

JavaCC: LOOKAHEAD MiniTutorial

1. WHAT IS LOOKAHEAD

The job of a parser is to read an input stream and determine whether or not the input stream conforms to the grammar. This determination in its most general form can be quite time consuming. Consider the following example (Example1.jj)

```
void Input() : {}
{
    "a" BC() "c"
}

void BC() : {}
{
    "b" [ "c" ]
}
```

In this simple example, it is quite clear that there are exactly two strings that match the above grammar, namely: `abc` and `abcc`

The general way to perform this match is *to walk through the grammar based on the string* as follows. Here, we use abc as the input string:

Step 1. There is only one choice here: the first input character must be 'a', and since that is indeed the case, we are ok.

Step 2. We now proceed on to non-terminal BC. Here again, there is only one choice for the next input character, it must be 'b'. The input matches this one too, so we are still ok.

Step 3. We now come to a **choice point** in the grammar. We can either go inside the [...] and match it, or ignore it altogether. *We decide to go inside.* So the next input character must be a 'c'. We are again ok.

Step 4. Now we have completed with non-terminal BC and go back to non-terminal Input. Now the grammar says the next character must be yet another 'c'. But there are no more input characters. *So we have a problem.*

Step 5. When we have such a problem in the general case, we conclude *that we may have made a bad choice somewhere*. In this case, we made the bad choice in **Step 3**. So we retrace our steps back to step 3 and make another choice and try that. This process is called backtracking.

Step 6. We have now backtracked and made the other choice we could have made at **Step 3**, namely, ignore the [...]. Now we have completed with non-terminal BC and go back to non-terminal Input. Now the grammar says the next character must be yet another 'c'. The next input character is a 'c', so we are ok now.

Step 7. We realize we have reached the end of the grammar (end of non-terminal Input) successfully. This means we have successfully matched the string abc to the grammar.

As the above example indicates, *the general problem of matching an input with a grammar may result in large amounts of backtracking and making new choices and this can consume a lot of time.* The amount of time taken can also be a function of how the grammar is written. Note that many grammars can be written to cover the same set of inputs – or the same language (i.e., there can be multiple equivalent grammars for the same input language).

For example, *the following grammar would speed up the parsing of the same language as compared to the previous grammar:*

```
void Input() :
{
{
    "a" "b" "c" [ "c" ]
}
}
```

while the following grammar slows it down even more since the parser has to backtrack all the way to the beginning:

```
void Input() :
{
{
    "a" "b" "c" "c"
    |
    "a" "b" "c"
}
}
```

One can even have a grammar that looks like the following:

```
void Input() :
{
{
    "a" ( BC1() | BC2() )
}

void BC1() :
{
{
    "b" "c" "c"
}

void BC2() :
{
{
    "b" "c" [ "c" ]
}
}
```

This grammar can match abcc in two ways, and is therefore considered *ambiguous*.

The performance hit from such backtracking is unacceptable for most systems that include a parser. Hence most parsers do not backtrack in this general manner (or do not backtrack at all), rather they make decisions at choice points based on limited information and then commit to it.

Parsers generated by JavaCC make decisions at choice points based on some exploration of tokens further ahead in the input stream, and once they make such a decision, they commit to it. That is, no backtracking is performed once a decision is made.

The process of exploring tokens further in the input stream is termed "looking ahead" into the input stream, hence our use of the term LOOKAHEAD.

Since some of these decisions may be made with less than perfect information (JavaCC will warn you in these situations, so you don't have to worry) you need to know something about LOOKAHEAD to make your grammar work correctly. *The two ways in which you make the choice decisions work properly are:*

- Modify the grammar to make it simpler.
- Insert hints at the more complicated choice points to help the parser make the right choices.

2. CHOICE POINTS IN JAVACC GRAMMARS

There are 4 different kinds of **choice points** in JavaCC:

1. An expansion of the form: (exp1 | exp2 | ...). In this case, the generated parser has to somehow determine which of exp1, exp2, etc. to select to continue parsing.
2. An expansion of the form: (exp)?. In this case, the generated parser must somehow determine whether to choose exp or to continue beyond the (exp)? without choosing exp.
3. An expansion of the form (exp)*. In this case, the generated parser must do the same thing as in the previous case, and furthermore, after each time a successful match of exp (if exp was chosen) is completed, this choice determination must be made again.
4. An expansion of the form (exp)+. This is essentially similar to the previous case with a mandatory first match to exp.

Remember that token specifications that occur within angular brackets <...> also have choice points. But these choices are made in different ways and are the subject of a different tutorial.

3. THE DEFAULT CHOICE DETERMINATION ALGORITHM

The default choice determination algorithm looks ahead 1 token in the input stream and uses this to help make its choice at choice points.

The following examples will describe the default algorithm fully. Consider the following grammar (Example2.jj):

```
void basic_expr() : {}
{
    <ID> "(" expr() ")"    /* Choice 1 */
    |
    "(" expr() ")"        /* Choice 2 */
    |
    "new" <ID>             /* Choice 3 */
}
```

The choice determination algorithm works as follows:

```
if (next token is <ID>) {
    choose Choice 1
}
else if (next token is "(") {
    choose Choice 2
}
else if (next token is "new") {
    choose Choice 3
}
else {
    produce an error message
}
```

In the above example, the grammar has been written such that the default choice determination algorithm does the right thing. Another thing to note is that the choice determination algorithm works in a top to bottom order, i.e. if Choice 1 was selected, the other choices are not even considered. While this is not an issue in this example (except for performance), it will become important later below when local ambiguities require the insertion of LOOKAHEAD hints. Suppose the above grammar was modified to (Example3.jj):

```
void basic_expr() : {}
{
    <ID> "(" expr() ")"    /* Choice 1 */
    |
    "(" expr() ")"        /* Choice 2 */
    |
    "new" <ID>             /* Choice 3 */
    |
    <ID> "." <ID>          /* Choice 4 */
}
```

Then the default algorithm will always choose Choice 1 when the next input token is <ID> and never choose Choice 4 even if the token <ID> is followed by a "." (a dot). You can try running the parser generated from the example above on the input id1.id2. It will complain

that it encountered a "." when it was expecting a "(" . Notice that, when you built the parser, it would have given you the following warning message:

Warning: Choice conflict involving two expansions at line 25, column 3 and line 31, column 3 respectively. A common prefix is: <ID>.
Consider using a lookahead of 2 for earlier expansion.

Essentially, JavaCC is saying it has detected a situation in your grammar which may cause the default lookahead algorithm to do strange things. The generated parser will still work using the default lookahead algorithm, except that it may not do what you expect of it. More on this later. Now consider the following example:

```
void identifier_list() : {}  
{  
    <ID> ( "," <ID> ) *  
}
```

Suppose the first <ID> has already been matched and that the parser has reached the choice point (the ("," <ID>) * construct). Here's how the choice determination algorithm works:

```
while (next token is ",") {  
    choose the nested expansion (i.e., go into the (...)* construct)  
    consume the "," token  
    if (next token is <ID>) consume it,  
    otherwise report error  
}
```

In the above example, note that the choice determination algorithm does not look beyond the (...)* construct to make its decision. Suppose there was another production in that same grammar as follows (Example5.jj):

```
void funny_list() : {}  
{  
    identifier_list() "," <INT>  
}
```

When the default algorithm is making a choice at ("," <ID>) *, it will always go into the (...)* construct if the next token is a "," . It will do this even when identifier_list is called from funny_list and the token after the "," is an <INT> . Intuitively, the right thing to do in this situation is to skip the (...)* construct and return to funny_list. More on this later.

As a concrete example, suppose your input was "id1, id2, 5", the parser will complain that it encountered a 5 when it was expecting an <ID>. Notice that when you built the parser, it would have given you the following warning message:

Warning: Choice conflict in (...)* construct at line 25, column 8.
Expansion nested within construct and expansion following construct
have common prefixes, one of which is: ","
Consider using a lookahead of 2 or more for nested expansion.

Again, JavaCC is saying that it has detected a situation in your grammar which may cause the default lookahead algorithm to do strange things. As above, the generated parser will still work using the default lookahead algorithm, except that it may not do what you expect of it.

We have shown you examples of two kinds of choice points in the examples above (1) "exp1 | exp2 | ...", and (2) "(exp)*". The other two kinds of choice points, (3) "(exp)+" and "(exp)?" - behave similarly to (exp)* and we will not be providing examples of their use here.

4. MULTIPLE TOKEN LOOKAHEAD SPECIFICATIONS

So far, we have described the default lookahead algorithm of the generated parsers. *In the majority of situations, the default algorithm works just fine.* In all situations where it does not work well, JavaCC provides you with warning messages like the ones shown above. *If you have a grammar that goes through JavaCC without producing any warnings, then the grammar is a LL(1) grammar.* Essentially, LL(1) grammars are those that can be handled by top-down parsers (such as those generated by JavaCC) using at most one token of LOOKAHEAD. When you get these warning messages, you can do one of two things.

Option 1: you can modify your grammar so that the warning messages go away. That is, *you can attempt to make your grammar LL(1) by making some changes to it.* The following (file Example6.jj) shows how you may change Example3.jj to make it LL(1):

```
void basic_expr() : {}
{
    <ID> ( "(" expr() ")" | "." <ID> )
    |
    "(" expr() ")"
    |
    "new" <ID>
}
```

What we have done here is to factor the fourth choice into the first choice. Note how we have placed their common first token <ID> outside the parentheses, and then within the parentheses, we have yet another choice which can now be performed by looking at only one token in the input stream and comparing it with "(" and ".". This process of modifying grammars to make them LL(1) is called **left factoring**.

The following (file Example7.jj) shows how Example5.jj may be changed to make it LL(1):

```
void funny_list() : {}
{
    <ID> ", " ( <ID> ", " ) * <INT>
}
```

Note that this change is somewhat more drastic.

Option 2: You can provide the generated parser with some hints that help it out in all the non-LL(1) situations that the warning messages bring to your attention.

All such hints are specified using either (1) setting the global LOOKAHEAD value to a larger value (see below) or (2) by using the LOOKAHEAD(. . .) construct to provide a local hint.

A design decision must be made to determine if **Option 1** or **Option 2** is the right one to take. The only advantage of choosing **Option 1** is that it makes your grammar perform better. JavaCC generated parsers can handle LL(1) constructs much faster than other constructs. However, the advantage of choosing **Option 2** is that you have a simpler grammar, one that is easier to develop and maintain, one that focuses on human-friendliness and not machine-friendliness.

Sometimes **Option 2** is the only choice, especially in the presence of user actions. Suppose Example3.jj contained actions as shown below:

```
void basic_expr() : {}
{
    { initMethodTables(); } <ID> "(" expr() ")"
    |
    "(" expr() ")"
    |
    "new" <ID>
    |
    { initObjectTables(); } <ID> "." <ID>
}
```

Since the actions are different, left-factoring cannot be performed.

4.1 SETTING A GLOBAL LOOKAHEAD SPECIFICATION

You can set a global LOOKAHEAD specification by using the option LOOKAHEAD either from the command line, or at the beginning of the grammar file in the options section. *The value of this option is an integer which is the number of tokens to look ahead when making choice decisions.* As you may have guessed, the default value of this option is 1, which derives the default LOOKAHEAD algorithm described above.

Suppose you set the value of this option to 2. Then the LOOKAHEAD algorithm derived from this looks at two tokens (instead of just one token) before making a choice decision. Hence, in Example3.jj, Choice 1 will be taken only if the next two tokens are <ID> and "(", while Choice 4 will be taken only if the next two tokens are <ID> and ".". Hence, the parser will now work properly for Example3.jj. Similarly, the problem with Example5.jj also goes away since the parser goes into the (. . .) * construct only when the next two tokens are ", " and <ID>.

By setting the global LOOKAHEAD to 2, the parsing algorithm essentially becomes LL(2).

Since you can set the global LOOKAHEAD to any value, parsers generated by JavaCC are called LL(k) parsers.

4.2. SETTING A LOCAL LOOKAHEAD SPECIFICATION

You can also set a local LOOKAHEAD specification that affects only a specific choice point. This way, the majority of the grammar can remain LL(1) and hence perform better, while at the same time one gets the flexibility of LL(k) grammars. Here's how Example3.jj is modified with local LOOKAHEAD to fix the choice ambiguity problem (file Example8.jj):

```
void basic_expr() : {}
{
    LOOKAHEAD(2)
    <ID> "(" expr() ")" /* Choice 1 */
    |
    "(" expr() ")"      /* Choice 2 */
    |
    "new" <ID>           /* Choice 3 */
    |
    <ID> "." <ID>        /* Choice 4 */
}
```

Only the first choice (the first condition in the translation below) is affected by the LOOKAHEAD specification. All others continue to use a single token of LOOKAHEAD:

```
if (next 2 tokens are <ID> and "(" ) {
    choose Choice 1
}
else if (next token is "(") {
    choose Choice 2
}
else if (next token is "new") {
    choose Choice 3
}
else if (next token is <ID>) {
    choose Choice 4
}
else {
    produce an error message
}
```

Similarly, Example5.jj can be modified as shown below (file Example9.jj):

```
void identifier_list() :      {}
{
    <ID> ( LOOKAHEAD(2) "," <ID> ) *
}
```

Note, the LOOKAHEAD specification has to occur inside the (...) * which is the choice is being made. The translation for this construct is shown below (after the first <ID> has been consumed):


```

while (next 2 tokens are "," and <ID>) {
    choose the nested expansion (i.e., go into the (...)* construct)
    consume the "," token
    consume the <ID> token
}

```

We strongly discourage you from modifying the global LOOKAHEAD default. Most grammars are predominantly LL(1), hence you will be unnecessarily degrading performance by converting the entire grammar to LL(k) to facilitate just some portions of the grammar that are not LL(1). If your grammar and input files being parsed are very small, then this is okay.

You should also keep in mind that the warning messages JavaCC prints when it detects ambiguities at choice points (such as the two messages shown earlier) simply tells you that the specified choice points are not LL(1). JavaCC does not verify the correctness of your local LOOKAHEAD specification, it assumes you know what you are doing, in fact, it really cannot verify the correctness of local LOOKAHEAD's as the following example of if statements illustrates (file Example10.jj):

```

void IfStm() : {}
{
    "if" C() S() [ "else" S() ]
}

void S() : {}
{
    ...    | IfStm()
}

```

This example is the famous **dangling else** problem. If you have a program that looks like:

```

if C1 if C2 S1 else S2

```

The else S2 can be bound to either of the two if statements. The standard interpretation is that it is bound to the inner if statement (the one closest to it). The default choice determination algorithm happens to do the right thing, but it still prints the following warning message:

```

Warning: Choice conflict in [...] construct at line 25, column 15.
Expansion nested within construct and expansion following construct
have common prefixes, one of which is: "else"
Consider using a lookahead of 2 or more for nested expansion.

```

To suppress the warning message, you could simply tell JavaCC that you know what you are doing as follows:

```

void IfStm() : {}
{
    "if" C() S() [ LOOKAHEAD(1) "else" S() ]
}

```

To force lookahead ambiguity checking in such instances, set the option `FORCE_LA_CHECK` to `true`.