# Adding preprocessor directives and comments to syntax trees

Andreas Bartho [1]

*Department of Computer Science*
*Technische Universität Dresden*
*Dresden, Germany*

**Abstract**

Parsers usually produce trees well suited for further machine processing. They do not contain preprocessor directives and usually also no comments. However, when the tree representation of source code is created for humans, e.g. for a code outline of an IDE, having comments and preprocessor directives would be beneficial for navigation and orientation. Unfortunately, combining them with the syntax tree is not easy. For comments there are usually many potential nodes to which they could belong. Preprocessor directives are also hard to include, because they can appear in positions where it is impossible to add them to the syntax tree of the source code. This paper presents an algorithm that tackles both problems. The problem of comment association is alleviated by the possibility to provide a configuration stating to which type of node a comment should be added. The problem of preprocessor directive incompatibilites is solved by intelligently splitting nodes of the syntax tree.

*Key words:* AST, Comments, Parser, Preprocessor, Source Code

## 1 Introduction

In the field of language translation source code is transformed into a syntax tree and then processed further. The syntax tree does usually not contain all information that were present in the original source code. Some tokens such as semicolons or parentheses are left out, preprocessor directives are interpreted before the actual parsing and are not part of the syntax tree. Sometimes, e.g. in `#IF` directives in C#, whole code sections are omitted during the parsing process. Comments are often ignored completely.

This approach is targeted at interpreting or compiling a program. In the area of code visualization, however, one would often like to have a much more

[1] Email: `andreas.bartho@tu-dresden.de`

*This paper is electronically published in*
*Electronic Notes in Theoretical Computer Science*
*URL:* `www.elsevier.nl/locate/entcs`

detailed syntax tree, including comments and preprocessor directives (from now on only called directives). Such a tree would be very useful in a code exploration tool, e.g. within a programming IDE. Comments and some directives can be used to structure the code, so adding them to a tree representation would be beneficial [2]. However, computing such a complete syntax tree is not easy.

One way could be to extend the parser. Unfortunately, comments are usually allowed between any two tokens, so the underlying grammar becomes both bloated and possibly ambiguous. Adding the directives is even more difficult. There are also cases where it is not feasible or possible to create a new parser and an existing one must be reused.

An alternative is to parse the code file multiple times with different special-purpose parsers and combine the resulting syntax trees of the normal code, the comments, and the directives.

In the following sections the problems arising with the second approach are discussed, along with their solutions. An algorithm implementing these solutions is also presented. The algorithm was developed for the tutorial development tool DEFT [3]. DEFT allows writing and maintaining programming tutorials. Source code fragments are added to the tutorial by dragging and dropping a node from a code outline. Therefore the tree in that outline must be very accurate and detailed.

The introduction of the algorithm is split into two parts. The first part in section 2 considers only the simpler case that nodes from a foreign tree are added to a base tree, as it is the case with comments. In section 3, when the reader already has some basic understanding of the algorithm, the "extended version" is presented, in which also nodes from the base tree can be added to the foreign tree. This is necessary for directives enclosing source code. Section 4 discusses related work and finally section 5 gives a summary and makes suggestions for future enhancements.

## 2 Combining syntax tree and comments

For two trees to be combined to one, some prerequisites must be met. Both trees should contain all tokens and must for each token store the start and end position (offsets). If not all tokens are present in the tree, at least the non-terminal nodes must contain accurate offsets.

Additionally, tree nodes should be ordered, that is, the leaf nodes of the syntax trees should contain the tokens in the same order as they appeared in the input file. While the ordering is optional it makes the tree combining algorithm more efficient. The algorithm presented in this paper relies on the

[2] Macro definitions, as they can be used in C, are excluded from the discussion in this paper.
[3] http://deftproject.org

tree containing all tokens and the nodes to be ordered.

### *2.1 Finding locations for comment nodes*

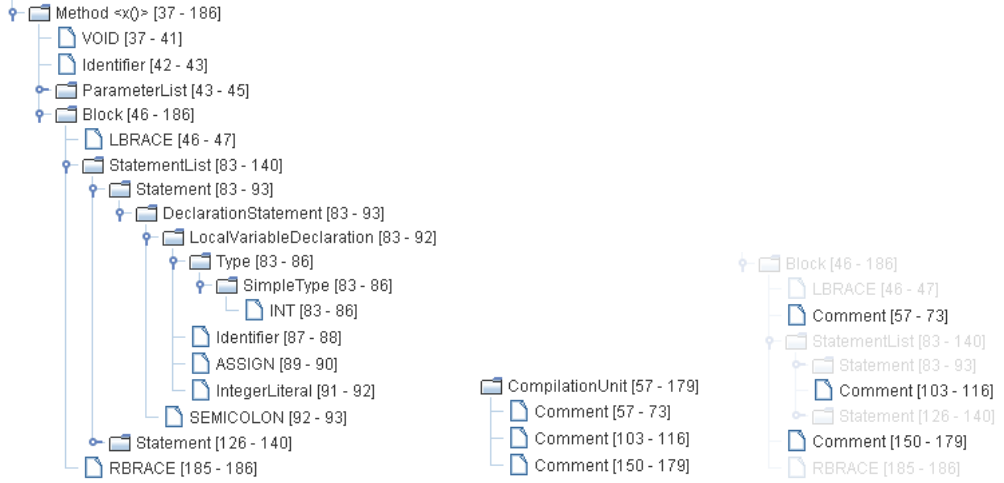
The simplest kind of tree combination is the addition of comments to a syntax tree. A tree of source code comments consists only of comment nodes that share the same parent (the root node). So it is effectively only a list. To add the individual comments to the source code tree, their new parent node must be identified. The new parent node has to embrace the start and end offset of the comment node. Furthermore, it must not have children that embrace the comment node. After such a node is found it is examined at which index in its child list the comment should be added, and finally the comment is added.

Sometimes, however, it is not clear what the new parent of a comment node should be. Consider the example from figure 1.

```
3     void x() {
4         //initializing i
5         int i = 0;
6         //computing j
7         int j = i + 1;
8         //j has finally been computed
9     }
```

Fig. 1. Method with comments

Parsing the code leads to the trees shown in figures 2a and 2b. The numbers in brackets denote the nodes' start and end offsets.




(a) AST of the Code (b) AST of the Comments (c) Naively combined AST

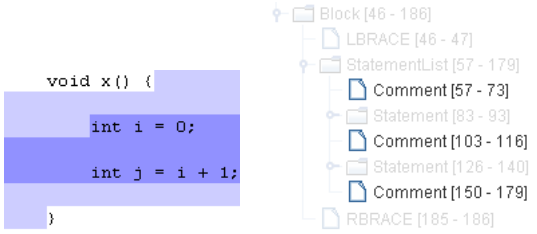
Fig. 2. Naive AST combining

To add the comments to the code AST, they must be inserted into the right "gaps", i.e. added between two nodes where the first node's end offset is

less or equal to the comment's start offset and the second node's start offset is greater or equal to the comment's end offset. This is shown in figure 2c.

Unfortunately, the first and the last comment are added to the `Block` node, as siblings of the `StatementList`. This was clearly not the intention. The comments were expected to be on the same hierarchy level as the second comment: as siblings of the `Statements`[4]. Upon closer examination of figure 2a it becomes clear that there are actually multiple possibilities to add the comments to the AST. The first comment, for example, can be a child of `Block` (after `LBRACE`), `StatementList`, `Statement`, and all the way down to `SimpleType`. There is no way to safely identify the destination of the comments to be added.

The solution is to make the tree combining algorithm configurable. It is necessary to specify that inserting nodes before and after `StatementList` is prohibited. On a technical level that means that `StatementList`'s range should not only span the range of its child nodes but also the preceding and following whitespace. That is, the `StatementList` should be treated as if it starts directly after the opening brace, at offset 47. Similarly, the end offset should be assumed 185, directly before the closing brace. These offsets will subsequently be called extended offsets. Figure 3a shows the source code with both normal offsets (dark) and extended offsets (light) of `StatementList` highlighted.




(a) Normal and Extended Offsets (b) Correctly combined AST

Fig. 3. AST combining with extended offsets

With the `StatementList` configured as described above, the combination algorithm can neither add the first comment between `LBRACE` and `StatementList` nor the third comment between `StatementList` and `RBRACE`. Instead, they are added one hierarchy level deeper as siblings of the `Statements`, as can be seen in figure 3b. Note that the displayed offsets are not the extended offsets. Those are only used while the combination algorithm runs and discarded afterwards.

Sometimes it is desirable to allow nodes only be added before some node but not after (or vice versa). Consider, for example, Javadoc comments for methods. A comment above a method usually belongs to that method. A com-

[4] For a discussion of how to associate comments to specific statements please see sections 4 and 5.

ment below, however, belongs most likely to the following method. Therefore the configuration in general consists of two parts: a set of node names that do not allow node insertion before them, and a set of node names that do not allow node insertion after them.

### *2.2 Simple algorithm for comment adding*

In the following an implementation of the previously explained behavior will be shown.

#### Procedure combineTrees

```
   Input    : baseTree: the tree to which the nodes of a second tree should be added
   Input    : foreignTree: tree whose nodes (the comments) should be added to baseTree
1  foreach TreeNode tn ∈ foreignTree.getChildren() do
2      tn.removeFromParent()
       // find node in baseTree to which to add tn
3      TreeNode newParent ← findSpanningNode(baseTree, tn)
4      int childIdx ← findIndex(newParent, tn)
       // add tn
5      newParent.addChild(childIdx, tn)
6  end
```

**Algorithm 1**: combineTrees(baseTree, foreignTree)

Procedure `combineTrees` shown in algorithm 1 is the starting point of the algorithm. For all the comment nodes from `foreignTree` an appropriate new parent node in the `baseTree` is searched with the help of function `findSpanningNode`. This new parent node will already have some children, so it must be examined, at which index to add the comment node. Once this is known, the comment node can be added to its new parent.

#### Function findSpanningNode

The most interesting part of the algorithm is finding the correct parent for the nodes from the `foreignTree`. This is done by function `findSpanningNode`, presented in algorithm 2.

The function implements a depth-first search for the deepest node in a tree which still spans some other node. For each child of `root` it is checked whether its start and end offsets embrace the start and end offset of `foreignNode`. If such an embracing node is found, the function is called recursively with that node. Eventually there will be no more embracing nodes, i.e. `foreignNode` fits somewhere between `root`'s children.

But as explained above, merely using the nodes' offsets for finding a new parent for a comment is not enough. Sometimes it is necessary to use the extended offset instead. This choice is delegated to `determineStartOffset` and `determineEndOffset` (see algorithm 3). Note that the check for `tnEnd`

```
   Input    : root: TreeNode, from which to recursively search a descendant TreeNode
              that still embraces foreignNode
   Input    : foreignNode: TreeNode for which to find the deepest embracing node under
              root
   Result   : root or the deepest descendant node of root that embraces foreignNode
 1 int fitFrom ← foreignNode.getStartOffset()
 2 int fitTo ← foreignNode.getEndOffset()
 3 foreach TreeNode tn ∈ root.getChildren() do
 4     int tnStart ← determineStartOffset(tn)
 5     int tnEnd ← determineEndOffset(tn)
       // if tn spans fitFrom and fitTo, check its descendants
 6     if tnStart ≤ fitFrom ∧ (tnEnd ≥ fitTo ∨ tnEnd = ∞) then
 7         return findSpanningNode(tn, foreignNode)
 8     end
 9 end
   // No child was found, so root is the node we were looking for
10 return root
```

**Algorithm 2**: findSpanningNode(root, foreignNode)

= $\infty$ has to do with the computation of the extended end offset. Details will be presented below.

### Function determineEndOffset

Function `determineEndOffset` returns for a node its end offset or its extended end offset, depending on the configuration.

If it is not forbidden to add other nodes after `node`, the normal end offset is returned. Otherwise the extended end offset must be computed and returned. That requires finding the token (leaf node) which follows `node`. Its start offset is the same as the extended end offset of `node`. The example code in algorithm 3 does a brute-force search over all tokens. While there are more efficient ways, this approach has not shown to be a bottleneck even for rather large trees. If `node` contains the last token, i.e. there is no token node after `node`, it is not possible to return a definite end offset. There might be comments at the end of the file and we do not know how long they are. Therefore we return infinity. This is also the reason for the check for infinity in function `findSpanningNode`.

Function `determineStartOffset` works similarly. The extended start offset of a node is the end offset of the preceding token. The extended start offset of a node which contains the first token is 0.

## 3 Combining arbitrary trees

While adding comments to a syntax tree is not difficult, things become harder for directives. In contrast to comments they can contain code themselves, which in turn can again contain directives, and so on. To make matters even worse, it is possible to write directives so that they "cut through" the code,

```
   Input    : node: TreeNode for which to determine the end offset
   Data     : tokenNodes: sorted array of all token nodes (leaves) from the base tree
   Data     : config: configuration describing when to use the extended end offset
   Result   : the end offset or the extended end offset of node, depending on config
 1 int endOffset ← node.getEndOffset()
   // config allows insertion after node?→return normal end offset
 2 if ¬config.isInsertAfterForbidden(node.getName()) then
 3     return endOffset
 4 end
   // otherwise compute extended end offset
 5 for int i ← 0; i < tokenNodes.length; i++ do
 6     TokenNode tn ← tokenNodes[i]
 7     int start ← tn.getStartOffset()
       // find first TokenNode after node and return its start offset
 8     if start ≥ endOffset then
 9         return start
10     end
       // node spans last token:  ext end offset is end of file
11     if i = tokenNodes.length - 1 then
12         return ∞
13     end
14 end
   // this is never executed
15 return ERROR
```

**Algorithm 3**: determineEndOffset(node)

i.e. it becomes impossible to combine the directives tree with the syntax tree.

### *3.1 Intersecting directives*

```
1 class Test {
2 #region A
3 }
4 #endregion
```

Fig. 4. Directive cutting Class body

The problem of directives intersecting code is illustrated in figure 4. The class body comes first, then starts the `#region` directive[5], then the body ends and finally the directive ends. After parsing and tree construction, the `Body` node of the C# tree and the `Region` node of the preprocessor tree would overlap. We say that such overlapping nodes are incompatible. In order for the preprocessor tree to be combined with the code tree, the incompatibility must be resolved. One way is to define that the node starting later (i.e. the node with the greater start offset) is always a descendant of the node

[5] The `#region` directive is used to mark a section of code, e.g. for code folding.[2]

starting earlier, even if the order of the leaf nodes in the resulting tree does not accurately mirror the source code. In the context of DEFT this was not possible, so another way has been chosen: one of the incompatible nodes must be split. In the presented algorithm the node starting earlier is left untouched, while the node starting later will be split. In the example the directive node would be split.

It follows a detailed explanation of the algorithm. After that there is a step-by-step example for a better understanding.

### *3.2 Complete tree combining algorithm*

#### Procedure combineTrees

The main loop of the algorithm is similar to that of the comment adding algorithm. For all children of the foreign tree a parent node in the base tree is searched. Then that child is inserted under the new parent.

```
   Input    : baseTree: the tree to which the nodes of a second tree should be added
   Input    : foreignTree: tree whose nodes should be added to baseTree
   // for all children of foreignTree's root node
1  foreach TreeNode tn ∈ foreignTree.getChildren() do
2      tn.removeFromParent()
       // find node in baseTree to which to add tn
3      TreeNode newParent ← findSpanningNode(baseTree, tn)
       // add tn and process recursively if necessary
4      insert(newParent, tn)
5  end
```

**Algorithm 4**: combineTrees(baseTree, foreignTree)

#### Function insert

Function `insert` is the main part of the algorithm. It handles insertion of compatible nodes and controls splitting of incompatible nodes. First it is checked if the child to be inserted is compatible with all its future sibling nodes (i.e. with the children of its new parent). If so, it can safely be added to its new parent. However, it is possible that some of the nodes will be displaced by the new child (e.g. methods being displaced by a `#region` directive). Those nodes must be identified, then the child is added to its new parent node at the correct index and the displaced siblings are removed and recursively inserted somewhere under the new child (e.g. the abovementioned methods are added under the `#region` node).

If, however, the child is incompatible with one of its siblings, either the sibling or the child must be split, depending which one starts later. The two split parts are then removed from their tree and added at their new position by recursively calling `insert`.

Algorithm 5 shows the implementation in pseudocode.

```
   Input    : parent: TreeNode under which to add childToAdd
   Input    : childToAdd: TreeNode to be added under parent
 1 TreeNode incompatibleSibling ← getIncompatibleSibling(parent, childToAdd)
 2 if ¬∃ incompatibleSibling then
       // no incompatibilities, inserting will work fine
 3     List lToMove ← findNodesToMove(parent, childToAdd)
 4     int childIdx ← findIndex(parent, childToAdd)
 5     parent.addChild(childIdx, childToAdd)
 6     foreach TreeNode tn ∈ lToMove do
 7         tn.removeFromParent()
 8         TreeNode newParent ← findSpanningNode(childToAdd, tn)
 9         insert(newParent, tn)
10     end
11 else
       // incompatibleSibling starts before childToAdd
12     if incompatibleSibling.getStartOffset() < childToAdd.getStartOffset() then
13         TreeNode split1, split2 ← split(incompatibleSibling, childToAdd)
14         split1.removeFromParent()
15         TreeNode newParent ← findSpanningNode(incompatibleSibling, split1)
           // no more incompatibilities, now insert split1 and split2
16         insert(newParent, split1)
17         split2.removeFromParent()
18         insert(parent, split2)
19     else
           // childToAdd starts before incompatibleSibling
20         TreeNode split1, split2 ← split(childToAdd, incompatibleSibling)
21         split1.removeFromParent()
22         TreeNode newParent ← findSpanningNode(childToAdd, split1)
           // no more incompatibilites, now insert split1 and childToAdd
23         insert(newParent, split1)
24         insert(parent, childToAdd)
25     end
26 end
```

**Algorithm 5**: insert(node, config)

The procedure has a number of helper functions which work rather straightforward. Most of them rely merely on simple offset comparison. Functions `findIndex` and `findParentNode` have already been explained for the comment adding algorithm. Function `findNodesToMove` finds all children of a node which are embraced by some other node (and will therefore later be displaced and moved under that node). Function `findIncompatibleSibling` tries to find a child node from `parent` which is incompatible with `child`.

### Functions split and splitAt

Function `split` deserves more attention. It takes two incompatible nodes as arguments and causes the second node to be split at the end offset of the first node. If the node to be split has descendants at the according offset, they are split, too. After the nodes are split, their children are divided up

accordingly.

Algorithms 6 and 7 show the node splitting in pseudocode.

```
   Input    : reference: TreeNode that is incompatible with nodeToSplit
   Input    : nodeToSplit: TreeNode to be split at end offset of reference node
   Result   : split1, split2: The two parts of nodeToSplit after the splitting
 1 int splitOffset ← reference.getEndOffset()
 2 List lToSplit ← findDescendantsToSplit(splitOffset, nodeToSplit)
 3 foreach TreeNode tn ∈ lToSplit do
 4     splitAt(splitOffset, tn)
 5 end
 6 TreeNode split1, split2 ← splitAt(splitOffset, nodeToSplit)
 7 return split1, split2
```

**Algorithm 6**: split(reference, nodeToSplit)

```
   Input    : splitOffset: offset at which nodeToSplit must be split
   Input    : nodeToSplit: TreeNode to be split
   Result   : split1, split2: The two parts of nodeToSplit after the splitting
 1 TreeNode split1 ← clone(nodeToSplit)
 2 TreeNode split2 ← clone(nodeToSplit)
   // divide children of nodeToSplit among split1 and split2
 3 if nodeToSplit.hasChildren() then
 4     foreach TreeNode c ∈ nodeToSplit.getChildren() do
 5         nodeToSplit.removeChild(c)
 6         if c.getEndOffset ≤ splitOffset then
 7             split1.addChild(c)
 8         else
 9             split2.addChild(c)
10         end
11     end
12 else
13     split1.setEndOffset(splitOffset)
14     split2.setStartOffset(splitOffset)
15 end
   // exchange nodeToSplit with split1 and split2 in parent node
16 if nodeToSplit.hasParent() then
17     TreeNode parent ← nodeToSplit.getParent()
18     int idx ← nodeToSplit.getSiblingIndex()
19     parent.removeChild(nodeToSplit)
20     parent.addChild(idx, split1)
21     parent.addChild(idx + 1, split2)
22 end
23 return split1, split2
```

**Algorithm 7**: splitAt(nodeToSplit, splitOffset)

The function `findDescendantsToSplit` finds, as already mentioned, recursively all descendant nodes of `nodeToSplit` that also need to be split. It

is very important that the result list `lToSplit` contains the nodes in a special order: from the deepest to the highest level. If this is not the case the calls to function `splitAt` in the following loop will not produce correct results.

Function `splitAt` replaces the node to be split with two new nodes. The children of the `nodeToSplit` are moved to these new nodes. The children before the `splitOffset` are added to one node, the children after `splitOffset` to the other. Then `nodeToSplit` is removed from the tree and the new nodes are added instead.

There are cases when `nodeToSplit` has no children, for example, when directives are involved. A C# `#region` node, for example, contains among a `StartRegion` and `EndRegion` node also an `InputSection` node[2], which in turn contains a long string that represents the code and the comments standing inside the region. This string has to be removed before the algorithm is run, because it would interfere with the tokens from the actual code tree. Therefore the `InputSection` node has no children. Splitting such a node means adjusting the offsets of the new nodes manually.

### *3.3 Example*

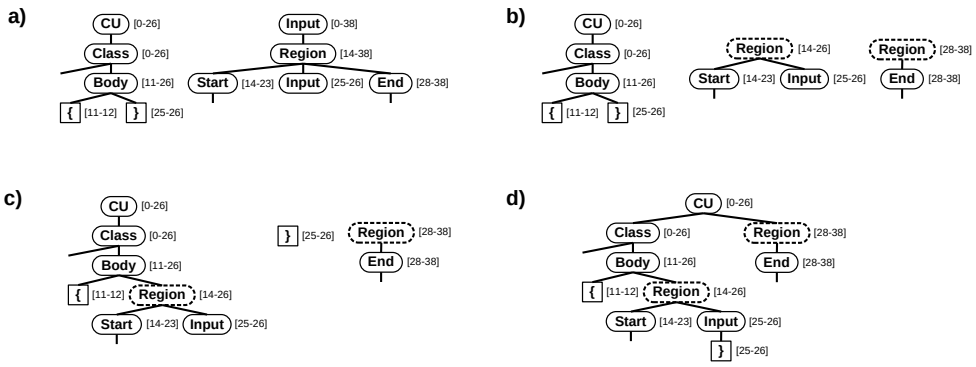



Fig. 5. Step-by-step example for tree combining

Figure 5 shows how the example from figure 4 would be transformed. In a) there are the original C# source code tree and the preprocessor tree (only the important parts are displayed). The algorithm starts by finding a new parent node in the C# tree for all children of the preprocessor tree's root node, i.e. `Region`. The new parent node is the `CompilationUnit` (`CU`). Next is a call of procedure `insert` with `CU` as `parent` and `Region` as `childToAdd`. However, `Region` overlaps with `Class`, and `Class` becomes the `incompatibleSibling`. The `Region` node must therefore be split as shown in b).

Due to the split the incompatibility has been resolved and the two parts can be added to the C# tree. The first part of the split `Region` node (variable `split1`) needs a new parent node. That parent node must be the `Body` node or one of its descendants. In the example the `Body` node itself is the new parent. Then `insert` is called recursively. All children of `Body` are compatible

with `Region`, but the closing brace will be displaced by `Region`. List `lToMove` therefore contains the closing brace. Then `Region` is added and the brace is removed, as shown in c). The new parent for the brace is `Input`. Another recursive call to `insert` adds the brace appropriately.

After the first part of the split `Region` node has been processed, the second part (denoted by variable `split2`) must be handled, too. For `split2` it is not necessary to find a new parent node. It must be added to `parent`. This was, after all, the original goal, before the `Region` had to be split.

Procedure `insert` is called again, `split2` is added to `parent` and the algorithm stops. The final result can be seen in d). In an optional postprocessing step the start and end offsets of the nodes can be recomputed. The `CU` node in the example should have an end offset of 38 instead of 26.

### *3.4 Combining multiple trees*

Often it is not enough to combine only two trees. In C#, for example, there might be the code tree, the preprocessor tree and the comment tree. Combining those takes two steps. First two trees are combined, e.g. code tree and preprocessor tree. The resulting tree is then combined with the comment tree. In general it is necessary to execute the algorithm n-1 times for n trees to combine. For each of these executions it is possible to use a dedicated configuration, making the approach very flexible.

## 4 Related Work

To the best of our knowledge, the topic of creating trees from code, directives and comments has received only little attention. In [5] the authors present multiple solutions to associate comments to to the right AST nodes in order to handle them correctly during refactorings. Comments are classified, depending on their relative position to statements: they can be leading, trailing or freestanding. The algorithm presented in our paper is generally suited to achieve that, too. One could create multiple comment trees, one for leading statement comments, one for trailing statement comments and so on, and add them to the code tree one after another with an appropriate configuration. Alternatively the check when to use the extended offset could be made more powerful.

The Eclipse[3] Java parser assigns comments to AST nodes. In contrast to the approach presented here the comments are not part of the actual AST. A comment mapper maintains a list of all comments and their mapping to AST nodes. This mapping is created by visiting the nodes of the AST and checking whether there are comments before (leading comments) or after them, but on the same line (trailing comments) and storing them in according arrays. This would potentially lead to the problems elaborated in section 2. However, in the case of the Eclipse Java-AST the consequences are not so bad because

there are no nodes such as `StatementList` with which the comments could be wrongly associated.

Many popular IDEs, such as Eclipse, NetBeans[4] or Visual Studio[6] offer code outlines, or navigators. Usually, however, they only show named entities of the code, e.g. classes, fields or methods. This allows an easy navigation in the code on a high level. For Eclipse there is a plugin, ASTView[1], which displays a 1:1 representation of the currently active file's AST and even shows to which nodes comments were associated by the comment mapper. However, due to the complexity of the displayed AST, ASTView is hardly suitable to find a specific element (e.g. statement) in the code. No tools could be found that try to combine both approaches. Nodes of the tree which are not necessary to represent the code structure can be omitted from the code outline. Nodes which have no meaning to the programmer, e.g. `Statement`, can be labeled so that one can immediately see, what part of the code they belong to. Figure 6 shows what such a simple outline looks like in DEFT.

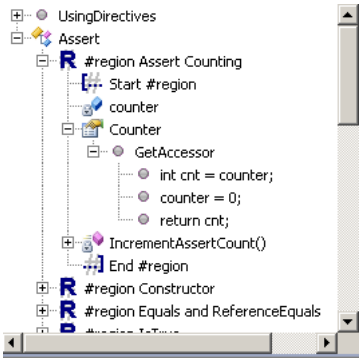



Fig. 6. Code outline with directives

## 5 Summary and outlook

In this paper we presented an algorithm to produce a syntax tree created from a normal program, comments and preprocessor directives. Such a tree can be useful for programming or software exploration tools.

The algorithm has been developed for the tutorial development environment DEFT, with which an evaluation will be made. Both performance of the algorithm and the clarity of the result tree are of interest. It is expected that there is rarely need to split tree nodes in code for programming tutorials, but it cannot be ruled out. For example, an author might want to show bad coding style with directives. The author should be easily able to navigate to the source code fragments he wants to document. Possibly some filter mechanisms to to hide the complexity of a tree with split nodes must be introduced.

While the algorithm itself works reliably, there are some open issues in closely related fields. One is the distinction of different kinds of comments: leading, trailing, and possibly others, depending on coding conventions. Ac-

cording to the discussion in section 4 it should be evaluated if the current configuration mechanism can handle this distinction of comment types efficiently, or if the configuration mechanism should be extended.

Furthermore the parser must be extended to handle problems introduced by directives such as `#if` in C#. It is, for example, possible to cut through statements, ending up with one statement start and two statement ends, as shown in figure 7.

```
6           int i =
7 #if A
8                   0;
9 #else
10                  1;
11 #endif
```

Fig. 7. if-directive cutting through a statement

It is necessary to think of a way to parse such code, preferrably in multiple passes with a standard parser, and to represent it properly in the syntax tree. The tree combining algorithm can then add the directive as explained in this paper.

## References

[1] *ASTView*.
URL http://www.eclipse.org/jdt/ui/astview/index.php

[2] *C# Language Specification*.
URL
http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf

[3] *Eclipse*.
URL http://www.eclipse.org/

[4] *NetBeans IDE*.
URL http://www.netbeans.org/

[5] Sommerlad, P., G. Zgraggen, T. Corbat and L. Felber, *Retaining comments when refactoring code*, in: *OOPSLA Companion '08: Companion to the 23rd ACM SIGPLAN conference on Object oriented programming systems languages and applications* (2008), pp. 653–662.

[6] *Visual Studio*.
URL http://msdn.microsoft.com/en-us/vstudio/products/default.aspx