Developing an Intelligent Editor for Microcomputers

T.S. McDermott
Department of Computer Science, University of Cape Town

Abstract
A language-independent syntax-directed editor for teaching programming languages on micro-computers is briefly described. The design of the configuring grammar and the internal tree-structure to be edited and interpreted is discussed. The necessity for the internal structure to be "natural", i.e. conformable to the user's semantic view of his program, is emphasised.

Introduction

This is a brief interim report on a project initiated by the author at the University of Tasmania in 1981, and now being pursued at the University of Cape Town. The goal is an efficient programming-support system for micro-computers (language-independent and machine-independent), built around an "intelligent" editor. By an "intelligent" editor, in this connection, we mean an editor which consults the syntax (both context-free and context-sensitive) of the programming language to be used, and thus guides a user in the construction of his program.

The building of such systems became a practical proposition with the invention of full-screen editing (1), and has already attracted the attention of several researchers (4), the best-known work being perhaps the Cornell Program Synthesizer of Teitelbaum et al (2). The present project grew out of adaptations made to the Apple UCSD Pascal System in 1980 when it was used to teach Pascal to first-year students at the University of Tasmania. A full account of these adaptations is to be found in (3). Here we shall mention only the attempts to make transparent the confusing division of programs into text form and code form.

In the UCSD system the program must first of all be edited in *text* form, then compiled into *code* form for execution. The compiler and editor are linked to this extent: if the compiler detects a syntax error in the edited text, one may, if one wishes, return with a diagnostic message to the editor at the offending point in the text, in order to edit out the error.

In the adapted system the code and text forms of the program are even more effectively linked: after a run-time error one may hold a post mortem, then decide either to continue running (normally or one Pascal statement at a time), or to return with a diagnostic message to the editor at the point in the text corresponding to the offending point in the code. Of course, editing out an error means compiling again from the beginning.

The further step, therefore, taken by the present project is to concertina the edit and compile stages into one, so that what is edited is not text but an already syntactically correct (partial) program *structure,* which can be directly interpreted as though it were code. Several advantages accrue. The student conceives a program from the start as an abstract structure, underlying its multiple concrete implementations as text or code. A program is not a piece of text, but a static structure representing the dynamic structure one wishes to impose upon a computational process. The imposing of the dynamic structure is the 'execution' of the program; the creating of the static structure is the 'editing' of the program. Moreover, the errorprone and wasteful process of editing structural algorithms into linear text, so that a compiler may translate them back again, disappears. This is especially rewarding on micros, where the micro is (largely) unemployed during editing, and the programmer during compiling.

In anticipation of an early move from Pascal to Ada, both editor and interpreter were designed to be configurable to a new language by reading in specifications of the new syntax and semantics. This report restricts itself to the editor and syntax specifications.

The editor relies on three main stores of information:

(1) GRAMMAR holds information common to all programs of the given language, and which therefore need not appear in
(2) TREE, which holds only information specific to the user program being edited, i.e. a record of editing *choices* made by the user from options available to him by the grammar.
(3) STRINGPOOL holds a record of the actual terminal strings with which the user displays his choices textually, e.g. the actual identifiers employed.

We will deal in some detail with the first two of these data-structures.

The Grammar

The grammar of the particular language to be used is read in at the start of the editing session in a binary form which we will call GRAMMAR, derived at some earlier time from a textual form which we will call the configuring grammar.

The configuring grammar derives from BNF; but with 'template'-productions consisting only of concatenations rigorously distinguished from 'menu'-productions, consisting only of alternations. Certain 'tokens' (such as IDENTIFIER, INTEGER, etc) have no productions in the grammar, but have built-in definitions in the editor. A language with deviant definitions of such tokens would need to link in a new library of such definitions.

Much effort was expanded to make the grammar "natural". The way it is structured affects the way the editor guides the user in the construction of his program, and must therefore reflect the way users think of this construction process. The single most important consideration here is that users look "through" the static structure of their text to the dynamic structure of the computation process they are trying to prescribe. Although an infinity of formal grammars are *capable* of defining the required static structure, only a few are "natural" in the sense that they are transparent to the semantic concerns of the user. For example, it is not "natural" to define statement-sequences recursively:

<statement-sequence> ::=
<statement>[; <statement-sequence>].

This definition actually defines not a statement-sequence but a statement-tree in which every component statement occupies a new logical level; but the user thinks of them as all sequential on one logical level, and rightly so, because he is looking "through" the textual sequentiality to the sequentiality of the dynamic process being prescribed. We therefore adopt the more "natural" linear definition:

statement-sequence ::= {statement_;}

where the braces represent a repeatable construct called a 'list', and the underline represents the only allowable point of exit

from the list. Of course, where recursion *is* sensible, e.g. to represent hierarchical block-structure, we *would* use it, and not the 'list' structure.

We also require that templates, menus and tokens be indicated by "natural" meta-names, usable in the editor's prompts to the user. So we talk of 'if' and 'statements', not of 'branch-statement' or 'serial statement'. This becomes especially important when context-sensitive syntax is in question. The present article does not deal with this, except to say that we follow the intuition underlying 'attribute grammars': that context-free syntax creates the tree-shape of the program structure, ensuring uniform development of sub-trees in whatever trees they are contextually embedded; and then context-sensitive syntax lays down certain rules of interaction between distant parts of the program tree. We conceive of these rules as definitions of macro-names, substitutable for the basic context-free metanames, e.g. in the menu for a 'type' we would replace the context-free name 'identifier' with the macro 'typeidentifier', defined to mean 'an identifier that has appeared in the subtree BLOCK.TYPES above.' Besides tree-structure information, GRAMMAR includes also display information needed for pretty-printing a text-window by the simple expedient of textually formatting the rules in the configuring grammar in exactly the way we wish the resultant program constructs to be formatted. This, with the special sign which prescribes textual indentation of the whole subtree derived from the following node, suffices to ensure a natural structural mapping from internal tree to text representation.

The Tree

The TREE is an internal representation of the user's program stored in the most condensed form possible, omitting information common to *all* programs of a given language and containing only the (possibly unfinished) record of editing choices made in developing a particular program. Fig. 2 illustrates the internal TREE of user choices which corresponds to the unfinished program of Fig. 1.

User choices are of four main kinds:

(1) *Options accepted.* A program can be written without variable declarations, but our example has opted for them, even though no actual declarations has yet been completed. So TREE contains nodes 3-5 and the display contains line 1.

(2) *Lists repeated.* This is another form of option. The statement-list at level 2 of our program contains two statements, as can be seen from the fact that STATEMENTS has generated two 'phrase 2's at level 3: one at node 7 and one at node 12 (see column P on Fig. 2).

(3) *A 'menunode'* in the TREE may be *particularised* by selecting one of its items: e.g. a STATEMENT node at node 7 has been so particularised to a REPEAT, whilst STATEMENT nodes at 8 and 12 still await particularisation.

(4) *A 'tokennode'* may be *particularised* by pointing it at a terminal string stored in the STRINGPOOL, e.g. the IDENTIFIER node 1 points to the string 'EXAMPLE' and that at node 11 to the string 'FINISHED'. In general, only one copy of each terminal string is kept in the STRINGPOOL; with consequent storage gains.

```
          0         1
           123456789 123456789
     0     program EXAMPLE;
     1     var     : ;
     2     begin
     3       repeat
     4
     5       until FINISHED;
     6
     7     end;
```

FIGURE 1. Text window.

TREE therefore can be thought of as representing

a) the *parse*-tree of the program according to GRAMMAR. As such it omits all unrequested optional nodes and list repetitions (henceforth referred to as 'holes').

b) the *abstract* parse-tree of the program. As such it omits all terminal symbols such as 'begin', 'while', ' ',';' belonging to the concrete syntax defined in GRAMMAR.

c) the abstract parse-tree of a *partially complete* program. As such it omits all development of nodes still awaiting some user decision, e.g. selection from a 'menu'.

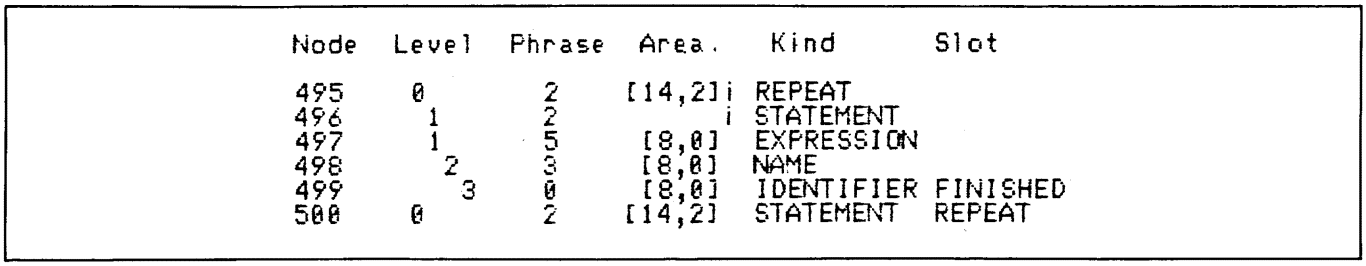
All such omissions are detectable since the editor knows both its current position in the TREE and its current position in the GRAMMAR, and moves both together. If options have not been accepted or lists not repeated this emerges from a comparison of the actual nodes present in TREE (and their so called 'phrase-numbers') with the totality of phrases a rule could generate (or repeat generating) according to GRAMMAR.

To conserve storage further the TREE is not stored as a linked data-structure but flattened into prefix form in an array with a 'level' indication at each node. (One bit suffices for a level indication however deep the tree goes! Developed nodes are already marked, so one knows all 'fathers'. For unique determination of the tree simply mark all 'youngest sons'.)

Every node thus contains *for tree definition purposes:* level and development flags together determining the structure of a tree, a 'phrase-number' determining (from the father's production-rule) the 'syntax-kind' of the node, and an extra byte-sized slot used either to indicate a choice at a menu-node (e.g. REPEAT at node 7), *or* to point to the stringpool associating e.g. a particular identifier with a tokennode, or a comment with a template-node; and *for screen display purposes:* an indicator ('i' on Fig. 2) of an indenting node, and the area of screen occupied by display of the subtree dependent on the node (e.g. BLOCK ends up six lines down, and three positions to the right of the current indent). Since this area must be updated at each insertion or deletion into the subtree, it is convenient always to hold the position in TREE and on screen of all ancestors of the current node, in a continually changing aux-

Node	Level	Phrase	Area	Kind	Slot	Paths 1	2	3
0	0	0	[4,7]i	PROGRAM		[0,0]	[0,0]	[0,0]
1	1	0	[7,0]	IDENTIFIER	EXAMPLE			
2	1	1	[3,6]	BLOCK		[0,1]	[0,1]	[0,1]
3	2	3	[0,1]	VARIABLES		[0,1]		
4	3	2		IDENTIFIER		[6,1]		
5	3	5		TYPE				
6	2	8	[3,5]	STATEMENTS			[0,2]	[0,2]
7	3	2	[14,2]i	REPEAT			[2,3]	
8	4	2	i	STATEMENT				
9	4	5	[8,0]	EXPRESSION				
10	5	3	[8,0]	NAME				
11	6	0	[8,0]	IDENTIFIER	FINISHED			
12	3	2	i	STATEMENT				[2,6]

FIGURE 2. Internal TREE.



Node	Level	Phrase	Area.	Kind	Slot
495	0	2	[14,2]	REPEAT	
496	1	2		STATEMENT	
497	1	5	[8,0]	EXPRESSION	
498	2	3	[8,0]	NAME	
499	3	0	[8,0]	IDENTIFIER	FINISHED
500	0	2	[14,2]	STATEMENT	REPEAT

FIGURE 3. Copy buffer with Keynode.

iliary stack called the PATH. PATHS for nodes 4,7 and 12 are iluustrated in Fig. 2.

The flattening of TREE is also a user-friendly feature. Linear movement around TREE is only presented as up-and-down level movement when such a conception has semantic importance. Travel around TREE takes place in 'node' mode (stopping only at nodes present in the tree, developed or undeveloped) or in 'phrase' mode (accessing also 'holes', i.e. nodes which *could* be present but are not yet). In either of these modes a user can request a jump to the start or end of the program, a move left or right a given number of nodes (or phrases), a move up or down a given number of levels or of textual indentations, or a move to the next compulsory inserting point as yet undeveloped.

The Editor in Action

The editor divides its display screen into two parts: an upper prompt area, and a lower text window in which a cursor marks the current editnode. (In Fig. 1 the cursor is at line 6, and node 12 is editnode). The editor executes by going through a SETPATH-PROMPT-EDIT cycle. First of all, the PATH to the current editnode is updated internally. If the current editnode is a non-optional template node then it is automatically inserted without prompt or user keystroke. In all other cases a user keystroke is required, and unless the user has shown by a premature keypress that he does not need prompting, the editor displays a threefold prompt.

Firstly, a flag at the cursor point names the node (or phrase) at which the user is stationed e.g. <STATEMENT> in our example. Secondly, the first line of the prompt area defines the valid editing steps at this point. Typical might be

Options: *Insert = rtn *Jump = esc →
*Exit *Phrasemode.

This says a control-I is valid and will result in a call to insert something at this point (the key <Return> can also be used). Other valid keys are "travel left one node",←, and "travel right one node", →; control-J (or <escape>) which allows one to access other modes of travel; control-E which exits from the editor; and control-P which switches travel from 'node' mode to 'phrase' mode. After insertion, the insertion option becomes invalid and will be replaced by a delete option (*Delete). All insertions and deletions take place immediately by way of a copy-buffer, an example of which is given in Fig. 3. This shows how the copybuffer would look immediately after a deletion of node 7 and its dependent subtree (nodes 8 to 11) from the example program. It contains a copy of the deleted nodes, followed by a key node which enables the editor to determine of any editnode position, whether copying the buffer (*Copy) would be a syntactically valid option.

When the editnode is a menu mode or a token node the prompt area also displays one or more menu-definition or token lines, e.g.

Menu: Assignment Procedurecall Statements If Case WHile Repeat For WIth Goto Null; or
Identifier: Letter Alphanumerics

In selecting from a menu the user need only type sufficient letters to determine uniquely his chosen item, e.g. R for REPEAT, WH for WHILE. In responding to a tokenprompt the user simply types the terminal string that he wishes to insert at the node in question. In both these cases no preliminary strike of control-I or <Return> is required since non-control characters themselves in a menu or token situation trigger the insert option.

A word must now be devoted to 'holes'. Normally the editor will neither prompt for nor permit a label before every statement. The user switches into Phrasemode when he *does* wish for such a prompt and permission. Not all possibly empty nodes are represented by holes. A Pascal null statement is represented by a menu item generating the nullstring on screen and a nullnode in TREE. To have made a STATEMENT an optional node or hole would have meant that the editor would normally never prompt for nor permit statements! As so far described, complex sequences of keystrokes can sometimes be required to deal with holes and menunodes that coincide or follow one another rapidly, as in the definition of EXPRESSION:

EXPRESSION ::= ?SIGN{{OPERAND MULTOP} ADDOP}?PREDICATE
PREDICATE::=RELOP}}OPERAND MULTIOP}ADDOP}

where SIGN,OPERAND,MULTOP,ADDOP and RELOP are all defined by menus. This complexity together with the mismatch between a user's linear conception and the grammar's nested generation of expressions, has led some researchers to abandon structure-editing inside expressions. The Cornell Program Synthesizer, for example, reverts to text-editing for what it calls 'phrases', followed by rapid syntaxchecking, and a request to retype the whole phrase if an error is detected. We have preferred to stick with the structured approach and simplify difficult situations with multiple "lookahead" menus which allow the user to skip across unrequired holes and insert in the required hole the required menu item, with a single keystroke. Further helps are the specification of a menu-item by actually typing a token consonant with that menuitem, rather than the item's metaname; and the specification of list-repetition by typing the appropriate symbol from the concrete syntax (e.g. the comma in a Pascal label-list or identifier-list). To implement these last enhancements GRAMMAR contains look-ahead sets for all such critical points in the grammar.

References

[1] Meyrowitz and van Dam. "Interactive Editing Systems" Computing Surveys 14, 3 September, 1982 pp 321-415.

[2] Teitelbaum, T and Reps, T. "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment". CACM 24, 9 September 1981 pp 563-573.

[3] Keen, C.D. and McDermott, T.S. "A New Tasmanian Apple Variety". Australian Computer Science Communication 3, 3 September 1981 pp 305-313.

[4] a) see bibliography of reference 2 above.
b) in "Tools and Notions for Program Construction" ed. D. Neel, C.U.P., 1981.
p.329 Douzeau-Gouge, Huet, Kahn, Lang, Levy. "A structure-oriented program editor".
p.337 ditto "Programming environments based on structured editors: the Mentor experience".
p.351 Habermann. "System development environments".
c) Batten, D.L. "Modular Design for Interactive Systems". Technical Reports 80-113 & 80-114, Queens University, Kingston, Ontario, Canada (with further bibliography).
d) Lakos, C and McDermott, T.S. "Interfacing with the user of a syntax-directed editor". Report 82-3, University of Tasmania, Australia.