

# Supporting the Designer in: Writing Nested Loop Programs using Eclipse

---

*Leiden Institute of Advanced Computer Science*

*Bachelor Project report*

*By: Jori van Lier - 0425478*

*Project supervisor: Bart Kienhuis, Assistant Professor, LIACS*

*July 15, 2008*

## ***Abstract***

An Eclipse editor plug-in for Nested Loop Programs written in the Matlab programming language, to be used by the LIACS-designed aggressive multithreaded Compaan compiler. The editor is built on extensions of the eclipse framework and provides certain advanced features like an automatic code formatter, code completion and code transformations.

# Contents

1	Context .....	3
2	Problems.....	3
2.1	Coding style .....	3
2.2	Time saving features .....	3
2.3	Complex transformations.....	3
2.3.1	Loop unfolding.....	4
2.3.2	Loop skewing .....	4
2.3.3	Other minor problems.....	5
3	Solutions.....	5
3.1	Formatting .....	5
3.2	Variables, code transformations .....	5
3.3	Other problems .....	6
3.4	Overview of added functionality .....	6
3.5	Concepts used .....	6
3.5.1	Eclipse editor infrastructure.....	7
3.5.2	JavaCC.....	7
3.5.3	Visitor concept.....	7
3.5.4	Line numbers .....	8
4	Implementation.....	9
4.1	Formatting.....	9
4.1.1	Parse tree changes .....	11
4.2	Reparse delay .....	11
4.3	Variable checking .....	12
4.3.1	Parse tree changes .....	14
4.4	Auto completion.....	14
4.4.1	Limitations .....	15
4.5	Transformations .....	16
4.5.1	Loop unfolding.....	16
4.5.2	Loop skewing .....	17
5	Results .....	19
5.1	Formatting.....	19
5.2	Variable checking and code completion .....	19
5.3	Transformations .....	19

5.3.1	Loop unfolding.....	20
5.3.2	Loop skewing.....	20
6	Conclusion.....	21
7	Bibliography.....	22

## 1 Context

This project builds on the foundation of an existing editor for Nested Loop Programs. This editor was very limited and a little buggy in a few areas, basically incomplete. The editor is built using the Eclipse environment and uses the excellent extensions available for plug-in development. Using this editor provides the designer with an easy way to write correct Nested Loop Programs. The source code can then be compiled by the aggressive parallel compiler Compaan (Compilation of Matlab to Process Networks), which is developed at LIACS.

Most of the core editor features already existed before the start of the project – generating the parse tree, creating the outline view, code syntax coloring etc. This project has focused more on supporting the designer by providing more advanced editor features, like automatic formatting, variable name checking, variable auto-completion and complex code transformations.

## 2 Problems

### 2.1 Coding style

Most developers have their own, distinctive coding style, whether it be C, Java, or PHP. Of course, it is good practice to learn to use a commonly accepted coding style, especially when working in a team. This works well, but it's easy to forget certain minor details, and going back over a lot of lines of code to correct them can be a very tedious job. It would be a lot easier and much faster to have some kind of automated procedure do this for you. A designer should focus on designing, not on correcting the amount of indentation in the code.

### 2.2 Time saving features

The previous section already contained a notion of taking work out of the hands of the designer. A designer should be able to focus on designing, and leave all tedious tasks to the software supporting the designer where possible. This isn't just a valid statement for tedious tasks. Anything that can reduce the amount of time needed for any action is usually a welcome gift. And not merely for the designer either, the employer will also appreciate to see tasks done as fast as possible.

Trivial things like looking up the name of a variable on the top of the document only takes a few seconds, but added up over a day's work for an entire development team this can be a lot. Having a list handy with all valid variables at any place in the document could be a great feature to improve efficiency.

### 2.3 Complex transformations

The Compaan (1) compiler is a very aggressive compiler with a lot of optimizations for usage in multi-core systems. Now, while it is convenient to have to compiler do most of the optimizing work,

wouldn't it also be nice to do some it if with a bit more control, on a source base level? We're defining two distinctive code transformation procedures for `for`-loops. One of them easily doable by hand without help, although tedious. The second is a lot more complex, and cannot easily be done manually, or at least not as fast as an automated procedure would do it.

### 2.3.1 Loop unfolding

The output of the unfolding transformation is an affine nested loop program which is functionally equivalent to the input program but with enhanced task-level parallelism. This approach to get the higher level of parallelism required is to copy a loop body a number of times in such a way that these copies are mutually exclusive.

In practical terms, this simply consists of duplicating the code inside a loop  $n$  times, using some modulo if-expression. In the simplest form, taking  $n = 2$ :

```
for i = 0 : 1 : 10,  
    [ a(i) ] = Func;  
end  
→  
for i = 0 : 1 : 10,  
    if mod(i,2) == 0,  
        [ a(i) ] = Func;  
    end  
    if mod(i,2) == 1,  
        [ a(i) ] = Func;  
    end  
end
```

### 2.3.2 Loop skewing

The skewing transformation creates a new nested loop program in which the bounds and the iterators are changed in a particular way to make the potential parallelism in the input affine nested loop program explicit.

The 'skewing' name stems from a certain matrix transformation, often used in other fields as well like computer graphics. See Figure 1 for a result. This transformation is done on a matrix that describes the iteration space of two or more loops. The dependency graph of the loops changes in a similar way.



Figure 1: A skewed rectangle

For example, take the following nested loop program:

```
for j = 1 : 1 : 4,
    for i = 1 : 1 : 3,
        [y(i), x(j)] = F(y(i), x(j));
    end
end
```

After applying the skewing transformation on the source code above, by skewing on the i-loop, this is the result:

```
for j = 2 : 1 : 4+3,
    for i = (max(1,j-4) : 1 : min(j-1,3)),
        [ y(i), x(j-i) ] = F(y(i), x(j-i));
    end
end
```

For further details on the technical background of this procedure, see this LIACS paper: (2).

### 2.3.3 Other minor problems

The NLP editor, in the state it was in before this project started, was not completely finished. There are a few minor quirks and issues that should be fixed. For example, the document is reparsed after every single change. This continuous reparsing tends to get quite slow on large input files. There are also some errors in the way the parse tree is generated, resulting in crashes for certain input texts, instead of an error marker. These exceptions should also be caught and handled appropriately.

## 3 Solutions

This section describes the solutions proposed to the problems outlined in the previous section. All solutions extend on the existing editor and use the Eclipse infrastructure in one way or another. This will be kept brief – see the next section for detailed information on the specifics regarding implementation.

### 3.1 Formatting

First of all, an automatic formatter has been implemented. This rewrites the entire code in a file or a smaller selection, based on the parse tree generated from the original code. Line numbers and comments are kept intact, while the nodes of the parse tree will be completely rewritten.

Most automatic formatting tools change the entire structure of the code, but due to the nature of this particular coding language a different approach is used. For example, the lack of opening and closing brackets (e.g. {, }) for if-statements and for-statements avoids the obvious choice of placing them on the same line or the next line, as in higher level languages like C++ and Java. Since there really is only one valid coding style with respect to these kind of control statements, it only seems logical to keep the original line numbers intact. These line numbers are stored in the node objects and can easily be requested as needed, by using `node.getBeginLineNumber()`.

### 3.2 Variables, code transformations

Secondly, a system has been added to check if variables are valid in a particular scope, together with an error marker system to indicate wrongly used variable names. This information on variables is also

used to implement a variable name suggestion system, also known as a auto-completion list, to be accessed directly in the code, at any location.

Furthermore, the desired code transformations which have been outlined in the previous section can be automated. They have been conveniently located in the editor context menu, allowing for complex code transformations with just a few mouse clicks. The designer has to place the cursor at the location which needs to be unfolded or skewed, right click, and select 'Loop unfolding' or 'Loop skewing'.

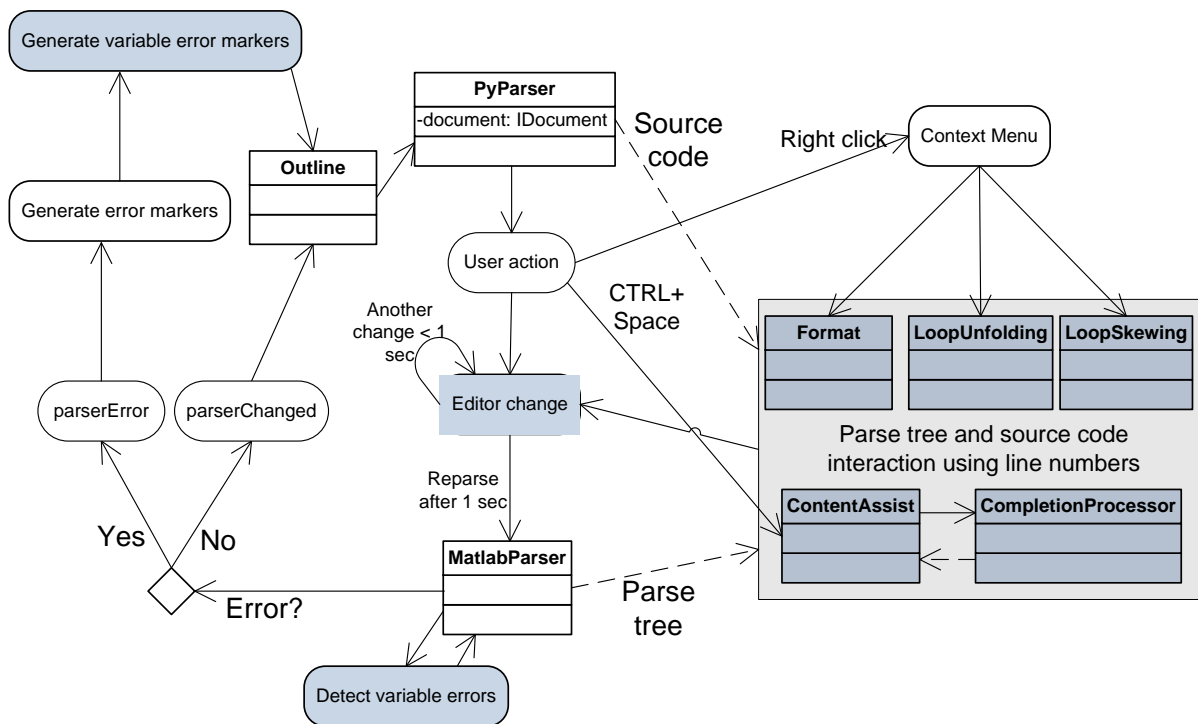
### 3.3 Other problems

A delay has been introduced for the reparsing of the code. Originally the editor reparsed after every key press, now it will only reparse if the designer has stopped typing for a certain amount of time.

Furthermore, some changes have been made to the way the parse tree is generated, either to fix a bug or as a requirement for one of the features.

### 3.4 Overview of added functionality

The following diagram shows the interaction of the classes and the control flow within one reparse cycle. Newly added functionality is marked in blue.



### 3.5 Concepts used

Here I will briefly outline the concepts that have been used for the implementation of the solutions.

### 3.5.1 Eclipse editor infrastructure

There are certain extension points in the Eclipse plug-in development environment which can be used as starting points for new features. These range from things as simple as an item in the editor context menus, to advanced features like context dependent auto complete lists.

Eclipse also supports visual feedback mechanisms which this editor uses to its advantage. An important one is the well known “red line” under errors, which has been used for the variable checking feature. Furthermore, there are default dialog windows for information, error messages and for asking questions. By using classes from the SWT toolkit (Standard Widget Toolkit (3)), it is also possible to design custom dialog windows.

### 3.5.2 JavaCC

Java Compiler Compiler (4) is an open source parser generator for Java. It generates a parser for a grammar provided in the Extended Backus-Naur Form notation, outputting the Java source code. JavaCC generates top-down parsers, which limits it to the LL(k) class of grammars. In particular, left recursion cannot be used. The tree builder that accompanies it, JJTree, constructs the tree from the bottom up.

### 3.5.3 Visitor concept

The visitor concept is a generic technique in object oriented programming languages to access elements of an object (private or public). An interface for a visitor has to be defined in advance, to allow implementation of custom visitors. This visitor can then be *accepted* by the target object, since the object type of the visitor matches the one in the method signature.

For the NLP editor, the visitor interface is defined in advance by JavaCC, and each visitor that is allowed to visit the object has to be an implementation of this interface. All usage of the visitor concept in this project has to do with accessing data of the parse tree.

Initiating a parse tree traversal using a visitor:

---

```
PyParser parser = editor.getParser();
SimpleNode root = parser.getRoot();
Visitor v = new Visitor();
root.jjtAccept(v, null);
```

---

Where the visitor is defined as follows:

---

```

public class Visitor implements MatlabParserVisitor {
    protected void _visitChildren(SimpleNode p) {
        int i = p.jjtGetNumChildren();
        SimpleNode n;
        for (int j = 0; j < i; j++) {
            n = (SimpleNode) p.jjtGetChild(j);
            n.jjtAccept(this, null);
        }
    }
    public Object visit(ASTIdentifier node, Object data) {
        _visitChildren(node);
        return null;
    }
    (...)
}

```

---

The 'visit' method is overloaded for each node type – 32 in total. The above code snippet only shows the visit method for the ASTIdentifier. To illustrate what a typical parse tree traversal looks like, assuming the `_visitChildren()` method is called in every `visit()` method, see Figure 2:

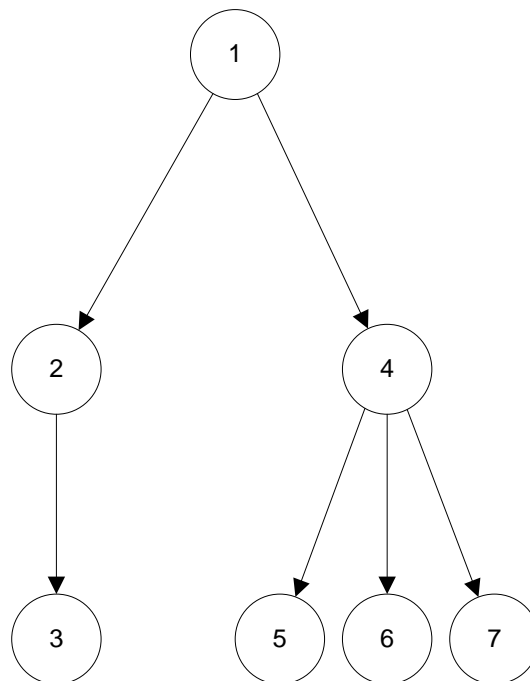


Figure 2: An example parse tree traversal. Numbers indicate order of visiting.

### 3.5.4 Line numbers

The parser generated by JavaCC is 100% java code, making it portable to an Eclipse plugin. It defines an interface 'Node' and an implementation class 'SimpleNode'. Every node in the parse tree inherits the SimpleNode class, granting it a lot of support methods and generic data storage with getters and setters. The variables 'beginColumnNumber' and especially 'beginLineNumber' are very important integers stored in these objects. It is the foundation on which all interaction between the parse tree and the source code is based. It is the only link between the source code and the parse tree, and is

used extensively in all of the new editor features. The code snippet below shows a line number request, for the first child of the root node:

---

```
SimpleNode child = (SimpleNode) root.jjtGetChild(0);  
int line = child.getBeginLineNumber();
```

---

To make the connection with the editor text, one can request this line as a string:

---

```
IDocument doc = editor.getDocumentProvider().getDocument(editor.getEditorInput());  
int lineOffset = doc.getLineOffset(line);  
int lineLength = doc.getLineLength(line);  
String lineText = doc.get(lineOffset, lineLength);
```

---

Alternatively, you could also use `doc.get()` to get the entire editor text, and take it from there using standard Java string manipulation techniques.

The method above is useful when the goal is to find the string that corresponds to a node in the parse tree. To do it the other way around, i.e. find a node that corresponds to a line in the source code, a search through the parse tree will have to be initiated. Imagine that we need to extract the upper bound and lower bound from the for-statement on a particular line. Take the visitor from section 3.5.3, and modify the `visit(ASTforStatement, Object)` method like this:

---

```
public Object visit(ASTforStatement node, Object data) {  
    if (node.getBeginLineNumber() == line) {  
        lowerBound = node.jjtGetChild(1).getName();  
        upperBound = node.jjtGetChild(3).getName();  
    }  
    _visitChildren(node);  
    return null;  
}
```

---

## 4 Implementation

This section shows a detailed overview of the implemented solutions.

### 4.1 Formatting

The automatic formatter combines two sources to generate the result:

- the nodes in the parse tree
- the textual contents of the editor (String)

The parse tree is used to reconstruct the source code by visiting all the nodes, while the editor text is used to extract the comments and restore these into the result string.

The first step is re-writing the entire parse tree. This is done using a visitor which starts at the root node, visiting the children in a linear top-down fashion. Different methods are accessed depending on the node type. For example, the code for visiting for-loop node:

---

```

public Object visit(ASTforStatement node, Object data) {
    append(getIdent() + "for ", node);
    ident++;
    int numChildren = node.jjtGetNumChildren();

    String[] separators = { " = ", " : ", " : ", ", " + getEndlineDelimiter() };
    for (int i = 0; i < 4; i++) {
        SimpleNode n = (SimpleNode) node.jjtGetChild(i);
        n.jjtAccept(this, null);
        append(separators[i], n);
    }

    for (int i = 4; i < numChildren; i++) {
        SimpleNode bodyNode = (SimpleNode) node.jjtGetChild(i);
        bodyNode.jjtAccept(this, null);
    }

    ident--;
    return null;
}

```

---

The first loop in the code snippet above rewrites the actual for-statement line by sending the visitor (using keyword: `this`) to its children in order, and appending the separators between them. This additional formatting like commas and equal signs has to be added by the automatic formatter on the fly, since this is not stored in the parse tree. The `append(String, SimpleNode)` method appends the string to the result `StringBuffer`, which eventually will contain the entire formatted code. The second argument is used to find the line number, by using `node.getBeginLineNumber()`.

The second loop in the `visit(ASTforStatement, Object)` method simply causes the visitor to accept each of the children in the body of the for-loop.

By using `n.jjtAccept()` on the children, the visit methods for these node types will get called, rewriting, for instance, the name of an iterator:

---

```

public Object visit(ASTIteratorIdentifier node, Object data) {
    append(node.getName(), node);
    return null;
}

```

---

Every node type has a visit method, but not every method actually rewrites code. For example, there is a node for complex expressions, but this is built from smaller parts such as constants and variables. There is no actual rewriting in the visit method for the complex expression, only a method call to visit the children.

After this process of rewriting the entire parse tree, the code still lacks the original comments. The next step is parsing the original text in the editor line by line, looking for the symbol that marks a comment: `'%%'`. When that symbol is found, a substring is taken using the length until the end line delimiter. The result `StringBuffer` is now accessed to find the correct offset and insert the comment, keeping the original line number intact

The final step is an Eclipse editor method call to replace the text. This also fires an 'editor changed' state, signaling all listener methods to begin processing (the reparse is, for example, added as such a listener). This step is undo-able with a single key press or mouse click, just like any other editor change.

#### 4.1.1 Parse tree changes

When working on the implementation for the automatic formatter, a new node type for the 'end' statement was introduced. The original parse tree implementation had the EBNF string '<END><NEWLINE>' inside the 'for' and 'if-else' statements, which required additional logic in the visitor to function properly. Simply rewriting 'end' when the visitor enters a `endStatement` node is a more elegant solution. This snippet shows the new node:

---

```
void endStatement () : { }
{
    <END> <NEWLINE>
    { }
}
```

---

Which is referenced by `controlStatement`:

---

```
void controlStatement() #void : {}
{
    (
        ifElseStatement() | forStatement()
    )
    endStatement()
}
```

---

## 4.2 Reparse delay

For implementation of the reparse delay, a one-second timer has been added. The timer is started after a change in the editor. When the timer runs out, the document is reparsed. Any additional change before the timer has finished, resets the timer, effectively delaying the document reparse by another second.

The method `reparseDocumentDelay()` has been added, which is called in the `documentChanged()` method of the editor's `IDocumentListener` listener. The timer action calls the original `reparseDocument()` method.

The code for the timer:

---

```

public void reparseDocumentDelay() {
    if (timer.isRunning()) {
        timer.restart();
    }
    else {
        timer.start();
    }
}
private ActionListener reparseActionListener = new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
        Display.getDefault().asyncExec(new Runnable() {
            public void run() {
                reparseDocument();
            }
        });
    }
};
private static int DELAY = 1000;
private Timer timer = new Timer(DELAY, reparseActionListener);

```

---

The `asyncExec` is needed to avoid SWT “invalid thread access” exceptions, caused by updating widget elements in the outline view.

### 4.3 Variable checking

Variable checking has been implemented to run simultaneously with the main code `reparse`, which means it is integrated in the the JavaCC file `MatlabParser.jjt`. During generation of the parse tree, all nodes call the `jjtreeOpenNodeScope()` method when first created and the `jjteeCloseNodeScope()` method when done processing. This includes generating all the children in that node. This behavior makes it an ideal location for implementation of the variable checking system.

Logic has been added to the close node scope method only, since the children need to be generated in order to do most of the checks. Note that this is different to the visitor methodology used in the implementation of other features. The variables are checked directly as the parse tree is being generated – not afterwards – so all checks are done on a partially finished parse tree.

There are separate stacks for each type of variable:

- Parameters, defined by `%parameter`.
- Variables, valid in the remainder of the file after initialization. Left hand side initialization:  
`[ a(i+1) ] = init.`
- Iterators, defined by a for-loop statement, only valid in the scope of the for-loop.

The following code snippet shows the adding of left-hand side defined variables to the variable stack:

---

```

if (n instanceof ASTleftVariableList) {
    for (int i = 0; i < n.jjtGetNumChildren(); i++) {
        SimpleNode variable = (SimpleNode)n.jjtGetChild(i);
        SimpleNode ident = (SimpleNode)variable.jjtGetChild(0);
        variables.push(ident.getName());
    }
}

```

---

This is the first step, keeping track of all defined variables. Now, variables on the right-hand side of a statement need to be defined on the left side first. The following snippet shows how it is checked if this is the case:

---

```

if (n instanceof ASTrightVariableList) {
    for (int i = 0; i < n.jjtGetNumChildren(); i++) {
        SimpleNode variable = (SimpleNode)n.jjtGetChild(i);
        ASTIdentifier identifier = (ASTIdentifier)variable.jjtGetChild(0);
        if (! ( variables.contains(identifier.getName()))) {
            ParseVarError error = new ParseVarError(
                identifier.getBeginLineNumber(),
                identifier.getBeginColumnNumber(),
                identifier.getEndColumnNumber(),
                "Undefined identifier: " + identifier.getName());
            varErrorList.add(error);
        }
    }
}

```

---

A new data structure was added to the parser to register the errors: `Vector <ParseVarError> varErrorList`. After the complete construction of the parse tree, this data structure is checked and any errors will be marked with a red error marker in the editor side line, and the typical 'red line'. `ParseVarError` is a simple data structure class which contains information such as the line number, column number, etc.

Parameters are added to the stack in a similar way. Iterators are done slightly differently: they are only defined in the scope of the for-loop. This means pushing iterators to the stack when they are created and popping iterators from the stack, once the parser is out of the for-loop node scope:

---

```

if (n instanceof ASTiteratorIdentifier) {
    iterators.push(n.jjtGetChild(0).getName());
}
if (n instanceof ASTforStatement) {
    iterators.pop();
}

```

---

Iterators and parameters need to be checked in the children of: `complexExpression` and `simpleExpression`. The following snippet details how this is implemented:

---

```

if (n instanceof ASTcomplexExpression ||
    n instanceof ASTsimpleExpression) {
    // Get all children identifiers of the node:
    Vector<ASTIdentifier> v = getIdentifiers(n, null);

    for (Iterator<ASTIdentifier> i = v.iterator(); i.hasNext(); ) {
        ASTIdentifier identifier = i.next();
        String name = identifier.getName();
        if (! (iterators.contains(name) || parameters.contains(name))) {
            newVarError(identifier);
        }
    }
}

```

---

### 4.3.1 Parse tree changes

The EBNF definition of the for-loop originally was:

---

```

<FOR> Identifier() "=" complexExpression() <COLON> Integer() <COLON>
complexExpression() ", "

```

---

This posed a problem in detecting the iterator. The open node scope method for the for-loop was not an option, since its children are not yet generated at that time. Doing it in the close node scope method would be too late, since variables in the body of the loop have already been checked. The solution was subtle: change the iterator from a generic ‘identifier’ node to ‘iteratorIdentifier’. It is now of a different object type, and by using Java’s ‘instanceof’ keyword it is easy to detect an iterator and add this to the stack.

## 4.4 Auto completion

The auto-completion feature is designed to be a suggestion list for parameters and iterators in the scope. It ties into extensions provided by the Eclipse infrastructure, giving it the exact same graphical appearance as the auto-completion in the default Java editor. The list of suggestions can also be narrowed down if the designer is already in the process of typing a variable name, by selecting on a prefix, taken on the left of the input symbol. For instance, if the two valid variables are ‘jS’ and ‘iS’ and the designer just typed ‘j’, only ‘jS’ is considered as a suggestion.

A regular editor contribution (implements `IEditorActionDelegate`) was added to the plugin manifest, with the key sequence `CTRL+Space`. The `run()` method in this action class starts the `CompletionProcessor` (implements `IContentAssistProcessor`).

First, the prefix – if any – is determined by checking characters on the left of the input symbol (the location of which stored is in the integer ‘offset’):

---

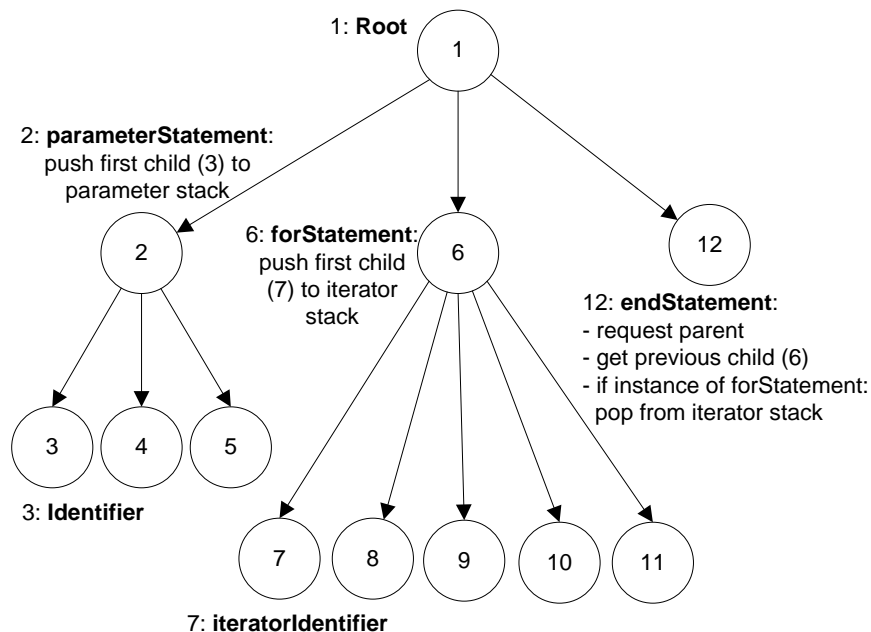
```

String prefix = "";
int index = offset - 1;
while (Character.isLetterOrDigit(text.charAt(index))) {
    prefix = text.charAt(index) + prefix;
    index--;
}
index++;

```

---

The next step is finding the valid variables at this offset in the source, using a visitor. On completion, these are placed in: `Stack<String> suggestions`. The following diagram shows the actions of the visitor in a typical NLP program:



The numbers in the nodes indicate the order of node processing by the visitor. Visitor processing ends when the following condition is met:

$$\text{Node offset (line number + column number)} \geq \text{cursor offset}$$

Any iterators that have not been popped at this time are in the scope of the current cursor location and should be in the suggestion list. The final step involves iterating over the suggestions to create the `CompletionProposals` and select on the prefix if required. An array of these proposals is returned and used by Eclipse to fill the suggestion list and display it on the screen.

---

```

for (String txt : suggestions) {
    if (txt.length() > 0) {
        if (!prefix.isEmpty()) {
            if (txt.startsWith(prefix)) {
                proposals.add(new CompletionProposal(txt, index,
                    length, txt.length()));
            }
        } else {
            proposals.add(new CompletionProposal(txt, offset, 0, txt
                .length()));
        }
    }
}
return proposals.toArray(new ICompletionProposal[proposals.size()]);
  
```

---

#### 4.4.1 Limitations

The current implementation is fairly limited in the sense that the contents of the suggestion list are not necessarily syntactically correct. The valid variables will be listed, but it will not check in advance

whether or not the suggestion will result in a parser error – that task is still up to the designer. To clarify this, take a blank line in a NLP program as an example. If ‘i’ is a valid iterator in the scope, this will be listed as a suggestion, but using this suggestion results in a parser error. One would expect iterators to be used inside an expression, not just anywhere, although they will be suggested anywhere.

## 4.5 Transformations

Both transformations are initiated from the editor context menu using a custom menu entry. The starting point of the transformation is the current cursor location.

### 4.5.1 Loop unfolding

Upon choosing ‘Loop unfold’ from the editor context menu, this functionality starts a walk through the parse tree. It traverses the parse tree as usual, until a node is reached with a start line equal to or greater than the start line of the selection. Starting from this node, the visitor goes upwards in the parse tree, until the root node is reached. The data that needs to be collected: all iterators in the scope, and for the inner for loop only: the start and end of the for-loop body, which contains the code to be duplicated. This is the method responsible for the data collection:

---

```
private void collectIteratorsInScope(SimpleNode n) {
    if (n instanceof ASTForStatement) {
        if (!_foundFirstForloop) {
            _foundFirstForloop = true;
            lastLine = n.getEndLineNumber();
            SimpleNode childInBody = (SimpleNode)n.jjtGetChild(4);
            if (childInBody != null) {
                firstLine = childInBody.getBeginLineNumber();
            }
        }
        SimpleNode iterator = (SimpleNode)n.jjtGetChild(0);
        iterators.add(iterator.getName());
    }
    SimpleNode parent = (SimpleNode)n.jjtGetParent();
    if (parent != null) {
        collectIteratorsInScope(parent);
    }
}
```

---

When this is done, a dialog is presented to the user. There are a couple of options to consider:

- Times to unfold loop (1-10)
- Expression to be used in mod statement

This dialog extends `org.eclipse.jface.dialogs.Dialog` and uses widgets from the SWT toolkit.

When these choices have been made, the unfolding procedure starts. After a few straight forward calculations to get the indentation-string and the offset and the length of the code inside the for-loop, the replacement text is generated. The following code snippet shows how this is implemented:

---

```
String loopContents = _doc.get(offset, totalLength);
loopContents = loopContents.replaceAll(endLine, endLine + " ");

String result = "";
for (int i = 0; i < times; i++) {
    result += ident + "if mod(" + expression + "," + times + ") == "
        + i + "," + endLine;
    result += " " + loopContents;
    result += ident.substring(0, ident.length() - 2) + "end" + endLine;
}
doc.replace(offset, totalLength, result);
```

---

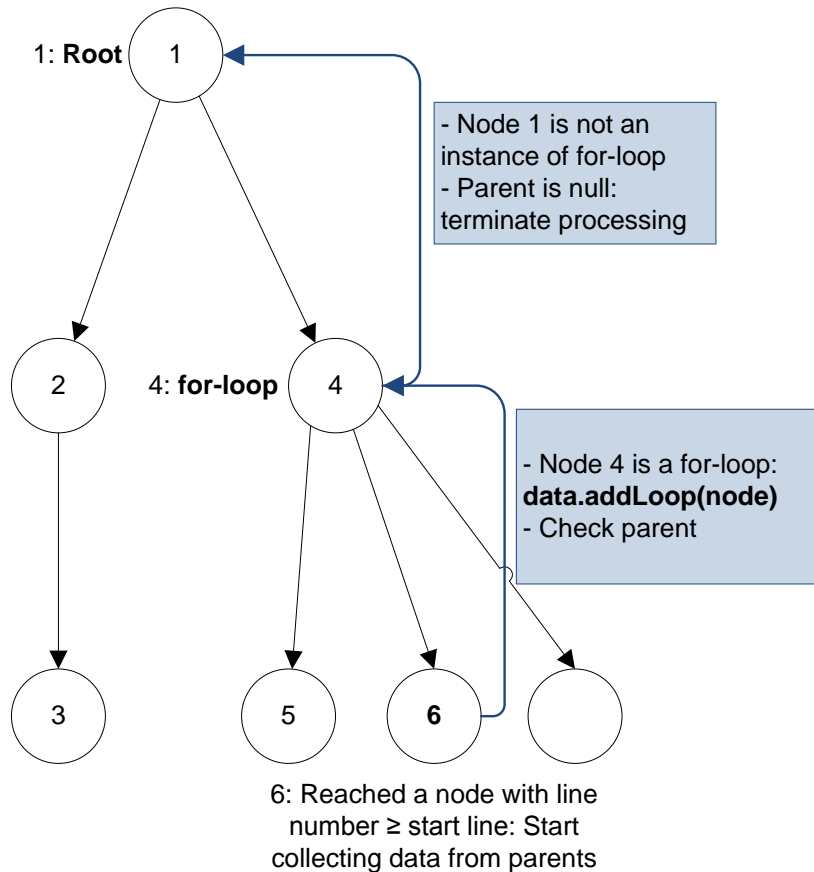
The loop content is duplicated multiple times, separated by modulo-if statements. See the results section (5.3.1) for a NLP code example.

#### 4.5.2 Loop skewing

Three classes have been designed to implement loop skewing:

- LoopSkewData: a data structure class
- LoopSkewDialog: used for input from the user into a dialog screen
- LoopSkewing: the actual skewing class

Similar to loop unfolding, the first step is a walk through the parse tree to collect information needed for the transformation. Below is an example if node 6 is at the cursor location. The blue arrows indicate `node.getParent()` calls.



The `data.addLoop(node)` method call in the diagram adds the line number, lower and upper bound and the iterator name of the for-loop to `data`, which is an instantiation of `LoopSkewData`. Note that loop skewing needs a minimum of two for-loops.

The next step is constructing the skewing matrix. This matrix is automatically generated to be the identity relation, with a skew on the first iterator. The default matrix is described by the Matlab string: `[1,0,0,0;0,1,1,0;0,0,1,0;0,0,0,1]`, which already works for two for-loops without any parameters involved. For each additional for-loop or parameter (in the lower or upper bound), an additional column and row will be added, in such a way that it keeps the identity relation intact. Finally, the skewing matrix is presented to the designer, who can modify the matrix as desired.

From here on, everything is automated. Next up is construction of the for-loop matrix. The iteration space of a for-loop can be described using two expressions:

- Lower bound  $\leq$  Iterator
- Iterator  $\leq$  Upper bound

A set of these expressions is created for each for-loop involved in the skewing process and added to a data structure. This is then converted to a matrix, to be used in the next step. The code snippet is responsible for generating the for-loop matrix:

---

```
List<Expression> expressionList = new ArrayList<Expression>();

for (int i = 0; i < data.getNbOfLoops(); i++) {
    Loop loop = data.getLoop(i);
    ExpressionParser r = new ExpressionParser();
    Expression e1, e2;
    try {
        e1 = r.getExpression(loop.getLowerBound() + "<=" +
            loop.getIterator());
        expressionList.add(e1);
        e2 = r.getExpression(loop.getIterator() + " <=" +
            loop.getUpperBound());
        expressionList.add(e2);
    } catch (ParseException e) {
        e.printStackTrace();
    }
}

SignedMatrix forloopMatrix = Convert.
    expressions2SignedMatrix(expressionList, dpVec);
```

---

Next, the for-loop matrix will be multiplied with the inverse of the skewing matrix to get the result, the skewed for-loop matrix:

---

```
JMatrix skewMatrixInv = Convert.uinv(skewMatrix);
forloopMatrix.rMul(skewMatrixInv);
SignedMatrix skewedMatrix = new SignedMatrix(forloopMatrix);
```

---

Finally, the skewed matrix will be converted back into a set of for-loops with new upper and lower bounds, to be placed into the code, replacing the existing for-loop statements. Iterators have

received an “S” suffix to denote that they have been skewed. The new names for the iterators are then string-replaced in the body of the inner for-loop to complete the skewing procedure.

## 5 Results

### 5.1 Formatting

Badly formatted, unstructured code:

```
17for l = 3 : 1 : M,  
18for m = 3:1:N - 1,  
19    if l+m <= 7,  
20        [ r2 ( l , m ) ] = Pass( r1 ( l- 1 , m - 2));  
21    end  
22        if l + m >= 8,  
23 [ r2(l,m) ] = Pass (r1( l, N-3 ));  
24        end  
25 [Sink(l,m)] = Pass( r2(l,m));  
26end  
27end
```

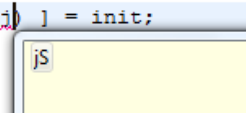
Formatted code after pressing CTRL+SHIFT+F or selecting the option from the context menu:

```
17for l = 3 : 1 : M,  
18    for m = 3 : 1 : N-1,  
19        if l+m <= 7,  
20            [ r2(l,m) ] = Pass(r1(l-1,m-2));  
21        end  
22        if l+m >= 8,  
23            [ r2(l,m) ] = Pass(r1(l,N-3));  
24        end  
25        [ Sink(l,m) ] = Pass(r2(l,m));  
26    end  
27end
```

### 5.2 Variable checking and code completion

‘j’ is not a valid variable in this scope, therefore it is marked with a red line and an error marker. Code completion (press CTRL+Space) at that location shows all valid variables that start with prefix ‘j’, in this case: ‘jS’.

```
9for tS = 1 : 1 : P,  
10    for jS = 1 : 1 : M,  
11        [ r1(tS,j) ] = init;  
12    end  
13end  
14
```



### 5.3 Transformations

Loop unfolding and skewing will be applied to this bit of code:

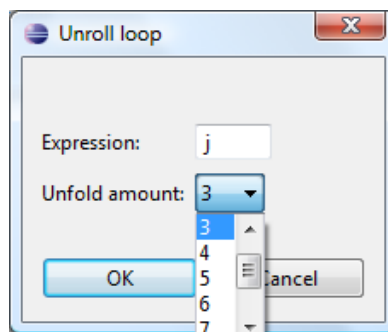
```

6 %parameter P 1 100;
7 %parameter M 1 100;
8 %parameter N 1 100;
9 for t = 1 : 1 : P,
10  for i = 1 : 1 : M,
11    for j = 4 : 1 : N,
12      [ r1(i+1,j-3) ] = init;
13    end
14  end
15end

```

### 5.3.1 Loop unfolding

This is the dialog, used to enter the parameters, and the corresponding result:



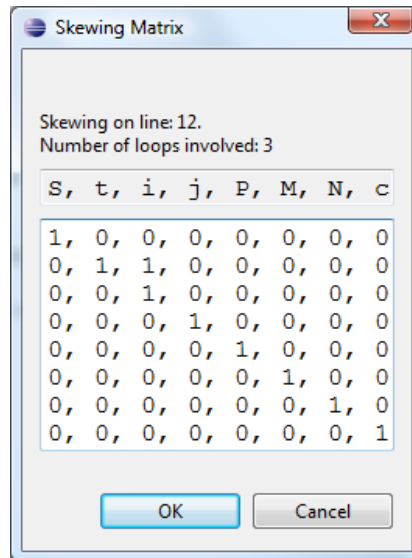
```

6 %parameter P 1 100;
7 %parameter M 1 100;
8 %parameter N 1 100;
9 for t = 1 : 1 : P,
10  for i = 1 : 1 : M,
11    for j = 4 : 1 : N,
12      if mod(j,3) == 0,
13        [ r1(i+1,j-3) ] = init;
14      end
15      if mod(j,3) == 1,
16        [ r1(i+1,j-3) ] = init;
17      end
18      if mod(j,3) == 2,
19        [ r1(i+1,j-3) ] = init;
20      end
21    end
22  end
23end

```

### 5.3.2 Loop skewing

The skewing matrix: an identity matrix, bar an extra '1' on the 'i'-column.



```

6 %parameter P 1 100;
7 %parameter M 1 100;
8 %parameter N 1 100;
9 for tS = 2 : 1 : P+M,
10     for iS = max(tS-P,1) : 1 : min(tS-1,M),
11         for jS = 4 : 1 : N,
12             [ r1(iS+1,jS-3) ] = init;
13         end
14     end
15 end

```

## 6 Conclusion

Eclipse provides an excellent plug-in infrastructure, which is easily adaptable to almost any language and can host a variety of features, custom or as implementations of pre-defined extension points.

It has also become clear that it is possible to bring complex code transformations to the source-code level in a convenient method. Eclipse has a host of refactoring features for Java built in the core editor, but the purpose of most of them is to make the code look neater. None actually change, for example, the iterated space of a for-loop. This skewing operation was implemented with the NLP programs in mind, but it could also be done for any other procedural language, like C or Java.

## 7 Bibliography

1. *Compaan Design bv*. [Online] [www.compaandesign.nl](http://www.compaandesign.nl).
2. **T. Stefanov, B. Kienhuis, E. Deprettere**. Algorithmic Transformation Techniques for Efficient Exploration of Alternative Application Instances. [Online] 2002.  
[http://ptolemy.eecs.berkeley.edu/~kienhuis/ftp/CODES\\_02.pdf](http://ptolemy.eecs.berkeley.edu/~kienhuis/ftp/CODES_02.pdf).
3. SWT: The Standard Widget Toolkit. [Online] <http://www.eclipse.org/swt/>.
4. JavaCC Home. [Online] <https://javacc.dev.java.net/>.