Zuse: A Structured Source Code Editor with Improved Usability

Chengbin Gao

June 21, 2016

Abstract

Computer programs can be edited by either text editors or structured editors. Many researches believed that structured editing can be eventually more natural and efficient for programming since program source code is originally highly structured. However, traditional text editors still remain the main trend since structured editors have suffered from usability problems.

This thesis regards structured editors as the next trend and is meant to improve their usability. The approach consists of designing an editing oriented tree structure, introducing a stack-managed Vi-like multi-mode user interface, and providing a system of macros to make tree manipulation intuitive. A structured editor application called Zuse[1] was developed to demonstrate these ideas.

1 Introduction

Computer programming languages are usually based on context free grammars. The source text is perfectly nested and the structure behind the text is a tree. Structured editors are program source editors that directly modify the underlying tree structure instead of its text representation. They are usually safer but less flexible than text editors, which makes them conceptually attractive but practically unusable. This thesis tries to provide both a safe and flexible structured editor.

Structured editors are safer than traditional text editors. Text editors force programmers to care about syntax details and manipulate them carefully. It is common to see errors caused by syntax in day to day programming work. Syntax mistakes cause both static (compile time) syntax errors and dynamic (runtime) logic errors. The static errors are caused because text editors give users redundant freedom such that arbitrary text can be input. The dynamic errors can be made by misnesting parenthesis or misusing punctuation without

[1] Zuse is recursively defined as "Zuse Usable Structured Editor". It is also from the name of Konrad Zuse, the designer of the first high-level programming language Plankalkül. [1]

conflicting the grammar. Both errors, especially dynamic errors, significantly annoy the programmer and damage the development efficiency. Structured editors can prevent all static errors and a subset of dynamic errors from the very beginning.

Structured editors are usually less usable than text editors. The reason can be simply summarized as that a change of code by a structured editor requires more finger and/or brain efforts than by a text editor[11]. There is no evidence to prove that the concept of structured editor itself has any unavoidable shortcoming, and the situation seems like that there is just missing an acceptable design.

2 Existing Structured Editors

The research of structured editors has lasted for more than 40 years and there exist many implementations. Here four of them are summarized in order to characterize the history in a perspective of usability.

Emily[6] was a classical structured editor that generates the source code along the grammar derivation rules. MENTOR[3] provided a different approach that the user modifies the source code by a command-line language MENTOL. ALOE[10, 5] had a much higher usability by providing a large set of operations and extendable programability. The Cornell[13] program synthesizer improved the usability by restricting structured editing to big statement-level structures and making expression editing base on text editing.

From the Cornell program synthesizer on, structured editors have tended to compromise with text editors. Tree editing has been applied to only structures like `if` or `for`, and the tiny messy parts such as expressions has been edited by plain text. [11, 9, 2, 7]

Using tree operations to edit statement level `if`'s or `for`'s does improve the development efficiency, but Zuse pays more attention to expression level structures since they are important sources for errors. One typical example can be:

```
if (x && f(z, a.b().c(g + f(d(e(), k((t)h, j((t)m)))))) ...
```

In this example, any modification should be done carefully, and a mistake can be made anytime within such an expression.

3 Overview of Zuse

Zuse is a structured editor for the Java language. As K. Osenkov pointed out, implementing a structured editor for an existing language is more realistic than making an editor or editor generator for arbitrary language or a newly invented language[11]. Therefore Zuse is deeply fit into the Java language and every language relevant logic is hard-coded. The language Java is chosen since it is widely used[14] and its grammar is simple enough for an experimental project.

Zuse aims to be both safe and usable. The safety is enhanced by orienting the language. User's operation is limited by the Java grammar and the perfect-nestedness is enforced during the whole editing process. All syntax matter is automatically handled. Semicolons, commas, parenthesis, spacing, indentation, and other keywords and punctuation just show up at the right place. Those are common in any structured editor. The starring features of Zuse is how it improves the usability. There are three usability improving features:

(i) An underlying tree structure specialized for editing

(ii) Stack-managed Vi-like multi-mode user interface

(iii) Macros facilitating intuitive bottom-up code generation

4 Feature I: The Edit Tree

Unlike many other structured editors, Zuse does not share the tree structure with the compiler's parser. The Zuse *edit tree* is specialized for human modification. Making editors share a tree with compilers saves computational power; however, regarding that programming is usually not a resource intensive task, Zuse drops considerations on computational efficiency in favor of the user interface design. Besides, a stand-alone tree decouples the editor from the compiler or the IDE[2], which makes it more portable as an independent module. Figure 1 shows the structural differences in these approaches.

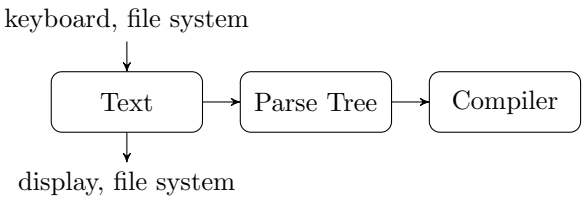
4.1 Node Classes and Node Types

To specialize the tree for editing, several changes are made such that the tree does not directly map to the original Java grammar. Nodes with different topological characteristics are classified into different *node classes*, and nodes of same class can have different *node types* according to their syntactical roles. A full list of node types is provided in Appendix A, which explains what a type of node represents.

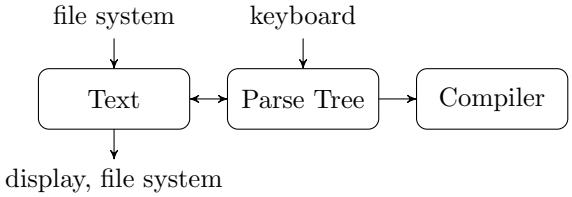
In order to improve the readability, we use SMALL CAPITAL font to mark the node types out regarding that they are named by common words. We also use BOP and DECL to denote "binary operator" and "declaration" to avoid long capital runs.

Figure 2 shows relations between all node classes. On the top level, the nodes in the edit tree can be either *scalar* or *internal*, where internal nodes can be either *fix-sized* nodes or *list* nodes, and list nodes have a subset called *binary operator list* nodes. The node classes are similar to the internal representation of ALOE[10], with the additionally introduced binary operator list node as well as many detailed differences in each class of nodes.

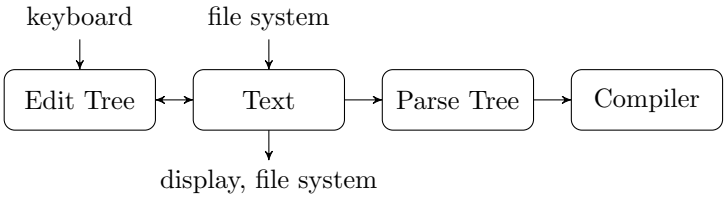
[2]IDE: integrated development environment, a software that integrates editors, compilers, debuggers, profilers, and other programming related tools.



(a) Text editor



(b) Structured editor with shared tree



(c) Structured editor with independent tree

Figure 1: Different types of editors

4.1.1 Scalar Node

Scalar nodes are leaves in the edit tree, which correspond to either a terminal symbol in the abstract syntax tree or a node of type META.

As an abstract syntax tree leaves out all auxiliary[3]punctuation and keywords, the terminal symbols correspond to either constant literals, identifiers, and some special keywords. Constant literals refer to strings, number, or `true`, `false` and `null`; and special keywords refer to `this`, `super`, or keywords representing primitive types like `int`, `long`, or `float`.

In the edit tree, we use three types of nodes to represent those terminal symbols — IDENT, STRING, and NUMBER. STRING and NUMBER represent string and number literals, while any other terminal symbol goes to the IDENT type. This discrimination is done by their different way of editing. A string consists of any character, a number consists of digits, and an identifier consists of a set of legal characters defined by the syntax. These three types of scalar nodes have their

[3]To help the code text to form a structure without giving information independently by itself.

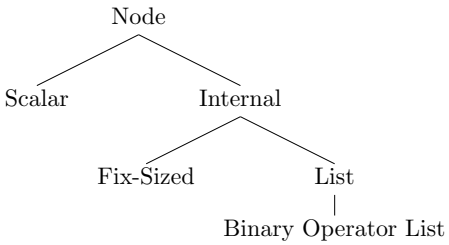


Figure 2: Node classes

own corresponding modifying mode, namely string input mode, number input mode, and identifier input mode. (§ 5.3)

A scalar node of type META[4]is a placeholder in the edit tree. It takes place of a node that has not been completely created in order to keep the whole tree structurally legal. META nodes are beginning points of a bottom-up tree generation, and a META node will be created whenever a new node can possibly be one of multiple types.

4.1.2 Fix-sized Node

Fix-sized nodes represent those non-terminal symbols in abstract syntax that have invariant number of children. Most of class-level and statement-level structures of Java is in this node class, such as `class` declarations, `for` statements, or variable declarations. Some of the binary operators also belong to this node class, like equality (`==`) or bitwise boolean operators (`&`, `<<`, etc.). It is noteworthy that the most frequently used binary operators, like addition (`+`) or method invocation (`obj.f(x)`), are not fix-sized nodes with size 2 but binary operator list nodes; and the `if` statements are also represented by list nodes instead of any fix-sized node. (§§ 4.1.3 and 4.1.4)

Fix-sized nodes of type DECL CLASS, DECL INTERFACE, DECL METHOD, CONTINUE, BREAK, RETURN, NEW CLASS, and NEW ARRAY have *hideable* children. This is because those nodes may or may not have a child at a specified position. For example, a class declaration can optionally inherit from a specified base class by writing `extends`, and/or optionally implement some interfaces by writing `implements`.

```
class A { }                              // none
class B extends A { }                    // 'extends' only
class C implements I { }                 // 'implements' only
class D extends A implements I { }       // both
```

Therefore, a node of type DECL CLASS may be of length 2, 3 or 4, and their exists two kind of class declarations of length 3. Without providing multiple types of class declaration nodes, we fix the node size to 4, and allow the `extends` and/or the `implements` child to be hidden. With hiding or showing those hideable

[4]The name *meta* takes from ALOE[10]. The similar concept is sometimes referred to as *non-terminal node* , *placeholder*, or *marker*. [6, 12, 13, 3]

children, the user can change a class declaration among the four types anytime. The same logic applies also to `throws` part in method declarations, returned expressions in `return` statements, target tags in `continue` or `break` expressions, and initializer in `new` expressions for classes or arrays.

Declarations of classes, class members, method parameters, and local variables can be preceded by modifiers like `public`, `static`, `final`, etc. These modifiers are not topological children of the corresponding fix-sized trees but their attached properties. They can be toggled on or off by the *modifier toggling mode* (§ 5.6).

4.1.3 List Node

List nodes are the complement of fix-sized nodes, which contain a variant number of children. List nodes typically have node types of MEMBER LIST, STMT LIST, DECL PARAM LIST, etc. The user can insert or delete an arbitrary number of children within those list nodes.

Different from the original Java grammar, statement nodes are always children of STMT LIST nodes. In the original grammar, statements are expanded by *compound statement*[4] — adding brackets around multiple statements makes them syntactically one single statement such that they can be put anywhere a statement can appear.

```
while (true)     // single statement as body
    eat();

while (true) {   // compound statement as body
    eat();
    sleep();
}
```

This language rule results in that the user has to put brackets whenever multiple statements are in a body. Although the user can simply always type brackets even there is only one statement, writing a single statement without bracket looks less verbose so that this style has been preferred by many coders. Then problems will occur when a single statement becomes multiple or vice versa. It takes time to add or remove brackets and errors will be caused if any mistake is taken.

```
while (true)
    eat();
    sleep();     // out of while body
```

Therefore, Zuse makes the bodies always a STMT LIST node, and brackets are automatically shown or hidden according to the number of statements inside.

It is noteworthy that the `if` statement is also represented by a list node. Usually people use `if` statements like this:

```
if (fish)
    swim();
else if (dog)
    bark();
```

```
else if (human)
    write(code);
else
    sleep();
```

Although the code is formed by multiple nesting `if-else` statements, intuitively it looks like a big list, and it is common to perform list operations — insert or delete conditions and their bodies — in such a list. Thus Zuse treats `if` a list that can have children of type IF CONDBODY or STMT LIST. A IF CONDBODY will be displayed as "`if (condition) body`", and a STMT LIST displays as is. The `else` keyword is the list separator for the `if` list, and a list separator always automatically appears or disappears between elements in a list node. Within an IF LIST, all STMT LIST will be surrounded by brackets if and only if at least one of them contains multiple statements. This is done because surrounding only a part of statements in an IF LIST results in a "super ugly" code style.

The similar rule applies also to `try` statements. They look like this:

```
try {
    go(shopping);
} catch (NoMoneyException e1) {
    complainAbout(e1);
} catch (NoTimeException e2) {
    complainAbout(e2);
} finally {
    go(home);
}
```

A node of type TRY LIST can have children of type STMT LIST or CATCH. Here a STMT LIST may represent either a `try` block or a `finally` block. The keyword `try` is always displayed before a TRY LIST since it is obligate, and the keyword `finally` will be displayed before a STMT LIST if and only if it is not the first child. A CATCH represents a `catch` block, which is a fix-sized node having two children: the exception declaration and the body statements.

Usually a list node can contain arbitrary number of children, but some list nodes become ill-formed if their number of children satisfy a certain condition. There are two such conditions — *ill-one* and *ill-zero*. Ill-one or ill-zero means that a list node has one or no child while this conflicts the Java grammar or the Zuse tree structure rules. Thoughtfully dealing with, and also making use of ill-one and ill-zero list nodes is important to this editor such that the term ill-one and ill-zero will be referred several times within this paper.

All binary operator list nodes are typically ill-one since they must have at least two operands. An example for ill-zero can be the TRY LIST:

```
try { go(shopping); }   // ok
try { }                 // ok
try                     // wrong: requires at least one child
```

4.1.4 Binary Operator List Node

Binary operator list nodes represent binary operators that are of same precedence and are often concatenated as a long expression. One most typical example is the addition (`+`) operator. It is definitely a binary operator, but people

usually write a long expression like "`a + b + c + d`". This looks like a list of four elements in human's intuition, but syntactically it is a deeply nested binary tree. (Fig. 3)

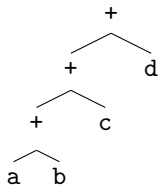


Figure 3: Tree form of `a + b + c + d`

Problems occurs when the expression needs to be modified. It is not trivial to find out which branch in the tree should be cut and how the new node should be linked. More brain effort would be required if this long expression were represented in a deeply nested binary tree. Therefore, they are always flattened into a plain single level list node unless the user explicitly increases the depth, in order to enable intuitive list insert and append operation.

Since high level languages are vivid and machine instructions are stiff, compilation is considered as a process of losing information. However, it is not true that the user knows more than the machine. The compiler generates the parse tree according to not only the source program but also a full set of language rules. An ordinary programmer[5]only remembers a subset of the language rules and forgets those less remarkable points. Therefore, it may cost the user more effort to face the parse tree instead of the plain text, and this is why the Zuse edit tree tries to contain no more information than the plain text source code. Making binary operators plain list relieves the user from having to remember whether the binary tree is left-first or right-first derived and inserting or removing elements in a long chain of binary operators becomes as simple as inserting or removing plain text.

Structured editors typically avoid making a list structure into a recursively self-nesting binary tree as the grammar definition does. Zuse advances one more step such that it tries to make nodes that were originally regarded as binary — such as binary operators — into plain lists.

All binary operator list node are defined ill-one. (§ 4.1.3) This affects behavior of the editor when remove operations or macros are triggered on the list. (§§ 5.1.4 and 6.2)

4.2 Reduced Complexity

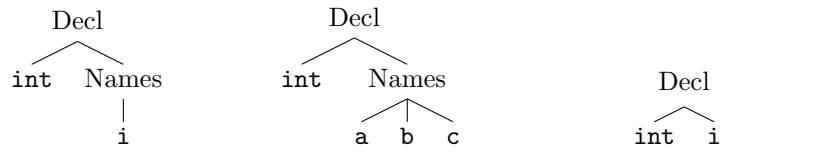
Compiler parsers do not care about the ease of modification of the tree, and they prefer to unify data structures among the whole tree in order to simplify processing. On the contrary, structured editors can break this uniformity to improve the modifiability.

[5]Except compiler makers and crazy people.

In the Zuse edit tree, many invisible or less-visible internal nodes are eliminated. For example, the Java variable declaration can declare either one or multiple variables in one statement.

```
int i;          // one variable
int a, b, c;    // multiple variables
```

In a machine-processing point of view, the declaration statement should be a binary tree whose left child is the type specifier and right child is a list of names[6]. (Figs. 4a and 4b) This reduces the complexity of compilers since they can deal with different length of name lists simply by a loop statement. However, this tree structure inconveniences the user of a structured editor if the cursor points to the character "`i`". The problem is that the user would not know whether the cursor points to the variable name itself or the list nesting it[7].



(a) One variable (b) Multiple variables (c) One variable (reduced)

Figure 4: Tree structure of variable declarations

There are two solutions for this problem. One is to leave the tree structure unchanged and design a set of algorithms such that the editor always does the right thing whether the cursor points to the name or the name list. This approach greatly increases the complexity of the editor's implementation. Another preferred solution is to automatically change a name list into simply a name whenever the size of the list is one such that the name becomes the direct child of the declaration. (Fig. 4c) Therefore, what the cursor points to in the underlying tree structure straightforwardly maps to what it points to in the screen. This automatic conversion from a single-element list to the nested single element is implemented by defining the nesting list *ill-one* (§ 4.1.3). When the user wants to add a name in the declaration, no distinction between name or name list is needed to be concerned by using the macros (§ 6).

The automatic conversion between a name and a name list is an example for eliminating invisible internal nodes. An typical example for eliminating less-visible nodes can be the integration of statements and expressions. In Java, an expression plus a semicolon yields an expression statement, where the semicolon often troubles the programmer. It is common to move an expression statement into somewhere a pure expression should appear without semicolon and vice versa. To prevent the user from eliminating or appending a semicolon manually, the edit tree dropped the concept of expression statement and regarded an expression appearing inside STMT LIST directly as a statement. The semicolon

[6]Precisely "name" should be called as *declarator*, which means an identifier that can be optionally assigned by an initial value.

[7]Actually in Zuse the user can still distinguish this by looking at the position the the "outer node" (§ 5.1.1). However, this still remains a problem as moving the cursor from the statement to the variable requires double stroking the F key, which is inconvenient.

is added as a list separator automatically such that it will also be removed automatically when the expression is removed.

Besides the semicolon in STMT LIST, `else` in IF LIST and `finally` in TRY LIST are also examples of list separators. The automatic appearance or disappearance of those list separators helps a lot when clipping elements from and pasting elements to those list nodes.

5 Feature II: The Mode Stack

Zuse uses a Vi-like multi-mode and single-keystroke-based user interface. Unlike Vi, the modes are not switched among each other but pushed and popped on a *mode stack*. The mode on the top of the stack becomes the activated one, and the other modes below the top represents those editing tasks that are temporarily suspended. The mode stack helps (but does not force) the user to think and operate in a depth-first-ordered workflow.

The modes are stateful objects. They record states like the cursor's position so that the cursor can be placed on the right place whenever the top of the mode stack changes.

Sometimes the top of the mode stack cannot handle a key stroke. Then the mode will pop itself and the key stroke will be *thrown* to the next stack top. This mechanism is for handling *macros* in most cases. (§ 6)

5.1 Normal Mode

Normal mode is the fundamental mode for cursor motions as well as non-constructive (moving or deleting) editing operations, and it is also the entrance of other modes. An instance of normal mode is pushed onto the stack when Zuse starts up and it remains unable to be popped.

5.1.1 Cursor Motion

As a structured editor, the cursor points to a node, either scalar or internal, instead of a character. The node under the cursor is displayed as a region with background of highlighted color. The parent node of the node under the cursor is also highlighted but with a different color. In order to facilitate the expression, the node under the cursor will be referred to as *the inner node*, and the parent node of the inner node will be referred to as *the outer node*. The type of both the inner mode and the outer mode affects the semantics of a keystroke.

The *tree-wise* motions are the very basic cursor motions. They are *focus-in*, *dolly-out*, *slip-back*, and *go-on*. The name "focus", "dolly", "slip" and "go" are chosen for ease of memorizing their key strokes: "F", "D", "S", and "G". These four key strokes correspond to move-to-first-child, move-to-parent, move-to-previous-sibling, and move-to-next-sibling respectively. When "F" is pressed,

the cursor will move onto the first child of the inner node if the inner node is internal, or perform no action if the inner node is scalar. When "D" is pressed, the cursor will move onto the outer node if the outer node is not the root, or perform no action otherwise. When "S" or "G" is pressed, the cursor will move onto the previous or the next sibling node of the inner node. These sibling motions are circular, thus pressing "S" when the cursor points to the first child will bring the cursor to the last child, and vice versa for pressing "D". These tree-wise motions, focus-in, dolly-out, slip-back, and go-on, are suggested to be used under most of cases.

The normal mode also supports *screen-wise* motions. Sometimes the user wants to move the cursor from one screen position to another screen position, and the relation between that two positions in the tree structure may not be trivial. In this case the user can use the Vi-style "K", "J", "H", or "L" to move the cursor up, down, left or right. All the screen-wise motion makes the cursor point to a scalar on the specified direction. When "J" or "K" is pressed, the "closest" scalar below or above the inner node will be selected. Since "J" and "K" are vertical motions, the vertical row distance is always considered at first. If multiple scalars are available in the closest row, the scalar having the closest "horizontal center" will be selected, where the "horizontal center" is the average of the beginning and ending column index of the scalar text. "H" and "L" will position the cursor onto the previous or next scalar node when the inner node displays within a single line (not necessary to be a scalar node). "H" and "L" will take no action if the inner node displays as multiple lines,

Supporting tree-wise motions and screen-wise motions simultaneously is a starring feature of Zuse. In a QWERTY keyboard, the keys for tree-wise motions, "S", "D", "F", and "G", lay directly under the left hand; and the keys for screen-wise motions, "H", "J", "K", and "L", lay directly under the right hand. This design may addict a user such that he or she cannot switch to other tools anymore.

There is another important type of motion called *big motion*. In most text editors, besides the basic up, down, left, and right motions, cursor can move to "far" positions such as the beginning of the line, the center of the screen, or the end of the file. Big motions is a similar feature that enables the user to quickly go to a "far" place at anytime and from anywhere. Unlike tree-wise motions and screen-wise motions, big motions are invoked by two key strokes. The first key stroke should be one of "S", "D", "F", and "G" (the keys for tree-wise motions) with the Shift key pressed, and the second key is taken from *type operands* (Appendix B). By those two key strokes, the cursor will move onto a node which the type operand indicates. The direction of the motion is defined by the first key stroke. "Shift+F" means to perform a breadth-first search downward along the tree from the inner node, while "Shift+D" means to perform a search towards to the root node. "Shift+S" or "Shift+G" means to search within the preceding or succeeding sibling node in a decreasing or increasing order. The type of the target node is specified by the second key, such as "Shift+M" stands for a DECL METHOD and "Shift+C" stands for a DECL CLASS. In addition to type operands, the second key can also be "S", which specifies a statement.

The most easy and frequently used big motion might be "Shift+D, S", which

bring the cursor to the statement level such that the user can browse along statements after editing an expression.

5.1.2 Insert and Append

Just like Vi, the keystroke "I" means insert and brings the user to a mode for new content generation. "I" can be pressed when the outer node's node class is list, When "I" is pressed, a new node will be created as the preceding sibling of the inner node. The type of the new node is determined by the type of outer node. For example, if the outer node's type is DECL PARAM LIST, the new node's type will become DECL PARAM; and if the outer node's type is CLASS LIST, the new node's type will become DECL CLASS. While those node can have different types of children, like STMT LIST, in most cases the new node's type will be META that is prepared for a bottom-up tree generation. After the creation of the new node, a mode for editing the new node will be pushed onto the mode stack.

If "I" is pressed when the outer node's class is a binary operator list, one more keystroke is required to determine the binary operator. This extra keystroke is referred to as *binary operator operand*, and a full list of them is available in Appendix B. Basically the binary operator operand is exactly the character of that binary operator. For example, hitting "+" on the keyboard will insert a term which adds with its succeeding sibling; or hitting "/" will insert a term which is divided by its succeeding sibling, Similar to other list nodes, inserting within binary list nodes also creates a new node and pushed its corresponding modifying mode onto the stack. In a binary operator list context, the new node's type will become META in most cases such that the expressions are generated in a bottom-up approach. One exception applies to the method call — the second child of that will become ARG LIST.

Similarly, pressing "O"[8]appends a new node after the inner node, and if the outer node is a binary operator list, the new node will operates with its preceding sibling.

Besides list nodes, insert and append operations can also be applied to fix-sized nodes when the inner node precedes or succeeds a hidden hideable node. In these cases, pressing "I" or "O" will make the hidden node show up and its corresponding modifying mode pushed. For example, given a method declaration:

```
public eat(Food lunch, Water drink) {
    ...
}
```

When the cursor points to the whole[9]formal parameter list, pressing "O" will make the hidden `throws` node appear.

```
public eat(Food lunch, Water drink) throws ?? {
    ...
}
```

[8]The key "A" is reserved for some features in the future version.

Then the user can type to edit the NAME LIST in the `throw` part.

In nodes of type DECL CLASS, their might be two adjacent hidden hideable nodes, which are the `extends` part and the `implements` part. In those cases, appending from the preceding class identifier will make the `extends` node show up and inserting from the succeeding member list will make the `implements` part show up.

If a fix-sized node that has only one child which is hideable, this hideable node is shown up by *punch* operation (§ 5.1.3) instead of insert or append operation.

5.1.3 Assart and Punch

Assart and *punch* operations share the key "F" with focus-in operations. The common point of those three kinds of operations is that they result the cursor in moving to the a child node of the inner node. Their difference is that focus-in operations do not create any new node while assart operations add a child in an empty list and punch operations make a hidden child show up.

Assart[10]is the way to create a very first child into an empty list node. Since in this structured editor the cursor always points to an existing node but any "null' positions, the cursor cannot move into an empty list and thus it is required to provide an operation to add the first child while the cursor is pointing to this empty list. After the creation of this first child, other child can be created by insert or append operation with the first child as a reference sibling.

Similar to inserting and appending, after pressing "F" on an empty list, a new node will be created and its corresponding mode for modification will be pushed onto the stack.

Ill-one and ill-zero list nodes (§ 4.1.3) do not require assart operations since their first (or even second) child will always be automatically created.

When the inner node is a fix-sized node that has only one hidden child, pressing F punches the hidden child such that it shows up. For example, when the inner node is a `return` statement with no return value, pressing "F" makes a new META node appears next to the `return` and therefore the user can type an expression for returning a value.

5.1.4 Remove

The remove operation is quite trivial — pressing "R" within a list will simply erase a child.

After removing, ill-zero and ill-one (§ 4.1.3) conditions are checked and related operations will be performed automatically. If the outer node is ill-one and

[9]Another parameter will be added if "O" is pressed while the cursor is *inside* the parameter list.

[10]To assart is to clear a forested land for farming. Here the assart operation opens up an unused empty list and "farm" new child into it.

there remains only one child in the list, an expose (§ 5.1.6) operation will be automatically performed; and if the outer node is ill-zero and there remains no child, remove operations will be recursively performed until the outer node is not an ill-zero list with no child.

For example, when we remove the "`b`" in the expression "`a + b`", the expression will become "`a +`", which is a typical ill-one since all binary operators should be "binary". Thus the sub-expression "`a`" will exposed (§ 5.1.6) from its parent "`+`" and finally the whole expression will become a single "`a`".

Remove operations can also be applied to fix-sized nodes. When "R" is pressed on a hideable but shown child, this child will be hidden and the node itself will also be deleted. In any other cases, pressing "R" in a fix-sized node will replace the node with a node of a default type and value. These default nodes are usually META nodes, and they can also be non-meta nodes in special cases, such as a member list within a class declaration.

5.1.5 Change and Modify

There are two ways to update a node: *change* and *modify*. To change a node, press "C" and the node will be replaced by a re-created node of proper type according to the context. Like inserting, a mode for modification will be pushed regarding the type of the newly created node. To modify a node, press "M" and a modifying mode for the inner node will be pushed immediately without generating any new node. Therefore, the change operation is to re-generate content and the modify operation is to modify the details. If "M" is pressed with the Shift key, the modifying mode will be push with the *clear* flag set. Then the old content of the inner node will be cleared on the next keystroke. (Similar to select a segment of text and press any character to override it in any text input box) The difference between "Shift+M" and "C" is that Shift+M does not change the type of the inner node while "C" might do since it is totally re-inputting.

5.1.6 Nest and Expose

Nest operations are also frequently used, which surround the inner node with a newly created outer node. A nest operation consists of two key strokes. The first key is either "N" or "Shift+N", and the second key is one of the *type operands*, which specifies the type of the nester node. A full list of type operands is available in Appendix B.

If the type of the nester is a fix-sized node of size one or a list node that is not ill-one (§ 4.1.3), pressing "N" or "Shift+N" usually makes no difference. The only exception happens to the increment and decrement (`++` and `--`) operators — "N" make it postfix and "Shift+N" makes it prefix. This is because "N" and "Shift+N" decides whether the nestee (the inner node) appears at left or right after nesting. This also applies to nester nodes of greater size — fix-sized nodes of at least length two or ill-one list nodes. In those cases, "N" makes the nestee the leftmost child of the nester, and "Shift+N" makes the it the rightmost

child. Within this paper, "N" and "Shift+N" will be named as *nest-as-left* and *nest-as-right* respectively.

If the nester's size is greater than or equals to two, modifying modes for construcing other children of the nester will be pushed onto the stack. The behavior is similar to any other operations like insert, append, assart, or punch.

The expose operation can be considered as the inverse of nest operations. The effect shows as Fig. 5. Pressing "X" causes the old outer node to disappear and the inner node becomes direct child of its previous grandparent. If the original outer node has more than one child, all the other child except the inner node will be deleted in this expose operation.

(a) Before (b) After

Figure 5: Effect of the expose operation

5.1.7 Yank and Clip

The power of reusing existing content dominates the power of an editor. Therefore, we prepared 26 clipslots in the clipboard and recommend these clipslots to be heavily used. Each of these clipslot is assigned a letter from A to Z, and a little boxed letter displayed on the top-right corner indicates the currently active clipslot. To switch between those clipslots, "P" should be pressed with the indicating letter following.

A yank operation is done by pressing "Y", which copies the inner node to the currently active clipslot. Whereas a clip operation can be done in three ways: "Shift+R", "Shift+C", and "Shift+X". "Shift+R" is the most trivial case — the inner node will be deleted with a copy of it goes to the clipslot. "Shift+C" is named as *clip-and-change* — a change (§ 5.1.5) operation will be done with the old content clipped to the clipslot. "Shift+X" is called *clip-and-expose*, and it works only when the outer node has exactly two children — the sibling node of the inner node will clipped to the clipslot before the expose operation takes place.

Paste operation, the counterpart of yank, is performed in tiny lexer modes (§ 5.2) instead of the normal modes.

5.2 Tiny Lexer Mode

Tiny Lexer Modes[11]are the most frequently pushed modes when nodes are to be modified. It is the corresponding mode for scalar nodes of type META.

[11]Lexer: lexical analyzer, a program that tokenize character stream.

Typically a tiny lexer mode pops itself immediately after receiving one key stroke, since this one key stroke usually determines the node type and thus the modifying node is no longer META. This one key stroke is taken from *tiny lexer operands*, and a full list of tiny lexer operands is available in Appendix B. Basically, tiny lexer operands do not need to be memorized deliberately because the tiny lexer mode works similar to a lexical analyzer. For example, if the key stroke is an alphabet, the tiny lexer mode will change the META node into an identifier node (type IDENT), make the key stoke the identifier's initial character, pop itself and immediately push an identifier input mode to wait for rest part of the identifier to be inputted. Same rule applies to literal constants — the META node will be changed into a NUMBER node or STRING node if the key stroke to the tiny lexer mode is a digit or a double quote (`"`), and afterward corresponding modifying modes will be pushed.

Being entrances for other scalar input modes is the basic role of tiny lexer modes. Another important role for them is to perform *character coupling*, which is a mechanism for inputting operators that consist of two or more characters, such as logical AND (`&&`), logical OR (`||`), miscellaneous assignments (`+=`, `-=`, `<<=`, ...), equality (`==`, `!=`), increment and decrement (`++`, `--`). If a tiny lexer mode receives a key stroke that is the non-first character of a multi-character operator, it will check whether the outer node has the matching type. For example, if the tiny lexer mode receives a `&`, it will check whether the outer node is of type bitwise AND (`&`) such that the two characters form a logic AND (`&&`). If the outer node is a bitwise AND, it will be changed into a logic AND. For another example, if the tiny lexer mode receives a `=`, it will check whether the outer node is of type ADD BOP LIST (`+` or `-`), MUL BOP LIST (`*`, `/`, or `%`)..., or ASSIGN (`=`) such that those two characters can form ASS ADD (`+=`), ASS SUB (`-=`), ..., or EQ (`==`), and the outer node will change its type if one of those matches.

An inequality expression such as "`a != b`" can also be generated by simply typing it as is. However, inequality (`!=`) is not generated by character coupling because its first character — logical NOT (`!`) is a prefix unary operator such that the `!` and `=` are not adjacent. After typing the `a` and `!`, we will get an `!a` since the `!` key is a macro (§ 6). As `!a` is a complete term without any meta node, there is no chance for a tiny lexer mode to be pushed. The mechanism to convert this "`!a`" into "`a != b`" is illustrated in § 6.

Basically, the Space key means to cancel and it pops the tiny lexer mode with deleting the underlying META node. However, an exception happens when the tiny lexer mode is pushed from an explicit nest-as-right operation (Shift+N, § 5.1.6). In this case, if the outer node is an ADD BOP LIST which has exactly two children (including the editing META), pressing Space will convert the outer node into a UNARY PLUS or UNARY MINUS according to the original operator. Therefore, to change an expression "`a`" into "`+a`", the required key sequence is Shift+N, +, and Space.

Tiny lexer mode is also the mode for pasting. Press Tab will copy the node in the current clipslot (§ 5.1.7) and use it to replace the modifying META node.

5.3 Scalar Input Modes

There are three kinds of scalar input modes in Zuse. They are *identifier input mode*, *number input mode*, and *string input mode*. One of these scalar mode will be pushed onto the stack when its corresponding type of scalar is to be modified.

The working logic of scalar input modes is to accept only those legal characters basically and map some of the illegal characters to auxiliary functions.

5.3.1 Identifier Input Mode

Identifier input mode will append a character to the modifying identifier when the keystroke stands a legal[12]character in an identifier. If the user stroke the space bar, the identifier input mode will be popped. If the keystroke is neither legal identifier character nor the space, the identifier input mode will be popped and the keystroke will be thrown to the next stack top. Typically that thrown key will be a macro that will be eventually handled by a fix-sized input mode or a normal mode. (§ 6)

A *scalar promotion* might be caused when an identifier input mode is explicitly popped by pressing Space. Scalar promotion is to convert an identifier into an internal node when one of the following the conditions are satisfied.

- The identifier is a Java keyword such that it should not be an identifier;
- The identifier seems like a type specifier within a local variable or member declaration.

In the first case, the convertion rule is straightforward. A `while` identifier will be converted into a `while` statement, an `if` will become an `if` statement, and same rule applied to `return`, `for`, and `do`. If the identifier is a modifier keyword, like `public` or `static`, the identifier will be promoted to a variable declaration.

In the second case, as the identifier seems like a type specifier, the scalar will be converted into a variable declaration with the scalar becoming the type specifier part. The predicate "seems like" is check by two steps. First, we check if the identifier is written in UpperCamelCase[13]and the outer node is able to include a variable declaration. Second, we check if the identifier is placed in a DOT BOP LIST where all the binary operators are member access such that it forms a *qualified name*[4], and the whole qualified name satisfies the previous first condition. For example, `MyClass` or `com.mysite.MyClass` are all considered as "seems like" a type specifier. This rule works in most cases thanks to the Java community strongly agree on that class names should be written in UpperCamelCase.

[12]Begin with `[A-Za-z$_]` succeeded by `[A-Za-z0-9$_]`[4]

[13]Here we define UpperCamelCase as a string with at least one lower case character whose initial character is capitalized. Since by this rule CAPITAL_RUN is distinguished away, single upper case character will not be considered as UpperCamelCase though.

After a scalar promotion, the corresponding fix-sized node input mode will be pushed onto the stack for continuing editing the promoted node. The offset property (§ 5.4) may be set to a non-zero value to avoid re-inputting type specifier when the promotion generates a variable declaration.

Scalar promotion is one important part to enable a natural bottom-up code generation in this structured editor.

5.3.2 Number Input Mode

Similar to identifier input mode, the number input mode appends character after the modifying number scalar when the keystroke maps to a digit. Besides the number keys, "M", "J", "K", "L", "U", "I", and "O" can also be mapped to a digit. The mapping is shown in Fig. 6. By these extra mappings, the user can quickly input numbers with the right hand just like dialing on the numeric keypad. This is extremely useful when the user inputs a long number like `3.141592653589793238462`6. After inputting the number, naturally pressing a space can pop away the number input mode.

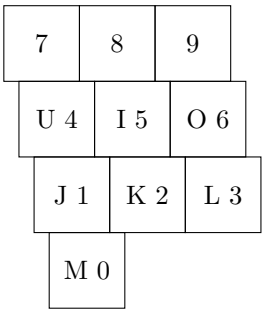


Figure 6: Extra number mapping

5.3.3 String Input Mode

Unlike other scalar input modes, the string input mode accepts all displayable character keystroke and appends it to the end of the modifying string, including the space. The string input mode can be popped by pressing a double quote (") key. However, if the number of suffixing backslash (\) is odd, the double quote will be appended to the string since it should be escaped.

5.4 Fix-Sized Node Input Mode

Fix-sized node input modes correspond to fix-sized nodes. As the source code is eventually constructed by terminal symbols, fix-sized node input modes do not generate content by themselves but by pushing other modes onto the stack. If that newly pushed mode is also an mode corresponds to an internal node, the

pushing will recurse until a scalar input mode is pushed. Therefore, fix-sized nodes are basically generated in a depth-first order.

A fix-sized node input mode is pushed with a parameter *offset* specified. The offset parameter indicates from which child of the modifying fix-sized node should the recursive pushing begin. In most cases, the offset is zero such that the tree expansion begins from the first child. A non-zero offset value works together with macros (§ 6) or scalar promotion (§ 5.3.1), and this is for skipping those already completed children.

After pushing a fix-sized node input mode, the corresponding mode for modifying the child at the position specified by the offset parameter will be push onto the stack immediately. As now the fix-sized node input mode is not the stack top, the user controls flow into the new stack top and the fix-sized node input mode waits for the stack top's popping. After the new stack top is popped, the fix-sized node input mode moves the cursor onto the next sibling of the fix-sized node, pushes the next mode onto the stack, and waits for becoming the stack top again. This repeats until all the child is modified by their corresponding mode, and when the mode for the last child is popped, the fix-sized node input mode pops itself. This is a typical depth-first ordered node creating process, where the fix-sized node input mode does not respond to any user key stroke directly.

There are some exceptions where the user can jump out of the depth-first workflow. The first exception happens when the fix-sized node input mode pushes a mode for modifying a child belongs to list node class. In this case the fix-sized node input mode does not push the modifying mode immediate but stops and waits for the user's instruction. Here three key strokes are legal: "F", Space, and "S". If "F" is pressed, the list node child will be actually edited and the fix-sized node input mode will pop itself meanwhile. If Space is pressed, the list node child will be skipped and the fix-sized node input mode moves onto the next sibling. If "S" is pressed, the fix-sized node input mode will pop itself with placing the cursor back to the previous sibling node.

This design is regarding that list editing usually takes a long time and the user will probably forget about the outer fix-sized node. Therefore in these cases we break out the workflow from the depth-first generation.

A fix-sized node input mode will not care about hideable children and all the hideable children will be skipped during the depth-first generation. They can be additionally added after the fix-sized node input mode pops itself.

Here is one get-all-together example to show how a fix-sized node input mode works. Suppose that a fix-sized node input mode for modifying a CLASS DECL (class declaration) is pushed onto the stack. Now this mode will immediately push a tiny lexer mode for inputting the class identifier. Then the user will input the identifier and press Space to pop the new mode away. As the fix-sized node input mode becomes the stack top again, it advances the cursor to the next sibling. However, the next two siblings — `extends` and `implements` are skipped without pushing any related modifying mode as they are hideable children. Therefore, the cursor will continue advancing and stop at the MEMBER LIST. This time no modifying mode will be pushed immediately since the list

node is a special case. Now the user has three options — pressing "F" to continue edting the MEMBER LIST, pressing "S" to slip-back to the identifier, and pressing Space to abort edting this CLASS DECL. Each option will pop the fix-sized node input node away, and if "S" or Space is pressed, the user can press "O" or "I" to edit the hideable nodes (§ 5.1.2).

5.5 List Node Input Mode

List node input modes are the corresponding modifying modes for list nodes. These modes accepts no user input but just simplify and unify the working logic of the editor. After they are pushed onto the stack, they immediately add a child to the modifying list node, pop themselves, and push a mode for the newly created child.

The node type of the newly added child is determined by the parent list node. In most cases, the type will be META such that the user can perform a bottom-up generation. There are only three exceptions — the child of DECL PARAM LIST will be DECL PARAM, the child of IF LIST will be IF CONDBODY, and the child of TRY LIST will be STMT LIST.

5.6 Modifier Toggling Mode

In the Java language, classes, interfaces, methods, and variables can be declared with *modifiers*. As described in § 4.1.2, the modifiers are stored as properties of the nodes instead of their children. Modifying the modifiers requires to push a *modifier toggling mode* from the normal mode. The color of the cursor will turn green when a modifier toggling mode is on the stack top.

The Java language has 11 keywords of modifiers in total. They are `abstract`, `final`, `static`, `native`, `synchronized`, `transient`, `volatile`, `public`, `protected`, `private`, and `strictfp`. The `public`, `protected`, and `private` are all for specifying access controls such that they cannot appear at same place. In a modifier toggling mode, pressing P can toggle a declaration between "nothing", `public`, `protected`, and `private`[14]. Besides the access modifiers, all the other keywords are just simply "to be or not to be". Pressing their corresponding key will switch the modifiers on or off. A list of key mapping is in Appendix B. After determined the modifiers for a declaration, pressing the space bar will pop the modifier toggling mode out of the stack.

Local variables and formal parameters of methods can be declared only with the `final` modifier. Thus pressing M in the normal mode when the cursor is pointing to these nodes will quickly switch the `final` keyword on or off without pushing a modifier toggling mode. In other types of declarations, a modifier toggling mode will be pushed and the user can switch on any modifier. This operation might result in a syntax error. For instance, declaring a class with `native` is illegal to the grammar. This is one example that Zuse sacrifices

[14]The key P is chosen because these modifiers all starts with P. The keys for other modifiers are also chosen in a similar way.

syntax safety to improve the flexibility. Syntax issues that are irrelevant to the perfect-nestness are generally dealt loosely and roughly.

6 Feature III: The Macros

A *macro* is an operation that contains a series of other basic operations and conditional checks in order to perform complex tasks with one or a few of key strokes. Macros are usually used for generating new contents in a bottom-up workflow. The design principle of macros is to simulate sequential text inputting that users have been used to. The system of macros solves the problem that many structured editors force the user to input content in an unfamiliar top-down approach[8].

For example, when the cursor points to a variable "`a`", sequentially type "+b" will nest the "`a`" with a new node of type ADD BOP LIST whose right child is "`b`". Then type "+c" will append a child "`c`" to the binary operator list such that the whole expression will become "`a + b + c`". In this example, the key "+" is the macro. The meaning of this "+" depends on its context — when we pressed the "+b", it became a nest operation; and when we pressed the "+c", it became an append operation. The details for how a meaning of a macro key stroke is determined is illustrated later. In one word, straightforwardly type "a+b+c" will get an expression "`a + b + c`".

Generally a macro can be triggered from any mode. If the top of the mode stack is a scalar input mode when a key for macro is pressed, the mode stack will pop its top until the top becomes a fix-sized node input mode or a normal mode, and afterwards the macro will take effect. In the former example, the identifier "`b`" was input in a tiny lexer mode pushed after nesting "`a`" with a "`+`". After typing "b", the tiny lexer mode recognized it as an identifier, popped itself, and pushed an identifier input mode to wait for the rest characters of the identifier to be inputted. However, we typed the macro "+" instead of continuing inputting the identifier, so the identifier input mode got popped and the macro took effect after the top of stack became the normal mode. As the macro worked, a scalar node with type META was appended to the addition list, and another tiny lexer mode was pushed for modifying the META node.

Macros can be triggered from any part of the source code, and nodes of different types correspond to different sets of macro operations. Those different macros can be categorized into two groups — statement level macros and expression level macros.

6.1 Statement Level Macros

The most frequently used macro key may be the Enter key. Pressing Enter inside a node of type STMT LIST appends a META node after the statement that contains the inner node. If the inner node is nested by multiple STMT LIST, this change will be taken to the innermost one; and nothing will happen if the inner node is nested by no STMT LIST. Similarly, pressing Shift+Enter inserts a META

node before the statement containing the inner node. Intuitively, Enter adds a META node below the current line, and Shift+Enter adds a META above the current line. After pushing the META node, a tiny lexer mode will be pushed for the bottom-up statement generation.

Macros are also frequently used in list-formed `if` blocks and `try` blocks. The common point of `if` and `try` is that the blocks can contain either "`(...){...}`" form or "`{...}`" form.

```
if (time.hour == 12) {              // a (...){...} form
    eat(lunch);
} else if (time.hour == 18) {       // a (...){...} form
    eat(dinner);
} else {                            // a {...} form
    play(game);
}

try {                               // a {...} form
    play(game);
} catch (NoGameException e) {       // a (...){...} form
    giveUp();
} finally {                         // a {...} form
    close(game);
}
```

In an IF LIST, an IF CONDBODY node has a "`(...){...}`" form, and a STMT LIST has a "`{...}`" form. In a TRY LIST, a CATCH node has a "`(...){...}`" form and a STMT LIST has a "`{...}`" form. Regarding these similarities of `if` and `try`, they share a set of macro operations — the "(" macro and the "{" macro.

In these list nodes, pressing "(" or "{" on a list element node will append a node of "`(...){...}`" form or "`{...}`" form. For example, pressing "(" on an `else-if` block inside a IF LIST will append another `else-if` block, and then the cursor will be placed inside the new condition to wait for the input.

The "(" and "{" macros can also be triggered when the cursor points to the body statement part of a "`(...){...}`" form for convenience.

Besides append operation, the "(" macro may also mean nest operation to `if` statement. The IF LIST is another example for ill-one (§ 4.1.3), thus an IF LIST containing only one `if condbody` will be automatically exposed (§ 5.1.6) into a bare IF CONDBODY. A bare IF CONDBODY represents a `if` statement without any `else` or `else-if`.

```
if (money == 0)
    cry();
```

In this case, when the cursor points to the whole IF CONDBODY or its second child STMT LIST, pressing "(" will nest the IF CONDBODY with an IF LIST, and another IF CONDBODY will be appended to the new list which is ready for the next input. This mechanism is also common to other ill-one list nodes — when there exists a list, the macro means to append; and when there is no list, the macro means to binarily nest-as-left.

Parenthesis macro can also be used to convert a variable (field) declaration into a method declaration. Pressing "(" when the cursor is on either a node of

type DECL VAR or the DECTOR[15]inside it, the DECL VAR will be converted into a DECL METHOD with corresponding return type and method identifier. This macro is usually used together with scalar promotion (§ 5.3.1). The typical way to input a method declaration is to type "ReturnType methodName(" — when "ReturnType" is typed and the space is pressed, the identifier will be promoted to a variable declaration; and when "methodName" is typed and the "(" is pressed, the variable declaration will become a method declaration. Then a fix-sized node input mode will be pushed onto the stack with its offset value set to 2 such that the cursor will point to the formal parameter list. Afterwards the user can press "F" to continue inputting the formal parameter declarations. (§ 5.4)

6.2 Expression Level Macros

Expression level macros are the main component to enable bottom-up expression inputting. Most of those macros are keys for binary infix operators and their usage is basically like typing in plain text.

When a key for infix binary operator is typed, the macro works according to the following steps:

1. Keep dollying-out if those two conditions are satisfied:
   - The precedence of the macro operator is lower than the precedence of the outer node's operator;
   - The inner node is the last child of the outer node.
2. Create a new node according to node type and node class for which the macro stands.
   - If the macro stands for a CAST, the inner node will be nested as a right child of a new CAST node;
   - If the macro stands for a binary operator list node whose node type is same as the outer node, the new node will be appended as the succeeder of the inner node.
   - If the macro stands for an ASSIGN and the inner node has type LOGIC NOT, the child of the inner node will be exposed as the new inner node, and the new inner node will be nested as a left child of a newly created node of type NEQ.
   - In other cases, nest the inner node as a left child of a newly created node of the specified type.
3. Push a modifying node for the other child of the newly created binary operator, or do nothing if this macro nested the inner node with an unary parent.

[15]Declarator, an identifier with an optional initial value.

Although the process seems quite verbose, its effect is straightforward. A macro can be considered as a "plain" input.

The first dollying-out step is for preventing unintentional nesting. For example, when we have an expression "`a * b`" and the cursor is on "`b`", pressing "+c" will result in "`a * b + c`" but not "`a * (b + c)`" since the former one looks "plainer". For another example, given an expression "`a * b * c`" where the cursor is on "`b`", pressing "+d" will result in "`(a * b + d) * c`" but neither "`a * (b + d) * c`" nor "`a * b + d * c`". Although the last one seems to be the "plainest" change, it is probably not intended by the user because the change consists of splitting a multiplicative term with an addition and simultaneously add a factor inside one of the split part. There is no reason for combining those two unrelated actions into one macro (or more generally, one key stroke), so the "`(a * b + d) * c`" is preferred as the correct result instead of "`a * b + d * c`".

The second step can be summarized as "append or nest". As mentioned in § 6.1, a macro related to an ill-one list node will check whether there already exists a list first, and perform either append or nest-as-left operation according the existence. All binary operator list node is ill-one. (§ 4.1.3) This explains how the "`a + b + c`" is generated plainly — when "+b" is typed, a nest-as-left operation is performed as there is no ADD BOP LIST yet; and when "+c" is typed, it is interpreted as an append operation since there is already a list.

The macro key "=" is dealt with specially when the inner node has type LOGIC NOT. By this rule, the user can get a "`a != b`" expression by typing "a!=b" straightforwardly. During this key sequence, the expression will become "`!a`" at the middle, but it will immediately become "`a != ??`" after the "=" macro takes its effect. This behavior is safe thanks to that an expression like "`!a = b`" is illegal to the Java language since "`!a`" is a rvalue[4] that cannot be assigned.

The Comma key is also a frequently used macro. List elements in PARAM LIST, ARG LIST, and DECTOR LIST can be quickly added by pressing Comma. Since DECTOR LIST is ill-one, "append-or-nest" will be checked as any other ill-one lists.

6.3 Macro Relaying

There is another function in the tiny lexer modes called *macro relaying*. Macro relaying is a feature to enable multi-keystroke macros and it always works after a character coupling (§ 5.2) where the first character is input by a macro.

If there is no macro relaying mechanism, problem occurs when we input an expression like "`a && b && c`". Here we have four "&" keys stroked in total. There is no problem when the beginning "`a && b`" is inputted. Things becomes messy when the forth "&" is stroked — the expression will become "`a && (b && ??)`", where the parenthesis is not expected. This happens because the expression had to be "`a && b & ??`" during the middle where the "`b & ??`" is child of the AND LIST as a whole subexpression[16]. Although the

bitwise AND is converted into a logic AND as the fourth "&" triggered a character coupling, the topology of the tree stays unchanged, and yields the unexpected parenthesis.

Therefore we introduced the macro relaying mechanism to solve this problem. If a tiny lexer mode is pushed within a context of macro, such a macro will be relayed after any character coupling. A macro relaying consists of reverting effects of the previous single-character macro, and re-perform all logic (including the iterative dolly-out and the "append-or-nest" check) of the new double-character macro afterwards.

6.4 Rationality of the Macro System

It is not contradictive to make a structured editor works like a text editor. The simulation for sequential plain text inputting is performed only when new contents are being generated. However, the major difference between editors is how they modify existing contents. If the user completed a source file sequentially from the first character to the last character without any modification and cursor moving, no editor would be needed at all. A structured editor edits, but not writes, source code along with the tree structure, Whereas new contents should be generated in a plain sequential way because that is how people speak and write natural language everyday. Therefore, the introduction of the macro system does not vacillate the editor between text editor and structured editor in any manner.

7 Use Cases

7.1 Hello World

In the demo version of Zuse, creating a new empty file is not supported yet. To create a hello world sample, a non-empty file is needed to be opened and things starts from clearing up the old content.

Before:

```
// anything in the existing source file
```

After:

```
public class Hello {
    public static void main(String[] args) {
        System.out.println("hello, world");
    }
}
```

Key sequence: R, I, `Hello`, Space, F, `void`, Space, `main`, "(", F, `String`, "[", Space, `args`, Space, D, G, F, `System.out.println`, "(", `", hello, world, "`, Shift+D, Shift+M, M, P, S, Space, D, D, M, P, Space.

[16] The bitwise AND has higher precedence than the logic AND.

7.2 Merging Assignment into Declaration

Before: (cursor points to "`42`") [17]

```
int i = 0;
i = 42;
```

After:

```
int i = 42;
```

Key sequence: Shift+R, K, L, C, Tab, J, D, R.

7.3 Switching Order of Statements

Before: (cursor points to "`eat();`")

```
eat();
sleep();
```

After:

```
sleep();
eat();
```

Key sequence: Shift+R, Enter, Tab.

7.4 Increasing Call Chain Level

Before: (cursor points to "`cat`")

```
cat.getHeight();
```

After:

```
cat.getBody().getHeight();
```

Key sequence: Dot, `getBody`, "(", Space

7.5 Surrounding Children with Different Parent

Before: (cursor points to the whole expression, clipslot A in use)

```
r = x < y ? x : y;
```

After:

```
r = min(x, y);
```

Key sequence: F, L, Y, P, B, L, Y, D, D, P, A, C, `min`, "(", Tab, Space, P, B, Comma, Tab.

[17] This was an example to show structured editor's weakness in expression-level edting[11].

7.6 Re-expressing a Logic

Before: (cursor points to "`cat`")

```
cat == dog
```

After:

```
cat.isDog()
```

Key sequence: X, Dot, `isDog`, "(", Space.

7.7 Packing Logic to Function

Before: (cursor points to the first "`num`")

```
num != 741 && num != 7474741
```

After:

```
isGoodNum(num)
```

Key sequence: Y, D, D, C, `isGoodNum`, "(", Tab.

7.8 Type Casting with Caring Precedence

Before: (cursor points to "`animal`")

```
animal.operatations.bark()
```

After:

```
((Dog) animal).operations.bark()
```

Key sequence: ")", `Dog`, Space.

7.9 Changing Subexpression with Caring Precedence

Before: (cursor points to "`tom`")

```
!tom.isPig()
```

After:

```
!(tom.isPid() && jerry.isDog())
```

Key sequence: D, N, &, &, `jerry.isDog`, "(", Space.

7.10 Switching Order of Operands

Before: (cursor points to "`expra`")

```
call(expra, exprb);
r = exprc * exprd;
```

After:

```
call(exprb, expra);
r = exprd * exprc;
```

Key sequence: Shift+R, Comma, Tab, J, Shift+R, Shit+N, *, P.

7.11 Nesting or Exposing

Before: (cursor points to "`lunch`")

```
eat(lunch);
drink(boil(water));
```

After:

```
eat(cook(lunch));
drink(water);
```

Key sequence: Shift+N, "(", `cook`, Space, J, L, D, X.

7.12 Adding Condition

Before: (cursor points to the whole IF CONDBODY)

```
if (money > 0)
    eat();
```

After:

```
if (money > 0)
    eat();
else
    sleep();
```

Key sequence: "{", `sleep`, "(", Space.

7.13 Accessibility Refactoring

Before: (cursor points to "`cat`")

```
cat.height = 20;
```

After:

```
cat.setHeight(20);
```

Key sequence: D, Shift+X, F, G, Shift+M, `setHeight`, "(", Tab.

7.14 Member or Static

Before: (cursor points to "`obj`")

```
func(obj);
```

After:

```
obj.func();
```

Key sequence: Shift+R, S, I, Dot, Tab.

7.15 Extracting Condition

Before: (cursor points to the condition)

```
if (expr1 == expr2)...
```

After:

```
boolean cond = expr1 == expr2;
if (cond)...
```

Key sequence: Shift+C, `cond`, Shift+Enter, `boolean`, Space, `cond`, =, Tab.

7.16 Yet Another Precedence Example

Before: (cursor points to "`i + 1`")

```
i + 1
```

After:

```
(i + 1) % period
```

Key sequence: N, %, `period`

7.17 Extracting an Expression Partially

Before: (cursor points to "`x`") [18]

```
if (x && f(z, a.b().c(g + f(d(e(), k()))))
    stmt;
```

After:

```
int v = a.b().c(g + f(d(e(), k())));
if (x && f(z, v))
    stmt;
```

Key sequence: L, L, L, D, Shift+C, `v`, Shift+Enter, `int`, Space, `v`, =, Tab.

[18] This change will be super erroneous in a text editor.

8 Conclusion

By the edit tree, the mode stack, and the macros, the usability of the structured editor Zuse has been improved. As a demo application, this version of Zuse has limited features and sometime might be buggy, but the basic ideas behind it actually improved the program editing efficiency as expected. It is sufficient to replace a real life editor with Zuse as long as we make Zuse a more robust software without changing its basic underlying concepts.

We have successfully shown that a structured editor for Java can be highly usable by several example cases. Therefore, it is also practical to apply these concepts in a program editor for more general C-family languages besides Java. Furthermore, since the concept "edit tree", "mode stack", and "macro" are irrelevant to Java, theoretically they can be applied to arbitrary language. In conclusion, structured editors are able to be the next trend in the field of program editing.

A Node Types in Each Node Class

Node class	Node type	Explanation	Note
List	CLASS LIST	the whole document	
	MEMBER LIST	class members	
	DECL PARAM LIST	method formal parameters	
	STMT LIST	a compound statement	
	IF LIST	compressed `if-else` blocks	ill-one
	TRY LIST	a `try-catch` block	ill-zero
	DECTOR LIST	declarators	ill-one
	ARG LIST	call arguments	
	NAME LIST	`throws`, `extends`, `implements`	ill-zero
Bop List	DOT BOP LIST	`.`, `()`, `[]`	ill-one
	MUL BOP LIST	`*`, `/`, `%`	ill-one
	ADD BOP LIST	`+`, `-`	ill-one
	AND BOP LIST	`&&`	ill-one
	OR BOP LIST	`||`	ill-one
Fix-sz (1)	RETURN	`return`	hideable child
	BREAK	`break`	hideable child
	CONTINUE	`continue`	hideable child
	THROW	`throw`	
	POST INC	`i++`	
	POST DEC	`i--`	
	PRE INC	`++i`	
	PRE DEC	`--i`	
	UNARY PLUS	`+a`	
	UNARY MINUS	`-a`	
	LOGIC NOT	`!a`	
	BIT NOT	` a`	
Fix-sz (2)	DECL VAR	`int a = 3, b;`	
	DECL PARAM	formal parameter	

	WHILE	`while`	
	DO WHILE	`do while`	
	IF CONDBODY	`if` or `else-if` block	
	CATCH	`catch`	
	CAST	type cast operator	
	LT, LEQ, GT, GEQ	`<, <=, >, >=`	
	INSTANCEOF	`instanceof`	
	EQ, NEQ	`==, !=`	
	SHL, SHR, SHRA	`<<, >>, >>>`	
	BIT OR	`|`	
	BIT XOR	`^`	
	BIT AND	`&`	
	ASSIGN	`=`	
	ASS ADD, ...	`+=, -=, ...`	
Fix-sz (3)	QUESTION	`cond ? a : b`	
	NEW CLASS	`new Cat()`	hideable child
	NEW ARRAY	`new Dog[4]`	hideable child
Fix-sz (4)	DECL CLASS	`class`	hideable child
	DECL INTERFACE	`interface`	hideable child
	FOR	`for`	
Fix-sz (5)	DECL METHOD	method declarations	hideable child
Scalar	IDENT	identifier	
	NUMBER	number literal	
	STRING	string literal	
	META	visible placeholder	
	HIDDEN	invisible placeholder	

B List of Key Mappings

Mode	Key	Function	Related Sections
Normal	F	Focus-in, assart, punch	§§ 5.1.1 to 5.1.3
	D	Dolly-out	§ 5.1.1
	S	Slip-back	§ 5.1.1
	G	Go-on	§ 5.1.1
	K	Up	§ 5.1.1
	J	Down	§ 5.1.1
	H	Left	§ 5.1.1
	L	Right	§ 5.1.1
	Shift+F	Big Focus-in	§ 5.1.1
	Shift+D	Big Dolly-out	§ 5.1.1
	Shift+S	Big Slip-back	§ 5.1.1
	Shift+G	Big Go-on	§ 5.1.1
	I	Insert	§ 5.1.2
	O	Append	§ 5.1.2
	R	Remove	§ 5.1.4
	C	Change	§ 5.1.5
	M	Modify	§ 5.1.5
	Shift+M	Clear-and-modify	§ 5.1.5

	N	Nest-as-left	§ 5.1.6
	Shift+N	Nest-as-right	§ 5.1.6
	X	Expose	§ 5.1.6
	Y	Yank	§ 5.1.7
	Shift+R	Clip	§ 5.1.7
	Shift+C	Clip-and-Change	§ 5.1.7
	Shift+X	Clip-and-Expose	§ 5.1.7
	P	Switch-clipslot	§ 5.1.7
Tiny lexer	Tab	Paste	§ 5.2
	Space	Cancel	§ 5.2
	`A-Za-z$_`	Input Identifier	§ 5.2
	`"`	Input String	§ 5.2
	`0-9`	Input Number	§ 5.2
	=, +, -	Character Coupling	§ 5.2
	`&, |`	Character Coupling	§ 5.2
	<, >	Character Coupling	§ 5.2
Identifier input	Space	Pop, scalar-promotion	§ 5.3.1
	`A-Za-z0-9$_`	Append character	§ 5.3.1
Number input	Space	Pop	§ 5.3.2
	`0-9`	Append digit	§ 5.3.2
	J, K, L, ...	Append digit	§ 5.3.2
String input	„	Pop	§ 5.3.3
	(any other)	Append character	§ 5.3.3
Fix-sized input	Space	Pop	§ 5.4
	S	Slip-back	§ 5.4
	F	Assart	§ 5.4
Modifier toggling	Space, M	Pop	§ 5.6
	A	Toggle `abstract`	§ 5.6
	F	Toggle `final`	§ 5.6
	P	Toggle access	§ 5.6
	Shift+P	Toggle access (reverse)	§ 5.6
	S	Toggle `static`	§ 5.6
	T	Toggle `transient`	§ 5.6
	V	Toggle `volatile`	§ 5.6
	N	Toggle `native`	§ 5.6
	C	Toggle `synchronized`	§ 5.6
Bop operand	Space	Cancel	§§ 5.1.2 and 5.1.6
	+	`+`	§§ 5.1.2 and 5.1.6
	-	`-`	§§ 5.1.2 and 5.1.6
	*	`*`	§§ 5.1.2 and 5.1.6
	/	`/`	§§ 5.1.2 and 5.1.6
	%	`%`	§§ 5.1.2 and 5.1.6
	(	method call	§§ 5.1.2 and 5.1.6
	.	member access	§§ 5.1.2 and 5.1.6
	[	array access	§§ 5.1.2 and 5.1.6

Node type operand	Space	Cancel	§§ 5.1.2 and 5.1.6
	., (, [	DOT BOP LIST	§§ 5.1.1, 5.1.2 and 5.1.6
	+, -	ADD BOP LIST	§§ 5.1.1, 5.1.2 and 5.1.6
	*, /, %	MUL BOP LIST	§§ 5.1.1, 5.1.2 and 5.1.6
	)	CAST	§§ 5.1.1, 5.1.2 and 5.1.6
	$	BIT AND	§§ 5.1.1, 5.1.2 and 5.1.6
	|	BIT OR	§§ 5.1.1, 5.1.2 and 5.1.6
	^	BIT XOR	§§ 5.1.1, 5.1.2 and 5.1.6
	!	LOGIC NOT	§§ 5.1.1, 5.1.2 and 5.1.6
	~	BIT NOT	§§ 5.1.1, 5.1.2 and 5.1.6
	<	LT	§§ 5.1.1, 5.1.2 and 5.1.6
	>	GT	§§ 5.1.1, 5.1.2 and 5.1.6
	=	ASSIGN	§§ 5.1.1, 5.1.2 and 5.1.6
	Shift+C	DECL CLASS	§§ 5.1.1, 5.1.2 and 5.1.6
	Shift+M	DECL METHOD	§§ 5.1.1, 5.1.2 and 5.1.6
	V	DECL VAR	§§ 5.1.1, 5.1.2 and 5.1.6
	I	IF CONDBODY	§§ 5.1.1, 5.1.2 and 5.1.6
Big move operand	S	statement	§ 5.1.1
Macro	(	Many situations	§§ 6.1 and 6.2
	{	Many situations	§ 6.1
	,	Many situations	§§ 6.1 and 6.2
	Enter	Append statement	§ 6.1
	Shift+Enter	Insert statement	§ 6.1
	[	Array access	§ 6.2
	.	Member access	§ 6.2
	)	Type cast	§ 6.2
	*	Multiply	§ 6.2
	/	Divide	§ 6.2
	%	Modulo	§ 6.2
	+	Add	§ 6.2
	-	Subtract	§ 6.2
	=	Assign	§ 6.2
	<	Less than	§ 6.2
	>	Greater than	§ 6.2
	!	Logic not	§ 6.2
	~	Bit not	§ 6.2
	&	Bit and	§ 6.2
	|	Bit or	§ 6.2
	^	Bit xor	§ 6.2

References

[1] Bauer, F. L., and Wössner, H. The Plankalkül of Konrad Zuse: a forerunner of today's programming languages. *Communications of the ACM 15*, 7 (1972), 678–685.

[2] Britefury. The Larch environment. http://www.larchenvironment.com. Accessed: 2016-06-14.

[3] Donzeau-Gouge, V., Huet, G., Kahn, G., and Lang, B. Programming environments based on structured editors: The MENTOR experience. Tech. rep., DTIC Document, 1980.

[4] Gosling, J. *The Java language specification*. Addison-Wesley Professional, 2000.

[5] Habermann, A. N., and Notkin, D. Gandalf: Software development environments. *Software Engineering, IEEE Transactions on*, 12 (1986), 1117–1127.

[6] Hansen, W. J. User engineering principles for interactive systems. In *Proceedings of the November 16-18, 1971, fall joint computer conference* (1971), ACM, pp. 523–532.

[7] JetBrains. Meta programming system. http://www.jetbrains.com/mps/. Accessed: 2016-06-14.

[8] Ko, A. J., Aung, H. H., and Myers, B. A. Design requirements for more flexible structured editors from a study of programmers' text editing. In *CHI'05 extended abstracts on human factors in computing systems* (2005), ACM, pp. 1557–1560.

[9] Lang, B. On the usefulness of syntax directed editors. In *Advanced Programming Environments* (1986), Springer, pp. 47–51.

[10] Medina-Mora, R., and Notkin, D. S. ALOE users' and implementors' guide. Tech. rep., DTIC Document, 1981.

[11] Osenkov, K. Designing, implementing and integrating a structured C# code editor. *Brandenburg University of Technology, Cottbus* (2007).

[12] Reps, T., and Teitelbaum, T. The synthesizer generator. In *ACM Sigplan Notices* (1984), vol. 19, ACM, pp. 42–48.

[13] Teitelbaum, T., and Reps, T. The Cornell program synthesizer: a syntax-directed programming environment. *Communications of the ACM 24*, 9 (1981), 563–573.

[14] TIOBE. TIOBE index for May 2016. http://www.tiobe.com/tiobe_index, 2016. Accessed: 2016-05-16.