THE BELL SYSTEM TECHNICAL JOURNAL
Vol. 62, No. 10, December 1983
Printed in U.S.A.

Generation of Syntax-Directed Editors With Text-Oriented Features

By B. A. BOTTOS* and C. M. R. KINTALA†

(Manuscript received March 16, 1983)

Often, syntax-directed editors rely solely on menu selection for program construction. We describe here the generation of *hybrid editors* that give a programmer the option of either (1) using menu selection and tree navigation as in a syntax-directed editor, or (2) entering text for parsing and navigating through the text as in a conventional editor at *any* stage during the expansion of a program. A prototype system, HEG (Hybrid Editor Generator), has been built to automatically generate such a hybrid editor from a high-level specification of a grammar for an application language. Each such generated hybrid editor is called an AGE (Automatically Generated Editor). We describe the HEG meta-language and briefly summarize the editing process in AGEs. We also describe possible extensions to the meta-language to describe program semantics, and the generation of the procedures to check those semantics during program construction.

I. INTRODUCTION

In the past, there has been a dichotomy between the way a development tool such as a text editor would manipulate the text of a computer program and the way another development tool such as a parser would manipulate the same text. The advent of syntax-directed editing has removed this difference by introducing the use of editors that store and manipulate programs entirely as (partially) instantiated syntax trees.[1–8] The question, though, of whether a program should be manipulated *only* in terms of its syntax tree, or also in terms of its

* Carnegie-Mellon University. † Bell Laboratories.

textual representation, is yet to be resolved.[8,9] Some editing systems purport to allow the programmer both points of view, but provide only one, predetermined, choice for the manipulation of each construct in the language.[7] We introduce here the concept of a *hybrid editor*, which integrates the tree-navigation and menu-selection capabilities of a syntax-directed editor with the text-navigation and the string-entry capabilities of a conventional editor. Either of these views of a program can be taken at *any* stage during the expansion of the program in a hybrid editor.

One of the features of the hybrid editors discussed herein is the fact that a complete editing system is automatically generated from a high-level description of a language. The concrete and abstract syntaxes of the language are both described by one context-free grammar; and both the syntax-directed editor, and the incremental parser for use with the editor, are generated from this grammar.

In this paper, we outline the overall design of a hybrid editor generator and describe briefly the interface presented to a programmer by the hybrid editor. We define the meta-language in which the application language is specified for the generator, and describe the operations performed on the language specification in the process of generating the editor. Based on this design, a prototype hybrid editor generator, called HEG, has been built and is being tested. We describe possible extensions to the meta-language to describe program semantics, and the generation of the procedures to check those semantics during program construction.

II. BACKGROUND

Although syntax-directed editing can be very helpful to a programmer who is unfamiliar with the language he is using, one who knows the syntax of the language may easily become frustrated with the plethora of menu choices that must be made to "write" a program substructure as simple as an assignment statement. For this reason, allowing the programmer the option of giving the editor a string to parse and insert into the partially expanded program is a desirable feature to have in such a menu-driven editor. We define a hybrid editor to be a syntax-directed editor with the ability to parse and to integrate into the program tree a string given at any time during the expansion of a program.

There have been attempts to generate syntax-directed editors automatically for several programming languages from specifications of the syntactic (given by a context-free grammar)[5] and the semantic (given using attributes for the symbols in the grammar) aspects of a language.[10] There is also a system in use that requires only the addition of a context-free grammar for any new language for which it is to operate.[4] (In this latter system, detailed knowledge of the structure of the grammar seems to be required in order to use the editor with any grace.) Usually, however, such editors are built individually for each programming language. This is because (1) high-level programming languages have many context-sensitive semantic constraints that cannot be expressed by a context-free grammar, (2) programmers using high-level languages usually need more local context-sensitive aid than syntactic help, and (3) the external interface that such an editor is expected to provide depends heavily upon the language it supports.

Little attention, however, has been paid to syntax-directed editors for special-purpose languages such as the input specification languages for Yet Another Compiler-Compiler (YACC)[11] and database interface languages such as HISEL.[12] Yet it is for these seldom used, but numerous, languages that a hybrid editor would be most useful. Users of such application programs often write their input in a file using a regular text editor and manually check conformity with the syntactic constraints imposed by the particular application program. Some of these application programs have parsers in their front ends to check the syntactic correctness of their input. (Sometimes these parsers are automatically generated from utilities such as YACC.[11]) Others just assume that their input is syntactically correct and abort when it is not. If, in place of the parser, a hybrid editor is available within the application program, the user can be guided by the program's editor during input construction.

A tool to automatically generate such editors for application languages can be quite useful for several reasons: (1) A special-purpose application language, unlike a programming language, is likely to change more rapidly with an evolving application program. (2) Many application programs are not frequently used and hence their idiosyncratic syntactic constraints are likely to be forgotten. (3) The amount of syntactic help from a hybrid editor can be determined by the programmer. (4) The derivation tree built by the hybrid editor may be used by the application program to process its input in a more structured manner than is often possible when only a textual view of the input is available to the program. It is for these types of languages that HEG-generated hybrid editors, called AGEs (Automatically Generated Editors), are most valuable.

III. THE EDITING PROCESS

In a HEG-generated hybrid editor for a given language, a program is internally represented by a derivation tree of the program in that language, along with a symbol table containing the programmer-defined character strings. (See Appendix A for an example of a short session with an AGE.) When the programmer wishes to expand a particular nonterminal in the tree by syntax-directed menu selection, and if there are two or more production rules for that nonterminal, then the AGE creates a one-item-at-a-time wraparound menu, on the bottom line of the screen, displaying the menu strings associated with those production rules. After a production rule is chosen for expansion, the internal node for the nonterminal is grown in such a way that the frontier of the subtree rooted at that internal node corresponds to the right-hand side of the production rule selected. If, however, the programmer wishes to forego menu selection, he/she may expand a nonterminal by entering a string that is derivable from the nonterminal. The editor will parse the string, build a subtree representing the string, and graft the subtree into the program tree at the node corresponding to the nonterminal. The string given by the programmer may be any combination of terminal strings and nonterminal names.

Even though the internal representation of a program is a tree, both tree and text interfaces are provided to the programmer for navigation through the program. At any moment, a tree cursor navigating around the internal nodes of the tree is available. The programmer sees the cursor spanning the entire frontier of the subtree rooted at the tree-cursor node. He can move the tree cursor along the internal nodes of the tree (using the commands up [^], down [V], left-sibling under the same parent [<], right-sibling under the same parent [>], next internal node of the same type under the same parent [N], etc). Or, he can navigate textually (using the commands n for next word, b for back word, CR for next line, - for previous line) through the program. The commands 'e' and 'u' provide syntax-driven expansion and unexpansion facilities. Commands 'p' and 'r' allow parser-driven nonterminal expansion and file-reading facilities.

When the programmer quits editing, both the text and the tree versions of the program are saved. The tree is saved by storing the leftmost derivation sequence of the production rules in the derivation tree and the symbol table. This sequence is used to reconstruct the tree for a later editing session on that program. The test is saved for possible use by other tools.

IV. ARCHITECTURE OF HEG

HEG produces a generalized table-driven syntax-directed editor, linked with a parser, for a given application language. The application language is specified by an AGE grammar in the meta-language described in the next section. A parser generator front end, named 'GENPAR', generates a YACC file and a LEX file[13] from this specification of a language. The parser generated from these two files is capable of parsing strings, which may contain nonterminal names, starting from any nonterminal of the grammar. The grammar tables used by the editor, the parser, and the generic editing routines are then linked together to generate an AGE for the given language.

Figure 1 shows the interactions between the different components of an AGE.

V. THE META-LANGUAGE

The "meta-language" for HEG is the grammar specification language in which the production rules of any context-free grammar are specified, along with the pretty-printing information required by the display utilities of the editor (e.g., indentation and color codes, if available) and the strings for the menu lists. As an example of the use of the language,

```
query           : ⟨QUERY⟩
                | "select" list "where" conds
                ;
```

is a normal production rule. Here, ⟨QUERY⟩ is a "user-friendly" name for the nonterminal symbol 'query'. The sequence of strings following the character '|' specifies a production for the nonterminal 'query'. Strings within double quotes in the production specification are terminal characters and others are nonterminals. The terminal character strings may also contain the following special characters denoting pretty-printing information: '\n' for a new-line, '~' for a blank character, '\t' for tabbing one position to the right and increasing the tab count by one for the starting positions of the subsequent lines, and '\T' for decreasing the tab count by one for the current and the subsequent lines.

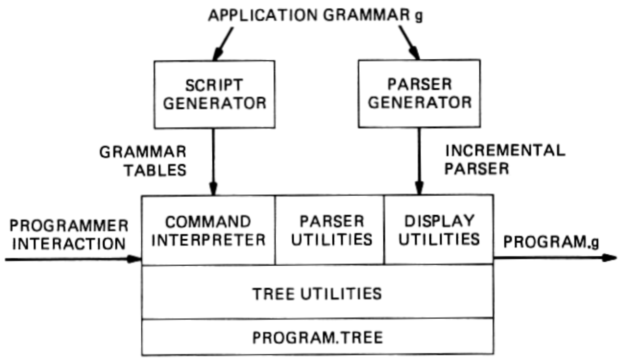


Fig. 1—Interactions between the components of an AGE.

Another example is

```
list          : ITEM_LIST
              |+item ","
              ;
```

This is an iterative production rule. It specifies that the nonterminal, 'list', can be expanded into one or more occurrences of the nonterminal, 'item', separated by the terminal character string ','. A '*', in place of the '+' above, would denote zero or more occurrences of the nonterminal.

There may be more than one production rule for a nonterminal; if so, all of the rules for the nonterminal must be specified in one rule set, each preceded by a '|'. For example, the following two sets of rules show several possible expansions for the corresponding nonterminals:

```
clause        : CLAUSE
              | field_name "~" op "~" constant
              | constant "~" op "~" field_name
              | field_name "~" op "~" field_name
              ;
op            : OPERATOR
              | "="
              | "!="
              | ">="
              | "<="
              | ">"
              | "<"
              ;
```

For each production rule, the grammar specification must include an associated "expansion character". This expansion character must be unique within the set of rules for that nonterminal. If there are two or more rules for the expansion of a nonterminal and if an instance of that nonterminal in the program is to be expanded, the programmer must indicate his choice by typing the expansion character associated with the desired rule. However, if the programmer wishes to examine all the choices for that nonterminal, the AGE creates a one-item-at-a-time wraparound menu on the bottom line of the screen, from which the programmer may choose a production. Therefore, the implementor (the person defining the grammar) must provide a "menu-item-string", which is displayed in the menu for each production rule. These menu items should be suggestive of the associated production rule for better communication with the programmer. When a particular rule is chosen for the expansion, the right-hand side of that rule replaces the left-hand side nonterminal node in the program's syntax tree.

To illustrate the provision of menu information in a grammar for HEG as above and to clarify the grammar specifications further, a complete set of rules for a database query language HISEL[12] is given below. Sentences (queries) in this language have the form:

select x1.fld1,x2.fld2, ⋯ where ⟨CLAUSES⟩ or ⟨CLAUSES⟩ ⋯

where 'x' is a cursor name, 'fld' is a field name, and ⟨CLAUSES⟩ is a conjunctive sequence of field conditions. The single characters following the menu items in the rules are the expansion characters. For the rule sets having just one production rule, meaningless menu-item strings (e.g., 'xxx') and expansion characters (e.g., 'x') are used for uniformity in the meta-syntax. Observe that there is no white space in the menu-item strings.

```
query        : ⟨QUERY⟩
             | xxx x "select" list "\nwhere~" conds
             ;
list         : ⟨ITEM_LIST⟩
             | xxx x + item ","
             ;
item         : ITEM
             | xxx x cursor "." field_name
             ;
conds        : ⟨CONDITIONS⟩
             | xxx x *disjunct "\n~~~or~"
             ;
disjunct     : ⟨CLAUSES⟩
             | xxx x +clause "~,~"
             ;
clause       : CLAUSE
             | field_op_const 1 field_name "~" op "~" constant
             | const_op_field 2 constant "~" op "~" field_name
             | field_op_field 3 field_name "~" op "~" field_name
             ;
op           : OPERATOR
             | equal = "="
             | not_equal ! "!="
             | greater_equal 1 ">="
             | less_equal 2 "<="
             | greater > ">"
             | less < "<"
             ;
```

The left-hand side nonterminal of the first rule in the grammar specification (i.e., 'query', above) is the starting nonterminal. If no production rule is available for a nonterminal (e.g., ‘field_name’), and that nonterminal appears during the derivation of a program, then it is assumed to expand into a character string to be provided by the programmer at the time of expansion. Such nonterminals are said to derive “identifiers”. A regular expression specification for any such nonterminal can be used to restrict the format of the identifier strings that a programmer may supply at the time of expansion. HEG provides a default specification for all nonterminals deriving identifiers and having no regular expression specification.

As another example, the following is a grammar in the meta-language (meta-grammar) for the meta-language described in this section. In fact, one can invoke an AGE for this meta-language to enter a grammar specification for any user-defined language.

```
gram          : ⟨AN_AGE_GRAMMAR⟩
              | xxx x cfrules “\n%%\n” rerules
              ;
cfrules       : ⟨CONTEXT_FREE_RULES⟩
              | xxx x +ruleset “\n”
              ;
ruleset       : ⟨A_RULE_SET⟩
              | xxx x nonterm-name “\t:~” user_name “\n” rules
                  “\n;\n\T”
              ;
rules         : ⟨RHS_OF_A_RULE_SET⟩
              | xxx x +rule “\n”
              ;
rule          : ⟨A_RULE⟩
              | normal-rule n “|~” menu_string “~” exp_char “~”
                  tklist
              | nonempty-rec-rule + “|~xxx~x~+” nonterm_name
              “~\“” separator “\””
              | empty-rec-rule * “|~xxx~x~*” nonterm_name “~\“”
                  separator “\””
              ;
tklist        : ⟨A_SEQ_OF_TOKENS⟩
              | xxx x +token “~”
              ;
token         : TOKEN
              | nonterminal-token n nonterm_name
              | terminal-token t “\“” terminal_str “\””
              ;
rerules       : ⟨REGULAR_EXP_RULES⟩
              | xxx x *rerule “\n”
              ;
```

```
rerule          : ⟨REG_EXP_RULE⟩
                | xxx x nonterm-name "~~\~##!" reg_expr
                ;
%%
exp_char        ~##![a-zA-Z0-9+*^<>&]
```

VI. THE PARSER GENERATOR FOR HEG

The tools YACC and LEX are used to produce a parser and a scanner for an AGE system. The parser generator front end, GENPAR, takes, as input, the AGE grammar specification for the desired language and produces YACC and LEX specification files. (Note that the use of YACC implies that the input grammar must be LALR(1) in order to generate a parser for an AGE system.)

The generated parser can be invoked to parse an input string representing the expansion of any nonterminal node in the program derivation tree. The string to be parsed can be any combination of terminal strings and nonterminal names that is derivable from the nonterminal at the "current" position of the editing cursor.

6.1 The parsing of an input string

When the programmer provides a string to be parsed, rather than making a menu selection, the AGE system opens a temporary file, writes the prefix *$$NONTERMINAL_NAME$$* to the file (where *NONTERMINAL_NAME* represents the nonterminal from which the programmer's input string is to be derived), and appends the input string. The parser is called to process the entire string contained in the file. The prefix is considered to be an integral part of the input string. If the portion of the string entered by the programmer cannot be derived from the nonterminal indicated by the prefix, the input will be considered syntactically incorrect. If a syntax error is found, the temporary file is saved in case the programmer should wish to edit the string and resubmit it to the parser.

The parser will parse its input in a "backtracking bottom-up" fashion, constructing a syntax tree to represent the derivation of the input string. After the tree is constructed, a preorder traversal of the tree is performed, producing a list of production numbers representing the leftmost derivation of the input string. This list is then passed back to the controlling program in AGE.

In an AGE, the YACC-generated parser is run subordinate to a "parser monitor" to provide a certain amount of parser backtracking, made necessary by the conflict resolution scheme of the LEX-generated scanner. For such a scanner, the lexical tokens desired are specified by regular expressions. When two or more of the given regular expressions match equal-length segments of the input (starting at the "current input position" of the scanner), and these are the longest matches possible, the scanner will select the regular expression that had been listed first (textually) in the input specification file as the "correct" match, and return the corresponding token number. If the parser is expecting one of the other possible matches at the current input position, the parser will find a "syntax error" where no error may, in fact, exist.

The parsers produced by the YACC program are standard shift/reduce ("bottom-up") parsers. The required backtracking is accomplished through the interaction of code at two levels of the parsing scheme. At the lowest level, the handling of multiple matches in the scanner is slightly modified by forcing the scanner to "reject" an "identifier" match, after linking the token number associated with the match into a "token map." In this manner, all possible matches for an "identifier" are linked into the token map for the parse. At the highest level, the parser monitor runs the shift/reduce parser, handing the parser token numbers from either the scanner or the token map, depending upon the current state of the parse. Then, if the YACC-generated portion of the parser discovers a syntax error in the input, the monitor can rotate the last set of entries in the token map to (temporarily) "forget" the "preferred" token number for the last "choice position" in the input stream, allowing the use of another possible token number for that position.* In this manner, no syntactically correct input will be declared to contain a syntax error, and an incorrect input will only be rejected after trying the allowable combinations of token numbers for the "identifier" positions.

6.2 What the parser generator does

Several transformations must be applied to the AGE grammar to produce a grammar specification that is acceptable to YACC, and that will specify a parser offering the features and enforcing the constraints desired. All the transformations are performed and the YACC grammar specification is produced in a single pass over the input grammar.

6.2.1 Starting the derivations from arbitrary nonterminals

For every nonterminal in the AGE grammar, two additional productions are generated. One such set of productions makes it possible to

*Due to the relative simplicity of most application languages (in terms of permissible "identifier" combinations and possible token map complexity), the erroneous "identifier" token number will usually be in the last set of entries in the token map. So, in most cases, only the last one or two "identifiers" need to be retried (if the input really is syntactically correct). In all cases, the LALR(1) property of the grammar should aid in the isolation of groups of points in the "identifier" cross-product space that could not possibly be characteristic of a correct parse. The points in these groups, then, need not be considered individually during parsing.

start a derivation from any nonterminal in the original grammar, even though YACC will allow the specification of only one start symbol for a grammar. Each such added production is of the form:

ppstart : AGE_TERM_20 clause

where "ppstart" is a newly defined start symbol for the grammar, "clause" is a nonterminal in the original grammar, and "AGE_TERM_20" is the token returned by the scanner when it reads the prefix (e.g., "$$clause$$") encoding the nonterminal from which the remainder of the input is to be derived. This type of production also allows the enforcement of the rule that the input string must be derivable from the nonterminal at the "current" node in the program tree.

6.2.2 Use of nonterminal names

The second set of productions generated for each nonterminal allows the acceptance of a "user-name" for a nonterminal, in place of a string that could be derived from that nonterminal, wherever the nonterminal may appear in an input sentence. Each of these productions is of the form:

clause : AGE_TERM_22

where "AGE_TERM_22" is the token returned by the scanner when it reads the user-name for the nonterminal on the left-hand side of the production (e.g., "clause").

6.2.3 Iteration in the grammars

The iterative specifications in the AGE grammar must be transformed into explicit left recursions, with the separators appropriately treated. Since the most general form of iteration in AGE grammars is that of a list item repeated zero or more times, with a nonwhite-space separator, that case is considered here. The AGE grammar specification for such a list might be:

clauses : *clause ","

where "clause" is the nonterminal to be repeated in the list structure and "," is the terminal separator to appear between list elements.

The YACC specification corresponding to the above production would be:

clauses : Xclause_2_0X

where Xclause_2_0X is a new nonterminal encoding the nonterminal to be repeated as the list elements (e.g. "clause"), the number of the terminal string to be used as the list element separator (e.g., "2"), and the minimum number of times the nonterminal must appear in the list (e.g., "0"). Such encoding permits the use of the list item nonterminal, "clause," as the repeated element in other lists with different separators and/or different minimal numbers of occurrences.

To derive the required number of occurrences of "clause," sets of productions of the following form must be added to the YACC grammar:

```
Xclause_2_0X   : e
               | Xclause_2_1X
```

where "e" represents the empty string. The second new nonterminal, Xclause_2_1X, denotes one or more occurrences of "clause" separated by occurrences of terminal number 2. For this second new nonterminal, sets of productions of the following form are added to the YACC grammar:

```
Xclause_2_1X   : clause
               | Xclause_2_1X AGE_TERM_2 clause
```

where "AGE_TERM_2" denotes the required separator ("terminal number 2") between elements of the list.

Note that other iterative specifications are specializations of the above case. If the list must be nonempty, only the second new nonterminal, with its associated productions, is generated. If the list element separator is not significant (i.e., if it is any form of white space), then no terminal is encoded in the new nonterminal(s) or included in any of the new productions.

6.2.4 Treatment of identifiers

In an AGE input grammar, there are no explicit productions for the derivation of character strings representing identifiers. Any nonterminal that does not appear on the left-hand side of any production, in an AGE grammar specification, may produce an identifier. Since there are no such implied rules in a YACC grammar specification, GENPAR must add explicit productions to the grammar to allow the reduction of terminal identifier strings to appropriate nonterminals. These productions are of the form:

```
cursr : AGE_IDENTIFIER_5
```

where an identifier number (e.g., "5") is specified only if the default lexical specification is overridden for the nonterminal on the left-hand side of the production (see below).

To handle the lexical specification for each terminal identifier, the grammar designer has two choices. The designer may use GENPAR's default specification, which allows identifiers to be any combination of letters, digits, and the characters "-", "_", and ".", which begins with a letter, and is not a reserved terminal string in the language. He may, instead, associate an arbitrary LEX specification with any nonterminal that may derive an identifier. The given specification would then be used to override the default identifier specification for that particular nonterminal.

6.2.5 *Treatment of white space*

In the generation process, all types of white space in the AGE grammar specifications of terminal strings are treated identically, and are considered insignificant to the specification of the generated grammar. For example, if a "PROGRAM" is specified as a "list of statements separated by new-lines", the YACC-generated parser will accept any "list of statements separated by any white space (e.g., blanks, tabs, or new-lines)" as a "PROGRAM".

6.3 *Generation of YACC actions*

If the input string to the parser is valid, AGE requires the output of the parser to be a list of rule numbers, symbol table indices, and list item occurrence counts representing the leftmost derivation of the input string. Since the YACC-generated parser is a shift/reduce parser, the order in which that parser uses the grammar productions will not represent a leftmost derivation of the input. To produce the proper input for the AGE monitor, the parser builds a tree to represent the derivation of its input string as it parses the input string, and, as part of the last production applied (reduction to the start symbol), performs the preorder traversal of the tree, generating the required list of numbers.

The building of this tree is the main activity of the "actions" generated for each production in the YACC specification. To generate code to appropriately link nonterminal sub-trees, the positions of all the nonterminals in a given production rule must be saved as the production is processed by the generator. This is accomplished by counting the tokens on the right-hand side of the given production rule and stacking the position numbers that correspond to nonterminals. Actions are then generated that call precoded subroutines that link the tree nodes together. These subroutines, and their associated data and type declarations, are constant across all grammars, and are included in the appropriate sections of each generated YACC grammar.

6.4 *The size of the processed grammar*

The growth of the grammar in the generation process is actually not as large as may be imagined. If n = the total number of nonterminals and r = the number of iterative nonterminals in the original

AGE grammar, at most $3n + 3r < 6n$ productions, $2n - 1$ nonterminals, and betweeen $2n + 1$ and $3n - r$ terminals are added to the grammar before YACC generates the parser.

VII. ATTRIBUTES AND SEMANTIC PROCESSING

One possible extension to this work would involve the addition of attributes to the grammars described above. These attributes could allow a certain amount of semantic checking of user programs, in addition to allowing interpretation and/or code generation (for simple application languages) during program construction. The basic principles behind attribute grammars are described elsewhere;[14] we shall only discuss the required extensions to our meta-language and the generation of evaluation functions for the attributes.

7.1 Extensions to the meta-language

The meta-language described in Section V can be enhanced to include attributes and their evaluation routines in each production. An example of the use of the enhanced meta-language is shown below.

```
clause          : CLAUSE
                  inherited: int temp_loc;;
                  synthesized: int next_temp;
                | field_op_const 1 field_name "~" op "~" constant
                  {
                  $0.next_temp = $0.temp_loc;
                  }
                | const_op_field 2 constant "~" op "~" field_name
                  {
                  $0.next_temp = $0.temp_loc;
                  }
                | field_op_field 3 field_name "~" op "~" field_name
                  {
                  $0.next_temp = $0.temp_loc + 2;
                  }
                ;
```

In general, the attributes and their evaluation rules are specified in a pseudo-C language notation. We provide data typing facilities for the attribute variables.* The inherited attributes of a nonterminal and their data types are listed after the key word "inherited", which appears after the user-name of that nonterminal. The synthesized attributes

*As with the attributed grammars for programming languages, our experience in using attributed grammars for application languages suggests that facilities that include standard libraries of functions, user-defined types, and global attributes are needed.

of that nonterminal are then listed in a similar fashion. The attribute evaluation rules are listed after each production specification, within braces. "$0.attribute_name" in these rules refers to the correspondingly named attribute of the nonterminal in the left-hand side of the production. "$i.attribute_name" refers to the correspondingly named attribute of the *i*th nonterminal in the right-hand side of the production.

For each inherited attribute of each nonterminal appearing in the right-hand side of a production rule, there must be an evaluation rule (a function call or an arithmetic expression) assigning a value to that attribute associated with that production rule. Similarly, for each synthesized attribute of the left-hand side nonterminal of a production rule, there must be a rule (a function call or an arithmetic expression) assigning a value to that attribute. These rules may take as arguments the inherited attributes of the left-side nonterminal and/or synthesized attributes of the nonterminals in the right-hand side. These rules must not, however, violate the properties defining *L*-attributedness[14] of the grammar in order for the proposed evaluation scheme to work.

For iterative production rules, the semantic specifications appear as:

```
disjunct     : ⟨CLAUSES⟩
              inherited:;
              synthesized:;
             |xxx x +clause "~,~"
              {
              int current_temp;
              init: {
                 current_temp = 0;
              }
              repeat: {
                 $1.temp_loc = current_temp;
                 current_temp = $1.next_temp;
                }
              }
             ;
```

The routine following the key word "init:" is executed when this production rule is initially chosen. Then, for each instance of the nonterminal "clause" under the parent "disjunct", the specification following the key word "repeat:" is used. The "$1" in the latter specification refers to the instance of the nonterminal "clause" whose attributes are being evaluated. These evaluation specifications must follow the same guidelines as those of the regular productions. Nonterminals deriving "identifiers" (i.e., those having no production rules) are assumed to have only one synthesized attribute, "value", whose value is the string entered by the programmer at the time of expansion.

7.2 Generation of the attribute evaluators

In any specification of an application language as above, each implementor-defined attribute evaluation function is associated with a specific production. For each distinct attribute in the grammar, the calls to these functions are collected into one generated evaluation function, which determines the appropriate implementor-defined function to call, based on the rule that produced the associated nonterminal. To make the required environment information accessible during the attribute evaluation, a pointer to the tree node at which the generated function is to be evaluated is passed as a parameter to the generated function. The evaluation functions for inherited attributes must be passed a pointer to the parent node of the nonterminal node with which the attribute is associated (i.e., inherited attributes are evaluated when the associated nonterminal appears on the right-hand side of a production). Synthesized attribute evaluation functions must be passed a pointer to the nonterminal node with which the attribute is associated (i.e., synthesized attributes are evaluated when the associated nonterminal is expanded).

Each generated evaluation function for an inherited attribute contains a section of code for each production in which the associated nonterminal can appear on the right-hand side. Similarly, each generated evaluation function for a synthesized attribute contains a section of code for each production in which the associated nonterminal appears on the left-hand side. Within each of these sections of code is the code for the appropriate implementor-defined function, along with the function calls required to evaluate the actual parameters of the implementor's function. The attributes are evaluated only as required, and the evaluation nesting is managed automatically by the C (target programming language) procedure calling mechanism.

VIII. REMARKS

The editor generator HEG, as described here (excluding the semantic analysis), currently exists and has been used for a variety of application languages. The novel aspects of this system include: (1) the ability to give a programmer the option to use menu selection or to enter strings containing terminal characters and nonterminal names at *any* stage during the expansion of a program in the user-language, and (2) the ability to generate a hybrid editor from a high-level specification of a user grammar. This system was an experiment to investigate only these aspects of editing. Various other related issues such as building a robust syntax-sensitive editor for higher-level languages (e.g., C language), experimenting with different ways of exhibiting the menus, and human-factors issues are being addressed separately.[5]

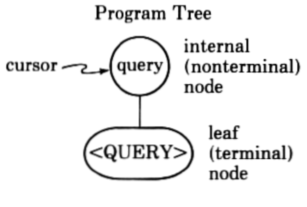
REFERENCES

1. V. Donzeau-Gouge et al., “A Structure-Oriented Program Editor: a First Step Towards Computer-Assisted Progamming,” Proc. of Int. Computing Symp., Antibes, June 1975.
2. P. H. Feiler and R. Medina-Mora, “An Incremental Programming Environment,” Carnegie-Mellon University, Computer Science Department Report, December 1980.
3. C. N. Fischer, G. Johnson, and J. Mauney, “An Introduction to Release 1 of Editor Allan Poe,” University of Wisconsin, Technical Report 453, 1981.
4. C. W. Fraser, “Syntax-Directed Editing of General Data Structures,” Proc. ACM SIGPLAN-SIGOA Symp. Text Manipulation, Portland, Oregon, June 1981, pp. 17–21.
5. E. R. Gansner et al., “SYNED—A Language Based Editor for an Interactive Programming Environment,” Spring COMPCON 83 San Francisco, California, February 1983, pp. 406–10.
6. M. R. Horton, “Design of a Multi-Language Editor with Static Error Detection Capabilities,” Ph.D. Dissertation, Computer Science, University of California, Berkeley, July 1981, p. 158.
7. T. Teitelbaum et al., “The Why and Wherefore of the Cornell Program Synthesizer,” Proc. ACM SIGPLAN-SIGOA Symp. on Text Manipulation, Portland, Oregon, June 1981, pp. 8–16.
8. S. R. Wood, “Z—The 95% Program Editor,” Proc. ACM SIGPLAN-SIGOA Symp. on Text Manipulation, Portland, Oregon, June 1981, pp. 1–7.
9. R. C. Waters, “Program Editors Should Not Abandon Text Oriented Commands,” ACM SIGPLAN Notices, *17,* No. 7 (July 1982), pp. 39–46.
10. T. Reps, “Optimal-time Incremental Semantic Analysis for Syntax-directed Editors,” Proc. 9th Annual Symp. on Principles of Programming Languages, Albuquerque, New Mexico, January 1982, pp. 169–76.
11. Stephen C. Johnson, “Yacc: Yet Another Compiler-Compiler,” Computer Science Technical Report #32, Bell Laboratories, Murray Hill, 1976.
12. E. R. Gansner et al., “Semantics and Correctness of a Query Language Translation,” Ninth Symp. Principles of Programming Languages, Albuquerque, New Mexico, January 1982, pp. 289–98.
13. M. E. Lesk, “Lex—A Lexical Analyzer Generator,” Computer Science Technical Report #39, Bell Laboratories, Murray Hill, 1975.
14. P. M. Lewis, D. J. Rosenkrantz, and R. E. Stearns, “Attributed Translations,” J. Computer and System Sciences, *9* (December 1974), pp. 279–307.

APPENDIX A

An Editing Session With ‘Hisel.age’

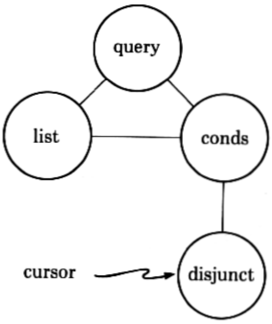
```
$hisel.age
program? test
Initializing the tree/text for test..
     Terminal Screen
1) ⟨QUERY⟩
```



```
:r <- user command - invisible
file name? temp
```

Assume that there is a file named 'temp' in the working directory and that it has the string "select ⟨ITEM_ LIST⟩ where ⟨CLAUSES⟩". AGE will read it, recognize the two nonterminal names, parse the string, create the subtree, and graft it to the root as shown below.

```
2) select ⟨ITEM_ LIST⟩
   where ⟨CLAUSES⟩
```

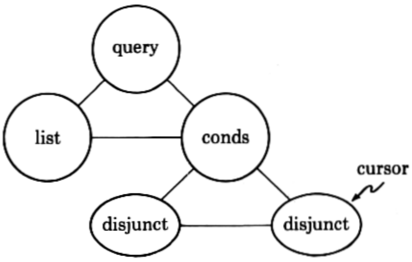


(Leaves are not shown)

```
:a
```

This will add one more instance of CLAUSES. The screen after this command will be:

```
3) select ⟨ITEM_ LIST⟩
   where ⟨CLAUSES⟩
      or ⟨CLAUSES⟩
```



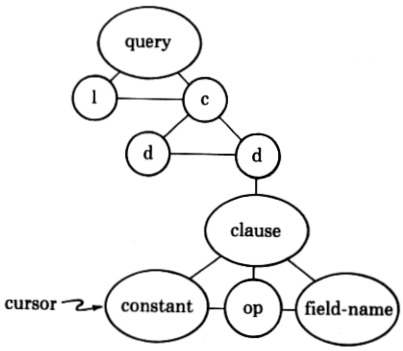
```
:e
:e
```

AGE will install a CLAUSE in place of ⟨CLAUSES⟩ after the first e command. Since there is a choice for the expansion corresponding to the next e command, the following menu items will appear one after another if the user asks for a menu:

```
expansion character (type ? for menu): ? <- user types this
field_op-const 1 y(es), n(ext), q(uit)? n <- user types this
const-op-field 2 y(es), n(ext), q(uit)? y
```

The screen after this will be:

```
4)  select ⟨ITEM_LIST⟩
    where ⟨CLAUSES⟩
       or constant OPERATOR field_name
```



:w

This command will write the above text into test.hisel file and the tree representation in test.lmd file.

:q

AGE will then print:

BYE! test hasn't been fully expanded; call me back later.

APPENDIX B

AGE Command Summary

Text-navigation commands:

cr - a carriage return : go to the next line
- - the character minus: go back one line
space-bar or n : go to the next word
b : go back one word

Tree-navigation commands:

^ : go to the parent of the current nonterminal
V : go to the first son of the current nonterminal
> : go to the right neighboring nonterminal under the same parent
< : go to the left neighboring nonterminal under the same parent
N : go the next unexpanded nonterminal
B : go backwards for the next unexpanded nonterminal

Nonterminal expansion commands:

p : parse text for the current nonterminal
r : read text from a file and parse it for the current nonterminal
e : expand the current nonterminal by menu selection
a : append an instance of a 'listy' (i.e., iterative) nonterminal
i : insert an instance of a 'listy' nonterminal
d : delete the current instance of the 'listy' nonterminal

Other commands:

U : unexpand nonterminal
w : write program
q : quit editing with AGE

AUTHORS

Beth A. Bottos, B.S.E.E., 1979, Purdue University; M.S. (Electrical Engineering), 1980, Stanford University; M.S. (Computer Science), 1983, Carnegie-Mellon University. From 1976 to 1979, Ms. Bottos was a participant in the Bell Laboratories Engineering Scholarship Program (BLESP). During that time, her work included telephone ringer characterization and modification and hardware design for a home appliance control system. From 1979 to 1981, she was involved in the development of the 1AVSS (Voice Storage System) and LADT (Local Area Data Transport) systems. Since 1981, Ms. Bottos has been a student in Computer Science at Carnegie-Mellon University. During that time, she has also worked for Bell Laboratories in the Advanced Software Department and in the Advanced Software Technology Laboratory.

Chandra M. R. Kintala, B.Sc. (Honors), 1970, Regional Engineering College, Rourkela, India; M.Tech. (Electrical Engineering), 1973, Indian Institute of Technology, Kanpur, India; Ph.D. (Computer Science), 1977, Pennsylvania State University; Assistant Professor, Computer Science Department, University of Southern California, 1977–1980; Bell Laboratories, 1980—. Mr. Kintala is a member of the Advanced Software Department. He has worked on the complexity of nondeterminism in various classes of Automata and Turing machines. His current interests include programming environments, compiler techniques for database query language translators and software graphics. Member, ACM, IEEE Computer Society, Sigma Xi, Phi Kappa Phi, and Who's Who in the East.