method with a return type is called or the end of the input is reached. The following pseudo code summarizes all steps of the scanning algorithm:

```
WHILE input is available
    find longest match
    save match result
    set dot to end of match

    IF token creation method has return value THEN
        CALL token creation method and RETURN result
    ELSE
        CALL token creation method and CONTINUE scanning
    END IF
END WHILE

RETURN default value
```

3.6 List of Default Code Sections

The following table summarizes all code sections which are generated independent of the input mode. The order in the table equals the order in the source code file.

Code Section	Description
Logo	The source code logo of AnnoFlex.
Statistics	A table which contains statistics about the scanner definition and the generated code.
Table constants	All constants related to the internal tables of the scanner.
Lexical state constants	All constants which are related to lexical states.
Helper constants	Contains all internal helper constants of the scanner.
Dot fields	Contains all fields related to the start position of the next scan.
Lexical state fields	Contains all fields which are related to the current lexical state.
Match fields	Contains all fields to manage the results of a single step of the lexical analysis.
Helper fields	Contains all internal helper fields of the scanner.
Table methods	Contains methods to create the internal tables of the scanner.
Dot methods	Contains methods for the management of the start position of the next scan.
Lexical state methods	Contains methods for the management of the current lexical state.
Match methods	Contains methods to get the values of the last match.
Scan methods	Contains all methods of the scanning algorithm.
Helper methods	Contains all internal helper methods of the scanner.

Table 17: All code sections that are independent of the input mode.

3.6.1 The `logo` section

The *logo* code section consists of a comment containing a logo that introduces the code area of the scanner. It can be enabled and disabled via the *logo* option, which is described in section 4.2.1.

```
//================================================
//     _                      _____ _
//    / \   _ __  _ __   ___ |  ___| | _____  __
//   / _ \ |  _ \|  _ \ / _ \| |_  | |/ _ \ \/ /
//  / ___ \| | | | | | | (_) |  _| | |  __/>  <
// /_/   \_\_| |_|_| |_|\___/|_|   |_|\___/_/\_\
//
//================================================
```

3.6.2 The `statistics` section

The *statistics* code section consists of a comment containing a table with characteristic values about the generated code. The table can be enabled and disabled via the *statistics* option, which is described in section 4.2.2. The statistics table has the following content:

```
/*************************************************
 *              Generation Statistics            *
 * * * * * * * * * * * * * * * * * * * * * * * * *
 *                                               *
 * Rules:            95                          *
 * Lookaheads:       1                           *
 * Alphabet length:  69                          *
 * NFA states:       576                         *
 * DFA states:       304                         *
 * Static size:      111 KB                      *
 * Instance size:    24 Bytes                    *
 *                                               *
 *************************************************/
```

The meaning of each property is explained in the following table:

Name	Description
Rules	The total number of lexical rules in the scanner definition.
Lookaheads	The total number of lookaheads in all regular expressions.
Alphabet length	The size of the virtual alphabet of the DFA.
NFA states	The total number of NFA states.
DFA states	The total number of DFA states.
Static size	This is a rough measure for the memory footprint of static data of the generated scanner class. The memory consumption of custom code is not included in this value.
Instance size	This is a rough measure for the memory footprint of non-static data of the generated scanner class. The memory consumption of custom code is not included in this value.

Table 18: Properties of the generated scanner.

3.6.3 The `tableConstants` section

The *tableConstants* code section contains constants for all tables of the DFA. They are used by the scanning algorithm to perform the pattern matching. All constants of this code section may not be accessed or manipulated by user code. They should be considered as private and non-existent. The *tableConstants* code section has the following class members:

Member	**Description**
characterMap	This is a mapping table which maps Unicode characters to characters of the virtual alphabet of the DFA.
transitionTable	This is the transition table of the DFA. It contains for each DFA state and each character of the virtual alphabet a reference to another DFA state.
actionMap	This is a mapping table which contains for each DFA state the corresponding action number.

Table 19: Members of the *tableConstants* code section.

3.6.4 The `lexicalStateConstants` section

The *lexicalStateConstants* code section contains all constants related to lexical states. All members of this code section are released for public use. Lexical states are described in detail in section 2.5. The *lexicalStateConstants* code section has the following class members:

Member	**Description**
lexicalStateEnum	Two or more enumeration constants which contain the ordinal numbers of all lexical states.

Table 20: Members of the *lexicalStateConstants* code section.

3.6.5 The `helperConstants` section

The *helperConstants* code section stores all generic and miscellaneous constants for which it makes sense to separate them from other code sections. All members of this code section are released for public use. The *helperConstants* code section has the following class members:

Member	**Description**
emptyCharArray	An empty char array which is used in reader mode to avoid null checks.

Table 21: Members of the *helperConstants* code section.

3.6.6 The `dotFields` section

The *dotFields* code section contains all fields related to the start position of the next scan. They may be accessed at any time but should not be manipulated by user code.

Use the appropriate dot methods in section 3.6.11 to change the dot. The *dotFields* code section has the following class members:

Member	Description
dot	The start position of the next scan.

Table 22: Members of the *dotFields* code section.

3.6.7 The `lexicalStateFields` section

This code section contains all fields which are related to lexical states. They may be accessed at any time but should not be manipulated by user code. Use the appropriate lexical state methods in section 3.6.12 to change the lexical state. For more information about lexical states have a look at section 2.5. The *lexicalStateFields* code section has the following class members:

Member	Description
lexicalState	The ordinal number of the current lexical state.

Table 23: Members of the *lexicalStateFields* code section.

3.6.8 The `matchFields` section

This code section contains all fields which are necessary to store the values of the last match. They may be accessed at any time but should not be manipulated by user code. For an input mode independent access to all text and character related properties of the last match it is recommended to use the methods in section 3.6.13. The *matchFields* code section has the following class members:

Member	Description
matchStart	The start of the last match.
matchEnd	The end of the last match (exclusive).
matchLookahead	The end of the last match (exclusive) including the lookahead.

Table 24: Members of the *matchFields* code section.

3.6.9 The `helperFields` section

This code section contains all internal helper fields of the scanner. They may not be accessed or manipulated by user code. They should be considered as private and non-existent. The *helperFields* code section has the following class members:

Member	Description
startState	The start state of the DFA of the current lexical state. It is only present if the scanner has more than one lexical state.
positionList	A list which contains all match end positions of a regular expression during a lookahead length determination. It is only present if there are lookahead expressions which do not have a constant part that can be used to compute the length directly.

Table 25: Members of the *helperFields* code section.

3.6.10 The `tableMethods` section

This code section contains all DFA table related methods. They may not be called by user code and should be considered as private and non-existent. The *tableMethods* code section has the following class members:

Member	Description
createCharacterMap	Unpacks the run-length encoded character map string and returns the resulting mapping table.
createTransitionTable	Unpacks the run-length encoded transition table string and returns the resulting two-dimensional mapping table.
createActionMap	Unpacks the run-length encoded action map string and returns the resulting mapping table.

Table 26: Members of the *tableMethods* code section.

3.6.11 The `dotMethods` section

This code section contains all methods for the access and manipulation of the start position of the next scan. All members of this code section are released for public use. The *dotMethods* code section has the following class members:

Member	Description
setDot	Sets the start position of the next scan to a new position.
getDot	Returns the start position of the next scan.

Table 27: Members of the *dotMethods* code section.

3.6.12 The `lexicalStateMethods` section

This code section contains all methods related to lexical states. All members of this code section are released for public use. Lexical states are described in detail in section 2.5. The *lexicalStateMethods* code section has the following class members:

Member	**Description**
setLexicalState	Sets the current lexical state of the scanner.
getLexicalState	Returns the current lexical state of the scanner.

Table 28: Members of the *lexicalStateMethods* code section.

3.6.13 The `matchMethods` section

This code section contains all methods which are available to access the properties of the last match. All members of this code section are released for public use. The *matchMethods* code section has the following class members:

Member	**Description**
getMatchStart	Returns the start of the last match.
getMatchEnd	Returns the first character after the end of the last match without the lookahead.
getMatchLookahead	Returns the first character after the end of the last match including the lookahead.
getMatchLength	Returns the length of the last match without the lookahead.
getMatchTotalLength	Returns the length of the last match including the lookahead.
getMatchLookaheadLength	Returns the length of the last lookahead.
getMatchText	Creates and returns a String with the characters of the last match without the lookahead.
getMatchTextRange	Similar to getMatchText but the sub-string can be adjusted by a start and end offset.
getMatchTotalText	Creates and returns a String with the characters of the last match including the lookahead.
getMatchLookaheadText	Creates and returns a String with the characters of the last lookahead.
getMatchChar	Returns a character relative to the start of the last match.

Table 29: Members of the *matchMethods* code section.

3.6.14 The `scanMethods` section

The *scanMethods* code section contains all methods which are related to the scanning algorithm. All members of this code section are released for public use. The *scanMethods* code section has the following class members:

Member	Description
getNextToken	Performs at the start position of the next scan a new lexical analysis and returns the result.

Table 30: Members of the *scanMethods* code section.

The signature and the behavior of the *getNextToken* method is highly configurable. Have a look at section 4.2 for more information.

3.6.15 The `helperMethods` section

This code section contains all internal helper methods of the scanner. Apart form the *hasNextChar* method they may not be called by user code and should be considered private and non-existent. The *helperMethods* code section has the following class members:

Member	Description
hasNextChar	Checks whether a specified position has a next character.
computeMatchEnd	Computes the match end position of a variable lookahead.

Table 31: Members of the *helperMethods* code section.

3.7 List of String-Mode Code Sections

The following table summarizes all code sections which are generated only if the input mode is *string*. The order in the table equals the order in the source code file.

Code Section	Description
String fields	Contains all fields which are necessary to store the string for which the lexical analysis should be performed.
Region fields	Contains all fields which are necessary to restrict the lexical analysis to a sub-region of the string.
String methods	Contains all methods for the management of the string fields.
Region methods	Contains all methods to set and get the properties of the sub-region of the string.

Table 32: All code sections of the *string* input mode.

3.7.1 The `stringFields` section

This code section contains all fields which are related to the `String` instance if the input mode is *string*. The fields may be accessed at any time but should not be manipulated by user code. Use the appropriate string methods in section 3.7.3 to change the string. The *stringFields* code section has the following class members:

Member	Description
string	The current string to be scanned.

Table 33: Members of the *stringFields* code section.

3.7.2 The `regionFields` section

This code section contains all fields which are related to the region property of the `String` instance if the input mode is *string*. The fields may be accessed at any time but should not be manipulated by user code. Use the appropriate region methods in section 3.7.4 to change the region. Have a look at section 3.4.1 for more information about regions. The *regionFields* code section has the following class members:

Member	Description
regionStart	The start of the scan region (inclusive) inside the string.
regionEnd	The end of the scan region (exclusive) inside the string.

Table 34: Members of the *regionFields* code section.

3.7.3 The `stringMethods` section

This code section contains all methods to access and manipulate the input string. All members of this code section are released for public use. The *stringMethods* code section has the following class members:

Member	Description
setString	Sets the current string to be scanned.
getString	Returns the current string to be scanned.

Table 35: Members of the *stringMethods* code section.

3.7.4 The `regionMethods` section

This code section contains all methods to access and manipulate the region. All members of this code section are released for public use. The *regionMethods* code section has the following class members:

Member	Description
setRegion	Sets the start and end of the region.
getRegionStart	Returns the start of the region.
getRegionEnd	Returns the end of the region.

Table 36: Members of the *regionMethods* code section.

3.8 List of Reader-Mode Code Sections

The following table summarizes all code sections which are generated only if the input mode is *reader*. The order in the table equals the order in the source code file.

Code Section	Description
Reader fields	Contains all fields which are related to the *Reader* instance which is used to read input characters.
Buffer fields	Contains all fields which are necessary to store the input characters.
Reader methods	Contains all methods which are related to the management of the *Reader* instance.
Buffer methods	Contains methods to access the character buffer.

Table 37: All code sections of the *reader* input mode.

3.8.1 The `readerFields` section

This code section contains all fields which are related to the `Reader` instance if the input mode is *reader*. The fields may be accessed at any time but should not be manipulated by user code. Use the appropriate reader methods in section 3.8.3 to change the reader fields. The *readerFields* code section has the following class members:

Member	Description
reader	The *Reader* from which the input characters are read.
readerStartCapacity	The initial size of the character buffer.

Table 38: Members of the *readerFields* code section.

3.8.2 The `bufferFields` section

This code section contains all fields which are related to the character buffer which is used as a backing store of the `Reader` instance if the input mode is *reader*. In contrast to the string fields, all buffer fields may not be accessed or manipulated by user code. They should be considered as private and non-existent. Use the appropriate buffer methods in section 3.8.4 to access the buffer. The *bufferFields* code section has the following class members:

Member	**Description**
buffer	The character buffer in which all characters of the *Reader* are stored.
bufferStart	The start position of the buffer relative to the first character of the input.
bufferEnd	The end position of the buffer relative to the first character of the input.

Table 39: Members of the *bufferFields* code section.

3.8.3 The `readerMethods` section

This code section contains all *Reader*-related methods. All members of this code section are released for public use. The *readerMethods* code section has the following class members:

Member	**Description**
setReader	Reinitializes the scanner by setting the Reader and the initial buffer size.
getReader	Returns the current reader.
getReaderStartCapacity	Returns the initial buffer size.

Table 40: Members of the *readerMethods* code section.

3.8.4 The `bufferMethods` section

This code section contains all character buffer related methods. All members of this code section are released for public use. The *bufferMethods* code section has the following class members:

Member	**Description**
getBuffer	Returns the current character buffer. The returned reference may change over time due to reallocations of the buffer.
getBufferStart	Returns the start position of the buffer (inclusive) relative to the first character of the input.
getBufferEnd	Returns the end position of the buffer (exclusive) relative to the first character of the input.

Table 41: Members of the *bufferMethods* code section.

4 Options

Options can be used to control how the code of the scanner is generated. They must be placed in the class comment of the scanner and are evaluated at the start of the code generation. Options represent global code generator settings which do not influence lexical rules and macros. The following sections describe the option syntax, summarize all available options and describe the functionality of each option in detail.

4.1 The `@option` Tag

Options must be specified with the `@option` tag inside the class comment of the scanner class. The tag starts the option declaration and is followed by an option name followed by an assignment character followed by an option-specific configuration text. The name of the option selects the option type and the option-specific text contains the data which is processed by the option. The syntax of this text varies from a single keyword up to a list of identifier/operator pairs. The general syntax of an option declaration can be summarized as follows:

```
/**
 * @option <name> = <value>
 */
public class MyScanner {
    // lexical rules
    // code area
}
```

4.2 List of Options

The following table lists all supported options in AnnoFlex:

Name	Description
logo	Specifies whether a logo of AnnoFlex should be generated at the start of the code area.
statistics	Specifies whether a statistics table with information about the scanner and its specification should be generated.
headings	Specifies whether and how headings for code sections should be generated.
methodName	Sets the name of the getNextToken method.
methodThrows	Sets the content of the throws clause of the getNextToken method.
defaultReturnValue	Sets the default return value of the getNextToken method.
inputMode	Specifies how the scanner input is read and processed.
bufferStrategy	Specifies which buffer model should be used if input mode is *reader*.
bufferIncrement	Specifies which formula should be used to increment buffer sizes.
functionality	Controls which scanner methods, fields and constants should be generated.
javadoc	Controls for which methods, fields and constants Javadoc comments should be generated.
visibility	Sets the visibility of methods, field and constants.
internal	Sets which methods should have *private* as their default visibility and a special name to avoid name conflicts with other methods.
noMatchAction	Sets which action should be performed inside the getNextToken method if no match could be found.

Table 42: All available options and their meaning.

4.2.1 The `logo` option

The *logo* option specifies whether the AnnoFlex code logo (described in section 3.6.1) should be generated at the start of the code area. The *logo* option has the following syntax:

```
Logo := "enabled" | "disabled"
```

If the specified value is *enabled* then the logo is generated. If the value is *disabled* then the logo is not generated. The default value is *enabled*.

4.2.2 The `statistics` option

The *statistics* option specified whether a table with statistical information (described in section 3.6.2) should be generated at the start of the code area. The *statistics* option has the following syntax:

```
Statistics := "enabled" | "disabled"
```

If the specified value is *enabled* then the statistics table is generated. If the value is *disabled* then the statistics table is not generated. The default value is *enabled*.

4.2.3 The `headings` option

The *headings* option specifies whether all code sections of the code area should obtain a descriptive heading. The *headings* option has the following syntax:

```
Headings := "enabled" | "disabled"
          | "small" | "medium" | "large"
```

If the specified value is set to *disabled* then no headings are generated. If the value is set to *enabled*, *small*, *medium* or *large* then headings are generated. *Enabled* is a synonym for *large*. Both generate headings with one line at the top and one line at the bottom of the section name. *Medium* generates only one line at the bottom and *small* generates no lines. The default value is *enabled*.

4.2.4 The `methodName` option

The *methodName* option specifies the name of the *getNextToken* method. It is described in detail in section 3.6.14. Please refer this section for more information. The *methodName* option has the following syntax:

```
MethodName := JavaIdentifier
```

The specified value must be a valid Java identifier. The default value is *getNextToken*.

4.2.5 The `methodThrows` option

The *methodThrows* option can be used to specify which *Throwable* classes the *getNextToken* method should have in its *throws* clause. The *methodThrows* option has the following syntax:

```
MethodThrows := Item { "," Item }

Item := LinkToJavaClassType
      | JavaClassType
```

Multiple *Throwable* values must be separated by comma. Each value can either be a Java class type or a link to a Java class type. Links should be preferred as this ensures that features of your IDE like *reference search* and *renaming* work also for these references. This option does not have a default value which means that no *throws* clause is generated.

4.2.6 The `defaultReturnValue` option

The default value of the *getNextToken* method can be specified with the *defaultReturnValue* option. The value of this option is used in all cases where the *getNextToken* method is not able to return a value. This is especially the case if the end of the input is reached. The *defaultReturnValue* option has the following syntax:

```
DefaultReturnValue := LinkToJavaConstant
                    | ArbitraryCharacterSequence
```

The specified value can either be a link to a Java constant or an arbitrary character sequence. If the value is a link then only the linked constant is used. If it is not a link then it is considered as an arbitrary character sequence and used "as is". The default value of this option depends on the return type of the *getNextToken* method which itself depends on the return type of the used lexical rules. All possible default values are summarized in the following table:

Return Type	**Default Value**
boolean	false
byte, short, int, long, float, double	-1
char	0
Reference	null

Table 43: Default values of the *defaultReturnValue* option.

The *arbitrary character sequence* value can be used to specify arbitrary Java code. However, this feature should be used with care as the code is located inside a comment and thus treated by your IDE as plain text. Features like *reference search* and *renaming* will possibly not work.

4.2.7 The `inputMode` option

This option specifies the *input mode* of the scanner. Input modes are described in detail in section 3.4. Please refer this section for further details. The *inputMode* option has the following syntax:

```
InputMode := "string" | "reader"
```

If the specified value is *string* then the *string* mode (`java.lang.String`) is used. If the value is *reader* then the *reader* mode (`java.io.Reader`) is used. The default value is *string*.

4.2.8 The `bufferStrategy` option

The *bufferStrategy* option specifies which kind of buffering should be used if the *input mode* is *reader*. The different types of buffering are explained in section 3.4.2. Please refer this section for further details. The *bufferStrategy* option has the following syntax:

```
BufferStrategy := "currentMatch" | "allCharacters"
```

If the specified value is *currentMatch* then the buffer strategy *current match* is used. Only the characters of the current match are available in this mode. Buffer sizes are $\mathcal{O}(maxMatchLength)$. If the value is *allCharacters* then the buffer strategy *all characters* is used. The buffer is constantly increased and stores all read characters. Buffer sizes are $\mathcal{O}(inputLength)$. The default value of the `bufferStrategy` option is *currentMatch*.

4.2.9 The `bufferIncrement` option

The *bufferIncrement* option specifies which formula is used to increment buffer sizes. This especially includes the main character buffer in *reader* mode but also all other buffers that are used by the scanner. The *bufferIncrement* option has the following syntax:

```
BufferIncrement := "goldenRatio" | "double"
```

If the specified value is *goldenRatio* then the buffer sizes are multiplied by 3/2. If the value is *double* then the buffer sizes are increased by doubling the size. The default value is *goldenRatio* which is the default behavior of many classes in Java.

4.2.10 The `functionality` option

The *functionality* option controls which scanner functionality should be generated. The *functionality* option has the following syntax:

```
Functionality := Item { Item }

Item := Member Modifier
      | MemberGroup Modifier

Modifier := "+" | "-"
```

The *functionality* option specifies a set of members for which code should be generated. Each member which is part of the set is generated. Members which are not part of the set are only excluded from the code generation if they are not required by other generated functionalities. If they are required, they are still generated. The modifiers of the *functionality* option have the following meaning:

Modifier	Description
+	Adds the specified member or group of members to the set of generated functionalities.
−	Removes the specified member or group of members from the set of generated functionalities.

Table 44: Modifiers of the *functionality* option.

The default value is "*all- readerMethods+ stringMethods+ regionMethods+ dotMethods+ getMatchStart+ getMatchEnd+ getMatchLength+ getMatchText+ getMatchChar+*".

4.2.11 The `javadoc` option

The *javadoc* option controls for which class members a Javadoc comment should be generated. The *javadoc* option has the following syntax:

```
JavaDoc := Item { Item }

Item := Member Modifier
      | MemberGroup Modifier

Modifier := "+" | "-"
```

The *javadoc* option specifies a set of members for which Javadoc comments should be generated. A Javadoc comment is only generated if the corresponding member is part of the set, otherwise no Javadoc comment is generated. The modifiers have the following meaning:

Modifier	Description
+	Adds the specified member or group of members to the set of members for which Javadoc comments are generated.
−	Removes the specified member or group of members from the set of members for which Javadoc comments are generated.

Table 45: Modifiers of the *javadoc* option.

The default value is *all+*.

4.2.12 The `visibility` option

The *visibility* option controls which visibility class members have. The *visibility* option has the following syntax:

```
Visibility := Item { Item }

Item := Member Modifier
      | MemberGroup Modifier

Modifier := "+" | "-" | "$" | "%"
```

The modifiers of the *visibility* option have the following meaning:

Modifier	**Visibility**
+	Public
-	Private
$	Protected
%	Package private

Table 46: Modifiers of the *visibility* option.

The default value is *"all- readerMethods+ stringMethods+ regionMethods+ dotMethods+ matchMethods+ scanMethods+"*.

4.2.13 The `internal` option

The *internal* option controls which methods have *private* as their default visibility and *Internal* as their name suffix in order to prevent name conflicts with other methods outside of the generated code. The *internal* option has the following syntax:

```
Internal := Item { Item }

Item := Member Modifier
      | MemberGroup Modifier

Modifier := "+" | "-"
```

The *internal* option specifies a set of members for which the internal feature should be active. The internal feature is only active if the corresponding member is part of the set, otherwise the internal feature is not active. The modifiers have the following meaning:

Modifier	Description
+	Adds the specified member or group of members to the set of members for which the internal feature should be active.
–	Removes the specified member or group of members from the set of members for which the internal feature should be active.

Table 47: Modifiers of the *internal* option.

The default value is *all-*.
A typical use case looks like in the following example:

```
/**
 * @option internal = setString+
 */
public class MyScanner {

    // custom method
    public void setString(String string) {
        setStringInternal(string);
        ...
    }

    // generated method
    private void setStringInternal(String string) {
        ...
    }
}
```

Using internal methods is similar to overriding methods in subclasses. The main difference is that both methods are part of the same class.

4.2.14 The `noMatchAction` option

The *noMatchAction* option specifies which action the scanner should perform if it could not find a match at the current dot. The *noMatchAction* option has the following syntax:

```
NoMatchAction := "error" | "continue" | "return"
```

If the specified value is *error* then a `java.lang.IllegalStateException` is thrown in order to indicate that the input is invalid. If the value is *continue* then the scanner moves the start position of the next scan one character forward and scans the input again. This is repeated until either a match could be found or the end of input is reached. If the *noMatchAction* option has the value *return* then the scanning stops and the *getNextToken* method returns without any action. If necessary a custom action can be performed outside of the scanner by checking for a *match length* of zero which can only happen in this case.

4.2.15 Examples

The following class comment contains examples for each option:

```
/**
 * @option logo = disabled
 * @option statistics = disabled
 * @option headings = small
 * @option methodName = yylex
 * @option methodThrows = {@link java.io.IOException}
 * @option defaultReturnValue = {@link Integer#MIN_VALUE}
 * @option inputMode = reader
 * @option bufferStrategy = allCharacters
 * @option bufferIncrement = double
 * @option functionality = all+
 * @option javadoc = all-
 * @option visibility = all- stringMethods+
 * @option internal = setString+
 * @option noMatchAction = continue
 */
```

5 Infrastructure

The following chapter gives an overview about technical details of AnnoFlex. It describes which steps are performed during a run of the code generator and which limitations and restrictions the code generation has. How the performance of AnnoFlex scales with increasing number of NFA and DFA states is shown in the last section.

5.1 The Generation Process

The following steps are performed in order to update the generated code of a scanner definition:

1. **Load file**: Loads the Java source code file which contains the scanner definition into memory.
2. **Parse file**: Parses the content of the Java file using an AnnoFlex-specific Java parser and reads in all AnnoFlex-specific processing instructions.
3. **Create NFAs**: Creates for each condition name and all corresponding regular expressions a NFA. The used algorithm is based on the Thompson's Construction Algorithm.
4. **Convert NFAs to DFAs**: Converts all NFAs into DFAs using the Powerset Construction Algorithm.
5. **Minimize DFAs**: All DFAs are minimized using Hopcroft's Minimization Algorithm.
6. **Generate code**: Computes the code of the scanner based on the minimized DFAs and the specified options. After the code has been computed it is integrated into the in-memory representation of the Java source code file.
7. **Write file**: Writes the updated Java source code file back to the file system.

5.2 Limitations and Restrictions

AnnoFlex has the following implementation-specific restrictions:

- The Unicode character support is limited to the first plane (Basic Multilingual Plane).
- The maximum number of lexical rules is limited to ~6400. The exact limit depends on the type of used regular expressions. It is caused by the 64k code limit of the class file format which can be reached by the action switch of the `getNextToken` method.
- Up to 2^31-1 NFA states are supported. The exact limit depends only on the amount of memory of the JRE.
- The maximum number of DFA states is limited to 32,767. This value is additionally limited by the 64k code limit of the class file format which can be reached by the string concatenation of the compressed DFA table data.

- The return type of methods of lexical rules is limited to primitive types and simple reference types. Array types and parameterized types (generics) are currently not supported.

5.3 Performance

The following table shows the memory consumption and processing time of AnnoFlex for the creation of various scanners. It shows how AnnoFlex scales with increasing number of NFA and DFA states.

Scanner Type	**# NFA States**	**# DFA States**	**Creation Time**	**Memory Consumption**
JavaScanner	565	304	~0.530 s	~14 MB
RegExScanner	740	413	~0.520 s	~14 MB
1 character dictionary	3	2	~0.320 s	~9 MB
10 words dictionary	78	65	~0.340 s	~9 MB
100 words dictionary	837	460	~0.390 s	~10 MB
1k words dictionary	8,532	2,617	~0.620 s	~22 MB
10k words dictionary	85,485	11,297	~1.6 s	~57 MB
20k words dictionary	170,590	18,343	~2.9 s	~124 MB
45k words dictionary	383,946	31,400	~4.5 s	~210 MB

Table 48: Scaling behavior of AnnoFlex for different scanner types.

JavaScanner is AnnoFlex's scanner for Java source code files. It can be found in the installation package under `source\org\annoflex\jdt\parser\JavaScanner.java`.

RegExScanner is AnnoFlex's scanner for regular expressions. It can be found in the installation package under `source\org\annoflex\regex\parser\RegExScanner.java`.

The *dictionary* scanners refer a synthetic scanner definition with one lexical rule of the form:

```
/**
 * @expr word1 | ... | wordn
 */
void createToken() {
}
```

The *creation time* value has been measured with the following PowerShell command:

```
Measure-Command {annoflex Scanner.java | Out-Default}
```

The *memory consumption* value has been determined with the Java VisualVM profiling tool which is part of the JDK. The specified value is the maximum of the "used heap" value of the "monitor" tab.

The following platform has been used to determine the values of the performance table.

- **CPU**: 2.6 GHz Intel Core 2 Duo Processor
- **RAM**: 8 GB DDR2 RAM
- **HDD**: 120 GB SATA II SSD
- **OS**: Microsoft Windows 8.1, 64-Bit
- **JRE**: Oracle Java SE 8u60, 64-Bit

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullmann. *Compilerbau.* Oldenbourg, 1999.

[2] Wilfried Brauer. *Automatentheorie.* Teubner, 1984.

[3] The Unicode Consortium. *The Unicode Character Property Model.* May 2015 (last access on 2016-11-18). URL: `http://unicode.org/reports/tr23/`.

[4] The Unicode Consortium. *Unicode Character Database.* June 2016 (last access on 2016-11-18). URL: `http://unicode.org/reports/tr44/`.

[5] The Unicode Consortium. *Unicode Regular Expressions.* Nov. 2013 (last access on 2016-03-20). URL: `http://unicode.org/reports/tr18/`.

[6] Russ Cox. *Regular Expression Matching Can Be Simple And Fast.* June 2007 (last access on 2016-03-20). URL: `https://swtch.com/~rsc/regexp/regexp1.html`.

[7] Jean Berstel, Luc Boasson, Olivier Carton and Isabelle Fagnot. *Minimization of Automata.* Jan. 2011 (last access on 2016-03-20). URL: `http://arxiv.org/pdf/1010.5318v3.pdf`.

[8] Gerwin Klein. *JFlex User‘s Manual.* Version 1.4.3. Jan. 2009.

[9] Alan M. Turing. “Computing Machinery and Intelligence.” In: *MIND* 59.236 (1950), pp. 433–460.