GED

A GENERALISED SYNTAX EDITOR

A Thesis Presented in Partial Fulfilment of the Requirements

for the Degree of Master of Science in Computer Science

at Massey University

Giovanni Serafino Moretti

1984

ABSTRACT

This thesis traces the development of a full-screen syntax-directed editor - a type of editor that operates on a program in terms of its syntactic tree structure instead of its sequential character representation.

The editor is table-driven, reading as input an extended BNF syntax of the target language. It can therefore be used for any language whose syntax can be defined in EBNF. Print formatting information can be included with the syntactic definition to enable programs to be pretty-printed when they are displayed.

The user is presented with a pretty-printed skeletal outline of a program with the currently selected construct highlighted and all required syntactic items provided by the editor. Any constructs with alternatives, such as "<statement>", which occurs in many languages, are initially denoted by a placeholder in the form of a non-terminal name (i.e. "<statement>") which is expanded when the user indicates which alternative is wanted. All symbols entered by the user are parsed immediately and any erroneous symbols rejected, making it impossible to create a syntactically incorrect program. The editor cannot detect semantic errors as no semantic information is available from the EBNF syntax. However the first use of all identifiers is flagged by the editor as an aid to the detection of undeclared identifiers.

A "help" area at the bottom of the screen continuously displays a list of the correct next symbols and the syntactic definition of the currently selected program construct. This display, together with a multi-level "undo" command and the provision of a skeletal program by the editor, provides a way of exploring the various constructs in a programming language, while ensuring the syntactic correctness of the resultant program.

Table of Contents

Chapter 1

Introduction

1 Program Preparation - The Traditional Approach

The most common method of program preparation involves the repeated use of a text-editor and a compiler. This method has an inherent limitation - even if the user is sitting at a terminal, it enforces an essentially batch mode of operation. The programs are prepared, and then submitted to a compiler for verification and translation. There are two error classes that could be eliminated if the editor itself was cognizant of the syntax of the programming language in use. The first class is composed of errors that violate the lexical grammar of the language and the second of errors in the constructive syntax - the productions that define how the lexical symbols may be combined.

Lexical Limitations

A text editor accepts programs, as an arbitrary sequence of characters, whereas logically a program is a sequence of unique symbols. Some of these symbols are required by the syntax, others occur in syntactically-ordered pairs or groups and some may be chosen by the programmer.

The only items in a program whose textual nature is significant are identifiers, numbers, strings and comments. These are composite items consisting of sequences of characters, and the fact that reserved words

are externally represented as sequences of characters is irrelevant and in this context misleading. It is irrelevant because although reserved words look like identifiers, they are treated in the syntax as unique symbols - a single incorrect character destroys the validity of a reserved word, whereas even several altered characters may leave a symbol still conforming to the syntax of an identifier.

More importantly, in this context it is misleading to treat reserved words as character sequences as it leads the user to think of a program as being composed of characters, not symbols. A text editor, having no knowledge of program syntax, manipulates the program as text, reinforcing this view.

Structural Limitations

A text editor has no knowledge of the syntactic structure of a program. Therefore common errors such as unbalanced bracketing symbols and the omission of required symbols are not recognised at a stage where it is possible to correct them easily. Only later, during the compilation of the program, will these errors be detected, and then immediate correction will be impossible.

If the editor knew the target language syntax then these syntactic errors could either be detected immediately and corrected, or prevented.

1.1 Integrated Programming Environments

The integration referred to here is that of the editor and the program that actually translates the user's program, be it compiler or interpreter. The most common such translators are interactive systems for the language BASIC but languages with dynamic data structures like APL, LISP and SNOBOL are also usually interpreted and often interactive.

Traditional interactive systems were in general originally designed for use with printing terminals and have had a line-oriented syntax - the slow speed of such terminals made the interactive editing of multi-line syntactic items impractical. Examples of this approach are interactive versions of BASIC, LISP, APL and the JOSS system although the most common by far is BASIC. For a language with an appropriate syntax, line oriented program entry is easy to use on both fast and slow speed terminals as the incremental parsing alerts the user to errors in a line as soon as that line is entered.

1.2 Interpretive BASIC Systems

The BASIC language was developed for teaching and was specifically designed to be interactive. The reasons for this are threefold:

(a) The input is checked for errors at the end of each line and erroneous lines may be corrected immediately.

(b) An altered program is immediately executable without the need to invoke a compiler or leave the BASIC system.

(c) A line trace is available during execution and it is possible interactively to find and alter the values of all variables for debugging purposes.

This first two of these are the most important, as having a single environment in which to create, edit and execute programs is an important contributor to BASIC's ease of learning and use. As the system can be left in "BASIC Mode", beginners do not need to learn about the operating system and editor environments.

1.3 Keyword Entry

A letter from Mr G.J. Tee of the Auckland University Computer Science Department contains a reference to what must be one of the earliest systems for the entry of complete keywords in a single keystroke: "I visited the Computer Centre at the University of Moscow during the International Congress of Mathematicians, in about June 1966. I saw there card punches being used to prepare ALGOL source programs, with the key-board including keys for the reserved words in ALGOL. For instance, one key had the Russian equivalents of BEGIN and END as the lower-case and upper-case symbols" [Tee 1983]. More recently the Sinclair ZX81 and the Spectrum microcomputers have their BASIC interpreters and keyboards arranged so that any keyword can be obtained by depressing (possibly in conjuction with a shift key) an appropriately labelled single key [Vickers 1980,1982]. This helps to avoid spelling errors and to ease program entry. The use of keyword entry reduces the program entry time simply by reducing the number of characters that need to be typed - this is especially valuable for

beginner who are often unfamiliar with a keyboard - and thereby reduces the opportunity for error. The editing of existing lines of program is also symbol oriented, with keywords being skipped, added and deleted as single entities. The systems are interpretive and check the syntax on a line-by-line basis which also contributes to their ease of use. This single keystroke token entry is the first form of syntax-directed program entry to be widely available.

1.4 Syntax-Directed Editing Environments

In the BASIC systems discussed in the previous section, the user is constrained by the syntax of language being entered and it is impossible to construct erroneous program units larger than a single line without the generation of an error message.

A contrasting technique made possible by the widespread availability of high-speed terminals has been the development of full-screen editors that provide an window into a file, instead of a view based on lines. Such editors may provide commands for editing the file in textual constructs - word processors deal with letters, words, lines, sentences, paragraphs and pages - or alternatively provide an editing environment in which the editing units are not textual but syntactic. Given the high speed at which the screen may be redrawn, the syntactic constructs need not be line-oriented and can therefore extend over several lines.

Syntax-directed editors permit the user to create programs that conform to the syntax of the programming language in use. The BASIC systems previously discussed are line-oriented examples of syntax-directed editing environments. More recently, syntax-directed editors for languages with a nested syntactic constructs have been developed.

These include the Cornell Program Synthesiser for PL/C (a subset of PL/1) [Teitelbaum 1981], the ALOE syntax-editor generator [Medina-Mora 1981], the POE editor for PASCAL [Fischer 1981] and the COPAS system for Pascal [Atkinson 1981]. The Z editor [Wood 1981] is a text editor but has features relating to program structure normally found only in true syntax-directed editors.

Each of these editors will be discussed to illustrate the user's view of the editor and the commands available. Where relevant the internal structure is also discussed.