

CHARACTER ORIENTED PROGRAM EDITING – HABIT OR NECESSITY?

Zorica Suvajdžin¹, Miroslav Hajduković¹, Žarko Živanov¹

Abstract. This paper advocates a viewpoint that the program text is not a simple string of characters but a complex structure built of components defined by the programming language. So, the program editing is not to be based on characters handling, but on handling components defined by the programming language. The paper describes how such handling can be done, and discusses properties of the suggested approach.

AMS Mathematics Subject Classification (2000): 68U15

Key words and phrases: editor, character oriented interface, programming language

1. Introduction

The traditional approach to program editing, based on character oriented user interface, is formed under the influence of the first character oriented input-output devices, intended to support a user-computer interaction. Although later appearance of graphical terminals and pointing devices has caused radical changes in the user interface, this has not touched essentially the program editors. This poses a dilemma on whether program editors have not been changed because there is no way to do so, or just because of inertia. This paper tries to resolve the dilemma by presenting a new approach to program editing. Further, it analyzes the characteristics of the presented approach.

2. User elementary activities during program editing

The program text can be seen as a complex structure composed of templates corresponding to the programming language statements. Presentation of such view of a program text is based on a subset of C programming language statements. This subset contains a variable definition statement, an assignment statement and an **if** statement, i.e. statements that can form a C function body. Table 1 contains the simplified templates of these three statements.

Each template of Table 1 contains zones with underscored names. Filling all zones of some template with suitable content leads to the completion of the corresponding C statement. Optional parts of templates are shown in italic. Parts of templates shown in non-italic are mandatory.

¹Department of Computing and Control, Faculty of Engineering, University of Novi Sad

<i>statement type</i>	<i>simplified template</i>
variable definition statement	type <u>new name</u> = <u>value</u> ;
assignment statement	<u>name</u> = operand;
if statement	if (operand <u>operator</u> operand) { <u>body</u> } else { <u>body</u> } ;

Table 1: Simplified variable definition, assignment and **if** templates.

A possible content of each template zone is known in advance. So the zone type can hold a name of an already defined type, the zone new name can hold a name of a previously undefined variable, the zone name can hold a name of an already defined variable, and the zone operand can hold a name of an already defined variable or constant. The zone value can hold a constant, the zone operator can hold one of the relational operators, and the zone body can hold a variable definition, assignment and **if** templates.

The zones could be nonterminal and terminal. The zone body is nonterminal, while all other zones are terminal. Only the nonterminal zones can hold templates.

When all zones of a template contain question-marks the template is empty. Figure 1 shows the empty variable definition template (with mandatory zones only).

? ?;

Figure 1: The empty variable definition template

Filling all zones of a template with acceptable contents leaves the full template (the filled content substitutes question-marks). Figure 2 shows the full variable definition template.

```
char ascii_code;
```

Figure 2: The full variable definition template

A C function body can be composed by filling the templates (from Table 1) to a special zone function body. The function body zone is similar to the zone body. The only difference is that the first one belongs to a function definition template, while the other belongs to an **if** template. Therefore, composing the C function body is done by inserting templates to the zone function body, and also

by deleting templates from this zone. After inserting, the template is to be filled. This is done by inserting contents to the template zones, and also by deleting these contents. The template filling can also include inserting of the optional zones, and their deleting. Therefore, the user's elementary activities during editing a program text composed from templates, consist of zone handling. The zones could be inserted and deleted. The same can be done with their contents.

3. Editor support to template zone handling

Insertion and deletion of zones or templates are supported by the editor operations **insert** and **delete**. The both of them are context sensitive operations, and are applied to the marked zone or to the marked template. The **insert** operation inserts a new zone (or a new template) behind the marked zone (or the marked template). The **delete** operation deletes the marked zone (if it is optional) or the marked template. The deleted item could be substituted by a question-mark.

Context sensitivity of **insert** and **delete** operations is possible only if marking is applied on the suitable text units. This is achieved by automatic marking the whole terminal zone, when the cursor is inside it, or by automatic marking the whole template, when the cursor is inside it, but outside of its zones. Figure 3 shows different cases of automatic marking on the example of a variable definition template.

```

char ascii_code;
char ascii_code;
char ascii_code;
char ascii_code;
char ascii_code;

```

Figure 3: The cases of automatic marking (the cursor has the form of gray rectangle, and marking is displayed by frames)

The zones and the templates build hierarchical structures. For example, the zone function body can contain an **if** template. This template contains the zone body with another **if** template and so on. This gives an idea to introduce an operation **mark (m)** intended to widen up marking to the first higher level. For example, when the zone body is marked, then the **mark** operation marks the whole **if** template containing the previously marked zone, and so on.

The cursor moving is supported by operations \rightarrow , \leftarrow , \uparrow , and \downarrow . The first two move the cursor left and right horizontally, and the last two move the cursor up and down vertically. An operation **?** moves the cursor to the first following question-mark.

Before inserting a new content to a zone it is necessary to enter the zone. This is done by the operation **enter**. After entering the zone, if it is empty, the question-mark can be substituted by a new content, while if it is a full terminal zone, then its old content can be substituted by a new content. The **enter** operation is not necessary for empty zones if marking of an empty zone causes by default entering this zone (to be filled with a new content).

The **enter** operation provides a new content by (1) input (for example, a new unique variable name in the case of the zone new name, or a constant in the case of the zone value) or by (2) selection of an alternative from a set of alternatives (for example, a type name from the set of defined types in the case of the zone type, or a variable name from the set of defined variables in the case of the zone name, or a template from the set of templates in the case of the zone body). Similarly, the **insert** operation also requires selection of an alternative from a set of alternatives. In any case selection is sensible only if there are more alternatives. In the opposite case, it is natural to choose the only option automatically.

Input of a new content is performed inside a special input dialog. In the coming examples this input dialog is represented by the rectangle shown in Figure 4.

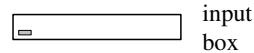


Figure 4: The input dialog

The user inputs the characters of a unique new variable name or a constant into the input box of the input dialog. The input box enables character editing, while the input dialog prevents incorrect characters input, and ensures that new variable names are unique, and that constant types are correct.

Selection of an alternative from a set of alternatives is done inside a special selection dialog. In the coming examples this selection dialog is represented by the split rectangle shown in Figure 5.

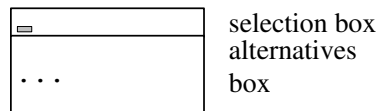


Figure 5: The selection dialog

The alternatives box contains all possible options, while the selection box mediates in selection of an option. The offered options are context sensitive (in the case of the zone type options consist of defined types names, in the case of the zone name options consist of defined variable names, in the case of the zone operator options consist of relational operator symbols, and in the case of the zone body options consist of statement symbols).

The input dialog and selection dialog enable the composed program text to be regular i.e. that is in accordance to the syntax of the programming language. Therefore, the described editor is called regular editor, and its usage regular editing.

4. The regular editing example

The example of regular editing is based on a C program fragment, shown in Figure 6.

```
char ascii_code;
char control_character;
if(ascii_code < 32)
  {control_character = 1;}
else
  {control_character = 0;}
```

Figure 6: The C program fragment

An example of regular editing is given in Table 2. The first column contains short description and keyboard keys needed to produce the screen state shown in the second column.

Table 2: The regular editing example


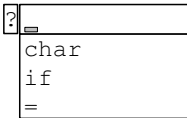
<i>Description</i>	<i>Screen state</i>
In this example regular editing starts by filling the zone <u>function body</u> .	
The enter key activates the selection dialog.	

Table 2: (continued)

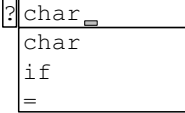
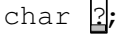
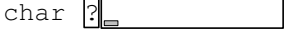
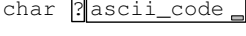
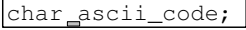
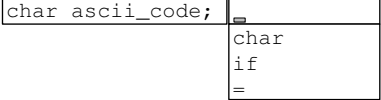
<i>Description</i>	<i>Screen state</i>
<p>The first option in the alternatives box is the type name char, because it is a symbol of a variable definition statement (to simplify discussion the number of types is limited to char type only). Also, there are two more options: symbols of assignment and if statements. The c key selects a variable definition template.</p>	
<p>The enter key confirms the selected template.</p>	
<p>The enter key activates the input dialog.</p>	
<p>The input box receives variable name characters a, s, c, i, i, -, c, o, d, e.</p>	
<p>The enter key confirms received variable name, and the ← key marks the variable definition template.</p>	
<p>The insert key activates the selection dialog. Its intention is to select a next template that is going to be inserted behind the marked template.</p>	

Table 2: (continued)

<i>Description</i>	<i>Screen state</i>
The c key selects the variable definition template. The enter key confirms the selected option. The enter key activates the input dialog. Its input box receives the new variable name characters: c, o, n, t, r, o, l, -, c, h, a, r, a, c, t, e, r . The enter key confirms the received variable name, and the ← key marks the just inserted template.	<pre>char ascii_code; char_control_character;</pre>
The insert key activates the selection dialog that intervenes in the selection of a new template. The i key selects the if template. The enter key confirms this choice.	<pre>char ascii_code; char control_character; if(2) {?};</pre>
The enter key activates the selection dialog to enable selection of an operand.	<pre>char ascii_code; char control_character; if(2) {?; ascii_code control_character</pre>
The a key selects the variable name <code>ascii_code</code> , and the enter key confirms this selection.	<pre>char ascii_code; char control_character; if(ascii_code) {?};</pre>
The insert key inserts two optional zones: <u>operator</u> and <u>operand</u> .	<pre>char ascii_code; char control_character; if(ascii_code 2 ?) {?};</pre>

Table 2: (continued)

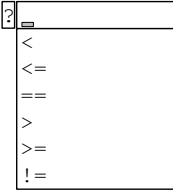
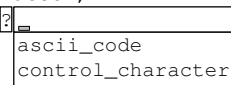
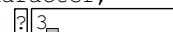
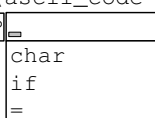
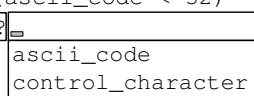
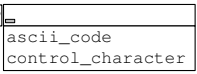

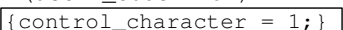

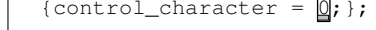
<i>Description</i>	<i>Screen state</i>
The enter key activates the selection dialog to enable selection of a relational operator.	<pre>char ascii_code; char control_character; if(ascii_code < ?) {?};</pre> 
The < key selects the operator less, and the enter key confirms selection. The ? and enter keys help to select the remaining operand.	<pre>char ascii_code; char control_character; if(ascii_code < ?) {?};</pre> 
The 3 key activates the input dialog instead of the selection dialog.	<pre>char ascii_code; char control_character; if(ascii_code < ?) {?};</pre> 
The 2 key ends the constant 32, and the enter key confirms this constant. The ? and enter keys enable filling the if statement body.	<pre>char ascii_code; char control_character; if(ascii_code < 32) {?} char if =</pre> 
The = key selects the assignment template, and the enter key confirms this choice.	<pre>char ascii_code; char control_character; if(ascii_code < 32) {? = ?};</pre>
The enter key activates the selection dialog to enable filling the left zone of an assignment variable template.	<pre>char ascii_code; char control_character; if(ascii_code < 32) {?} ascii_code control_character</pre> 

Table 2: (continued)

<i>Description</i>	<i>Screen state</i>
The c key selects the variable name control_character , and the enter key confirms this selection. The ? and enter keys help to choose a new value of the previously selected variable.	<pre>char ascii_code; char control_character; if(ascii_code < 32) {control_character = ?</pre> 
The 1 key activates the input dialog instead of the selection dialog.	<pre>char ascii_code; char control_character; if(ascii_code < 32) {control_character = ?1</pre> 
The enter key confirms the received constant. The m m keys mark the just filled zone.	<pre>char ascii_code; char control_character; if(ascii_code < 32) {control_character = 1;}</pre> 
The insert key inserts the optional zone.	<pre>char ascii_code; char control_character; if(ascii_code < 32) {control_character = 1;}; else {?};</pre> 
The keys enter , = , enter , enter , c , enter , ? , enter , 0 and enter help to fill the optional zone.	<pre>char ascii_code; char control_character; if(ascii_code < 32) {control_character = 1;}; else {control_character = 0;};</pre> 

Traditional character oriented editors demand 112 keystrokes to input the C program fragment from Figure 6. To input the same fragment by the regular editor 78 keystrokes are needed, or even 66 strokes if only the automatic entering to the marked empty zones is presumed (in the second case the enter operation is not needed). Therefore, in the presented example the regular editor reduces the number of keystrokes comparing to traditional editors by 41%. During the usage of the implemented prototype of the regular editor [4] reduction of keystrokes number was between 30% and 60%, comparing to traditional editors.

The previous example shows that the regular editor leads users, and does not require exact knowledge of the programming language syntax. It also prevents

all editing errors (lexical, syntax and semantic errors that can be detected in compile-time).

5. Regular editor discussion

The described regular editing prevents errors only during initial program text inserting. This means that changing of an already inserted program text can leave inconsistencies, despite the regular editing. For example, a C program fragment:

```
int number;
...
number = 1000;
```

becomes incorrect when the variable number type is changed from int to char. Such inconsistencies could be removed if the regular editor would automatically intervene and expel the name number from all zones where its usage becomes unacceptable. Therefore, after the described change of the variable number type, and after the editor intervention, the previous C program fragment looks like:

```
char number;
...
? = 1000;
```

Consequences of the regular editor automatic intervention on a program text can be very extensive, so it is important to offer the undo operation to cancel thoughtless changes and their consequences. A regular editor automatic intervention on a program text would also cover propagation of the changes like variable name changes. For example, if the name number is changed to the name count, then, after the regular editor intervention the previous C program fragment becomes:

```
int count;
...
count = 1000;
```

The regular editing influences the operations find, copy&paste and cut&paste. The find operation activates the special selection dialog to choose text to be retrieved. The copy (cut) operation denotes the marked text, and the paste operation inserts the denoted text behind or instead of the marked text (the denoted text is moved from the previous to the new position in the case of the operation cut). The paste operation is specific in the sense that the denoted text can not be inserted everywhere. For example, if a content of the zone operand is denoted, then it can not be inserted into the zone operator. Besides,

the paste operation activates the special dialog (1) in the case that the copy operation denotes a global variable or a function definition statement, and (2) in the case that the copy&paste (cut&paste) operation moves statements from the body of one function to the body of the other function. In the first case the dialog purpose is to change the names of the copied variable or the function to preserve their unique appearance. In the second case the dialog purpose is to exchange the local variables names of the source function (used in moved statements) for the local variables names of the destination function. The regular editing requires a special treatment of expressions in order to provide an automatic handling of the parentheses and the unary operations.

6. On implementation of the regular editor prototype

The idea of the regular editing is worked out, applied and checked [4] in the development of the Structure Text programming language environment (International Standard IEC 1131-3 [2]). This programming language is intended to be used for Programmable Controllers (PLC). The developed programming environment comprises the Structure Text editor (partly implemented), the translator from Structure Text programming language to C programming language, C compiler, the linker, the PLC loader and the PLC executive support. The environment uses the existing C compiler, the linker and the PLC executive support, and the editor, the translator and the PLC loader are developed. This environment was used to develop a complex application for controlling a big industrial section. The biggest program text composed with Structure Text Editor contains about 3000 lines. User's reactions to the implemented prototype of a regular editor [4] were very favorable, although it is not completely clear if these impressions can be generalized, because there were a few users and specific ones (only PLC programmers). The main advantages of the implemented regular editor prototype include:

- avoiding editing errors,
- guiding the user and
- reducing significantly the number of keystrokes.

The fact that the user is guided in the process of regular editing, implies that the user does not need to be familiar with the syntax of the programming language. The reduction of the keystroke number is the result of the following:

- names are assigned only once, than selected,
- keywords are automatically inserted into the program text,
- program text marking is automatic and
- results of the program text modifications are automatically propagated down the rest of the program text.

All these advantages are important for novice programmers, non-professional programmers (like PLC programmers), but should not be neglected by the professionals either.

7. Regular editor generator

In order to implement the idea of regular editing of the program text, the reditor must have knowledge of programming language syntax and semantics. This means that regular editors must be adapted to individual programming languages. Therefore, it is important to facilitate this adaptation as much as possible. This can be done by developing a software tool for generating reditors specialized for particular programming languages. The aim of such generator would be to automate the reditor's adaptation to the programming language. The generator input must contain some form of the programming language description, and the generator output would be the program code for the reditor specialized for the given programming language. The generator input should contain the following items:

- the lexical rules (the base for generating a lexical filter in a lexical dialog)
- the syntax rules (the base for generating a syntax filter in a syntax dialog)
- the set of all data types with their characteristics (name, length, ...)
- the set of all applicable operators, defined for data types
- the set of possible name attributes (each name can have an attribute assigned) and
- semantic sets, where each set contains a description of names that are used in the syntax dialog.

The generator output is produced from the regular editor implementation core, modified in accordance to the given input.

8. Regular editor review

In some respect, regular editing is similar to structure editing [1]. Both approaches try to avoid editing errors. The difference of these approaches is in the way this aim is fulfilled. Structure editing is character oriented and editing errors are discovered by lexical and syntax analysis. Regular editing is not character oriented and editing errors are prevented. It could be said that regular editing imposes the way of editing a program text and for that reason limits the user's freedom, because it offers only a subset of all available actions from traditional editors. So, it is natural to wonder if this restriction has serious consequences, in the sense of preventing the user of doing what he/she wants. For example, during the traditional editing, the user can transform a while statement into an if statement directly replacing the string "while" with the string "if". Although regular editing does not allow exactly the same method, it does not prevent the user from making the same effect either by copying the content of the zones condition and body from the while regular template into the if regular template, or applying a special operation introduced just for such modification of the program text. The previous suggests that regular editing, in principle, does not prevent arbitrary program text modifications, but offers eventually indirect way of their accomplishing, or at least demands existence of

operations with such purpose (which makes sense only when its usage frequency is high enough). Therefore, this paper does not discuss regular editing limitations, which it objectively introduces, but its indisputable advantages, which it definitely offers. Authors are not aware of any paper with similar subject. So the proposed regular editor is to be compared only to the latest commercial editors like Microsoft Visual C# editor from .NET development environment [3]. Similarity of these two kinds of editors is in the attitude to facilitate editing. They differ in the way how it is done. The editors like Microsoft Visual C# editor have the following characteristics:

- they try to improve readability of the program text by analyzing it and showing keywords in different color
- they analyze the program text in editing time to reveal lexical and syntax errors
- they offer to users possible options like method names of classes in editing time.

Such approach implies active participating of the user. The user makes an independent step and editor warns him about possible errors. The idea of regular editor is to lead the user only through correct states preventing him from doing any erroneous step. In this way, the user can compose only a correct program text, comprised of the elements supported by the used compiler.

9. Conclusion

This paper describes a new approach to program editing. The approach is based on handling the programming language statements as integral components of a program text, in contrast to traditional editors that treat a program text as a simple and unstructured string of characters. It is shown that the proposed approach can lead to a shorter and easier editing and avoiding editing errors.

References

- [1] Aho, A.V., Sethi, R., Ullman, J.D., Compilers: Principles, Techniques, and Tools, Addison Wesley, 1985.
- [2] International Standard IEC 1131-3, Programmable Controllers – Part 3: Programming Languages, IEC, 1993.
- [3] Microsoft Visual C# .NET development environment 2002.
- [4] Suvajdzin, Z., Structured syntax driven editor for ST programming language, Master thesis, Faculty of Engineering, Department of Computer Science, University of Novi Sad, 2000.

Received by the editors January 5, 2002