Generation of Run-Time Environments

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027

Abstract

Attribute grammars have been used for many years for automated compiler construction. Attribute grammars support the description of semantic analysis, code generation and some code optimization in a formal declarative style. Other tools support the automation of lexical analysis and parsing. However, there is one large part of compiler construction that is missing from our toolkit: run-time environments. This paper introduces an extension of attribute grammars that supports the generation of run-time environments. The extension also supports the generation of interpreters, symbolic debugging tools, and other execution-time facilities.

1. Introduction

Integrated programming environments are rapidly replacing the traditional tools used by programmers to edit, compile and debug their programs. The key components of an integrated programming environment are a standard user interface and a common database. A large number of prototype and teaching programming environments are now built using structure editing technology, which supports both of these features (for example, Tektronix' Magpie [3], the University of Wisconsin at Madison's Editor Allen Poe [8], Brown University's Pecan [21] and Carnegie-Mellon University's Gandalf [20]). Each of these environments consists of an integrated collection of tools that may be applied incrementally as the programmer writes and tests her programs. In some cases, the tools are automatically applied without the explicit intervention by the programmer. For example, type checking and symbol resolution are performed automatically as the program is created and modified; code generation and some code optimization may also be performed incrementally.

The early structure editor-based programming environments, such as the Cornell Program Synthesizer [24], were entirely hand-coded. Then Medina-Mora [19] demonstrated that a structure editing environment can be generated from an *environment description*. A program called an *environment generator* combines an environment description with the common editor kernel to produce the desired programming environment. The person who writes the environment description is called the *implementor* of the programming environment while a person who uses the programming environment to write her programs is called a *user*.

An environment description has two components, the syntax description and the semantics description. The *syntax description* includes the abstract syntax (or structure) of the programming language and the concrete syntax (or user interface) for programs in the language. It is now well-established that a syntax description is written in a declarative notation similar in style to a context-free grammar or BNF. It is very easy to write a syntax description for a conventional programming language. It might take two days for an implementor to write the syntax description for Pascal and as much as two weeks for the more complex syntax of Ada. A syntax description alone can be used as an environment description if no semantics processing is required. An environment generator can combine the syntax description with the editor kernel to produce a pure syntax-directed editor that supports program editing and enforces correct syntax.

The *semantics description* specifies all the processing performed by the environment, *i.e.*, everything the environment does that is not among the standard facilities provided by the editor kernel. In other words, the syntax description describes the program database and the semantics description describes the tools that operate on the database. The semantics processing of a programming environment is performed by a collection of tools that are knowledgeable about the particular programming language. The tools are often divided into two categories: the tools that handle static (compile-time) semantics and the tools that handle dynamic (run-time) semantics. The implementor of a programming environment describes the semantics processing in terms of the static and dynamic properties of the programming language. *Static properties* can be determined by inspection of the program while *dynamic properties* reflect the interaction between the user and the programming environment. Compilation is described in terms of static properties, while the run-time support is characterized by dynamic properties.

1.1. The Problem of Semantics Description

The description of static and dynamic properties is very difficult. In contrast to the syntax description, there is no commonly accepted form for the semantics description of a programming environment, and the development of a semantics description for a relatively simple environment such as the Gnome teaching environment [10] can take many months. However, there are two methods of semantics description, action routines and attribute grammars, that have been more widely used than their competitors in the generation of integrated programming environments.

Action routines were proposed by Medina-Mora in [6]. The semantics processing is written as a set of routines in a conventional programming language. A particular routine is associated with each rule in the abstract syntax. The corresponding routine is automatically invoked by the editor kernel when an editing command is applied to any object that is instance of the rule. Action routines were adapted from the semantic routines associated with parser generators such as YACC [15].

Attribute grammars, originally proposed by Knuth [18], have become a standard technique for compiler generation [5, 9]. Attribute grammars were adapted to interactive programming environments by Reps [4, 23]. The semantics processing is written as a set of attribute definitions associated with each rule in the abstract syntax. An incremental evaluator automatically re-evaluates all attributes whose values may have changed as the result of a modification of the syntax tree.

Action routines, attribute grammar and other mechanisms have proved extremely useful for speeding up the development of programming environments. However, none of these methods fulfill all the requirements of an implementor of a programming environment. The basic problem with action routines is that the design, implementation and debugging of the routines is tedious and error-prone compared to the ease with which a syntax description can be developed. The implementor has to consider the possible orders in which the action routines might be invoked and how this effects the integrity of the data structures that represent the auxiliary information maintained by the action routines. In contrast, dependencies among attribute definitions are handled automatically by the attribute evaluator.

However, attribute grammars are useful for describing only for a certain subclass of program development tools. They handle static properties quite well, but are unsuited to the description of dynamic properties because of the *derived* nature of the attributes. The value of each attribute is calculated from the source program and other attributes. By definition, it cannot depend in any way on the history of modifications to the source or of the execution of the source. This feature is exactly what is desired for static semantic checking and for code generation, but it is inappropriate for the dynamic properties of run-time environments where the semantic state depends on the history of the program execution.

1.2. A Solution

This problem has been solved by *action equations,* an extension of attribute grammars that supports the expression of *history* or dynamic properties. This is done by embedding *equations* similar to attribute definitions in an event-driven architecture. The *events* activate equations in the same sense that user commands trigger action routines. The editor kernel orders the evaluation of active equations according to the commands invoked by the user and the dependencies given by the equations. Equations that apply at all times are not attached to particular events and these correspond exactly to attribute grammar definitions.

This event-driven nature is one of the crucial differences between action equations and attribute grammars. Another way of stating this difference is that action equations support multiple events while attribute grammars support only one event, the ***change*** event. In the attribute grammar paradigm, the ***change*** event can be received from the user and propagated by attribute definitions to other definitions. In the action equations paradigm, the ***change*** event, standard events and implementor-defined events can all be received from the user and/or sent from action equations to other equations. (The *standard events* include create, delete, access, exit and enter.) This makes it easy to invoke particular operations in response to particular user commands, as described in Section 5.

An important implication of multiple events is that an incremental evaluator for action equations can easily handle circular dependencies among the equations while incremental evaluators for attribute grammar require non-circularity. This is because an equation activated by an event is evaluated exactly once in response to the event rather than (conceptually) re-evaluated repeatedly until quiescence. This distinction is built into the incremental evaluation scheme for action equations. The result is an easy mechanism for describing and implementing iteration and recursion that cannot be matched by attribute grammars. This is explained in Section 3.

Another crucial distinction between the two paradigms is that attribute grammars are applicative while action equations support certain non-applicative mechanisms. Attribute grammars require that attribute definitions be purely functional. This means that an attribute definition is re-evaluated (until quiescence) only when the program changes, and then the attribute definition is restricted to replacing the old value with an entirely new value. These applicative restrictions are removed in the new paradigm. An action equation may be re-evaluated due to the receipt of an event, and then the equation is permitted to directly modify the current value of the attribute. This permits a more efficient implementation of the run-time stack of activation records and other run-time support structures. This is discussed in Section 4.

1.3. Compile-Time Semantics

Since attribute grammars are a strict subset of action equations, compile-time semantics are handled in exactly the same manner in the two paradigms. Symbol resolution, type checking, flow analysis for anomaly detection and source-level optimizations, and code generation are described using standard attribute grammar mechanisms.

Compile-time semantics are not discussed further in this paper. The rest of this paper is concerned with run-time semantics. Section 2 gives an overview of the differences between attribute grammars and action equations. Section 3 discusses the description of interpreters using action equations. Section 4 describes the generation of run-time support for program execution and Section 5 explains additional support for language-oriented debugging facilities. Section 6 briefly discusses the implementation of action equations and Section 7 summarizes the significant contributions of this research.

2. Introduction to Action Equations

The action equations paradigm was developed as an extension to attribute grammars and follows the same basic form. An attribute grammar associates a set of attributes with each rule in a context-free grammar. Each attribute is given a function that uniquely determines its value. The function may take as arguments the other attributes associated with the same rule. In the case where the rule defines a non-terminal, the function may also take as arguments the values of the children; in the case of a terminal, the function may also take as an argument the value of the terminal (for example, an identifier, a string or an integer). An attribute grammar follows the format illustrated in Figure 2-1. Each association between an attribute and its defining function is called an *equation*.

Action equations extend this attribute grammar notation to treat some of these equations as active while others are passive. If f_i in Figure 2-1 is *active*, then the ith attribute (a_i) necessarily has the value $f_i(arg_1, ..., arg_n)$. In contrast, if f_j is *passive*, then the value of the jth attribute (a_j) does not necessarily have the value $f_j(arg_1, ..., arg_n)$. It retains its most recent value rather than being updated to maintain this constraint. Active equations correspond to attribute definitions in

$\texttt{rule}_i$

`{ Attribute definitions for` $\texttt{rule}_i$`. }`

    $a_k := f_k(arg_1, \ldots, arg_n)$
    ...
    $a_l := f_l(arg_1, \ldots, arg_n)$

...

$\texttt{rule}_j$

`{ Attribute definitions for` $\texttt{rule}_j$`. }`

    $a_o := f_o(arg_1, \ldots, arg_n)$
    ...
    $a_p := f_p(arg_1, \ldots, arg_n)$

Figure 2-1: An Attribute Grammar

attribute grammars, while passive equations are an innovation of action equations.

The implementor may specify that some equations are permanently active, as in attribute grammars. Other equations may be changed from passive to active status. After a passive equation has been activated, it returns to passivity immediately after the corresponding attribute has been updated. Passive equations are indicated by attaching each passive equation to an event. Our *event* corresponds to a message in an object-oriented language (such as Smalltalk [14]) and the equations attached to the event correspond to a method. (Another paper [13] describes action equations as an object-oriented description language.) The collection of equations associated with a particular rule are normally listed as in Figure 2-2. The equations not attached to any event are permanently active while the equations attached to a particular event are passive until the event is received by an object defined by the rule.

As with messages in object-oriented systems, events are normally generated by objects (the exception is events that represent user commands). Therefore, it must be possible to send an event as well as receive an event. This is supported by further extending attribute grammar-style equations to propagate events as well as apply functions. The *propagate equation* is illustrated in Figure 2-3. More than one destination attribute may be given. The propagation takes place if and only if the equation becomes active.

`rule`

`{ Permanently active equations. }`

    $a_i := f_i(arg_1, \ldots, arg_n)$
    ...
    $a_j := f_j(arg_1, \ldots, arg_n)$

`{ Passive equations for` $\texttt{event}_1$`. }`

$\texttt{event}_1$ `-->`
    $a_k := f_k(arg_1, \ldots, arg_n)$
    ...
    $a_l := f_l(arg_1, \ldots, arg_n)$

...

`{ Passive equations for` $\texttt{event}_m$`. }`

$\texttt{event}_m$ `-->`
    $a_o := f_o(arg_1, \ldots, arg_n)$
    ...
    $a_p := f_p(arg_1, \ldots, arg_n)$

Figure 2-2: Attaching Equations to Events

```
{ A propagate equation is always
  attached to some event with
  respect to some rule. }
```

propagate *<event>* to *<object>*

Figure 2-3: The Propagate Equation

3. Implementation of Interpreters

Action equations support the implementation of conventional control constructs using events and propagate equations. Sequencing, selection, iteration and branching can be expressed with simple combinations of these two facilities. Consider the description of iteration shown in Figure 3-1. There is a circularity among the sources and destinations of the propagate equations that describe the flow of control through the loop. This circularity would be prohibited in an attribute grammar but is no problem for action equations.

The interpretation works as follows. The loop statement receives the run event. The user of the programming environment sent the run event by doing two things. First, he moved the program cursor to highlight the part of the program he wanted to execute; this might be the entire program, any statement enclosing the loop statment, or just the loop statement. Then he selected the run command. In response, the command interpreter sent the run event to the object highlighted by the cursor.

When the loop statement receives the run event, the "<u>propagates</u> run <u>to</u> init" equation is activated. This propagate equation sends the run event to the init component of the loop statement. When the run event is received by the initialization statement, the equations (not shown) that execute this statement are activated. After completion of these equations, then the "<u>propagates</u> run <u>to</u> cond" equation is activated. This sends the run event to the cond component of the loop statement.

When the run event is received by the condition expression, it activates the equations (not shown) that evaluate this expression and set the result attribute to the value of the expression. Then the "<u>propagates</u> run <u>to</u> if cond.result then body" equation is activated. If the result attribute of the cond component is true, then the run event is

```
{ Abstract syntax for loop
  statement.  The loop non-
  terminal has four children;
  three are statements and
  one is an expression. }

loop =>
    init: STATEMENT
    cond: EXPRESSION
    reinit: STATEMENT
    body: STATEMENT

{ Passive equations
  for loop statement. }

run -->
    propagates run to init

{ The "run on init, reinit"
   notation is simply short-hand
   for writing each of the two
   events separately with the
   equations attached to each. }

run on init, reinit -->
    propagates run to cond

{ When the run event is received
  by the cond component, its result
  attribute is set by a constraint
  such as
  "result := <semantic function>"
  that is associated with the particular
  expression. If the new value of
  result is false, then the
  propagation does not occur. }

run on cond -->
    propagates run to
        if cond.result then body

run on body -->
    propagates run to reinit
```

Figure 3-1: Loop Statement Syntax and Semantics

sent to the body component of the loop statement.

The run event continues propagating around the loop, through the body, the re-initialization and the condition until evaluation of the condition expression returns false. When this happens, the run event of the "propagates run to if cond.result then body" equation is not sent at all. Control returns to the source of the original **run** event that started the loop execution.

4. Run-Time Support for Program Execution

An interpreter or a run-time environment for compiled code must implement the memory management required for subroutine invocation and data manipulation. This is supported by (1) extending attribute grammars further to permit any objects, not just attributes, to take part in equations and (2) augmenting the resulting action equations with the view definition notation described by Garlan [12]. The first change permits action equations to modify the source program as well as the values of attributes, introducing a potentially dangerous source of side-effects. However, this ability is used not to modify the source program constructed by the user but to modify alternative views of the source program that represent the internal state of the run-time environment.

A *static view* consists of a collection of rules that define an abstract syntax. Figure 4-1 gives two static views for a procedure: Source and Execution. The Source view describes the standard abstract syntax for a procedure, while the Execution view defines an activation record for a procedure. During program execution, an activation record is pushed onto the run-time stack to represent a procedure invocation.

Notice that both views define a name component as an instance of the identifier definition (identdef) terminal. Consequently, this component has exactly the same value in both views. Because of the 'ReadOnly' qualification given in the Execution view, the name component can be created (and deleted) with respect to the Source view but not with respect to the Execution view. This stands to reason, since it is (generally) not appropriate to change the name of a procedure during execution.

Both views also define parameters and locals components, but in each case with different types. In the Source view, both parameters and locals are vardefs. A vardef (not shown) consists of a name and a type. In the Execution view, both parameters and locals are blocks: terminals that represent a block of memory. The name and type of a

```
view Source

procedure =>
    name: identdef
    parameters: seq of vardef
    locals: seq of vardef
    body: seq of STATEMENT

view Execution

{ The default qualification
  is 'ReadWrite'. }

procedure =>
    name: identdef
        qualified 'ReadOnly'
    parameters: seq of block
    locals: seq of block

{ "ref to" indicates a
   symbolic reference to an
   object elsewhere in the
   same or another syntax tree.
   The reference terminal can
   be used to extend a rigid
   syntax tree to an
   arbitrary graph structure. }

    pc: ref to STATEMENT
    body: seq of STATEMENT
        qualified 'ReadOnly'
```

Figure 4-1: Two Static Views of Procedure Syntax

parameter or local variable are not accessible in the Execution view and the block of memory is not accessible in the Source view. The block of memory is used for storing the value of the actual parameter or the local variable during program execution.

Figure 4-2 gives the Execution view for the enclosing program: a stack of activation records, an expression stack and a heap for dynamic storage allocation. The entire internal state of an interpreter or a run-time environment can be described by a static view in this manner.

```
view Execution

program =>
    stack: seq of procedure
    estack: seq of block
    heap: block
```

Figure 4-2: Execution View of Program

So far, only pieces of abstract syntax have been associated with a particular static view. However, each action equation must also be explicitly associated with a particular view. Figure 4-3 shows part of both the syntax and semantics of a procedure call statement. Actual parameter evaluation and parameter passing are not illustrated. In this example, the run event and the action equations attached to the run event are associated with the Execution view.

```
view Execution

{ An "identuse" is
  an identifier use. }

call =>
    name: identuse
    actuals: seq of EXPRESSION

{ Defsite points to
  the definition of
  the called procedure. }

    defsite: ref to procedure
             qualified 'ReadOnly'

run -->
    ^program->stack :=

{ The "@" operator
  dereferences a reference
  and the "#" operator
  indicates concatenation. }

        copy @defsite #
             ^program->stack

    propagate run to @defsite
```

Figure 4-3: Execution View of Call Statement

The equation attached to the run event in Figure 4-3 has the effect of pushing a copy of the called procedure. Since this is the Execution view rather than the Source view, it is an activation record rather than the source for the procedure that is actually pushed. The propagate equation initiates execution of the invoked procedure.

Notice how simple it is to describe procedure invocation. This is why Garlan's view mechanism was adopted for the action equations paradigm. It would be more difficult to express the same semantics processing using attributes rather than views. It would be relatively easy to declare the stack and heap as attributes rather than components, but the manipulation of activation records would be more complex. Without views, the implementor would have to explicitly construct the appropriate activation record on each procedure invocation and explicitly maintain references to the shared information.

5. Interactive Execution and Debugging

The only requirements of interactive execution and debugging that cannot be satisfied by the facilities previously described are those that involve direct interaction with the user of the programming environment. These requirements are met using the delay equation and Garlan's display views [11].

A *display view* describes how a static view is displayed on a terminal or workstation screen. It also supports editing commands such as create an object and exit the current object with the cursor. These editing commands are always applied to the underlying attributed syntax tree in terms of the displayed representation. Display views are sufficient to support both display and modification of the internal state of the programming environment. During debugging, the user can display the Source and Eexecution views of her program in different windows.

The same mechanism can be used for stream input/output, where the stream is described in the IO view of the program. Implementation of the write statement is extremely easy. The value to be written is concatenated to the end of the I/O stream, which is automatically displayed on the screen. The read statement is slightly more difficult and requires the addition of the *delay equation.* The function of the delay equation is to request a particular event and delay the other active equations until the user sends the event. The user does this by selecting the command that corresponds to the event. The delay equation implements the read statement by suspending program execution until the user has entered another line of input.

The delay equation has the form illustrated in Figure 5-1, where *<object>* is an address expression and *<event>* is any standard or implementor-defined event that can be selected by the user as a command. The evaluation of a delay equation means that the currently active equations cannot be evaluated until *<event>* has been received by *<object>*. The active equations are suspended when the delay equation is evaluated and awakened when the event occurs at the object. This happens when the user performs some activity that has the effect of sending the standard or implementor-defined event to the appropriate object.

```
{ A delay equation is always
  attached to some event with
  respect to some rule. }

delay until <event> at <object>
```

Figure 5-1: The Delay Equation

The delay equation is the means for implementing debugging facilities such as breakpoints and singlestepping. Program execution would be suspended at a marked statement or after every statement, respectively, until the user invokes the **continue** command. Figure 5-2 illustrates the description of breakpoints. The user of the programming environment would insert a break statement into the body of a procedure (in the execution view) to indicate a breakpoint.

```
view Execution

{ The break statement
  has no components. }

break => nil

run -->
    delay until continue
```

Figure 5-2: Break Statement Syntax and Semantics

6. Implementation

The implementation of the action equations paradigm is explained in [17]. The basic idea is that action equations are translated to (1) dependency graphs and (2) procedures written in a special tree-oriented programming language. The dependency graphs are used by the structure editor kernel to determine the order in which to invoke the procedures corresponding to active equations. Each procedure implements a single action equation. The algorithm is an extension of Reps' incremental attribute evaluation algorithm for attribute grammars [22].

Two versions of the tree-oriented programming language and corresponding kernels have been implemented, one in Pascal for the DOSE generic structure editor (on the Perq workstation) [16, 7] and one in C for the ALOE structure editor generator (on Unix™) [1, 19]. The translator has not yet been completed. Only toy environments have as yet been described using action equations, but several real environments have been implemented using the tree-oriented programming language directly.

7. Conclusions

The primary result of the research described in this paper is the development of a new paradigm within which implementors can easily develop the run-time components of structure editor-based programming environments. The action equations paradigm represents a significant improvement over previously proposed paradigms. The major improvement over action routines is the declarative style of notation. This raises the level of abstraction at which implementors can describe semantics processing and drastically eases the debugging and enhancement of their programming environments. The major improvement over attribute grammars is the simple means for expressing dynamic as well as static properties of the programming environment. This allows the implementor to specify both code generation and the run-time support for program execution.

Action equations can also be used for generation of compilers and run-time environment as separate tools, apart from programming environments, in the same manner that attribute grammars are currently used for compiler generation. The implementation in this case is much easier, since the action equations are evaluated once, until quiescence, with respect to the whole program rather than incrementally as the program is modified.

Acknowledgements

The research discussed in this paper was performed in partial fulfillment of the requirements for the degree of Doctor of Philosophy at Carnegie-Mellon University. I would like to thank my advisor, Nico Habermann, for his many years of technical and personal support. I would like to thank David Garlan for developing the view mechanism that provides the structural support for action equations and for countless hours of fruitful discussion. I would also like to thank the Hertz Foundation for supporting me for five of my years in graduate school. This research was supported in part by the Software Engineering Division of CENTACS/CORADCOM, Fort Monmouth, NJ.

References

[1] Vincenzo Ambriola, Gail E. Kaiser and Robert J. Ellison.
An Action Routine Model for ALOE.
Technical Report CMU-CS-84-156, Carnegie-Mellon University, Department of Computer Science, August, 1984.

[2] David R. Barstow, Howard E. Shrobe and Erik Sandewall.
Interactive Programming Environments.
McGraw-Hill Book Co., New York, NY, 1984.

[3] Norman M. Delisle, David E. Menicosy and Mayer D. Schwartz.
Viewing a Programming Environment as a Single Tool.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* April, 1984.

[4] Alan Demers, Thomas Reps and Tim Teitelbaum.
Incremental Evaluation for Attribute Grammars with Applications to Syntax-directed Editors.
In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages (POPL).* January, 1981.

[5] Rodney Farrow.
Generating a Production Compiler from an Attribute Grammar.
IEEE Software 1(4), October, 1984.

[6] Peter H. Feiler and Raul Medina-Mora.
An Incremental Programming Environment.
IEEE Transactions on Software Engineering SE-7(5), September, 1981.

[7] Peter H. Feiler and Gail E. Kaiser.
Display-Oriented Structure Manipulation in a Multi-Purpose System.
In *Proceedings of the IEEE Computer Society's Seventh International Computer Software and Applications Conference (COMPSAC '83)*, pages 40-48. November, 1983.

[8] Charles N. Fischer, Gregory F. Johnson, Jon Mauney, Anil Pal, Daniel L. Stock.
The POE Language-Based Editor Project.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* April, 1984.

[9] Harald Ganzinger, Knut Ripken and Reinhard Wilhelm.
Automatic Generation of Optimizing Multipass Compilers.
In *Information Processing 77*, pages 535-540. North-Holland Publishing Co., New York, NY, 1977.

[10] David B. Garlan and Philip L. Miller.
GNOME: An Introductory Programming Environment Based on a Family of Structure Editors.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* April, 1984.

[11] David B. Garlan.
Flexible Unparsing in a Structure Editing Environment.
Technical Report CMU-CS-85-129, Carnegie-Mellon University, Department of Computer Science, April, 1985.

[12] David B. Garlan.
Representational Transformations for Tools in Structure Editing Environments.
PhD thesis, Carnegie-Mellon University, 198x.
In progress.

[13] David Garlan and Gail E. Kaiser.
MELD: An Object-Oriented Language for Describing Features.
Submitted to the ACM Conference on Object Oriented Programming Systems, Languages, and Applications, 1986.

[14] Adele Goldberg and David Robson.
Smalltalk-80 The Language and its Implementation.
Addison-Wesley Publishing Co., Reading, MA, 1983.

[15] S.C. Johnson and M.E. Lesk.
Language Development Tools.
The Bell System Technical Journal 57(6), July-August, 1978.

[16] Gail E. Kaiser.
Tree Manipulation Language User's Manual.
Technical Report RTL-84-TM-106, Siemens Research and Technology Laboratories, February, 1984.

[17] Gail E. Kaiser.
Semantics of Structure Editing Environments.
PhD thesis, Carnegie-Mellon University, May, 1985.
Technical Report CMU-CS-85-131.

[18] Donald E. Knuth.
Semantics of Context-Free Languages.
Mathematical Systems Theory 2(2), June, 1968.

[19] Raul Medina-Mora.
Syntax-Directed Editing: Towards Integrated Programming Environments.
PhD thesis, Carnegie-Mellon University, March, 1982.

[20] David Notkin.
The GANDALF Project.
The Journal of Systems and Software 5(2), May, 1985.

[21] Steven P. Reiss.
Graphical Program Development with PECAN Program Development Systems.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* April, 1984.

[22] Thomas Reps, Tim Teitelbaum and Alan Demers.
Incremental Context-Dependent Analysis for Language-Based Editors.
ACM Transactions on Programming Languages and Systems (TOPLAS) 5(3), July, 1983.

[23] Thomas Reps and Tim Teitelbaum.
The Synthesizer Generator.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments.* April, 1984.

[24] Tim Teitelbaum and Thomas Reps.
The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.
Communications of the ACM 24(9), September, 1981.
Appears in [2].