Contrasting Approaches of Two Environment Generators: The Synthesizer Generator and Pan

Barbara Staudt Lerner
Computer Science Department
University of Massachusetts
Amherst, MA 01003
blerner@cs.umass.edu

April 20, 1993

Abstract

The Synthesizer Generator and Pan are two popular environment generators. While having the same basic goal of assisting in the development of interactive environments, they use rather different mechanisms to reach this goal. This paper discusses the three basic descriptive components used in environment generation: syntactic, semantic, and user interface descriptions. The contrasting approaches of the Synthesizer Generator and Pan are presented, along with a discussion of the consequences resulting from their respective mechanisms.

1 Introduction

Software development environments are tools to support the development of software. Ideally, these environments provide support for the entire software lifecycle, coordinating the efforts of many people, and supporting a particular software engineering process. Such an all-encompassing environment is a complex piece of software itself. Thus, researchers have created environment generators to assist in the development of software development environments.

Environment generators are one category of tool in the more general class of application generators, which includes such tools as compiler-compilers, expert system generators, and spreadsheet generators. An environment generator is a tool that takes a description of an environment as input, processes the description and combines it with a large collection of reusable code to produce an environment as output. Depending on the environment generator, an environment description may include a description of the syntax and semantics of the language(s) to be manipulated by the environment, a description of the software process to be supported by the environment, and a description of the user interface. The reusable code typically provides such functionality as file manipulation, display manipulation, and command interpretation that is shared by all generated environments.

The power of environment generators stems from extensive reuse of a common environment kernel and the use of declarative descriptions to specify an environment. By combining these two techniques, an environment implementor can specify a new environment in a concise way, yet develop a powerful environment. Of course, the flip side of the coin is that the generated environments

may contain features that the environment implementor does not want. As with any application generator, the environment generator makes certain decisions over which the environment implementor has no control. It is therefore important for environment generators to provide an appropriate balance between reuse of common facilities and flexibility in specification [Kru92].

The earliest form of environment generators were editor generators, which produced syntax-directed editors. These early environments supported programming-in-the-small, that is, a single programmer working on a small program. As such, they were most commonly used in educational settings and for rapidly prototyping editors to support development of new programming languages. Today's environment generators can produce more sophisticated environments, environments with a much less obtrusive user interface, environments intended for programming-in-the-large, multilingual environments, environments that coordinate the efforts of multiple programmers, and environments to manipulate structured documents, not just source code.

While environment generators have been used to create numerous environments, it would be premature to claim that current environment generators can easily produce any environment imaginable. In this article, we compare and contrast the capabilities of two environment generators so that the software practitioner can determine if they would be useful for his/her work. We have chosen to present the Synthesizer Generator [RT89] and Pan [BGV92] since they are well known and present contrasting mechanisms to support environment generation, leading to an interesting comparison. This paper is necessarily an incomplete survey due to its length and omits important contributions made by numerous other projects including Gandalf [HGN91], Mentor [DGHKL84], Centaur [BCD+88], PSG [BS92], IPSEN [ELN+92], Pecan [Rei85], Mjolner [MHM+90], Yggdrasil [Cap85b], GIPE [HKKL86], and ASDL [KS89]. We also limit the discussion to environment generation and do not attempt to survey the much more general area of software development environments.

2 Syntactic Descriptions

A syntactic description defines the internal representation of documents as well as the external presentation of documents. The internal representation is typically an abstract syntax tree, while the external presentation is typically text. To be useful, an environment generator also needs mappings between the internal representation and the external presentation. The mapping from the external representation to the internal representation is used for parsing. The mapping from the internal representation to the external representation is used for display purposes, and is called *unparsing*. With the Synthesizer Generator, the environment implementor specifies an internal representation, an external presentation, and mappings between the internal representation and the external presentation explicitly. With Pan, the environment implementor defines an internal representation and a textual presentation using grammars, while Pan derives the mappings between these representations. In this section, we examine these two methods for providing syntactic descriptions.

2.1 Synthesizer Generator

With the Synthesizer Generator, an implementor describes a language with two distinct grammars, an abstract grammar and a concrete grammar. The abstract grammar defines the abstract syntax tree representation and the unparsing rules, while the concrete grammar defines a BNF-like syntax and tree construction rules for parsing. Both the abstract and concrete grammars are defined

as collections of phylum definitions. Each phylum definition groups together the productions that produce that phylum. A phylum can be thought of as a type, with the productions being functions returning that type. While there are often strong similarities between the abstract and concrete phyla, they are kept distinct to provide maximum flexibility in defining the two grammars.

2.1.1 Abstract Grammar

An abstract syntax phylum definition has three parts: abstract syntax productions, unparsing declarations, and attribute declarations. (Attribute declarations will be discussed in Section 3.1.)

Abstract Syntax An abstract syntax *statement* phylum might be defined as

```
statement : EmptyStat ()
          | While (exp statement)
          | Return (return-exp)
```

Each production has the form *x0 : op (x1 x2 ... xk)* where *op* is an operator and each *x* is the name of an abstract syntax phylum. The right hand side defines the subtrees that can be derived from the lefthand phylum. The operator labels the root of the subtree, while the operator arguments identify the phylum of each child in the subtree.

Unparsing The mapping from the internal representation to text is defined using *views*. Up to two unparsing schemes can be provided for each production in each view. One unparsing scheme is the primary unparsing scheme. The second scheme is the alternate unparsing scheme and is typically used for elision. The user can select which view to display an entire document in and which unparsing scheme of that view to use for each subtree in the document.

The Synthesizer Generator provides two types of views: *dense views* and *sparse views*. In a dense view, the nodes of the abstract syntax tree that will be unparsed can be determined by knowing only the shape of the tree and the operators at each node of the tree.

In a sparse view, the nodes that are unparsed depend on the values of attributes of those nodes, resulting in a more dynamic mapping. For example, an error view can be defined as a sparse view that unparses only those parts of the tree that have errors. In particular, the unparser first identifies all nodes with a non-null error attribute. It then finds their least common ancestor and begins unparsing at that point. The effect is that only the portion of the document that contains errors at any given time is displayed. The contents of the error view changes as errors are introduced and removed from a document even if the shape of the tree or the operators in the tree do not change.

Unparsing schemes do not refer to the children of a node by name. Instead, they contain a marker where each child should appear. Thus, the children are implicitly unparsed from left to right. For example, the unparsing scheme associated with a while statement could be defined as

```
statement : While ["while" @ "do" %t%n @ %b]
```

The @ symbols indicate that the next child should be unparsed. *%t* increases indentation. *%n* inserts a newline. *%b* decreases indentation. It is also possible to unparse attributes by inserting the attribute name into the unparsing scheme.

```
Statement   ::=   (WHILE Exp DO Statement)   {Statement.ast = While(Exp.ast, Statement.ast) }
                | (RETURN Return-exp)        {Statement.ast = Return (Return-exp.ast)}
```

Figure 1: Concrete Syntax in the Synthesizer Generator

Since there is no syntactic distinction between children in unparsing schemes, it is not possible to vary the order in which children are unparsed, to unparse the same child more than once, or to skip a child if any subsequent children should be unparsed. However, since attributes are referenced by name, they can appear in any order or be repeated.

2.1.2 Concrete Grammar

The concrete grammar is a BNF-like grammar used to develop a parser. The parser is generated by YACC, thereby requiring that the concrete grammar be LALR(1). The parser builds a parse tree using this concrete grammar.

The implementor defines the translation from a parse tree to the corresponding abstract syntax tree in the following manner. In each concrete syntax phylum, the implementor declares an attribute whose phylum is an abstract syntax phylum.[1] The implementor then defines an attribute equation to set the value of the attribute in each concrete syntax production of that phylum. The value is usually the result of applying an abstract syntax operator to the abstract syntax trees created while parsing the children of the current concrete syntax production. Therefore, parsing and creation of the abstract syntax tree are both done in a bottom-up manner and occur in parallel.

Figure 1 shows the concrete syntax phylum definition to parse statements. *ast* is the name of the attributes holding the abstract syntax tree corresponding to a particular parse tree. For example, when a *while statement* is parsed, an abstract syntax tree is created whose root is labeled with the abstract syntax operator *While*. The children of the root are the abstract syntax trees produced when parsing the respective children of the *while-statement*.

2.1.3 Relationship between the Abstract Grammar and Concrete Grammar

The example shown above exhibits a 1-1 relationship between the abstract syntax phyla and the concrete syntax phyla. Furthermore, there is also a 1-1 relationship between abstract syntax productions and concrete syntax productions. However, neither of these relationships is necessary. By separating the abstract syntax and concrete syntax, the grammars can be quite different, thereby allowing each to be customized for its intended purpose. Furthermore, the input language, as defined by the concrete grammar, and the display presentation, as defined by the unparsing declarations of the abstract grammar, can be quite different. For example, the input language could be concise to speed entry by experienced users, while leaving the displayed presentation verbose to maintain its readability.

Also, notice that parsing is a two step process. The YACC-generated parser produces a parse tree which is then converted to an abstract syntax tree using the tree-building attribute equations in

[1] Attributes are described more completely in Section 3.1 which discusses the Synthesizer Generator's semantic processing capabilities.

```
Expression      :  Null ()
                |  Sum (Expression, Expression)
                |  Diff (Expression, Expression)
                |  Prod (Expression, Expression)
                |  Quot (Expression, Expression)
                |  Constant (Integer)

Binding         :  Bind (Identifier, Integer)

Environment     :  NullEnv ()
                |  EnvConcat (Binding Environment)

ExpCommand      {  inherited Environment env ;
                   synthesized Binding b;
                   synthesized expression ast; };

ExpCommand      ::=  ('.' Identifier)
                     {  ExpCommand.b = lookup (Identifier, ExpCommand.env);
                        ExpCommand.ast = with (b): (Binding (*, intvalue): Constant (intvalue));}
```

Figure 2: Flexible Binding of Abstract and Concrete Syntax in the Synthesizer Generator

the concrete syntax. This allows the use of attribution during parsing so that the abstract syntax tree can depend on context, not just the input string.

For example, Figure 2 demonstrates this flexibility in an example of a desktop calculator where the user's input can be the name of a constant, while the value inserted in the abstract syntax tree is the constant's value (adapted from [RT91, p. 76]). The concrete syntax for *ExpCommand* is of the form *.Identifier*. When an ExpCommand is parsed, the attribute equations look for a binding of the identifier in the current environment. If one is found, *intvalue* is bound to the integer value in the binding. The abstract syntax tree produced uses the *Constant* operator defined in the *Expression* abstract syntax phylum. The child of the *Constant* operator is the integer value found in the binding. (We have omitted details of how the binding and environment attributes are created, or how a binding is found in an environment.)

2.2 Pan

Pan's internal representation is defined by an abstract syntax and its external presentation is defined by a concrete syntax. Pan determines the relationship between the abstract syntax and concrete syntax automatically through a technique called *grammatical abstraction* and then generates the parsing and unparsing functions.

2.2.1 Grammatical Abstraction

Unlike the abstract syntax of the Synthesizer Generator, Pan's abstract syntax definitions typically contain a great deal of syntactic sugar, more closely resembling the concrete syntax notation of the Synthesizer Generator. In fact, a Pan environment can be generated using only an abstract syntax.

Concrete Syntax

```
<stmt>      →  <if-stmt> ";"
<if-stmt>   →  <if-part> <else-part>
<if-part>   →  if <expr> then <stmts>
<else-part> →  ϵ
            |  else <stmts>
```

Abstract Syntax

```
<stmt>  →  <expr> <stmts>
        |  <expr> <stmts> <stmts>
```

Figure 3: Grammatical Abstraction in Pan

In that case, the internal representation is the parse tree. In practice, an implementor would only define concrete syntax productions for those parts of the grammar for which the parse tree would be an awkward internal representation. For example, the concrete syntax might use a number of productions to define precedence in expressions, which could be represented implicitly in an abstract syntax tree.

Grammatical abstraction requires that the abstract syntax and the concrete syntax define languages that can easily be mapped to each other. Each nonterminal in the abstract syntax must correspond to a nonterminal in the concrete syntax. Furthermore, the right hand side of each abstract nonterminal production must correspond to a string of nonterminals and terminals that can be derived from the corresponding concrete nonterminal. Some ways in which the abstract and concrete syntaxes may differ are the following:

- Terminals in the concrete syntax that represent keywords and tokens can be omitted from the abstract syntax if they can be inferred.
- Nonterminals in the concrete syntax can be omitted from the abstract syntax by substituting the (abstract version of their) definitions for where they are used.

Consider the example shown in Figure 3 taken from [BGV92, p. 106]. In this example, the abstract syntax nonterminal <*stmt*> corresponds to the concrete syntax nonterminal <*stmt*>. The remaining nonterminals and tokens shown in the concrete syntax have been abstracted away in the abstract syntax. The right hand side of the first abstract statement production corresponds to the concrete string *if* <*expr*> *then* <*stmts*> ϵ, which is derivable from the concrete nonterminal <*stmt*>. Similarly, the right hand side of the second abstract nonterminal <*expr*> <*stmts*> <*stmts*> corresponds to the concrete string *if* <*expr*> *then* <*stmts*> *else* <*stmts*>, which is also derivable from the concrete nonterminal <*stmt*>.

Parsing is accomplished using a modification of the Jalili-Gallier incremental parsing algorithm [JG82] which provides LR parsing. The Jalili-Gallier algorithm supports incremental parsing by incrementally modifying the parse tree as the user modifies the text. Pan's modifications support incremental modification of the abstract syntax tree directly, rather than the parse tree.

Unparsing is accomplished by reversing the mapping from the abstract to the concrete grammars. As a consequence, Pan automatically provides a unique unparsing for each document. In addition, the

environment implementor and sophisticated users (called *view style designers* in Pan's terminology) can develop new views using CommonLisp, much as Emacs can be extended by sophisticated users. A view style designer must have at least some knowledge of the abstract syntax and can use the semantic information maintained by Pan, if desired. Since the full generality of CommonLisp is available, new views can be quite sophisticated. They can traverse the abstract syntax tree and examine any semantic information available to produce either textual or graphical views.

2.2.2 Relationship between the Abstract Syntax and Concrete Syntax

Pan provides a great deal of flexibility in binding abstract syntax to concrete syntax, while allowing the common case where the syntaxes are the same to be easily shared. However, since the abstract syntax tree construction algorithms are based upon grammatical abstraction rather than explicitly provided by the implementor as in the Synthesizer Generator, Pan lacks the full generality found in the Synthesizer Generator. On the other hand, since the mappings are produced automatically, Pan guarantees that it can parse any document that it unparses. This guarantee is missing from the Synthesizer Generator.

3 Semantic Descriptions

The semantic description of an environment typically defines the semantic rules of the language. For example, a language's type-checking rules and inheritance mechanism would be described in the semantic section of an environment description. The semantic description may also describe the environment's semantics, such as coding styles, version control mechanisms, and software processes.

3.1 Synthesizer Generator

The Synthesizer Generator's semantic descriptions are written using attribute grammars. An attribute grammar is a context-free grammar enhanced with attribute declarations and attribute equations. An attribute declaration can be attached to either a phylum or a production in either the abstract or concrete grammar. The attribute's type is itself a phylum. An attribute might represent a semantic structure not normally visible to the user, such as a symbol table. Alternatively, an attribute might contain part of an abstract syntax tree, as is done to support the translation of parse trees to abstract syntax trees, which was described in Section 2.1.2. Another use of attributes is to mark nodes that have a particular meaning and define a sparse view to display those nodes, as described for defining error views in Section 2.1.1. Attribute equations are attached to the productions in the grammar and define how to compute the values of the attributes.

Attributes are divided into two categories: *synthesized* and *inherited*. The value of a synthesized attribute is computed by attribute equations defined with the productions of the phylum declaring the attribute. Its value is typically computed by a function applied to the attribute values of the children of the node containing the synthesized attribute. The value of an inherited attribute is computed by attribute equations defined with the productions that use the phylum declaring the attribute. Its value is typically computed by applying a function to the attribute values of the parent of the node containing the inherited attribute. Information is shared among different nodes in the tree by passing the value around from parent to child and vice versa. This form of sharing is one of the frequently

cited criticisms of attribute grammars, as a straightforward implementation of a large attribute, such as a symbol table, will result in a lot of copying of the attribute value when some small portion of the attribute changes, as when a new variable is declared. Various researchers have proposed mechanisms to support more efficient handling of such attribute propagation by allowing non-local propagation of attributes [HT86, Hed91, BC85, DRZ85, Hoo86, JF85].

Consider the parsing example in Figure 1. *ast* is an attribute representing the abstract syntax tree being constructed. The value of the *ast* attribute is computed by applying the *While* operator to the abstract syntax tree attributes associated with the children of the parse tree for the while statement. Since the abstract syntax tree of the *Statement* depends upon the abstract syntax trees of the *Statement*'s children, *ast* would be declared as a synthesized attribute.

A symbol table is an example of an attribute that would be synthesized for some phyla and inherited by others. The symbol table would be synthesized during the processing of the declarations section of the abstract syntax tree. It would then be inherited by the statements section of the corresponding block in the abstract syntax, so that variable references could be semantically verified. For example, the *env* attribute in Figure 2 represents a collection of bindings. The attribute is synthesized as bindings are inserted, which occurs when they are declared (not shown in the figure). The attribute is then inherited by the portion of the tree where expressions are entered, so that it can be used to look up specific bindings.

The collection of attribute equations for a language description must obey the following properties:

- Each production must have exactly one equation defining the attribute value for each synthesized attribute associated with the production or the production's phylum.
- Each production must have exactly one equation defining the attribute value for each attribute inherited by its children.
- There may be no circular dependencies in the attribute equations.

The first two properties ensure that all attributes will be assigned values. The last property ensures that there is a unique value for each attribute in a tree. If this property does not hold, the attribute evaluation algorithm may enter an infinite loop. It is this last property that makes implementation of dynamic semantics infeasible with attribute grammars. Suppose we had a value attribute associated with each node in expression subtrees. If a program had no loops or procedure calls, this would suffice. However, the value of an expression node in a loop depends upon which iteration of the loop is being considered. In other words, there is neither a unique value for the expression nor a bounded number of values, in the general case. Similarly, the value of parameters in a procedure depends upon which call is being considered.

By default, attributes are evaluated eagerly. That is, after each change made by the user, the incremental evaluation algorithm is executed to update the values of all affected attributes. However, the implementor may indicate that some attributes should be evaluated on demand, in which case they are only evaluated when needed for some other purpose, such as to update the display. In addition, the user can turn off all semantic checking if so desired.

3.2 Pan

Pan's semantic descriptions are written using a logic constraint language, Colander [Bal89], which combines logic programming and consistency maintenance. A logic constraint grammar is

a context-free grammar with goals attached to the productions in the grammar. There may also be goals that are independent of the productions. These goals initialize global data. The environment's semantics are defined with rules.

Pan performs semantic analysis using unification as in Prolog. First, the semantic analyzer attempts to satisfy the goals that are independent of the productions. Then, it attempts to satisfy the goals associated with the nodes in the tree, where the goal associated with a node is the same as the goal associated with the production used to create the node. Evaluation stops when either all the goals have been satisfied, or no more goals can be satisfied. In the first case, the document satisfies the specified semantics. In the second it indicates that there is a semantic error in the document. As with attribute grammars, the semantic analyzer will not terminate if it encounters circular dependencies among the goals.

As goals are satisfied, tuples are added to a logic database. This information can be accessed as needed to satisfy other goals. It does not need to be explicitly passed up and down the tree as in an attribute grammar. Colander supports the concepts of collections and contexts in the database. A collection groups together related tuples, such as the tuples corresponding to the declarations within a scope. A context is a group of collections representing the portion of the database that is used to satisfy a subtree's goals, such as the set of collections representing all scopes visible in a subtree.

For example, consider Figure 4, derived from [BGV92, pp. 113-5]. This is a very simple example involving the definition and use of identifiers. When a definition is created, *?Scope* is bound to the context in which the definition occurred. *?Name* is bound to the string representing the identifier. The next predicate tests that there is no declaration of the name in the given scope. If this predicate fails, the corresponding error message is produced. Otherwise, a new entity is created. Two tuples are added to the database concerning the entity. The first says it is an identifier, while the second says it is declared with the identifier's name in the current scope.

The *use* production binds *?Scope* and *?Name* in a similar manner. It binds *?Entity* to the entity declared with that name in the scope. If *?Entity* cannot be bound, an error message is produced. It also checks if the database contains a tuple indicating that *?Entity* is an identifier, and reports an error if it does not.

Pan keeps track of the dependencies between the abstract syntax tree and the tuples in the database. When the tree is modified, Pan automatically removes the tuples from the database that depend on the removed portion of the tree and attempts to satisfy a set of new goals based on the new portion of the tree.

Colander's declarative notation relies on logic programming constructs exclusively. Thus, the entire semantics of an environment is written in the same notation, rather than relying on an imperative language to define semantic functions, as the Synthesizer Generator does. It may therefore be easier to create and maintain Pan semantic descriptions than the Synthesizer Generator's. As with attribute grammars, Colander cannot be used to define dynamic semantics. An open question is whether the logic inferencing can be as fast as attribute evaluation since it is implemented with backtracking search. Thus far, Pan has introduced contexts to improve performance, but has not otherwise concentrated on efficiency issues.

4 User Interface

The environments produced by environment generators share one very important characteristic: they provide highly interactive, language-specific support for software development. Rather than

```
<def>  →  "DEF" id
       :- context ($$, ?Scope),
          string-name ($id, ?Name),
          not (<declared (?Name, ??), ?Scope>):
              "Invalid redeclaration of ?Name in the scope.",
          new-entity (?Entity),
          assert (<type-of ("id"), ?Entity>),
          assert (<declared (?Name, ?Entity), ?Scope >).

<use>  →  "USE" id
       :- context ($$, ?Scope),
          string-name ($id, ?Name),
          <declared (?Name, ?Entity), ?Scope>):
              "No identifier named ?Name is in the scope.";
          <type-of ("id"), ?Entity>:
              "?Name is not declared as an identifier.".
```

Figure 4: Pan Semantic Description Example

the traditional edit-compile-debug cycle, generated environments perform syntactic and semantic analysis during the edit phase typically using a compiler only for code generation.

There are two basic approaches to syntactic analysis. In the first approach, the user is able to freely enter any string of text, which is then parsed to determine its syntactic legality. This is the form of input most commonly associated with text editing. A second means of ensuring syntactic legality is by making it impossible to enter input that is syntactically incorrect. This is the approach taken in programs that have a menu-based user interface such as syntax-directed editors.

Environments also can apply a variety of policies regarding how semantic errors are handled. The user might be prevented from constructing a semantically ill-formed document, the user might be allowed to create a semantic inconsistency but immediately be warned of the inconsistency, semantic inconsistencies might be noticed immediately but not reported immediately, or semantic analysis might be delayed until it is explicitly requested by the user.

Another issue distinguishing syntax-directed editors from text editors is how the user navigates through the document. In a text editor, the user's current focus of attention is identified by a small cursor the size of a single character. The user can generally move this cursor backwards and forwards by characters, words, or lines.

In syntax-directed editing the user's focus of attention is identified by a larger two-dimensional character that highlights an entire programming construct, which corresponds to a subtree in the abstract syntax tree. Most syntax-directed editors support both structural and line-oriented navigation. In structural navigation, the cursor can move up to a parent in the abstract syntax tree, down to a child, or left or right to a sibling. In line-oriented navigation, the cursor can move down a line, for example, but it remains a structural cursor. Rather than highlighting a character on the next line, the cursor will highlight a structure on the next line.

4.1 Synthesizer Generator

Synthesizer Generator environments support both syntax-directed editing and a limited form of incremental parsing. The environment implementor defines the commands in the syntax-directed interface using transformation declarations. A transformation declaration specifies how to transform one subtree into another. They are typically used to define commands that replace placeholders in the document with subtrees corresponding to some syntactic structure, such as an *if-statement*. However, they can also be used to define commands that replace any arbitrary structural pattern with another structure. For instance, a transformation declaration could specify how to transform a procedure declaration into a function declaration, and vice versa, or how to rewrite a boolean expression using DeMorgan's Laws.

Through transformation declarations, the Synthesizer Generator has again taken a very flexible approach to the development of the user interface, since it separates the concerns of the definition of the internal representation from the definition of the syntax-directed interface. However, this approach requires the environment implementor to explicitly define all of the commands that construct subtrees, when most of them could be trivially derived from the abstract syntax.

Incremental parsing is accomplished using the parsing declarations described in Section 2.1.2 and *entry declarations*. An entry declaration creates a binding between an abstract syntax phylum and an attribute in a concrete syntax phylum. When the user attempts to edit a document textually, the environment looks at the phylum at the root of the subtree identified by the user's cursor. It then locates the entry declaration defined for that phylum to determine which concrete syntax phylum's parsing rules to use. After the user has completed the textual modifications and submitted the modified text for parsing, the environment parses it using the parsing rules defined for the concrete syntax phylum in the entry declaration. The parsing rules must build an abstract syntax tree whose root phylum is the abstract syntax phylum of the entry declaration and store the tree in the attribute identified in the entry declaration. After parsing is complete, the old abstract syntax subtree is replaced with the subtree stored in the attribute identified in the entry declaration. The example below shows an entry declaration that binds the *statement* abstract syntax phylum to the *ast* attribute of the *Statement* concrete syntax phylum.

```
statement ~ Statement.ast
```

There are several limitations to the incremental parsing provided by the Synthesizer Generator. First, the environment implementor must explicitly provide entry declarations to allow parsing. Second, the user must always edit complete syntactic constructs and produce syntactic constructs of the same phylum as the original. Third, the user is not allowed to leave text editing until the modified text is syntactically correct.

Whether the user uses syntax-directed editing or incremental parsing, the Synthesizer Generator requires the document to be syntactically correct at all times. The environment implementor can control when semantic checking is done and reported to the user to some extent by using *demand* attributes whose values are only computed when they are needed elsewhere. The user can also turn off semantic analysis if desired.

4.2 Pan

Syntax-directed editing has been criticized for numerous reasons. It unnaturally constrains the experienced user by requiring documents to always be built top-down. Furthermore, syntax-directed

editing typically requires the user to know implementation details of how the editor represents the document. (See [VGB92, Van92, Min92] for more detailed criticism.)

The user interface for Pan environments differs markedly from those for the Synthesizer Generator. Pan generates a syntax-recognizing interface rather than a syntax-directed interface. Pan supports a very general algorithm for incremental parsing that allows arbitrary text editing, and also allows the user to leave arbitrary syntactic (and semantic) errors in the document.

Parsing and semantic analysis are done incrementally, but only at the user's request. This has the advantage that the user is not interrupted with error messages when editing. However, to get the effect of eager analysis, the user must request analysis frequently.

An important aspect of Pan's user interface is its treatment of syntactically and semantically incorrect documents. Pan's philosophy is to treat the incorrect portions of a document as *variances*, rather than errors. In this spirit, variances are noted by placing a small graphical character (called a *glyph*) in the upper right corner of the screen to indicate the presence of a variance. The glyph may appear and disappear as the user modifies the document. The user only sees the messages corresponding to the glyph when he/she is ready. The user is never forced to change a document to remove the variances. This contrasts strikingly with Synthesizer Generator environments, which require that documents at least be syntactically correct at all times. Nevertheless, Pan attempts to provide as much incremental analysis as possible despite variances.

In addition, the Pan designers have paid considerable attention to the appropriate use of color and font styles such that they are truly informative rather than simply cluttering the screen. For example, the text in the document that corresponds to a syntactic variance is displayed in the same color as the glyph that indicates that variances exist. Thus the user can quickly identify, and possibly fix, syntactic variances without seeing an error message.

In addition to the text-editing interface, Pan also provides a structural interface based on the notion of *operand classes*, which can be dynamically created. Membership of a node in an operand class is determined by evaluating a predicate on the node. While operand classes can be used to create a syntax-directed interface to a Pan environment, they are much more general. A typical Pan environment would have operand classes corresponding to "language variance" and "query result". A user can select an operand class, like "language variance", and then use structural navigation to walk through his/her document, stopping only at nodes in the selected operand class, in this case those nodes that represent variances.

Operand classes provide a mechanism similar to the Synthesizer Generator's sparse views with two exceptions. First, Pan supports dynamic creation of operand classes, essentially allowing the user to define new ways to traverse his/her document, while only the environment implementor can do so in the Synthesizer Generator. Second, Pan's operand classes are orthogonal to the view mechanism. Thus, the same text can be displayed, but the interpretation of the structural navigation commands varies depending on the operand class being used. The Synthesizer Generator's sparse views combine the modified view and navigation into a single mechanism.

The user interface of a Pan environment is fully customizable, in the same spirit as Emacs. The environment implementor or user can define key bindings, fonts, menu contents, new commands, textual or graphical views, etc. using Common Lisp. Since Pan is written in Common Lisp, the customizations can access any functions defined by Pan, at least in principle. Thus, Pan supports full generality, but does not provide a simple declarative language to describe common textual representations as the Synthesizer Generator does.

5 Conclusions

Each environment generator surveyed here has made contributions to the field. However, neither provides all the features we would like to have. If we could choose components from each system and freely combine them, we might end up with the following hybrid. Pan's user interface is superior, not just in its syntax-recognizing approach, but also in more general user friendliness. Its treatment of ill-formed documents as variances rather than errors gives the user more control of the environment. It also uses fonts, colors, and other visual clues effectively to assist the user in understanding the state of the document.

A declarative notation for describing semantics would facilitate the development of environments, although, as previously noted, attribute grammars and logic constraint grammars are incapable of providing dynamic semantics. Kaiser's work on action equations [Kai85] addresses this problem by combining the attribute grammar notation with an event architecture. Action equations can be attached to productions and evaluated in the same manner as attribute equations. However, action equations can also be defined to be triggered only when certain events happen, such as when an instruction is executed. This allows action equations to capture the historical information necessary to support dynamic semantics.

While the syntactic description mechanisms provided by Pan are not as general as those provided by the Synthesizer Generator, the grammatical abstraction mechanism seems to support the types of differentiation needed between the abstract and concrete grammars, especially in light of the fact that the internal representation is hidden from the user.

We would also choose other features from environment generators that were not presented here. In particular, we would borrow ideas from Gandalf's support for programming-in-the-large [KSH89] and grammar evolution [GKS86] and Mercury's [KKM87] support for programming-in-the-large. These features allow more realistic environments to be developed and maintained over time with support for large documents, many programmers working concurrently, cooperating languages, and an internal representation that can evolve as the requirements of the environment change over time.

The systems presented here use radically different methods to define semantic processing. However, semantic descriptions are still rather difficult to write in either notation. Part of the problem is that the notations are very low-level and do not attempt to assist the definition of semantics by providing higher-level mechanisms to express common features of programming languages or environments, such as symbol table generation, type checking, definition of inheritance rules, version control, etc. Some work has been done to assist in the binding of names to definitions, such as [VL88, Cap85a, Rei83], and more is needed to facilitate other common types of semantic processing.

Also, while the environments produced by environment generators can support all phases of the software lifecycle in principle, the vast majority of environments have been developed for the coding phase. There are two reasons for this. First, until recently, the environment technology could only support a single language and a single programmer. To support all phases of the lifecycle, we need a design notation, a programming language, tests and their results, etc., all managed by the environment in a cooperative fashion. Second, until recently, software engineering processes were typically managed in ad hoc fashions. With the development of support for programming-in-the-large and the research into processes, the time is ripe to experiment with generated environments for more of the software lifecycle, and some work is beginning in that regard [SLC92].

Finally, both systems produce environments that are basically closed from interaction with externally-developed tools. While they can be integrated with other tools through procedure calls,

the interfaces are not clean or well thought-out. Clearly, if environment generation technology is to succeed, this limitation must be overcome. The Synthesizer Generator is attacking this problem by adding SoftBench style messages [Ger90] to support integration.

Acknowledgements

I am indebted to numerous people for assisting me with developing this paper. Tim Teitelbaum advised me about the Synthesizer Generator through numerous conversations. Susan Graham, Michael Van De Vanter, and John Pasalis provided assistance with Pan. I also want to thank Nico Habermann, Rick Adrion, Michael Van De Vanter, David Miller and Richard Lerner for their comments on drafts of this paper.

References

[Bal89] Robert Alan Ballance. *Syntactic and Semantic Checking in Language-Based Editing Systems*. PhD thesis, Computer Science Division — EECS, University of California, Berkeley, December 1989. Available as Report No. UCB/CSD 89/548.

[BC85] G. Beshers and R. Campbell. Maintained and constructor attributes. In *Proceedings of the ACM SIGPLAN Symposium on Language Issues in Programming Environments*, pages 34–42, Seattle, WA, July 1985.

[BCD+88] P. Borras, D. Clement, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. In *SIGSOFT 88: 3rd Symposium on Software Development Environments*, pages 14–24, Boston, November 1988.

[BGV92] Robert A. Ballance, Susan L. Graham, and Michael L. Van De Vanter. The Pan language–based editing system. *ACM Transactions on Software Engineering and Methodology*, 1(1):95–127, January 1992.

[BS92] Rolf Bahlke and Gregor Snelting. Design and structure of a semantics-based programming environment. *International Journal of Man-Machine Studies*, 37(4):467–479, October 1992.

[Cap85a] M. Caplinger. Structured editor support for modularity and data abstraction. In *Proceedings of the ACM SIGPLAN Symposium on Language Issues in Programming Environments*, Seattle, Washington, July 1985.

[Cap85b] Michael Caplinger. *A Single Intermediate Language for Programming Environments*. PhD thesis, Department of Computer Science, Rice University, Houston, Texas, 1985. Available as Rice COMP TR85-28.

[DGHKL84] Veronique Donzeau-Gouge, Gerard Huet, Giles Kahn, and Bernard Lang. Programming environments based on structrued editors: The MENTOR experience. In David R. Barstow, Howard E. Shrobe, and Erik Sandewall, editors, *Interactive Programming Environments*, chapter 7, pages 128–140. McGraw-Hill, New York, 1984.

[DRZ85] A. Demers, A. Rogers, and F. K. Zadeck. Attribute propagation by message passing. In *Proceedings of the ACM SIGPLAN Symposium on Language Issues in Programming Environments*, pages 140–147, July 1985.

[ELN+92] G. Engels, C. Lewerentz, M. Nagl, W. Schafer, and A. Schurr. Building integrated software development environments part I: Tool specification. *ACM Transactions on Software Engineering and Methodology*, 1(2):135–167, April 1992.

[Ger90] Colin Gerety. A new generation of software development tools. *Hewlett-Packard Journal*, 41(3):48–58, June 1990.

[GKS86] David Garlan, Charles W. Krueger, and Barbara J. Staudt. A structural approach to the maintenance of structure-oriented environments. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 160–170, Palo Alto, California, December 1986.

[Hed91] Gorel Hedin. Incremental static semantic analysis for object-oriented languages using door attribute grammars. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems: Proceedings of the International Summer School SAGA*, volume 545 of *Lecture Notes in Computer Science*, pages 374–379, Prague, Czechoslovakia, June 1991. Springer-Verlag.

[HGN91] Nico Habermann, David Garlan, and David Notkin. Generation of integrated task-specific software environments. In Richard F. Rashid, editor, *CMU Computer Science: A 25th Anniversary Commemorative*, Anthology Series, chapter 4, pages 69–97. ACM Press, Reading, Massachusetts, 1991.

[HKKL86] J. Heering, G. Kahn, P. Klint, and B. Lang. Generation of interactive programming environments. In *ESPRIT'85:, Status Report of Continuing Work, Part I*, pages 467–477. North-Holland, 1986.

[Hoo86] R. Hoover. Dynamically bypassing copy rule chains in attribute grammars. In *Proceedings of the 13th ACM Symposium on Principles of Programming Languages*, pages 14–25, St. Petersburg, Florida, January 1986.

[HT86] R. Hoover and T. Teitelbaum. Efficient incremental evaluation of aggregate values in attribute grammars. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, pages 39–50, Palo Alto, California, June 1986. Available as SIGPLAN Notices, July 1986.

[JF85] G.F. Johnson and C.N. Fischer. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, New Orleans, Louisiana, January 1985.

[JG82] Fahimeh Jalili and Jean H. Gallier. Building friendly parsers. In *Proceedings of the Ninth Annual ACM Symposium on the Principles of Programming Languages*, pages 196–206, Albuquerque, New Mexico, January 1982.

[Kai85] Gail E. Kaiser. *Semantics for Structure Editing Environments*. PhD thesis, Carnegie Mellon University Department of Computer Science, Pittsburgh, Pennsylvania, May 1985.

[KKM87] Gail E. Kaiser, Simon M. Kaplan, and Josehpine Micallef. Multiuser, distributed language-based environments. *IEEE Software*, 4(6):58–67, November 1987.

[Kru92] Charles W. Krueger. Software reuse. *ACM Computer Surveys*, 24(2):131–183, June 1992.

[KS89] Bernd Kramer and Heinz-Wilhelm Schmidt. Developing integrated environments with ASDL. *IEEE Software*, pages 98–107, January 1989.

[KSH89] Charles W. Krueger, Barbara J. Staudt, and A. Nico Habermann. Scaling up integrated software development environment databases. In *Proceedings of the 1989 ACM SIGMOD Workshop on Software CAD Databases*, pages 74–78, Napa, CA., February 1989.

[MHM+90] B. Magnusson, G. Hedin, S. Minor, et al. An overview of the Mjolner/Orm environment. In *Proceedings of the 2nd International Conference TOOLS (Technology of Object-Oriented Languages and Systems)*, pages 635–646, Paris, June 1990. Available as Lund University TR LU-CS-TR:90-57.

[Min92] Sten Minor. Interacting with structure-oriented editors. *International Journal of Man-Machine Studies*, 37(4):399–418, 1992.

[Rei83] S. P. Reiss. Generation of compiler symbol process mechanisms from specifications. *ACM Transactions on Programming Languages and Systems*, 5(2):127–163, April 1983.

[Rei85] Steven P. Reiss. PECAN: Program development systems that support multiple views. *IEEE Transactions on Software Engineering*, SE-11(3):276–285, March 1985.

[RT89] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Langauge-Based Editors*. Texts and Monographs in Computer Science. Springer–Verlag, New York, 1989.

[RT91] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator Reference Manual Release 3.5*. GrammaTech, Inc., Ithaca, NY, 1991.

[SLC92] A. V. Staa, C. J. P. Lucena, and D. D. Cowan. Talisman - a process-model driven software engineering environment. Technical Report CS-92-36, University of Waterloo Computer Science Department, Waterloo, Ontario, Canada, September 1992. Describes an environment generator for process-centered environments. So high-level as to be meaningless.

[Van92] Michael Lee Van De Vanter. *User Interaction in Langauge-Based Editing Systems*. PhD thesis, Computer Science Division — EECS, University of California, Berkeley, December 1992. Availabe as UCB/CSD-93-726.

[VGB92] Michael L. Van De Vanter, Susan L. Graham, and Robert A. Ballance. Coherent user interfaces for language-based editing systems. *International Journal of Man-Machine Studies*, 37(4):431–466, October 1992.

[VL88] Scott Vorthmann and Richard J. LeBlanc. A naming specification language for syntax-directed editors. In *Proceedings of the 1988 International Conference on Computer Languages*, pages 250–257, Miami Beach, Florida, October 1988. IEEE Computer Society Press.