Programming Techniques and Data Structures
M. Douglas McIlroy Editor

The Cornell Program Synthesizer: A Syntax-Directed Programming Environment

Tim Teitelbaum and Thomas Reps
Cornell University

Programs are not text; they are hierarchical compositions of computational structures and should be edited, executed, and debugged in an environment that consistently acknowledges and reinforces this viewpoint. The Cornell Program Synthesizer demands a structural perspective at all stages of program development. Its separate features are unified by a common foundation: a grammar for the programming language. Its full-screen derivation-tree editor and syntax-directed diagnostic interpreter combine to make the Synthesizer a powerful and responsive interactive programming tool.

Key Words and Phrases: programming environment, program development system, syntax-directed editor, template, diagnostic interpreter, source language debugger.

CR Categories: 4.12, 4.13, 4.30, 4.42, 4.43

I. Introduction

The Cornell Program Synthesizer is an interactive programming environment with integrated facilities to create, edit, execute, and debug programs. Our goal was to develop a unified programming environment that stimulates program conception at a high level of abstraction, promotes programming by step-wise refinement, spares the user from mundane and frustrating syntactic details while editing programs, and provides extensive diagnostic facilities during program execution.

We attained these goals by making the Synthesizer syntax-directed; both editing and execution are guided by the syntactic structure of the programming language.

The grammar of the programming language is embodied in a collection of *templates* predefined for all but the simplest statement types. Programs are created top-down by inserting new statements and expressions at a cursor position within the skeleton of previously entered templates. In general, the editing cursor can only be moved from one template to another and from one template to its constituents, and not simply from one line of text to another. Templates reinforce the view that a program is a hierarchical composition of syntactic objects, rather than a sequence of characters.

Runtime diagnostic facilities are likewise syntax-directed. Discrete computational units of execution correspond exactly to the syntactic units of the editor. When *tracing*, the screen cursor indicates the location of the instruction pointer in the source code as the program executes. When *single-stepping*, the user controls execution with respect to the hierarchical template structure of the program. The Synthesizer consistently demands a structural perspective.

The Synthesizer's editor is a hybrid between a tree editor and a text editor. Templates are generated by command, but expressions and assignment statements are typed one character at a time. It is impossible to make errors in templates because they are predefined. Errors in user-typed text are detected immediately because the parser is invoked by the editor on a phrase-by-phrase basis. By precluding the creation of syntactically incorrect files, the Synthesizer lets the user focus on the intellectually challenging aspects of programming.

Because code is generated each time a template or phrase is inserted, execution can follow editing without delay. Execution is suspended when a missing program element is encountered, but can be immediately resumed after the required code has been inserted. Thus, incomplete programs are executable; program development and testing can be conveniently and rapidly interleaved.

The design and implementation of the Program Synthesizer began in May 1978, and demonstrable prototype versions were operational under UNIX as well as on Terak (LSI-11) microcomputers by December 1978 [24, 25]. The Synthesizer, first used in classes at Cornell in June 1979, currently serves about 1500 of our introductory programming students a year. The Synthesizer has also been adopted for elementary programming instruction at Rutgers University, Princeton University, and Hamilton College.

The first language implemented for the Synthesizer was PL/CS, an instructional dialect of PL/I [5, 23]. PL/CS had previously been defined to serve as a vehicle for research on batch-oriented, error-correcting compilers [6] as well as for program verification [4]. We are currently developing a version of the Synthesizer for Pascal.

The development of the Cornell Program Synthesizer was supported in part by the National Science Foundation under Grants MCS77-08198 and MCS80-04218.

Authors' present address: Tim Teitelbaum and Thomas Reps, Department of Computer Science, Cornell University, 405 Upson Hall, Ithaca, NY 14853.

Certain individual features of the Synthesizer have appeared in previous systems:

—immediate phrase-by-phrase syntax analysis in BASIC [16],
—syntax-directed, program generation in EMILY [12] and the Fortran Language Anticipation and Prompting System [20],
—full-screen editing in the RAND editor, the Berkeley display editor ex[15], and ged[22],
—tree-editing in INTERLISP [26] and MENTOR [9],
—selective hierarchical file display at SRI [10],
—windowing in the Programmer's Assistant [27],
—screen-oriented execution monitoring in CAPS [28], and
—reverse execution in EXDAMS [3], PL/C [29] and BIDOPS [14].

The combination of these facilities in the Synthesizer has made it a powerful and responsive programming tool.

Related work in progress on language-specific programming environments includes LISPEDIT [1], and PDE1L [19] at IBM, Gandalf at CMU [11, 13], ALBE at Yale [17], and COPE at Cornell [2].

II. Program Editing

Programs are edited by inserting and deleting at the cursor position in a screen of text. The body of text expands either horizontally or vertically, as necessary, to accommodate insertions; it contracts when characters or lines are deleted. The screen serves as a window into a file. Whenever the cursor moves to a location in the file not contained on the screen, the file shifts automatically to include the new cursor position within the window. All screen modifications are essentially instantaneous on a high speed video terminal.

A. Files, Templates, Phrases, and the Cursor

A Synthesizer file is an object with hierarchical structure, not just a sequence of characters and lines. Files are composed of two kinds of elements: templates and phrases.

A *template* is a predefined, formatted pattern of characters and punctuation marks. The key words, punctuation, and indenting format of a template cannot be altered. The template provides an immutable framework for the insertion of additional program units. *Placeholders* identify the locations where these insertions are permitted. Each placeholder designates the syntactic class of permissible insertions. For example, the template for a conditional statement is:

```
IF (condition)
   THEN statement
   ELSE statement
```

where *condition* and *statement* are placeholders.

A *phrase* is an arbitrary sequence of typed symbols. For example, each of the following lines is a phrase:

```
k > 0
'not positive'
```

Both phrases and templates can be inserted into other templates at locations designated by placeholders, replacing the given placeholder. This nesting of templates, one within another, can occur to any depth.

All modifications of program text occur relative to the current position of the *editing cursor.* Although the cursor can be moved anywhere within a phrase, it can only be positioned at the leftmost symbol of a template or placeholder. The upper leftmost symbol of a template denotes the entire template including its constituents. The cursor never appears within the key words and punctuation marks of a template or in the margins. In general, the editing cursor is advanced from one template to another and from one template to its constituents. It is possible to position the cursor only where insertions and deletions are allowed.

Thus, the Synthesizer views the following partially developed file segment as a hierarchical composition of nested templates and phrases rather than three independent lines of text. The cursor, indicated by □, is positioned at the *statement* placeholder.

```
IF ( k > 0 )
   THEN □statement
   ELSE PUT SKIP LIST ( 'not positive' );
```

The cursor control keys move the cursor forward and backward through the program. For want of better names, we refer to the keys as **left,**[1] **right, up,** and **down.** Despite this nomenclature, the effect of the control keys is defined with respect to the one-dimensional reading order of a program, not the two-dimensional coordinate system of its display. Thus, both **right** and **down** move the cursor forward through the program; **left** and **up** move it backwards. Because much of the program text is immutable, the cursor jumps in logical increments, not character by character. Although **right** and **down** both move the cursor forward, their units of increment differ.

Up and **down** move the cursor one program element at a time, stopping only once per template, phrase, or placeholder. The following file segment shows with underlines all the possible stopping points for the cursor when the **up** and **down** keys are used:

```
IF (k > 0   )
   THEN statement
   ELSE PUT SKIP LIST ('not positive');
```

Left and **right** differ from **up** and **down** by also moving the cursor to every character within a phrase:

```
IF ( k > 0   )
   THEN statement
   ELSE PUT SKIP LIST ('not positive' );
```

[1] Boldface words such as **left** denote single keys of the terminal.

Other commands move the cursor in logical increments greater than **up** and **down** according to the nesting structure of the templates. The two-key sequence **long down** advances the cursor to the next element at the same structural level; the sequence **long up** moves backward similarly. The **diagonal** key moves the cursor to the immediately enclosing program element. The sequence **long diagonal** moves the cursor to the top of the program.

B. Insertions

Files are created top–down by inserting templates and phrases into existing templates where they are instantly displayed on the screen. Templates are not typed: they are generated by command. Each insertion occurs at the position of the editing cursor. Insertions can be made in any order, but only when the cursor is located at a placeholder. In order to insert the template

PUT SKIP LIST (*list-of-expressions*);

into

```
IF ( k > 0 )
   THEN [s]tatement
   ELSE PUT SKIP LIST ( 'not positive' );
```

the user types the command .p, and then strikes **return**. The insertion is instantly displayed on the screen, with the cursor automatically advanced to the placeholder *list-of-expressions*:

```
IF ( k > 0 )
   THEN PUT SKIP LIST ([l]ist-of-expressions );
   ELSE PUT SKIP LIST ( 'not positive' );
```

A template inserted by command is always syntactically correct for two reasons:

(1) The command is validated to guarantee that it inserts a template permitted at the current cursor position. (For example, typing the command .p with the cursor positioned at *list-of-expressions* is an error.)
(2) The template is a predefined unit. Because it is not typed, it contains no typographical errors.

Phrases, unlike templates, are explicitly typed at the position of the editing cursor. As the first character of a phrase is typed, the placeholder disappears. The phrase's right context shifts correspondingly, one character at a time, to accomodate the insertion. In the example above, ");" is the right context of *list-of-expressions*. It shifts one character at a time to the right as a phrase is typed at *list-of-expressions*. Phrases and their right contexts are automatically continued on consecutive lines, as necessary:

```
IF ( k > 0 )
   THEN PUT SKIP LIST ( the number k is strictly g
                reater than zero'[] );
   ELSE PUT SKIP LIST ( 'not positive' );
```

Because phrases are typed by the user, their syntactic correctness must be validated. Typed text is parsed as soon as the cursor is directed away from the phrase. Thus, the moment the user strikes **return** in the example above, the missing quote is detected, an error message is printed on the top line of the screen, and the editing cursor is positioned as close to the site of error as possible. The display would then appear:

```
IF ( k > 0 )
   THEN PUT SKIP LIST ( [t]he number k is strictly g
                reater than zero' );
   ELSE PUT SKIP LIST ( 'not positive' );
```

In this case, because the error detection mechanism properly positions the cursor, the required quotation mark can be inserted in a single keystroke. The phrase and its right context shifts right and down in response to this insertion:

```
IF ( k > 0 )
   THEN PUT SKIP LIST ( '[t]he number k is strictly
                greater than zero' );
   ELSE PUT SKIP LIST ( 'not positive' );
```

Directing the cursor away from the phrase invokes the parser again. Because the phrase is now syntactically correct, the cursor is positioned as directed. Phrases are prettyprinted and redisplayed after being parsed successfully.

C. Modifications

Structural changes to the program are accomplished by removal and insertion of whole templates and phrases. This highly disciplined mode of modification guarantees the structural integrity of the program at every step. The position of the editing cursor always denotes a whole program unit: template, phrase, or placeholder. Thus, it is not necessary to specify line limits in order to remove an entire program unit:

delete (move the template or phrase to the file DELETED),
clip (move the template or phrase to the file CLIPPED),
.mv f (move the template or phrase to the file f).

After a program unit has been removed, the display is immediately redrawn and the original placeholder reappears. The editing cursor can then be repositioned and the file segment reinserted:

insert (insert the contents of CLIPPED at the current cursor position),
.ins f (insert the contents of file f at the current cursor position).

In the example below, because the cursor is positioned on the IF, it denotes the whole code segment shown:

```
[I]F ( k > 0 )
   THEN PUT SKIP LIST ( 'the number k is strictly
                greater than zero' );
   ELSE PUT  SKIP LIST  ( 'not positive' );
```

Suppose we wish to enclose this IF-statement within a WHILE-loop. In a conventional, line-oriented, text editor, the lines

```
DO WHILE ( condition );
    END;
```

would be inserted before and after the IF-statement in separate editing steps. The first of these editing steps would drastically alter the structure of the program; in fact, it would make it temporarily incorrect due to the unbalanced DO-END pair. Such modifications have been the bane of incremental compilation schemes.

By contrast, in the Synthesizer, the two lines are part of one template and are therefore inserted simultaneously. First, the IF-statement is temporarily removed from the file, leaving the cursor positioned at a {*statement*} placeholder:

```
[{]statement}
```

Next, the WHILE-loop is inserted and the cursor advanced into the body of the loop to a subordinate {*statement*} placeholder:

```
DO WHILE ( condition );
    [{]statement}
    END;
```

Finally, the IF-statement is inserted into the body of the loop, automatically indented further to the right:

```
DO WHILE ( condition );
    [I]F ( k > 0 )
        THEN PUT SKIP LIST ( 'the number k is strict
                    ly greater than zero' );
        ELSE PUT SKIP LIST ( 'not positive' );
    END;
```

The special function keys enable this manipulation to take place in a sequence of just seven keystrokes:

clip	(remove the IF-statement),
.dw return	(insert the WHILE-loop),
return	(move the cursor to {*statement*}),
insert	(reinsert the IF-statement within the WHILE-loop).

The reverse manipulation, extracting the IF-statement and discarding the WHILE-loop, would require only four keystrokes:

clip	(remove the IF-statement),
diagonal	(move the cursor to DO),
delete	(delete the WHILE-loop),
insert	(reinsert the IF-statement).

Individual phrases are modified by first positioning the cursor anywhere within the phrase, and then modifying individual characters. As each change is typed, the context surrounding the phrase adjusts instantly. Character insertions are made simply by typing; there is no separate insert mode. Using special function keys, it is possible to erase characters forwards or backwards either one at a time or all the way to the boundary of the phrase. Whenever modified, a phrase is checked for syntactic correctness, exactly as if it had just been introduced for the first time. If every character of a phrase is deleted, the appropriate placeholder automatically reappears.

Templates, unlike phrases, cannot be modified—they are immutable. Insertions are permitted within templates only at the positions designated by placeholders. The predefined key words and punctuation marks of a template cannot be changed. In fact, it is not even possible to position the editing cursor within the characters of a template.

An initial and overriding goal of the Synthesizer was to guarantee that programs were completely correct at every stage of their development. Any modification that introduced an error was to be prevented. Context-sensitive constraints of the syntax forced us to retreat from that position. For example, consider the problem of changing the type of a program variable. Because a key word such as FIXED is part of an immutable declaration template, it is necessary to delete one declaration and insert another. However, the implemented dialect of PL/I requires that all variables be declared; deleting the declaration would introduce undeclared variable errors in every phrase referencing the variable.

Rather than having a separate mechanism to make such modifications atomic operations, the Synthesizer tolerates invalid phrases, highlighting them with the complemented character font until corrected. Thus, the moment a declaration is deleted, all phrases containing the undeclared variable are highlighted. When the new declaration is inserted, all are redisplayed in the normal font.

To illustrate this, consider deleting the declaration of the variable temp from:

```
DECLARE ( temp ) FIXED;
DECLARE ( m, n ) FLOAT;
temp= m;
m= n;
n= temp;
```

Errors introduced in the first and third assignment statements because of undeclared variables are highlighted:

```
DECLARE ( m, n ) FLOAT;
temp= m;
m= n;
n= temp;
```

When temp is redeclared, uses of temp become correct and are again displayed in the normal font:

```
DECLARE ( m, n, temp ) FLOAT;
temp= m;
m= n;
n= temp;
```

Users often forget declarations until reminded by an *undeclared variable* error message. Tolerance of invalid

phrases simplifies correction of this common error. When an error is detected in a phrase, any movement of the cursor away from the phrase overrides the error prevention mechanism and highlights the invalid phrase. The proper declaration can be inserted at any time.

As an additional benefit, the ability to override the error prevention mechanism allows free-form ideas to be sketched temporarily in the file without regard to the syntax of the programming language. These informal program notes and plans remain highlighted until they are either completed correctly or deleted. Although invalid phrases are permitted in programs, the overall structural integrity of files is maintained because the correct nesting of templates is always enforced.

D. Syntactic Iteration and Optional Placeholders

As described thus far, creating a file in the Synthesizer is exactly analogous to deriving a sentence with respect to a context-free grammar for the given programming language. Although a nontechnical vocabulary has been adopted in the presentation, the following correspondencies should be clear:

placeholder	nonterminal symbol
template	right side of a production
command	the name of a production
insertion	derivation
file	derivation tree
cursor position	node of derivation tree
file display	sentential form

In addition, the editor incorporates the usual metasyntactic formalism for syntactic iteration:

{*placeholder*}	zero or more occurrences of *placeholder*

Conceptually, each item in such a list is preceded and followed by {*placeholder*}. However, these placeholders are only displayed when the cursor is positioned there (or when there are no occurrences in the list). In fact, **return** is the cursor motion that means:

advance the cursor to the next template, phrase, or placeholder *including iterated placeholders.*

The following sequence of display snapshots illustrates repeated use of **return** to advance the cursor:

Original screen	After 1 **return**	After 2 **returns**	After 3 **returns**

[x]= 0;
y= 0;
	x= 0;
	[{]*statement*}
	y= 0;
		x= 0;
		[y]= 0;
			x= 0;
			y= 0;
			[{]*statement*}

When entering templates and phrases, each insertion terminated by **return** has the desired effect of advancing the cursor to the next possible placeholder for an insertion. Cursor motions other than **return** do not reveal iterated placeholders.

Optional components of templates are denoted by square brackets:

[*placeholder*]	zero or one occurrence of *placeholder*

In order to avoid excessive clutter on the screen, such optional placeholders are not normally displayed. A separate cursor motion is used to reveal them and to position the cursor there. For example, from

```
[D]O WHILE ( condition );
     {statement}
     END;
```

the move-to-optional-component command advances the cursor to

```
[[]loop-name:] DO WHILE ( condition );
   {statement}
   END;
```

E. Comment Templates

The limited number of lines displayed on video terminals hampers editing large files. *Comment templates* provide a mechanism for hiding details of a file, thereby allowing more of the program to be displayed. The comment template also provides a mechanism for controlling the speed as well as the scope of runtime diagnostic monitoring. A comment template is a single program unit expressing a program specification together with its refinement [7]. It is the structural unit used to express computational abstractions in programs:

```
/* comment */
  {statement}
```

The two placeholders, *comment* and {*statement*}, are part of *one* template. The {*statement*} part of the template is indented to show that it is the refinement of the specification provided in the *comment* part. Thus, a comment is not an arbitrary lexical insertion into the program; rather, it is a structural unit in its own right. The *scope* of a comment is the list of statements in its refinement.

The display of the refinement of a comment can be suppressed simply by striking the **ellipsis** key. For example,

```
/* Swap m and n */
    temp= m;
    [m]= n;
    n= temp;
PUT SKIP LIST (m, n);
```

would be redisplayed instantly as

```
/* Swap m and n */
    [.]..
PUT SKIP LIST (m, n);
```

Having hidden the details of the code, more of the program fits on the screen while the comment remains displayed to specify what the hidden refinement does. The hidden code at . . . can easily be revealed by striking **ellipsis** again. Thus, comment templates provide selective display of the hierarchical structure of files. The ellipsis feature is intentionally coupled with comment templates in order to encourage program documentation.

Comment templates serve a purpose during execution as well as editing. During program execution, . . . is considered a single atomic step for the runtime flow-tracing and pacing features described in the next section. For example, suppose the execution pace were set at a half-second per step. Then, . . . would be executed in as close to half a second as possible, regardless of the complexity of the statements hidden below the . . . . Thus, judicious use of . . . provides selective control of the speed and scope of diagnostic monitoring.

The ellipsis feature of comment templates provides an incentive to use comments during program development, rather than after the fact. Because it rewards a skillful, precise use of comments, this feature promotes good programming style and method.

III. Execution of Programs

Because programs are translated and maintained in interpretable form during editing, there is no compilation delay between editing and execution. Whenever execution is suspended, control returns to the editor, a printed message explains why execution was suspended, and the file cursor is positioned in the source program at the point of suspension. It is possible to run incomplete programs. Execution is suspended whenever a placeholder is encountered and can be resumed after the missing program element has been inserted. After most editing changes to a program, it is still possible to resume execution; certain changes, such as modifying a declaration, destroy the possibility of resuming execution.

The high transmission rate of a video display terminal allows the incorporation of unique runtime debugging aids. Display-oriented monitoring facilities provide a window into the computer through which one observes a running program. Their power is enhanced when combined with syntax-directed commands for controlling execution.

The flow of execution through the program can be traced using the screen cursor to indicate the location of the instruction pointer at each moment. The stopping places for the cursor during flow tracing correspond to the structural units of the editor—one cursor jump for each template and phrase. Thus, the editor and the interpreter share a unified view of program structure—separate editable units are seen as separate computational units. During flow tracing, the program display is automatically redrawn whenever control passes outside the display window or a procedure is called. Judicious use of the ellipsis feature can eliminate the trace of uninteresting sections of code and minimize undesirable redrawing of the program display.

Flow tracing at full speed provides a visible performance measure: the distribution of light intensities at the various cursor locations clearly indicates the fraction of time spent there. A *pace* feature allows the user to slow execution to any speed.

A syntax-directed *single-step* feature permits manual control of program execution. There are five ways to specify the step size of each resumption in terms of the template and phrase structure of the file

resume	execute one step of the current program element,
long resume	execute all steps of the current program element,
return	complete a list of statements,
diagonal	complete the enclosing template,
long diagonal	complete the enclosing procedure.

Consider stepping through the following program segment:

```
DO WHILE ( k < n );
    IF ( k>0 )
        THEN PUT SKIP LIST ( 'the number k is strict
                    ly greater than zero' );
        ELSE PUT SKIP LIST ( 'not positive' );
    k= k + 1;
    END;
```

resume would advance the cursor to k>0, **long resume** would advance the cursor to k= k + 1; by executing the entire IF-statement, **return** would complete the statements in the body of the loop and position the cursor at k<n, **diagonal** would position the cursor at DO until the loop is completed, and **long diagonal** would position the cursor at the top of the procedure until control returns to the calling procedure.

In this way, the Synthesizer maintains a unified view of both static and dynamic program structure. The syntactic units of the editor are the computational units of the interpreter.

Selected variables can be monitored during execution by displaying their names and values in a separate partition of the screen. The result of each assignment to a variable immediately appears on the screen replacing the previous value displayed for that variable. Currently, only one element of an array is displayed at a time. A least-recently updated replacement strategy is used when there is not enough room to display all monitored variables.

The facilities described above allow one to observe an error as it occurs. The *pace* feature, the *step* feature, and the pause statement serve as a throttle, providing control over the rate of execution, and thus providing a better chance for seeing errors. The program can be run at top speed until reaching the vicinity of the bug,

whereupon it can be paced or single-stepped until the error is observed.

It is easy to overshoot the mark, so the Synthesizer has a gear shift as well; a recently implemented reverse execution facility allows the program to run backwards a bounded number of steps. As with forward single-stepping, the step-size of each backwards step is specified in terms of the syntactic structure. Using the visual feedback provided by flow tracing and variable monitoring, and by alternating the direction of execution in the vicinity of a bug, the user is able to converge swiftly and precisely on the error. While reverse execution capabilities have been implemented by others [3, 14, 29], the novelty in the Synthesizer is the presence of enough immediate visual feedback to make the feature meaningful as a real-time control mechanism.

IV. Advantages of Templates

The Synthesizer's use of templates is central to achieving our design goals. The integrated behavior of templates and the cursor enforces the proper view that a program is a hierarchy of structurally nested components. Each template insertion is syntactically correct because template commands are only valid in appropriate contexts, and the templates are predefined. A program developed on the Synthesizer is always well-formed, regardless of whether it is complete or not.

Templates eliminate mundane tasks of program development. Typographical errors in structural units are impossible; indentation is automatic, both when a template is introduced and when it is moved, and errors cannot be introduced by modification because templates are immutable.

Placeholders in templates serve both as prompts and as syntactic constraints, by identifying places that can or must be refined, as well as by restricting the range of choices to legitimate insertions. The template-generated skeleton simplifies incremental compilation by bounding the extent of the program affected by modifications.

Template insertion is an economical mode of program entry with a corresponding economical implementation. Short commands insert long templates; because so few keystrokes are needed, typing mistakes are effectively minimized, and program entry is rapid. When statements are synthesized by command, there is no need for a parser to analyze the text. Thus, a template-based environment is ideal for microcomputers where space is in short supply.

Templates correspond to abstract computational units; because they are both inserted and manipulated as units, the process of programming begins and continues at a high level of abstraction; the user is never mired in syntactic detail. At runtime, templates provide a framework for the structured, single-step debugging facility. Thus, templates provide a unified view of both static and dynamic program structure.

V. Integration of Text and Structure

The Synthesizer is based on the premise that programs are not text. Although we want to promote the structural perspective, we recognize that abstract programs must be viewed and manipulated with respect to some concrete textual representation. The hybrid design of the Synthesizer seeks a pragmatic balance between the extremes of a derivation-tree editor and a text editor.

We believe we have successfully interleaved structural and textual features so that shifts between the two perspectives occur smoothly and spontaneously. Partitioning a language such as PL/I into templates and phrases seems natural and lets each construct be edited in an appropriate manner. User competence and comfort within the environment is enhanced by the persistence and uniformity of the template-phrase distinction throughout the system. The dual interpretation of the cursor as both text pointer and tree pointer seems intuitive: When the cursor is moved, it is perceived as a point stepping through program *text* in increments dictated by syntactic structure; when it stops, the cursor designates an entire *structure* that can be clipped or deleted as a unit.

Although, on the whole, we believe the Synthesizer accomplishes a harmonious integration of text and structure, occasionally, tension between the two perspectives leads to confusion and some inconvenience. One common difficulty stems from the fact that different abstract objects have identical representations as text. For example, consider trying to move the phrase k = 0 to *statement* in the code segment below:

```
IF ( k = 0 )
   THEN statement
```

Although k = 0 is textually correct as an assignment statement,[2] it cannot be moved to *statement* because it is an instance of the syntactic class *condition*. By preventing such implicit syntactic transformations, a structure editor may detect modifications with unforeseen and unintended consequences. However, when the consequences are intended, requiring a special mechanism to divorce a phrase of text from its syntactic classification is inconvenient.

A second example of tension between the textual and structural perspectives is the result of our decision to differentiate between declarations of formal parameters and local variables in the procedure template:

```
name: PROCEDURE (parameters);
      {parameter declaration}
      {declaration}
      {statement}
      END name;
```

Although the templates for declaring parameters and local variables are distinct,

[2] We ignore the missing semicolon for the sake of discussion.

```
DECLARE ( list-of-parameters ) FIXED;
DECLARE ( list-of-variables ) FIXED;
```

once expanded they appear the same, as in:

```
name: PROCEDURE ( j )
      DECLARE ( j ) FIXED;
      DECLARE ( k ) FIXED;
      {statement}
      END name;
```

Now suppose we wish to change k to be a parameter of the procedure. Because k is declared as a local variable, we cannot just add it to the parameter list on the first line, even though the resulting program would be correct *as text.* This is particularly confusing because the declaration for k not only looks like a parameter declaration but appears to be in the right place.

This example also illustrates an interplay in the Synthesizer between *incremental* error detetction and *left-to-right* error detection. Whenever a phrase is created or modified, the Synthesizer's local incremental error detection mechanism verifies that the new phrase is consistent with the previous state of the program. Any inconsistency leads to an immediate error message. As we have seen, adding k to the parameter list is an error because, at the time the change is attempted, k is already declared to be a local variable.

However, when the user overrides the local error prevention mechanism and allows an invalid phrase to remain in the program, the Synthesizer abandons the incremental viewpoint in favor of a static, left-to-right notion of correctness. The incrementally incorrect parameter list j,k turns out to be correct, whereas the declaration of k as a local variable is invalid because k already appears as a parameter:

```
name: PROCEDURE ( j, k );
      DECLARE ( j ) FIXED;
      DECLARE ( k ) FIXED;
      {statement}
      END name;
```

The declaration for k must now be moved from the {*declaration*} part of the procedure template to the {*parameter declaration*} part. However, as in our first example, the problem is not just a question of the template being in the wrong position. The existing declaration is bound to the syntactic category {*declaration*} and cannot be moved to any other placeholder; it must be deleted and a new parameter declaration created.

A third example of inconvenience in the Synthesizer's hybrid design is a result of the immutability of templates. Modifications that change only a few characters of program text may require a significant amount of restructuring. For example, if the program were text, a WHILE-loop could be changed into an UNTIL-loop by substitution of just five characters. In the Synthesizer, this change must be accomplished by moving the constituents of the existing WHILE-template into a newly inserted UNTIL-template. Although such modifications can be made rapidly using the **clip, insert,** and **delete** keys, they are admittedly awkward.

To alleviate such inconveniences imposed by structural constraints, we are building mechanisms that streamline these operations, yet enforce a discipline that emphasizes the abstract computational meaning of program units. For example, template-to-template transformations allow controlled changes in a single step. Positioning the cursor at a DO WHILE and typing the command .du could transform a WHILE-template into an UNTIL-template, leaving its constituents in corresponding places. Similarly, a local variable declaration could be transformed into a parameter declaration and repositioned in one step.

Besides purely syntactic transformations, a more powerful collection of semantics preserving transformations is also possible. For example, in one operation,

```
DO j= k to n by 1;
   {statement}
   END;
```

could be transformed into one of several equivalent alternative representations:

```
j= k;
DO WHILE ( j ≤ n );
   {statement}
   j= j + 1;
   END;
```

```
j= k;
IF ( j ≤ n )
   THEN DO UNTIL ( j > n );
        {statement}
        j= j + 1;
        END;
```

A similar transformation capability would allow procedures to be extracted from in-line code. The user would specify a section of program and the variables to become formal parameters. Then, in one operation,

```
/* Swap m and n */
   temp= m;
   m= n;
   n= temp;
```

would be replaced by

```
call swap (m, n);
```

with the appropriate procedure created automatically:

```
/* Swap m and n */
swap: PROCEDURE ( m, n );
   DECLARE ( m, n ) FLOAT;
   DECLARE ( temp ) FLOAT;
   temp= m;
   m= n;
   n= temp;
   END swap;
```

Such transformations will add considerable editing

power but retain the disciplined viewpoint of the rest of the system.

A final example illustrates an awkwardness arising not from the structural constraints of the Synthesizer, but from the textual constraints of a language whose concrete syntax was defined to be unambiguous for parsers. Inserting the template

IF (*condition*)
  THEN *statement*

into

IF (*condition*)
  THEN [s]*tatement*
  ELSE PUT LIST ('whose else am i?');

leads to an inconsistency between the explicitly derived structure (an IF–THEN within an IF–THEN–ELSE) and the structure implied by the parser-oriented concrete syntax (an IF–THEN–ELSE within an IF–THEN). Although tempted to adopt the derived interpretation (because prettyprinting easily distinguishes one interpretation from the other), we elected, instead, to maintain compatibility with PL/I. Therefore, we prevent such an insertion and require that the user provide a compound statement explicitly.

There are many possible alternative designs, among them the following four: a) the compound statement could be inserted automatically when necessary; b) a compound statement could be displayed automatically when necessary; c) the IF–THEN–ELSE template could be defined as

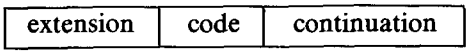
IF (*condition*)
  THEN DO; {*statement*} END;
  ELSE DO; {*statement*} END;

d) the IF–THEN template could be eliminated thereby requiring that every conditional statement have an ELSE-clause. In this final case, the display of an empty ELSE clause could be suppressed unless necessary for disambiguation.

VI. Implementation

A. File Trees

Synthesizer files are represented internally as executable derivation trees. Each template or phrase is represented in this tree by a separate node. The pointers connecting nodes are, in fact, **goto** instructions for the interpreter; the null pointer is a **halt** instruction. Nodes are variable length; each is composed of three sections:

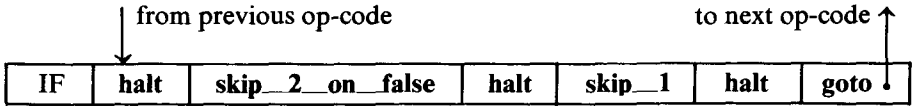


The *extension* identifies the node type and contains any other information needed to generate the display of the node but not necessary to execute it. The *code* section contains interpretable op-codes for executing the node. The *entry point* of the node is the first byte of the code section. The *continuation* contains a **goto** linking this node to the next op-code to be executed. The target of this **goto** is either the entry point of a sibling node or an interior op-code of a parent node.

For example, the template

IF (*condition*)
  THEN *statement*
  ELSE *statement*

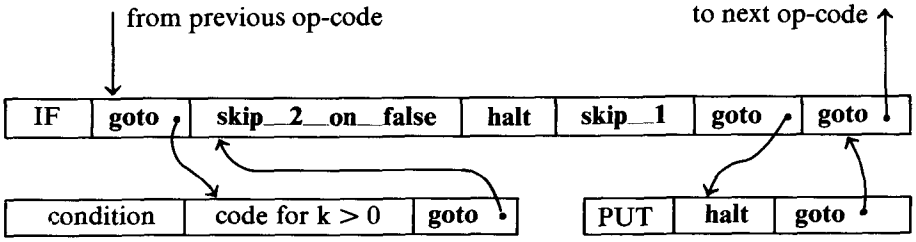
has the internal representation given below.



This node is tagged in the extension as an IF-node. It contains op-codes that implement the proper control flow and three **halt** instructions that represent the unexpanded placeholders. When the template has been expanded to

IF (k > 0)
  THEN *statement*
  ELSE PUT LIST (*list-of-expressions*);

a link to Polish postfix code for the phrase k > 0 replaces the first **halt** op-code, and a link to the node for the PUT-statement replaces the third **halt** op-code. A **halt** instruction remains for the other *statement* placeholder:



The interpreter is classical: it executes straight line code and **goto** instructions. It is completely blind to the structure of the tree and requires neither recursion nor a stack to execute a file tree. Access to variables and procedure definitions is through a symbol table.

The editor walks the tree using the same **goto** pointers as the interpreter. Each cursor position designates one of the nodes of the tree. Cursor motion is defined with respect to a preorder traversal. There are no backward pointers; thus, backward cursor motion is implemented internally by going all the way around.

B. Declarations

As demonstrated in Sec. II.C, declarations present a special problem: modifying a declaration can simultaneously introduce errors and correct errors at other locations in the program. Internally, information about identifiers is stored in a symbol table. When a declaration is modified, the Synthesizer discards the old symbol table and traverses the tree in preorder reparsing and redoing the semantics of every phrase. Phrases with errors are marked as invalid and are printed in the highlighted font when the screen is redrawn. Because the allocation of variables within an activation record is recomputed in the process of reconstructing the symbol table, access to the variables of a suspended activation record is lost in the process. Therefore, execution cannot be resumed after such modifications.

C. Displaying the Tree

The print representation of a file is generated from the tree; a text representation is not saved. The external representation of each kind of template is stored in a table. The entries of this table alternate between terminal strings and placeholder-descriptors. For example, the IF-template is encoded as:

"IF ("
condition-descriptor
") \{\nTHEN"
statement-1-descriptor
"ELSE"
statement-2-descriptor
"\}\r"

The placeholder-descriptors identify the placeholders and their positions within the code section of an internal node. The terminal strings contain key words, punctuation marks, and formatting control characters that are interpreted on output. For example,

\{ means move left-margin right one unit,
\n means line-feed, carriage-return to current left-margin,
\} means move left-margin left one unit,
\r means carriage-return to current left-margin.

The print routine traverses the tree in preorder, simultaneously keeping track of position within the external representation of the appropriate template. Each terminal string encountered is printed and its formatting commands obeyed. Each phrase is translated from postfix to infix for display. (The parentheses of a phrase are saved in the extension of the node encoded one bit per operator.)

As the tree is traversed for display, a table mapping internal node addresses to external screen coordinates is updated. This table is used both for cursor motion in the editor, and at runtime for the trace feature.

D. Implementation of Debugging Features

The tracing, pacing, and single-step features are implemented by taking appropriate action on the interpretation of each **goto** leading to a new node.

When tracing, each **goto** uses the map from internal node addresses to screen coordinates to determine the new cursor position. If the map is not defined for a given target node, then the cursor lies outside the window and the program is redrawn with the new cursor position centered in the window. Traced programs are never permitted to run any faster than one cursor update per refresh of the video screen in order to avoid stroboscopic effects such as loops that appear to run backwards. When pacing, the interpreter waits appropriately at each **goto** before continuing execution. When stepping, the interpreter waits for a resume command before continuing.

The variable-monitoring feature is implemented in a straightforward manner: a table mapping identifiers to screen positions is maintained. Assignment to a monitored variable is detected by the interpreter whereupon the appropriate position is updated on the screen.

Reverse execution also has a straightforward implementation: the forward execution interpreter maintains a history file of the flow of control and the values destroyed by assignments to variables. The reverse execution interpreter restores values and updates the screen to give the illusion of the program executing backwards.

VII. The Synthesizer Generator

Continuing research and development of the Synthesizer will increase its power, versatility, and range of application complementing the unique syntax-directed mechanisms the environment already provides. For example, global data flow analysis techniques will be used to answer queries about static program structure, as in [18]. The video display can be used to express static relationships between components of a program; the multiple fonts of a terminal can be exploited to highlight regions of interest. For example, the programmer might request the highlighting of all uses or all assignments to a variable *X*. Alternatively, the analysis can be keyed to the present location of the editing cursor. For example, the programmer might request the highlighting of all assignments to *X* that can account for its value at the present cursor location, or all possible uses of *X* that can

be reached from the present cursor location.

To facilitate such further development, we are implementing a language-independent system for generating Synthesizer-like systems from a grammatical specification of a given programming language. An attribute grammar will be used to define the syntax, display format, and semantics of each template and phrase. In our application, where program units are inserted and deleted in arbitrary order, semantic analysis must be both incremental and reversible. For this purpose, attribute grammars have the advantage of expressing semantics and context-sensitive constraints applicatively and on a modular basis; the arguments to each semantic function are imported explicitly from neighboring nodes in the derivation tree.

Because propagation of semantic information through the tree is implicit in the formalism, an incremental attribute evaluator can update the appropriate attribute values in conjunction with each editing operation. In particular, because the attribute dependencies are known, the evaluator can delete semantic information automatically when program units are deleted; a separate mechanism to undo semantics is not needed. We have described one such incremental attribute evaluator in [8]; more recently, we have developed an optimal-time incremental evaluator that runs in time proportional to the number of attribute values that actually must be changed [21].

Acknowledgments. Many people have participated in the development of the Synthesizer. We are deeply indebted to A. Demers for many stimulating discussions and for writing the LSI-11 operating system kernel; his insights and assistance have been invaluable. We are also extremely grateful for the generous help of J. Archer, R. Conway, M. Fingerhut, D. Gries, C. Hauser, S. Horwitz, D. Jacobs, R. Johnson, D. Krafft, S. Mahaney, and R. Olsson.

Received 5/80; revised and accepted 4/81

References

1. Alberga, C.N., Brown, A.L., Leeman, G.B., Mikelsons, M., and Wegman, M.N. A program development tool. Conference Record of the 8th Ann. Symp. on Principles of Programming Languages, Williamsburg, VA, Jan., 1981, 92–104.
2. Archer, J., Conway, R., Shore, A., and Silver, L. The CORE user interface. Tech. Report No. TR80-437, Dept. of Comptr. Sci., Cornell Univ., Ithaca, NY, Sept. 1980.
3. Balzer, R.M., EXDAMS–EXtendable Debugging and Monitoring System, AFIPS Proc. V. 34 (SJCC 1969), 567–580.
4. Constable, R., and O'Donnell, M.J. *A Programming Logic.* Winthrop, Cambridge, MA, 1978.
5. Conway, R. and Constable, R. PL/CS–A disciplined subset of PL/I. Tech. Rept No. 76-293, Dept. of Comptr. Sci., Cornell 1976.
6. Conway, R. *Primer on Disciplined Programming Using PL/CS.* Winthrop, Cambridge, MA, 1978.
7. Conway, R. and Gries, D. *An introduction to programming—a structured approach using* PL/I *and* PL/C. Winthrop, Cambridge, MA, 1979, 135–137.
8. Demers, A., Reps, T., and Teitelbaum, T. Incremental evaluation for attribute grammars with application to syntax-directed editors. Conference Record of the 8th Ann. Symp. on Principles of Programming Languages, Williamsburg, VA, Jan. 1981.
9. Donzeau-Gouge, V., Huet, G., Kahn, G., Lang, B., and Levy, J.J. A structure-oriented program editor. Tech. Rept, IRIA–LABORIA, France 1975.
10. Engelbart, D.C. and English, W.K. A research center for augmenting human intellect. AFIPS Proc. V. 33 (FJCC, 1968).
11. Feiler, P.H. and Medina-Mora, R., An incremental programming environment. Dept. of Comptr. Sci., Carnegie-Mellon Univ., Pittsburgh, PA, April 1980.
12. Hansen, W. Creation of hierarchic text with a computer display. Ph.D. Thesis, Comptr. Sci. Dept, Stanford University, Stanford, CA, June 1971.
13. Habermann, A.N. An overview of the Gandalf project. Comptr. Sci. Res. Rev. 1978–79, Carnegie-Mellon Univ., Pittsburgh, PA, 1979.
14. Hodgson, L.I., and Porter, M. BIDOPS: A bi-directional programming system. Dept. of Comptr. Sci., Univ. of New England, Armidale, N.S.W., Australia, 1980.
15. Joy, B. Ex Reference manual. Dept. of Electrical Eng. and Comptr. Sci., Univ. California, Berkeley, CA, 1977.
16. Kurtz, T.E. BASIC. SIGPLAN Notices, Aug. 1978.
17. Lewis, J.W. and Porges, D.F. ALBE/P: a language-based editor for Pascal. Dept. of Comptr. Sci., Yale Univ., New Haven, CT.
18. Masinter, L.M. Global program analysis in an interactive environment. Xerox PARC Report SSL-80-1, Jan. 1980.
19. Mikelsons, M. and Wegman, M.N. PDE1L: The PL1L program development environment principles of operation. Res. Rept RC8513, IBM, Thomas J. Watson Research Center, Yorktown Heights, NY, Nov. 1980.
20. Pinc, J.H. and Schweppe, E.J. A Fortran language anticipation and prompting system. Proc. ACM Nat. Conf., Atlanta, Georgia, 1973.
21. Reps, T. Optimal-time incremental semantic analysis for syntax-directed editors. Tech. Report No. 81-453, Dept. of Comptr. Sci., Cornell University, Ithaca, NY, March 1981.
22. Skinner, G. Ged user documentation. Dept. of Comptr. Sci., Cornell Univ., Ithaca, NY,
23. Teitelbaum, T. A formal syntax for PL/CS. Tech Rept 76-281, Dept. of Comptr. Sci., Cornell Univ., Ithaca, NY, 1976.
24. Teitelbaum, T. The Cornell Program Synthesizer: a microcomputer implementation of PL/CS. Tech. Report No. TR79-370, Dept. of Comptr. Sci., Cornell Univ., Ithaca, NY, June 1979.
25. Teitelbaum, T. The Cornell program synthesizer: A tutorial introduction. Tech. Report No. TR79-381, Dept. Comptr. Sci., Cornell Univ., Ithaca, NY, July 1979, Revised Jan. 1980.
26. Teitelman, W. INTERLISP reference manual. Xerox PARC, 1974.
27. Teitelman, W. A display-oriented programmer's assistant. Xerox PARC, March 1977.
28. Wilcox, T.R., Davis, A.M., and Tindall, M.H. The design and implementation of a table driven, interactive diagnostic programming system. *Comm. ACM 19*, 11 (Nov. 1976), 609–616.
29. Zelkowitz, M. Reversible execution as a diagnostic tool. Ph.D. Thesis, Dept. of Comptr. Sci, Cornell Univ., Ithaca, N.Y., Jan. 1971.