

# Building a Hybrid Editor

Martijn van Steenbergen

May 28, 2008

## Abstract

Manipulating an AST produced by a parser is usually reasonably straightforward. However, when converting the AST back to source code, comments and layout often get lost. Requiring that comments and layout are maintained as much as possible during mutations on the AST proves an interesting challenge, and its consequences on the design of the parser and AST are explored here.

## 1 Introduction

Most editors are either plain text editors (such as Notepad, Smultron) or graphical-oriented editors (such as Word, OmniGraffle). In the former case, the user has full control over the precise contents of the file, while in the latter case the precise underlying format is usually hidden from view. A few editors are both kinds at the same time: they allow the user to freely switch between the graphical view and its textual source. We call these *hybrid editors*. This introduces many new challenges: suddenly, ease of reading and editing of the graphics' source code becomes important, because users need to be able to work with it directly. Also, now there are two (or maybe more) radically different views on the same model which need to stay synchronized at all times. From 2003 through 2007 I have built such an editor for Eljakim IT B.V. together with Timon Bijlsma, and in this article I am going to tell you in more detail about the challenges of converting between text and AST.

The code snippets in this paper are written in Java 5 and use generics and annotations.

## 2 An introduction to EQL

The editor was built for a declarative domain-specific language called EQL. EQL is like a properties file (key-value pairs), but then hierarchical, typed and with simple references. Available primitive types are string, integer, float, boolean, enums and typed lists. References are only allowed to top-level declarations. For example, an EQL file could read:

```
string name = "EQL";
integer count = 5;
float width = 3.14;
list of integer sequence = [1, 2, 3, 4, count, 6];
```

Hierarchy and grouping is introduced by defining classes and their object instances. A class is simply a collection of property names and their types. Defining a class introduces a new type which can again be used as type for properties in other classes. Properties can have default values defined in the class definition. Classes can extend other classes, adding new properties, narrowing an existing property's type and/or overriding its default value. For example:

```

class Box {
    float x, y;
    float width, height;
}

class Text (Box) {
    string text;
    FontDesc font = FontDesc { // default value
        fontFamily = "Arial";
        fontSize = 12.0;
    };
}

class FontDesc {
    string fontFamily;
    float fontSize;

    boolean bold = False;
    boolean italic = False;
}

```

The precise syntax and semantics of EQL are beyond the scope of this article, but they are based on ideas from other languages and their meaning is usually easily guessed.

### 3 Non-pretty printing

In general, it is relatively straightforward to convert an AST back to text in such a way that when the resulting text is parsed the original AST is obtained again. Because parsers tend to discard whitespace, using it only to separate tokens, the conversion process can freely choose where to put newlines and horizontal whitespace, producing pretty text that is easy for humans to read. This process is referred to as *pretty printing*, sometimes even if the produced text is not neatly formatted. Figure 1 illustrates the conversion between source text and the AST. For pretty printing, the following equation should hold:

$$\text{parse} \circ \text{print} = \text{id} \tag{1}$$

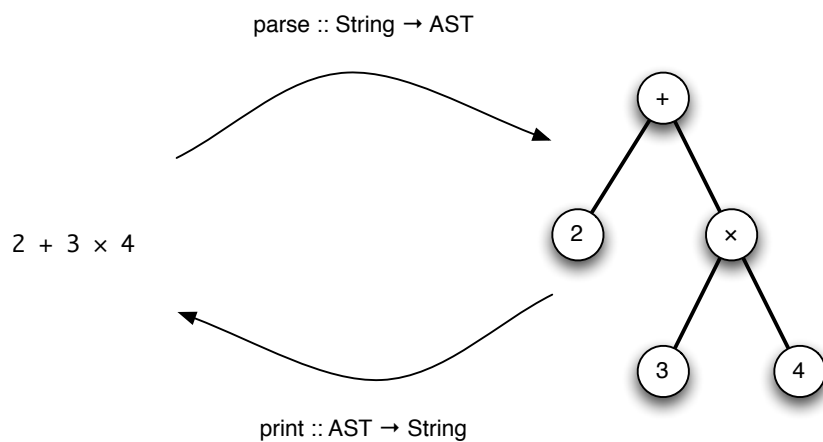


Figure 1: Converting between text and AST.

When not only the graphical but also the textual representation of the AST is to be edited by users, as is the case with hybrid editors, it is desirable that the equation's reverse holds too:

$$\text{print} \circ \text{parse} = \text{id} \tag{2}$$

The function *id* is the identity function, while the operator  $\circ$  denotes function composition and is defined as follows:

$$(f \circ g) x = f (g x) \tag{3}$$

However, when parsers discard whitespace, equation 2 cannot hold. Even worse, most parsers not only discard whitespace; comments and redundant parentheses are lost too. Imagine adding comments in the source, switching to the graphical view, making changes and then going back to the source view to find out that all your comments are gone. This is generally unacceptable.

One solution to this problem is to save all information and store layout and comments in the AST, so that the print operation has enough information to fully reproduce the original source text. Then equation 2 will hold, too. The printer is then no longer really a *pretty printer*.

But where to store this extra information in the AST? Because whitespace and comment tokens can appear between any two tokens, there is no single logical place in which to store this information. Reserving references for all possible positions of whitespace in an AST is just as bad as writing a parser that explicitly mentions all possible locations of whitespace: it clutters the code and makes it very hard to understand and maintain.

### 3.1 Tokens as part of the AST

Instead of reserving references for all possible positions of whitespace in the AST, we can do something more generic: besides saving the AST returned by the parser, we also remember the list of tokens returned by the lexer. This list is implemented as a doubly linked list: each token in the list has references to its preceding and to its succeeding token.

Next, each node in the AST has a reference to its starting token and a reference to its ending token. For example, in the expression `[1,2,3]` the list node in the AST has a reference `begin` to the token representing `[` and a reference `end` to the token representing `]`. The individual number literals have their corresponding tokens as both `begin` and `end` token. This is illustrated in figure 2. In this way we can immediately map from AST nodes to token intervals, and from token intervals we can reconstruct the original source code by visiting the list's nodes in order.

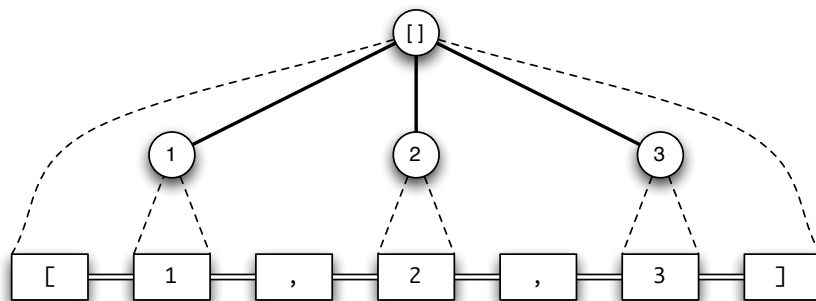


Figure 2: An AST with its nodes referring to intervals in the token list.

### 3.2 Special tokens

JavaCC's lexer allows for three kinds of tokens: regular tokens, special tokens, and skipped tokens. Regular tokens are what you expect them to be: they serve as input for the parser. Skipped tokens, too, are what you expect: they are discarded, and the parser never gets to see them. Whitespace is often defined as a skipped token in JavaCC grammars.

The so-called *special* tokens are somewhere in between regular and skipped tokens: they are not directly part of the stream of tokens the parser reads from, but neither are they completely discarded. Instead, they are stored in a separate reference called `special` in token nodes in the linked list. A special token is referred to by the token directly following it, so when visiting the tokens from beginning to end the special tokens can be seen as dangling behind other tokens.

We can conveniently use this to solve our problem: instead of discarding whitespace and comments, we mark them as special tokens and so they become part of the lexer's output without the parser noticing them; yet we can still access them once the AST has been built. Figure 3 shows what the token list looks like for an example piece of source text. The source text has one comment followed by the definition of an integer field. The comment and whitespace tokens are explicitly present in the list of tokens, but are not part of the main flow. The special tokens' exclusion from the flow is not only convenient during parsing, but also later when the token list is manipulated, as we will see in section 5.

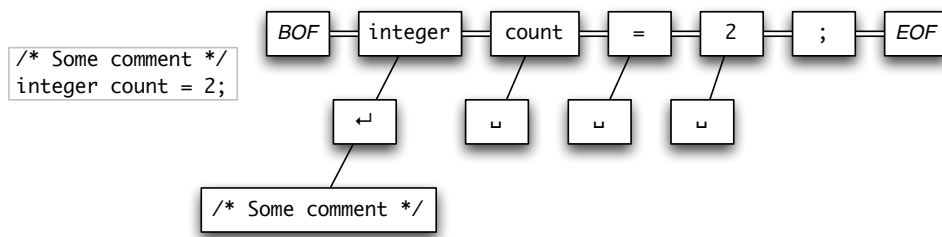


Figure 3: Special tokens dangle behind the meaningful tokens and form no part of the main flow.

The choice to have the special tokens be referred to by their succeeding tokens is arbitrary. However, considering how documentation generators such as Javadoc and Haddock expect documentation to appear before language constructs, this choice is usually the more convenient one: we can ask for an AST node's token interval, and follow the `special` reference of the interval's begin token. The symmetric alternative would require an additional step.

To be able to end a document with special tokens, there is an extra EOF token that does not contain any characters but marks the end of the token list and holds a reference to any special tokens the document might end with. There is a similar BOF token at the beginning of the token list. This way, the document root's interval can remain constant, even when the document's contents change: its begin and end token are always the same BOF and EOF tokens. This will come in handy later on.

Figure 4 shows Java class `Token` and the `print` method for reconstructing the source code from the linked list. The `image` field contains the actual textual representation of the token.

## 4 Commands and the undo stack

The Command design pattern [Wik08a] is a beautiful and extremely useful pattern when building editors. The idea of the Command design pattern is to represent each mutation on the model as a parametrized Command object. These commands can be used to implement many useful things, including:

- An undo history. When a command is executed, it is pushed onto the undo stack. Undoing causes the last command to be popped and unexecuted.

```

class Token {

    Token previous, next, special;
    String image;

    void print(StringBuffer buf, Token end) {
        // Print dangling special tokens, if any.
        if (special != null) {
            special.print(buf, end);
        }

        // Append this token's text.
        if (image != null) {
            buf.append(image);
        }

        // Print this token's successors.
        if (this != end && next != null) {
            next.print(buf, end);
        }
    }

    ...
}

```

Figure 4: Part of class `Token`, including its `print` method. Access modifiers such as `public` and `private` are left out for brevity.

- Serialization of mutations. Mutations can be stored in files or sent over the network.
- Transactions. When executing a whole list of commands and one of the commands' execution fails, the preceding commands can be undone so the whole transaction is rolled back.

Each command implements the following methods:

- `boolean canExecute()` — a specific mutation is not always possible in the current state of the model.
- `void execute()` — when `canExecute` returns true, applies the mutation to the model.
- `boolean canUndo()` — after having been executed, tells whether the mutation can be undone.
- `void undo()` — when `canUndo` returns true, undoes the mutation.
- `void redo()` — usually the same as `execute`, but some mutations may want to do something special when they are executed a second time.
- `String getLabel()` — returns a label used to describe the mutation. It usually depends on the mutation's parameters.

Commands offer a modular way of thinking about mutations. This also helps in section 5, where we will reimplement the commands on the AST as corresponding commands on the token list.

#### 4.1 Commands on the EQL model

There are only a few primitive mutations possible on the EQL AST:

- `ListInsertCommand` to insert an element into a list. Parameters: the list, the new element, and its desired index.
- `ListDeleteCommand` to delete an element from a list. Parameters: the list and the index of the element to delete. The command takes care of saving the deleted element in case the operation needs to be undone.

- `SetPropertyCommand` to set a property's value on an object. Parameters: the object, the name of the property and the new value.
- `RemovePropertyCommand` to restore a property to its default value. Parameters: the object and the name of the property to remove.
- `RenameCommand` to rename top-level declarations. Parameters: the declaration to rename and the new name.

Every time the command takes care of saving old data in case an operation needs to be undone. In `SetPropertyCommand` the property's old value is remembered, for example.

Then there is the `CompoundCommand` which is the *composite* in the Composite design pattern [Wik08b]. It groups a list of commands together and executes them in order and undoes them in reverse order. It could be used to implement the transactions mentioned above. It is especially useful when building a compound command where the child commands depend on the compound command's arguments or the model's state. Compound commands can be nested arbitrarily deep. For example, a `SetBoundsCommand` in our editor is a compound command consisting of four child `SetPropertyCommands`: two for the coordinates and two for the width and height. A `CreateBlockCommand` is a compound command that adds a block on the screen, and its constructor creates the necessary child commands based on the arguments and the model's current state:

- If the list of blocks does not exist yet, it installs a `SetPropertyCommand` to create it.
- Then it adds a `ListInsertCommand` to insert the new block in the list of blocks.
- A `SetBoundsCommand` is added to correctly position the block on the screen. Note that we just defined `SetBoundsCommand` to be a compound command itself.
- If the block is a `Line` (a special kind of block), its `LineDirection` property is set.

Undoing a `CreateBlockCommand` then automatically undoes only those subcommands that were executed in the first place. In this way it is very easy to compose complicated commands from simpler ones.

## 5 Keeping the AST and the token list synchronized

The graphical editor directly manipulates the AST. However, when switching back to the textual view after having made changes in the graphical view, we want the token list that is stored alongside the AST to reflect the changes made to the AST. Recall figure 4: the new source is directly generated from the token list.

This is where the modular approach of the commands comes in handy again: to do this, we only need to consider how every primitive (non-compound) command discussed above affects the token list. This in turn requires some primitive operations on the token list. We answer these questions in this section.

### 5.1 Kinds of intervals

Intervals are pointers to a begin and end token in a linked list of tokens. In most cases the list will extend more to the left and the right of the interval, but this need not be the case. Figure 5 illustrates all four possible cases. We call an interval a *prefix* if the list does not extend to the left, and a *suffix* if it does not extend to the right. If an interval is both a prefix and a suffix, it is called *maximal*. When defining operations on intervals, these four cases need to be considered. Some might require special care, while others may be rejected completely.

### 5.2 Operations on token intervals

For the operations on token intervals, we need two new operations on Tokens. They are listed in figure 6.

Let's take a look at the token interval operations. Note that in each case the interval whose method is called is referred to as *this interval*.

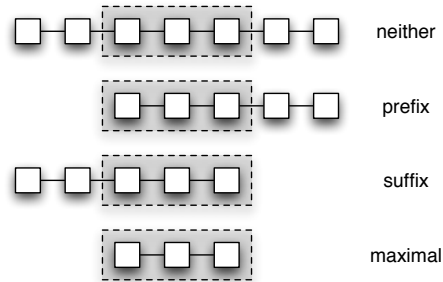


Figure 5: Both ends of an interval may or may not be connected to more tokens. These cases must be considered when defining operations on intervals.

```

class Token {
    ...

    Token(String image) {
        this.image = image;
    }

    void linkWith(Token next) {
        this.next = next;
        if (next != null) {
            next.previous = this;
        }
    }
    ...
}

```

Figure 6: Another part of class Token. See also figure 4.

- `TokenInterval(Token begin, Token end)` — constructs a new interval.
- `TokenInterval(Token t)` — convenience constructor for `TokenInterval(t, t)` to construct a token that spans only one interval.
- `insertAfter(Token t)` — the token is part of a different token list. Insert this interval into that chain after the specified token. This interval must be maximal.
- `insertBefore(Token t)` — dual of `insertAfter`.
- `append(TokenInterval ti)` — the supplied tokens are inserted after this interval, and this interval is extended to include the inserted tokens. The supplied interval must be maximal and will become non-maximal.
- `prepend(TokenInterval ti)` — dual of `append`.
- `replaceBy(TokenInterval ti)` — the supplied interval takes the place of this interval. The supplied interval must be maximal; this interval will become maximal.
- `cut()` — the interval is cut from its context, linking `begin.previous` with `end.next`, if they exist. This interval becomes maximal.

### 5.3 The AST commands' duals

Now that we have operations on token intervals, we can take a look at the token-based variants of the commands. `RenameCommand` is the easiest: everywhere the old name is replaced by the new name, the old token is replaced by a new one with the new name.

The other commands are a bit trickier. There are three types of lists in the AST: the list of top-level declarations, the list of properties in an object and the list of child elements of a list value. All three are implemented using the same low-level data structure: a *muli*, short for mutable list. `Mulis` are normal lists and support all common list operations, but also allow clients to register event listeners through the `MuliListener` interface; see figure 7. The reason for the special name "muli" is that "list" already has so many meanings, including the standard class `java.util.List`.

```
interface MuliListener<E> {
    void elementInserted(E newElement, int index);
    void elementRemoved(E oldElement, int index);
}
```

Figure 7: The generic `MuliListener` interface.

Whenever an element is inserted in or removed from a list, its token representation needs to be updated. How the tokens change depends on the kind of the list. The objects that update the tokens are called *token manipulators*; they are implemented as `MuliListeners`, à la the Model-View-Controller design pattern [Wik08c].

Updating the tokens of the top-level declarations and object properties is the easiest, and their algorithms are very similar: each element ends with a semicolon, and there are no separator tokens in between the elements. Deletion of an element results in a `cut()` operation on that element's token interval. Insertion is pretty easy, too: we ask for the new element's token interval and insert it right after the end of the token interval of the element preceding the newly inserted element. The only case in which this does not work is if there is no preceding element, i.e. if the new element was inserted at the beginning of the list. In that case we need the *opening token* of the list construct, and in fact this is the only way in which the token manipulators of the list of top-level declarations and of a list of properties differ: for an object, the opening token is the `{`, while for the list of declarations it is the document root's BOF token we mentioned earlier.

List values are slightly trickier, because their child elements are separated by commas. We cannot treat these commas the same way we treated the semicolons above, because the commas are not part of the elements' intervals, and there is no comma after the last element in the list.

When inserting a new element into a list, we need to distinguish three cases:

- If the list is empty, we insert the elements' tokens right after the list's opening bracket. No comma is necessary.
- If the list is not empty and the element is inserted at the beginning of the list, we insert the element's tokens after the list's opening bracket and insert a comma after the element's tokens.
- Otherwise, we insert the element's new tokens after the preceding element's tokens, then insert a comma before the new element's tokens.

Deleting an element requires three similar cases:

- If the removed element was the only element in the list, no comma needs to be deleted.
- If the element was the first in the list, the comma after the element is removed.
- Otherwise the comma before the element is removed.

We have now implemented all token-variants of the commands we introduced in section 4. The `SetPropertyCommand` can be seen as a conditional removal of a property element (if the property had an old value), and an insertion of a new property element. We can do even better: if the property has an old value, we only need to `replaceBy` the old value's tokens by the new value's tokens, preserving layout and comments of the property name and equals sign.

You might wonder where the commands for adding and removing top-level declarations are. The answer is that the `ListInsertCommand` and `ListDeleteCommand` are actually `MuliInsertCommand` and `MuliDeleteCommand` and work on any `muli`! They do not need to know about the token manipulators, because those are installed on the `mulis` as normal event listeners.

#### 5.4 Updating a token's column and line data

Part of the editor is the Outline View. This view shows the tree structure of the currently opened document, including top-level declarations, lists and properties (recursively). The selection in this tree always matches the caret's current position in the text. In order to offer this functionality, the editor needs to map from text position to tree position. We have enough information to do this: the token list allows us to map from text position to token position, and we can search the AST for the deepest element whose interval contains that token. However, the first step of this algorithm requires running time proportional to the offset of the caret.

To speed this up, we can store position information with the tokens. This way, we can skip the mapping from text position to token position and directly walk the tree looking for intervals that include a specific text position. We can store this position information in several ways. Two obvious ways are to store an absolute offset in the document (a simple integer), or to store line/column information. Which of those two is more convenient depends mostly on what information is needed most often. Either way, Eclipse's standard text editor maintains an efficient translation between these two representations, so for querying it does not matter much which one we choose.

For updating, though, there is a difference: imagine that due to some mutation some tokens are inserted, but these tokens are all on the same line. If we store an absolute offset, all following tokens until the end of the document have to be visited to have their positions updated. If we store line/column positions, we will find that because the mutation affected just one line, the positions of the tokens on subsequent lines are still up-to-date. In other words, when updating locations, we can stop as soon as we notice that the new, updated location of a token equals the old one. Figure 8 shows part of the algorithm.

#### 5.5 Tokens for fresh AST elements

So far we have only manipulated AST elements whose tokens were constructed by the parser. When we create fresh AST elements, there is no original source code for them, and hence no tokens exist yet. Yet when we make them part of the main document AST, the token manipulators will ask these new elements for their tokens. How to solve this?

```

class Token {
    ...

    int beginLine, beginColumn;
    int endLine, endColumn;

    void recomputeSucceedingTokenLocations(boolean smart) {
        Token preceding = this;
        Token t = next;
        while (t != null) {
            boolean changed = t.recomputeLocation(preceding);
            if (smart && !changed) {
                // Since this token's location didn't change, all upcoming
                // tokens' locations won't change anymore either, so we can stop
                // here.
                break;
            }
            preceding = t;
            t = t.next;
        }
    }

    boolean recomputeLocation(Token preceding) {
        if (specialToken != null) {
            // The special token is the actual preceding token. Let the special
            // token recompute its location first.
            specialToken.recomputeLocation(preceding);
            preceding = specialToken;
        }

        // Note the operator "|": both methods are always executed, in order.
        boolean changed = recomputeBeginLocation(preceding)
            | recomputeEndLocation();

        return changed;
    }

    ...
}

```

Figure 8: Tokens keep track of their location in the document. Part of the algorithm for recomputing locations is shown.

We will have to generate proper tokens for these elements on the fly. Therefore, apart from references to a token interval, each AST element also contains a prettyprinting method for constructing new tokens. Whenever an element's token interval is requested, the element checks if the interval is available already. If it is, it simply returns it; otherwise, it quickly generates them before returning them.

```

abstract class Expression {

    TokenInterval tokens;

    TokenInterval getTokenInterval() {
        if (tokens == null) {
            tokens = generateTokens();
        }
        return tokens;
    }

    abstract TokenInterval generateTokens();

    ...
}

class EList extends Expression implements Multi<Expression> {

    @Override
    protected TokenInterval generateTokens() {
        TokenInterval ti = new TokenInterval(new Token("["]);
        ti.append(new Token("]"));
        return ti;
    }

    ...
}

```

Figure 9: Part of class Expression and EList, demonstrating the lazy creation of tokens for empty elements.

This lazy behaviour is very convenient, because it allows the algorithms for constructing the new tokens to be significantly simpler: each element only has to *generate tokens for the trivial, empty case*. For example, a list only cares about creating the tokens needed for an empty list (`[]`), and an object only creates the tokens for an object with no properties (`ClassName {}`). Figure 9 shows the implementation of the token generation for list expressions.

Only trivial elements have to be pretty-printed from scratch, because all non-trivial cases will be created by starting with a trivial case and performing mutations on it until the desired value is reached. When the first AST mutation on a trivial element is executed, the installed token manipulator updates the token list accordingly, and in doing so requests that element's token interval. In other words, the first time a fresh element is asked for its tokens is always when that element is still in trivial state. Furthermore, each mutation knows how to update the token list in both trivial and non-trivial cases.

## 5.6 Parentheses

Parsers usually discard parentheses, simply yielding the value between them. Could this cause any trouble?

If the parser indeed discards them, the parent AST element will not know there are tokens for them without inspecting the token list. Imagine a list with one element: an integer. And just to tease we surround that integer by (unnecessary) parentheses. Then we tell the AST to delete the integer from the list. The list's token manipulator asks the integer for its token interval, cuts it from its context and is content. The token list however now reads `[()]` — incorrect syntax! — while we were expecting `[]`.

What if the parser, when reading `( expr )`, reset `expr`'s interval to include the parentheses? Then the above scenario would work out okay. But now the node itself will not know if and by how many parentheses it is surrounded. Recall how we inserted an element into an empty list: we inserted the new element's tokens after the opening bracket of the list. A naive list might assume that to reach its opening bracket it simply needs to follow the `begin` pointer of its interval, but now this is no longer valid as that pointer might point to an opening parenthesis instead.

The reason we are in trouble here is that, just like with whitespace and comments, the parser is throwing tokens away again. And so the solution seems obvious: do not throw them away! Instead, introduce parentheses as explicit nodes in the AST. Now parentheses tokens will never be a surprise anymore; instead, this problem has been lifted to the AST: everywhere a child expression appears, it could be a parentheses node. Furthermore, code that manipulates expressions where priorities matter now need to be very careful: it has become easy to create ASTs that are invalid. For example, the tree on the left in figure 10 is invalid, because it could never be generated by the parser and therefore the equation `parse ◦ print = id` cannot hold anymore. Invalid trees, however, can be easily detected, and detecting invalid parts and making them valid can be done as a last step, right before the source is generated from the token list.

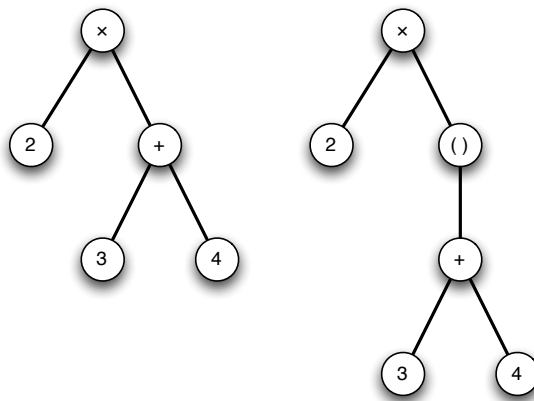


Figure 10: An invalid AST (on the left) can be made valid by inserting a parentheses node at the correct location (on the right).

## 6 Comparison with Proxima

Proxima is a presentation-oriented editor developed by Martijn Schrage at Utrecht University. It is explained in great detail in Martijn's thesis [Sch04]. Proxima is more powerful and complex than the editor we described here, because it is generic over the models it can edit. Specific parsers and presentation schemes can be plugged in to make Proxima suited for specific domains. Still, Proxima shares many challenges with our editor. Some of them are solved in the same way, while

others have been tackled in a different way. And while our editor is implemented in Java, Proxima is written in Haskell. Two important implementation differences are:

- Proxima reserves explicit fields for the (non-whitespace) tokens that do not belong to child elements in the AST nodes. Like in our implementation, each token may contain some "special" tokens. While this clutters the AST, there is also an advantage: no manual manipulation of the token list is necessary, as the tokens come and go automatically with the AST nodes. Here, too, nodes know how to generate their tokens if they have not been set yet (for example, when they do not originate from the parser). It might be possible that this different approach stems from Proxima's functional environment: there is no explicit way to manipulate references in Haskell, and our approach heavily depends on references.
- Mutations are not necessarily captured in command objects in Proxima. Our command objects were useful because of their modularity when implementing mutations on the token list and for supporting undo, but the former reason is no longer valid (see previous point), and the latter is done by simply keeping a list of complete ASTs. This is not such a bad idea as it may seem, because large parts of the AST maybe be (automatically) shared by the compiler.

## 7 Conclusion

Requiring source code generated from an AST to be as similar as possible to the original source code from which the AST was parsed in the first place is an interesting problem that has significant consequences on the design of the parser and model. While there are different ways of tackling this challenge, they share common ideas: do not let the parser throw away information! Instead, make all layout information part of the AST somehow, and update it as the AST's structure itself is updated.

## References

- [Sch04] Martijn M. Schrage. *Proxima – a presentation-oriented editor for structured documents*. PhD thesis, Utrecht University, The Netherlands, Oct 2004.
- [Wik08a] Wikipedia. Command pattern — wikipedia, the free encyclopedia, 2008. [Online; accessed 27-May-2008].
- [Wik08b] Wikipedia. Composite pattern — wikipedia, the free encyclopedia, 2008. [Online; accessed 27-May-2008].
- [Wik08c] Wikipedia. Model-view-controller — wikipedia, the free encyclopedia, 2008. [Online; accessed 27-May-2008].