A MULTI-LANGUAGE SYNTAX-DIRECTED EDITOR

Donald J. Bagert, Jr.
Northeast Louisiana University

Donald K. Friesen
Texas A&M University

Abstract

A limitation of current selection-entry syntax-directed editors is that a particular editor is limited to manipulating programs in one particular programming language. A primary goal of the IMEGS research project was to construct a selection-entry syntax-directed editor which can manipulate programs in a wide variety of languages. An IMEGS prototype has been developed which contains a multi-language editor.

1. Introduction

Syntax-directed editors (SDEs) provide many advantages over the standard text editor/compiler programming environment: the editor has knowledge of the language syntax, compilation is incremental, and execution may be halted and restarted [Teitelbaum and Reps 1981]. However, a limitation of syntax-directed editors to date has been that a particular SDE can be used as an environment for only one language.

A selection-entry syntax-directed editor is defined as an SDE which specifies most of the parse tree ordinarily constructed by a compiler through the use of templates chosen by the programmer during program construction [Allison 1983]. At the lower levels of the language definition are phrases [Teitelbaum and Reps 1981] which are input in text-entry mode and are compiled immediately after input.

There have been several attempts to bring about a greater degree of language independence to selection-entry syntax-directed programming environments. The Synthesizer Generator project at Cornell [Reps and Teitelbaum 1984] was an extension of the Cornell Program Synthesizer, which introduced many of the concepts currently used in SDEs [Teitelbaum and Reps 1981]. To design a language for an SDE to be constructed by the Synthesizer Generator, an input file would be created describing the syntax and semantics of the language, including its nonterminals and productions. The output of executing the Synthesizer Generator with this input file would be the desired selection-entry syntax-directed programming environment. A similar editor generator, called ALOEGen, was developed at Carnegie-Mellon [Medina-Mora and Notkin 1981]. In the case of both editor generators, a different programming environment would be developed for each language defined for a generator.

Recently, there has been some research at Texas A&M on creating a language-independent selection-entry SDE [Heyman and Lively 1985]. In this method, the editor reads in a file with the production templates for the desired language. However, the editor was used for program creation only; after exiting the editor, the program would have to be submitted to a standard compiler and run-time environment.

The IMEGS (An Incremental Multi-language Editor Generator and System) research project [Bagert 1986] was initiated with two primary goals: to have one SDE be able to manipulate programs in a variety of block-structured languages, and to provide an environment which allows IMEGS system users to incrementally construct their own languages to be used in the system editor. An IMEGS prototype has been constructed at Texas A&M which contains a selection-entry syntax-directed editor which can be used for manipulation of programs in several different languages. This paper reports the results of the implementation of the prototype editor.

2. The Editor Interface

When invoking IMEGS, the user is to supply the name of the language to be used within the IMEGS environment. (There is an IMEGS system security module which can limit the access that each user has to each language.) If the language exists, the information in the file for that language is input and serves as parameters for the system editor.

After successfully entering the IMEGS environment, the user may choose the editor mode in order to create or modify programs in the desired language. (There are other modes, for language definition and modification of security, which will not be discussed in this paper.) The interface to the editor is similar to the Cornell Program Synthesizer [Teitelbaum, Reps, and Horwitz 1981]. Templates may be specified at the higher levels of the language, while text-entry mode is used to input phrases at the expression level. Incremental compilation of IMEGS

© 1987 ACM 0-89791-218-7/87/0002-0300 75¢

phrases is accomplished by means of am operator-precedence parser.

3. An Internal View of the Prototype

3.1 Language Representation

The IMEGS prototype was implemented on a VAX 11/750 running under Berkeley UNIX 4.2 using the C programming language. The list of all IMEGS languages is in the file languages. There is a separate file for each language L defined within the IMEGS system. The definition of L can be classified in four sections, as is shown in Table 1.

Table 1. Language definition sections.

- security section
- lexical information section
    - character set definition
    - identifier name format
    - data type definition
    - constant format
    - reserved word definition
- operator information section
    - operator types
        - infix
        - unary prefix
        - unary postfix
        - unary group
        - binary group
    - symbol definition
        - priority levels
        - intermediate code instructions
    - associativity rules
- nonterminal/production section
    - template-oriented categories
    - text-oriented categories

The security section specifies what kind of access each system user has to L. The lexical information section supplies information needed for the lexical analysis of phrases in L.

The operator information section specifies the definition of operators to be used in phrases for the language L. There are five types of operators defined in the prototype: infix, unary prefix, unary postfix, unary group (e.g., using parentheses for grouping), and binary group (e.g., using brackets for subscript references in Pascal). The infix, unary prefix, and unary postfix operators are referred to as non-group operators while the unary and binary group operators will be referred to as group operators.

A set of symbols have been defined for each non-group operator type. A symbol may be a name such as mod or a delimiter such as +. Each symbol has an associated priority level which determines the hierarchy of operations in L. For each symbol, several intermediate code instructions have been defined; for example, the infix symbol + might be defined for both integer and real addition. The intermediate code instruction set has been predefined by IMEGS. Each instruction associated with a symbol by the user creating the language can be defined for different data types; for instance, real addition for the symbol + can be defined as legal for two real arguments or a mixture of integer and real arguments. In this manner, any coercion rules within phrases can be defined very precisely.

The definition of group operators has a few differences from the definition of non-group operators such as +. For instance, symbols such as + are not defined to stand for intermediate code instructions; instead, a pair of group operators is defined to perform a particular operation associated with an intermediate code instruction. Legal group operators in the prototype are (and), [and], and { and }. Also, due to the nature of the operator-precedence parser, unary group operators do not need an associated priority level.

Within the operator definition section, associativity rules are defined for each priority level.

The nonterminal/production section specifies what nonterminals have been defined for the language L and the production templates defined for each nonterminal. There are two classes of categories in IMEGS. The template-oriented categories are nonterminals which are placeholders for templates, while the text-oriented categories are nonterminals which are placeholders for phrases.

For each template-oriented nonterminal there has been defined a series of production templates. The template syntax is variant of Backus-Naur Form ([Horowitz 1984], page 51) and consists of a series of sections. A section can consist of a nonterminal or a sequence of terminal symbols. For example, the template

```
if {boolean_expression}
then {statement}
else {statement}
```

consists of six sections, three nonterminal sections and three terminal sections. For each section there is a set of prettyprinting rules, which are part of the semantic attributes [Knuth 1968] associated with the program. The next section describes other types of attributes associated with each template.

Text-oriented categories are used for the input of phrases. There are also semantics associated with these categories, which are discussed in the next section.

3.2 Program Representation

A selection-entry syntax-directed editor such as the one used in IMEGS internally represents its program by its corresponding parse tree. The IMEGS program parse tree consists of a series of nodes, with the root as {program}, the start symbol of all IMEGS languages. Expansion of a nonterminal (by the specification of a template) in the program requires the creation of a child for each section defined for the new template. Specification of a phrase in the place of a text-oriented nonterminal will require the creation of a node for the phrase as the single child of that nonterminal.

There are four types of attributes associated with the parse tree nodes representation of the program semantics. Every node has some associated prettyprinting rules. There are four attribute values associated with each node representing the starting line, starting column, ending line, and ending column of the portion of the sentential form text derived from this node.

Every nonterminal node in the parse tree is associated with attributes necessary for the maintenance of the program symbol table. The symbol table has the form of a tree corresponding to the parse tree. The upper part of the symbol table tree contains attribute nodes that are used to traverse the tree, while the lower parts of the symbol table tree contain entries which are referenced by the intermediate code during program execution. There are two types of entries: declared name entries, for names actually defined by the programmer within the program, and internal name entries, which are used for parsing and compiling of expressions in the prototype for storing constants and the results of operations performed during execution.

The intermediate code program generated during program creation is also in the form of an attribute tree. Each intermediate code instruction is in the form of a quadruple. The first argument of the instruction is a numeric value which corresponds to a member of the instruction set. The remaining arguments, which represent two input arguments and a result argument (not all instructions use each argument), each consist of two parts: a symbol table reference and an intermediate code reference. Most arguments only one of the two references. An exception is for expressions, which use the intermediate code reference to execute the code associated with the expression and the symbol table reference to store the result.

These are various types of static semantic errors that need to be checked during incremental compilation. Each text-oriented nonterminal and each production template in the language may have static semantic rules associated with it. For text-oriented nonterminals, the prototype editor has data type checks; for instance, the nonterminal {boolean_expression} would have a rule requiring the expression just parsed to be of type boolean. For productions, the static semantic rules can be more complex. For example, a static semantic rule could define the coercion rules for an assignment statement (remember, the coercion rules discussed in the previous section were for phrases only).

4. Conclusions

At the beginning of this paper, it was stated that one of the primary goals of the IMEGS research project was to have one selection-entry syntax-directed editor be able to manipulate programs in a variety of block-structured languages. The IMEGS prototype editor, using the language definition scheme described in Section 3.1, has been used to construct programs in various different languages. The language definition required that the languages be defined both syntactically and semantically. Language syntax was achieved by using a modified BNF syntax for the template rules and operator precedence and lexical definitions for the phrases. Semantics were achieved by augmenting the parse tree with various attributes. It is hoped that the IMEGS project will continue to contribute to making syntax-directed environments more language-independent.

References

Allison, L. 1983. Syntax–directed program editing. *Software – Practice and Experience*, 13, 5, 453–465.

Bagert, D.J. 1986. *IMEGS: An Incremental Multi–language Editor Generator and System.* Dissertation to be published, Texas A&M University, College Station, TX.

Horowitz, E. 1984. *Fundamentals of Programming Languages, Second Edition*, Springer–Verlag, Berlin–Heidelberg, Germany.

Knuth, D.E. 1968. Semantics of context–free languages. *Mathematical Systems Theory*, 2, 127–145.

Medina–Mora, R. and Notkin, D.S. 1981. *ALOE Users' and Implementors' Guide.* Department of Computer Science, Carnegie–Mellon University, Pittsburgh, PA.

Reps, T. and Teitelbaum, T. 1984. The Synthesizer Generator. *SIGPLAN Notices*, 19, 5, 42–48.

Teitelbaum, T.; Reps, T.; and Horwitz, S. 1981. The why and the wherefore of the Cornell Program Synthesizer. *SIGPLAN Notices*, 16, 6, 28–33.

Teitelbaum, T. and Reps, T. 1981. The Cornell Program Synthesizer: A syntax directed programming environment. *Communications of the ACM*, 24, 9, 563–573.