

JavaCC: TokenManager Mini Tutorial

The JavaCC lexical specification is organized into a set of "lexical states". Each lexical state is named with an identifier. There is a standard lexical state called **DEFAULT**. The generated token manager is at any moment in one of these lexical states. When the token manager is initialized, it starts off in the **DEFAULT** state, by default. The starting lexical state can also be specified as a parameter while constructing a token manager object.

Each lexical state contains an ordered list of regular expressions; the order is derived from the order of occurrence in the input file.

There are four kinds of regular expressions: **SKIP**, **MORE**, **TOKEN**, and **SPECIAL_TOKEN**.

All regular expressions that occur as expansion units in the grammar are considered to be in the **DEFAULT** lexical state and their order of occurrence is determined by their position in the grammar file.

A token is matched as follows: all regular expressions in the current lexical state are considered as **potential match candidates**. The token manager consumes the maximum number of characters from the input stream possible that match one of these regular expressions. That is, the token manager prefers the **longest possible match**. If there are multiple longest matches (of the same length), the regular expression that is matched is the one with the **earliest order of occurrence** in the grammar file.

As mentioned above, the token manager is in exactly one state at any moment. At this moment, the token manager only considers the regular expressions defined in this state for matching purposes.

After a match, one can specify an action to be executed as well as a new lexical state to move to. If a new lexical state is not specified, the token manager remains in the current state.

The regular expression kind specifies what to do when a regular expression has been successfully matched:

- **SKIP**: simply throw away the matched string (after executing any lexical action).
- **MORE**: continue (to whatever the next state is) taking the matched string along. This string will be a prefix of the new matched string.
- **TOKEN**: create a token using the matched string and send it to the parser (or any caller).
- **SPECIAL_TOKEN**: Special tokens are like tokens, except that they do not have significance during parsing - that is the BNF productions ignore them. Special tokens are, however, still passed on to the parser so that parser actions can access them. Special tokens are passed to the parser by linking them to neighbouring real tokens using the field "specialToken" in the Token class. Special tokens are useful in the processing of lexical entities **such as comments** which have no significance to parsing, but still are an important part of the input file.

If an <EOF> is detected in the middle of a match for a regular expression, or immediately after a MORE regular expression has been matched, an error is reported.

After the regular expression is matched, the lexical action is executed. During the execution of a lexical action, all the variables (and methods) declared in the **TOKEN_MGR_DECLS** region (see below) are available here for use. In addition, the variables and methods listed below can also be used.

Immediately after the execution of lexical actions, the token manager changes state to that specified (if any) and performs the action specified by the kind

of the regular expression (SKIP, MORE, ...). If the kind is TOKEN, the matched token is returned. If the kind is SPECIAL_TOKEN, the matched token is saved to be returned along with the next TOKEN that is matched.

The following variables are available for use within lexical actions:

(1) StringBuffer image (READ/WRITE): (different from the "image" field of the matched token) is a StringBuffer variable that contains all the characters that have been matched since the last SKIP, TOKEN, or SPECIAL_TOKEN. You are free to make whatever changes you wish to it **so long as you do not assign it to null** (since this variable is used by the generated token manager also). If you make changes to "image", this change is passed on to subsequent matches (if the current match is a MORE).

The content of "image" DOES NOT automatically get assigned to the "image" field of the matched token. If you wish this to happen, you must explicitly assign it in a lexical action of a TOKEN or SPECIAL_TOKEN regular expression.

Example:

<DEFAULT>

MORE : { "a" : S1 }

<S1>

MORE :

{

 "b"

 {

 int l = image.length()-1;

 image.setCharAt(l, image.charAt(l).toUpperCase());

^{^1}

^{^2}

 } : S2

}

<S2> TOKEN :

{

 "cd" { x = image; } : DEFAULT

^{^3}

}

The value of "image" at the 3 points marked by (1, (2), and (3) are:

- At 1: "ab"
- At 2: "aB"
- At 3: "aBcd"

2. `int lengthOfMatch` (READ ONLY):

This is the length of the current match (is not cumulative over MORE's).

See example below. You should not modify this variable.

Example: using the same example as above, the values of "lengthOfMatch" are:

- At ^1: 1 (the size of "b")
- At ^2: 1 (does not change due to lexical actions)
- At ^3: 2 (the size of "cd")

3. `int curLexState` (READ ONLY):

This is the index of the current lexical state. You should not modify this variable. Integer constants whose names are those of the lexical state are generated into the ...Constants file, so you can refer to lexical states without worrying about their actual index value.

4. `InputStream` (READ ONLY):

This is the input stream of an appropriate type. The stream is currently at the last character consumed for this match. Methods of `InputStream` can be called. For example, `getEndLine` and `getEndColumn` can be called to get the line and column number information for the current match. `InputStream` may not be modified.

5. `Token matchedToken` (READ/WRITE):

This variable may be used only in actions associated with `TOKEN` and `SPECIAL_TOKEN` regular expressions. This is set to be the token that will be returned to the parser. You may change this variable and thereby cause the changed token to be returned to the parser instead of the original one. It is here that you can assign the value of variable "image" to

"matchedToken.image". Typically that's how your changes to "image" has effect outside the lexical actions.

Example: if we modify the last regular expression specification of the above example to:

```
<S2> TOKEN :  
{  
  "cd" { matchedToken.image = image.toString(); } : DEFAULT  
}
```

Then the token returned to the parser will have its ".image" field set to "aBcd". If this assignment was not performed, then the ".image" field will remain as "abcd".

6. void SwitchTo(int):

Calling this method switches you to the specified lexical state. This method may be called from parser actions also (in addition to being called from lexical actions). However, **care must be taken when using this method** to switch states from the parser since the lexical analysis could be many tokens ahead of the parser in the presence of large lookaheads. When you use this method within a lexical action, you must ensure that it is the last statement executed in the action (otherwise, strange things could happen). If there is a state change specified using the **" : state "** syntax, it overrides all switchTo calls, hence there is no point having a switchTo call when there is an explicit state change specified. In general, calling this method should be resorted to only when you cannot do it any other way. Using this method of switching states also causes you to lose some of the semantic checking that JavaCC does when you use the standard syntax.

Lexical actions have access to a set of class level declarations. These

declarations are introduced using the following syntax:

```
token_manager_decls ::=  
    "TOKEN_MGR_DECLS" ":"  
    "{" java_declarations_and_code "}"
```

These declarations are accessible from all lexical actions.

Example 1: Comments

```
SKIP :  
{  
    "/*" : WithinComment  
}
```

<WithinComment>

```
SKIP :  
{  
    "*/" : DEFAULT  
}
```

<WithinComment>

MORE :

```
{  
    <~[]>  
}
```

Example 2: String Literals with actions to print the length of the string:

```
TOKEN_MGR_DECLS :  
{  
    int stringSize;  
}
```

MORE :

```
{  
  "\\" {stringSize = 0;} : WithinString  
}
```

<WithinString>

TOKEN :

```
{  
  <STRLIT: "\\"> {System.out.println("Size = " + stringSize);} : DEFAULT  
}
```

<WithinString>

MORE :

```
{  
  <~["\n","\r"]> {stringSize++;}  
}
```


HOW SPECIAL TOKENS ARE SENT TO THE PARSER:

Special tokens are like tokens, except that they are permitted to appear anywhere in the input file (between any two tokens). Special tokens can be specified in the grammar input file using the reserved word "SPECIAL_TOKEN" instead of "TOKEN" as in:

SPECIAL_TOKEN :

```
{  
  <SINGLE_LINE_COMMENT: "//" (~["\n","\r"])* ("\n"|"r"|"r\n")>  
}
```

Any regular expression defined to be a SPECIAL_TOKEN may be accessed in a special manner from user actions in the lexical and grammar specifications. This allows these tokens to be recovered during parsing while at the same time these tokens do not participate in the parsing.

JavaCC has been bootstrapped to use this feature to automatically copy relevant comments from the input grammar file into the generated files.

Details: the class Token now has an additional field:

Token specialToken;

This field points to the special token immediately prior to the current token (special or otherwise). If the token immediately prior to the current token is a regular token (and not a special token), then this field is set to null. The "next" fields of regular tokens continue to have the same meaning - i.e., they point to the next regular token except in the case of the EOF token where the "next" field is null. The "next" field of special tokens point to the special token immediately following the current token. If the token immediately following the current token is a regular token, the "next" field is set to null.

This is clarified by the following example. Suppose you wish to print all special tokens prior to the regular token "t" (but only those that are after the regular token before "t"):

```
if (t.specialToken == null) return;
// The above statement determines that there are no special tokens
// and returns control to the caller.
Token tmp_t = t.specialToken;
while (tmp_t.specialToken != null) tmp_t = tmp_t.specialToken;
// The above line walks back the special token chain until it
// reaches the first special token after the previous regular
// token.
while (tmp_t != null) {
    System.out.println(tmp_t.image);
    tmp_t = tmp_t.next;
}
// The above loop now walks the special token chain in the forward
// direction printing them in the process.
```