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

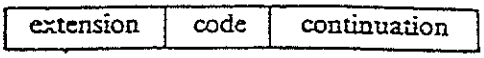
IF (*condition*)
  THEN DO; {*statement*} END;
  ELSE DO; {*statement*} END;

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:

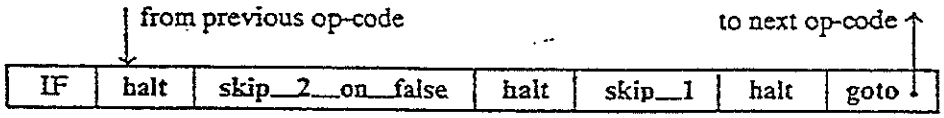


The *extension* identifies the node type and contains any other information needed to generate the display of the node but not necessary to execute it. The *code* section contains interpretable op-codes for executing the node. The *entry point* of the node is the first byte of the code section. The *continuation* 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*

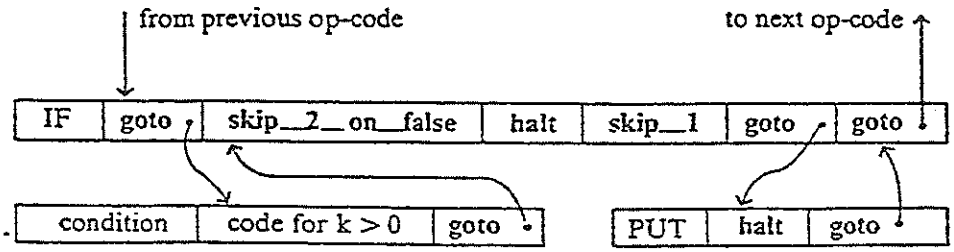
has the internal representation given below.



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. PDEIL: The PL1L program development environment principles of operation. Res. Rept RC8513, IBM, Thomas J. Watson Research Center, Yorktown Heights, NY, Nov. 1980.
20. Pine, 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.

Experiences with the PSG - Programming System Generator

G. Snelting*

Institut für praktische Informatik
Technische Hochschule Darmstadt

Abstract The programming system generator developed at the Technical University of Darmstadt generates sophisticated interactive programming environments from formal language definitions. From a formal, entirely nonprocedural definition of the language's syntax, context conditions and denotational semantics, it produces a hybrid editor, an interpreter and a library system. The editor allows both structure editing and text editing, guaranteeing immediate recognition of syntax and semantic errors. The generator has been used to generate environments for PASCAL, MODULA-2 and the formal language definition language itself. A brief description of the generated environments and the definition language is given, and our experiences with formal language definitions are discussed from the language definer's point of view as well as from the programmer's point of view using the generated environments.

1. Introduction

The Programming System Generator PSG developed at the Technical University of Darmstadt generates language-dependent interactive programming environments from formal language definitions. From a formal definition of a language's syntax, context conditions, denotational semantics and additional information it produces an integrated software development environment. One of the major components of a PSG environment is a powerful hybrid editor which allows structure oriented editing as well as text editing. In structure mode, the editor guarantees prevention of both, syntactic and semantic errors, whereas in textual mode it guarantees their immediate recognition. The editor is generated from the language's syntax and context conditions. Furthermore, a PSG environment includes an interpreter which is generated from the language's denotational semantics. A language-independent library system is part of a PSG environment.

The basic units for editing and interpreting are called fragments. A fragment is an arbitrary part of a program, for example a statement, a procedure declaration or a whole program. Fragments are internally stored as abstract syntax trees. Fragments may be incomplete, that is, subcomponents may be missing. Missing subcomponents are called templates. Bottom-up system development is provided by combining fragments, while the fragments themselves are constructed top-down.

The editor supports two input modes, which may be mixed freely by the user. In textual mode, the editor behaves like a normal screen-oriented text editor with the usual capabilities to enter, modify, delete, search etc. text. By keystroke, incremental syntactic and semantic analysis are invoked. If the input was error-free, the text will be pretty-printed and editing may proceed. If any syntactic or semantic errors are detected, an error message will be displayed by a menu-driven error recovery routine. Earliest possible detection of both syntactic and semantic errors is guaranteed: As soon as a fragment cannot be embedded into a syntactically and semantically correct program, it will be classified as erroneous. For semantic errors, this works even if declarations of e.g. variable types are still missing.

In structured mode, programs are developed in menu-driven refinement or modification steps. The menus are generated according to the abstract syntax of the language. The usual structure oriented commands are offered to the user, such as refinement of a structure, selection from alternatives of a syntactic class, modification, insertion, and deletion of substructures, zooming of substructures, copying of substructures etc. However, the menus are filtered dynamically by the context analysis, such that only those menu-items producing syntactically and semantically correct refinements after selection will be offered to the user. Thus, in structural input mode, neither syntactic nor semantic errors can occur. In addition the user may retreive the context information which has been derived so far. For example, he might ask the system which variables are already declared, which variables are still undeclared, what possible types the undeclared variables may possess etc*.

Like the other system components, the interpreter is able to handle arbitrary incomplete fragments. As long as control flow in the interpreted fragment does not touch any syntactically incomplete structure, the fragment can be interpreted without difficulties. If flow of control encounters a template,

* Work of this author was supported by the "Deutsche Forschungsgemein-

* According to our philosophy, declaration before use is not required. An undeclared variable is considered a semantic error as soon as the last template offering the possibility of declaring that variable has been deleted

the editor will be invoked asking the user to enter the missing parts of the fragment. Alternatively, the language definer may force the interpreter to ask the user for e.g. values of uninitialized variables or missing expressions.
A language-independent fragment library sytem where fragments are stored as abstract syntax trees is also part of a generated environment. Reading, writing and rewriting of fragments is automatically performed by the editor if required. Deletion of fragments requires an explicit user command. PSG environments offer the facility of redirecting input to external text files. Furthermore, fragments may be written in pretty-printed style onto external files.

2. What the language definer has to do

One of the most important goals during the development of PSG has been the definition of a formal language definition language covering the whole spectrum of a language's syntax, context conditions and dynamic semantics as well as all of the additional information required by an interactive environment e.g. menu texts or pretty-printing information. Thus, the language definer working with PSG is offered a formal, nonprocedural definition language. This is in striking contrast to most existing environment generators, which frequently support only the formal definition of the syntactic aspects of a language. For example, the language definer working with GANDALF [Hab82a] has to write so-called action routines in an ordinary programming language; these action routines will perform tasks such as type checking, code generation etc. Using the Cornell Program Synthesizer (CPS) [Rep81], which is based on attributed grammars, the language definer has to code certain attribute functions in the language C.
A PSG language definition consists of three major parts: the definition of the syntax, the context conditions, and the denotational semantics of the language. The first part is mandatory, the others are optional. Syntax and semantics definition rely on well-known concepts. However, new concepts based on AI technology had to be developed for defining and checking context conditions, due to the specific requirements of interactive environments where programs are usually incomplete containing e.g. pending variable declarations.

Definition of the syntax

The syntax definition part starts with the definition of the lexical structure of the language, which is used to generate a scanner. The language definer has to specify all reserved words and all delimiters (special symbols). Each lexical entity is given a name. For PASCAL, this looks as follows*:

```
if -> 'IF';
then -> 'THEN';
else -> 'ELSE';
becomes ->' := ';
equal ->' = ';
sem ->';';
```

etc.
The abstract syntax, which forms the second part of the syntax definition, is the core of any language definition. All other parts of a language definition refer to the abstract syntax. Abstract syntax rules look like this:

```
CLASS statement = assignment, forstatement, compound, ifstatement, call, ... ;
NODE assignment :: variable expression;
NODE forstatement :: Id expression to_or_downto expression statement;
NODE compound :: statementlist;
LIST statementlist = statement+;
NODE ifstatement :: expression statement [statement];
NODE call :: Id [parameterlist];
LIST parameterlist = expression+;
CLASS variable = Id, record_ref, array_ref, pointer_ref;
CLASS expression = variable, constant, addition, subtraction, ... ;
NODE addition :: expression expression;
```

etc.
CLASS rules describe syntactic alternatives. NODE rules define substructures of a syntactic entity. Substructures which are optional are enclosed in square brackets. The number of a node's substructures is fixed, although they may be of different syntactic type. LIST rules define syntactic entities with a variable number of substructures of the same syntactic type.
In a PSG environment, fragments are internally represented by abstract syntax trees. Missing substructures of a node are represented by tree templates; they serve as placeholders for pending refinements. Missing sublists of a list are called list templates, they may be moved, deleted and inserted freely within a list.

* In the following, all examples refer to PASCAL

immediately that 'i' has type integer (or a subrange thereof), that 'a' is a one-dimensional array with index and component type integer, and that the still missing right-hand side of the assignment must also be compatible with integer. If a user types 'TRUE' as the right side, a semantic error must immediately be reported. In addition, the menu for the right-hand side template should be filtered in such a way that the menu item for the constant 'TRUE' will not be displayed, as well as all other non-integer expression items.

The classical methods follow the scheme: first inspect the declarations and collect information about e.g. types of variables, then use this information to check type incompatibilities in expressions etc. This scheme does not work in the above example.

The concept of context relations [Hen84] has been developed to overcome these difficulties with the classical methods. The basic idea is to compute a set of still possible attributes for each node of an incomplete fragment. A collection of still possible attribute assignments to the nodes of a fragment is called a context relation. If such a relation consists of exactly one tuple, the context information is unambiguous. If a relation is empty, a semantic error has been detected. It can be shown that the context relation of a composite fragment is just the natural join of the relations of its subfragments. Therefore context conditions may be computed incrementally during editing. As context relations are in general of infinite size, they are represented in a finite way using so-called term form relations with variables. The basic idea is to describe the set of possible attributes by a grammar, the so-called data attribute grammar. Infinite sets of attributes are then represented by incomplete derivation trees according to the data attribute grammar; in addition these derivation trees may contain arbitrary functional dependencies between (sub)trees.

To specify context conditions, the language definer first has to define the scope and visibility rules of the language. This information is used to determine whether all the different occurences of an identifier in a fragment actually denote the same "abstract" identifier. If so, their corresponding sets of still possible attribute values may be intersected.

The second part of the context conditions definition is the specification of the data attribute grammar. Here, the structure of the attributes of the language is defined. Typical rules look like this:

```
NODE attribute :: type class;
CLASS type = simple_type, array_type, set_type, ... ;
CLASS simple_type = arithmetic, ordinal;
CLASS ordinal = integer, boolean, subrange, enumeration, ... ;
NODE settype :: ordinal;
NODE arraytype :: index_types type;
LIST index_types = ordinal+;
CLASS class = variable, ctype, constant, procedure, function, ... ;
```

-etc.

The attribute format definition forms the third part of the context conditions definition. Similar to the format definition of the context-free part of the language definition, it specifies how attributes shall be displayed to the user if he looks at the symbol table.

The last and most important part of the context condition definition is the specification of the so-called basic relations, which must be specified for all terminals and each node rule of the abstract syntax. As the context relation of a fragment is the join of the relations of its components, specification of the basic relations provides enough information to analyse each fragment incrementally. A basic relation consists of a set of tuples which define a (possibly infinite) set of attribute assignments to the components of a node rule resp. a terminal. For instance, the basic relation of a syntactic integer number consisting of a single tuple might be:

```
Int: MK-attribute(integer, constant);
```

which specifies that an integer number has type integer and is a constant. More sophisticated specifications can be obtained by using variables, which specify that certain subattributes must be identical. The basic relation for an assignment

```
assignment :: variable expression
```

contains three tupels, which use the variable TYPE:

```
assignment: NIL MK-attribute(TYPE, variable) MK-attribute(TYPE, computational)
          | NIL MK-attribute(real, Variable) MK-attribute(integer, computational)
          | NIL MK-attribute(TYPE, function) MK-attribute(TYPE, computational);
```

which says that in an assignment either

- the left-hand side is a variable of a certain TYPE, and the right-hand side is an expression of the same TYPE, or
- the left-hand side is a real variable, and the right-hand side is an integer expression, or
- the left-hand side is a function identifier with a certain result TYPE,

The structure oriented commands and menus offered to the user are generated according to the abstract syntax. For example, each template is associated with a menu of refinement possibilities. However, this menu is dynamically filtered with respect to context conditions (see below).
The concrete syntax, which is the third part of the syntax definition, is used to generate an incremental parser. The concrete syntax is restricted to full LL(1) grammars. It includes transformation rules which specify how to build abstract trees from textual input. Thus the concrete syntax is actually a string-to-tree transformation grammar. Concrete syntax rules look like this:

```
statement ::= ...
            | NODE for, Id, becomes, expression, to_or_downto, expression,
                  do, statement => forstatement
            | NODE begin, statementlist, end => compound
            | NODE Id, optparameterlist => call;
statementlist ::= LIST statement+-sem;
optparameterlist ::= [lp,parameterlist,rp];
to_or_downto ::= TERMINAL to | TERMINAL downto;
```

etc.
The NODE, LIST and TERMINAL keywords and the '[', ']', and '=>' delimiters specify how to build the abstract tree during the parsing process. However, the situation is not always that simple. Frequently, a concrete syntax does not merely reflect the rules of the abstract syntax, due to operator precedences or left-factorization used to avoid LL(1)-conflicts. For example,

```
expression ::= simple_expression, simplexpr_tail;
simple_expression ::= factor, ...;
simpleexpr_tail ::= UPDATENODE equal, simple_expression => equal_expr
                  | EMPTY;
```

Here, the UPDATENODE and EMPTY rules will construct a correct equal_expr node, although the rules reflect operator precedence and are left-factorized.
The parser will parse any input entered in textual mode. It accepts arbitrary valid prefixes of any input conforming to the syntactical category of a given template. If any syntax errors are detected, a recovery routine will compute a menu comprising all local correction possibilities, which is presented to the user. The user may then correct his input either in textual mode or by selection among the menu items.

Being the fourth part of the syntax definition, the format definition is a tree-to-string transformation grammar which is used to construct the external textual representation of an abstract tree. Prettyprinting information is part of the format definition:

```
forstatement => ! for Id becomes expression to expression do statement [2];
ifstatement => ! if expression then statement[2] (statement[2] -> ! else,);
```

In the example, '!' means start of a new line, and indentation factors may be specified inside square brackets. Parentheses are used to specify conditional formatting: the keyword 'ELSE' will be displayed only if the optional else-part of an 'ifstatement' is indeed present. Conditional formatting is used also to re-insert parentheses into expressions if neccessary due to operator precedence (note that parentheses are discarded during parsing and that operator precedences are reflected by the abstract tree's structure). A string-to-tree-to-string transformation which is performed by parsing textual input, building the abstract tree and pretty-printing the abstract tree must yield the original input text exactly except for spaces, newlines and redundant parentheses.
In the last part of the syntax definition, headers and menu texts have to be specified which are used to generate the textual representation of templates and menus. For each name occuring in the abstract syntax an external name has to be specified:

```
statement -> 'Anweisung';
ifstatement ->'Bedingte Anweisung';
```

For each syntactic class, menu texts have to be specified:

```
statement -> 'Zuweisung', 'FOR-Anweisung', 'Verbundanweisung', ... ;
```

For purposes of generality, syntactic entities may posses different external names, depending on their occurence in templates or in menus.

The definition of context conditions

The context analysis of PSG has been of special interest, since the classical methods like attributed grammars [Knu68] turned out to be inadequate even if attribute evaluation is performed incrementally [Rep83]. Consider the following situation: In a PASCAL program-fragment, the variables 'a' and 'i' have not yet been declared or used, and a declaration-template is still present. Now the user enters an incomplete assignment:

```
a[a[i+1]]:=
```

Although 'a' and 'i' are still undeclared, the context analysis must derive

and the right-hand side is an expression of the same TYPE.*
During editing, an inference engine is used to derive context information from the basic relations as demonstrated in the above example. Note the similarity to the AI-paradigms of inference-rule-based deduction systems.

The definition of semantics

Within the PSG system, the dynamic semantics of a language is defined in denotational style [Gor79]. The denotational semantics is used to generate an interpreter. The semantic functions are defined in a META-IV-like [Bjo78] extension of type-free lambda calculus. This metalanguage supports high-level concepts like lists and maps and allows the definition of higher-order-functionals of arbitrary rank. The terms of the metalanguage are used as an universal intermediate language. If a fragment is to be executed, it will be translated into a term of the metalanguage, using the definitions of the semantic functions. This term will be interpreted, that is, reduced to normal form. The resulting term is the result of program execution.
In contrast to systems like SIS [Mos79] our interpreter allows interaction with the user during program execution in order to supply input data, to enter values of uninitialised variables etc.
The definition of the semantics consists of three parts. First of all, a set of auxiliary functions to be used elsewhere in the semantics definition may be defined. For example, the definition of a "distributed concatenation" function for a list of lists (which is supposed to be used in several distinct semantic functions for different types of lists) looks as follows:

```
disconc = LAM list_of_lists. IF NULL list_of_lists THEN <>
                  ELSE CONC HEAD list_of_lists, (disconc TAIL list_of_lists);
```

Here, LAM denotes functional abstraction, parentheses denote functional application. NULL is a test for the empty list, CONC, HEAD and TAIL have their usual meanings, and '<>' denotes the empty list.
The main part of the semantics definition comprises the semantic functions for each syntactic entity. In a PASCAL-subset without GOTOs and side effects of functions, the meaning of a statement may be defined as a functional which maps environments onto functions which map states to states. The meaning of an expression is a functional which maps environments onto functions from states to values. An environment is a map which maps identifiers to <location, descriptor> pairs. A state is a map which maps locations to values. Thus, the semantic function for a conditional statement might look as follows:

```
ifstatement: LAM env. LAM state. IF (( |[ expression ]| env) state)
                  THEN (( |[ statement 1 ]| env) state)
                  ELSE (( |statement 2:LAM env.LAM state.state| env) state);
```

The '|[' and ']|' brackets are the "meta-brackets" which denote the meaning functions of the subcomponents of a node. The special form '|' .. '|' is used for subcomponents which are optional (as the ELSE-part in our example). If the optional subcomponent is missing, the function following the colon will be used.
The third part of the semantics definition describes the meanings of the executable fragments. Typical examples are

```
procedure_declaration: '', ERROR 'Procedure declaration is not executable';
statement: 'Result of statement execution with no variables declared
                  or initialized:', (( |[ statement ]| []) []);
```

where '[]' denotes the empty map. Note the difference between the result of a 'statement' execution specified here and the semantic function for the syntactic class 'statement', to which the above definition refers.

3. Experiences with the generator and the generated environments

Until now, environments have been generated for Algol60, PASCAL, MODULA-2, the language definition language itself, and some experimental specification languages. The language definition environment has been used intensively not only by the members of the project team, but also by lots of students, as most of our environments have been generated as part of diploma theses. At Kaiserslautern, PSG has been used along with other systems (GAG [Kas80] and GANDALF) in student projects for the implementation of a PASCAL-subset. The PASCAL-environment was used to implement other parts of the PSG system. Thus, we feel that by now we have gathered substantial experience and that we are able to compare our approach to others, especially GANDALF and CPS.

The benefits of a formal language definition language

In [Hab84], Habermann states that "the state of the art has not reached the point where all of the task-specific (i.e. language specific) part (of an environment) can be formally described and automatically generated". However, our experience with PSG indicates that this point has been reached by now, at least for languages of a complexity not greater than that of e.g.

* For the sake of readability, this specification does not exactly reflect the ISO standard "assignment compatibility"

PASCAL.

The use of a formal language definition language has many advantages:

- In view of the power and complexity of the generated environments, PSG language definitions are very short. Typically, they vary in size between 240 lines for an Algol60 environment without context conditions and semantics and 3600 lines for a MODULA-2 environment including full specification of context conditions and denotational semantics.
- The expressive power of the language definition language allows concentration on the relevant aspects of a language definition. The language definer does not have to concern himself with minor details such as the organization of symbol tables etc.
- PSG language definitions are safe, since all inconsistencies in a definition are detected at generation time.*
- The modular design of the language definition language improves readability and reliability. It allows the independent definition of the syntactic, context dependent, and semantic aspects of a language, once the abstract syntax has been defined.
- a formal language definition language is an ideal tool during the development of new languages. In a "language design lab", language definitions are easily modified and tested.

As a consequence, the amount of manpower to generate an environment is small: A moderately awake graduate student with some background in programming languages and some initial knowlegde of the PSG user interface will specify and debug an Algol60 definition without context conditions and semantics within ten days. The MODULA-2 environment including full specification of context conditions and denotational semantics was defined as part of a diploma thesis within eight months. Thus, the use of a formal language definition language allows the quick generation of correct, reliable and powerful programming environments.

The benefits of the hybrid editor approach

In [Fei84], Kaiser and Feiler state for structure oriented editors that "in order to modify an expression, ... , the user must understand the underlying tree representation and enter a tedious serious of tree oriented clip, delete and insert commands. Unfortunately, complete parsing of all expressions is also nonoptimal". This is true not only for expressions, but also for arbitrary structured statements as well as for any syntactic entity including complete programs. In [Rep81], Teitelbaum and Reps state that "(the change of a while loop into a repeat loop) 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 ..., they are admittedly awkward". Within a PSG environment, problems of this kind do not exist, since users may switch freely between textual mode and structure mode. Furthermore, our experience indicates that experienced programmers prefer textual mode not only for modifications, but also to enter e.g. a sequence of statements or even a whole procedure. Since the parser accepts arbitrary incomplete input and, in case of syntax errors, generates a menu of all possible local recovery actions, textual input mode seems to be quite attractive for users who know the concrete syntax of their language. Furthermore, arbitrary parts of a fragment may be read in from an external textfile. On the other hand, unexperienced users tend to prefer structured mode. By simply selecting menu items, they need not bother about syntactic details which they do not know. Thus, the possibility to mix textual mode and structure mode freely seems to be the most flexible, general, and user friendly solution to the dichotomy of viewing programs either as text or as structure.

The benefits of dynamic context sensitive menu filtering

In [Hab82b] Habermann states that "we believe that preventing mistakes is far superior to making the user fix them. ... (however) as to semantic errors it is difficult to see how to avoid them". Within a PSG environment, structured mode prevents syntactical and semantical errors due to the dynamic context-sensitive menu filtering. This feature is not provided by any other environment known to us. In textual mode, the user may always type arbitrary nonsense, but syntactical and semantical errors will be detected immediately. This guarantees that programs are correct at every stage of their development.

We believe that our environments support syntactic and semantic error prevention in the best possible way. There is, however, one problem in connection with certain modifications: if a user modifies e.g. a procedure declaration by adding an extra parameter, context incompatibilities will occur at each place where the procedure is called. If the calls are modified first, they will become incompatible with the procedure declaration. Although it might be considered bad programming style to modify the types of objects in an uncontrolled manner, the user can circumvent such situations

* At the moment, this is not true for the semantics definition, as it is based on type free lambda calculus. However, the implementation of a type inference algorithm allowing handling of polymorphism, overloading and coercions is about to be completed (see [Let84]).

by temporarily deactivating the context analysis. It is planned to modify the context analysis in a way that enables it to tolerate faulty subtrees temporarely.

Drawbacks in generality and performance

PSG is not the ultimate system, as there remain several unsatisfying points. A formal language definition language enables language definers to generate environments in a rapid and reliable way. However, the current implementation of the definition language imposes some restrictions on the class of languages which may be defined with PSG.

First of all, if the concrete syntax of a language cannot be made LL(1), the language cannot be defined within PSG. It is, however, difficult to see how to incorporate a more powerful parsing technique. Bottom-up techniques such as incremental LR parsing ([Cel78]) do not fit in our framework. LL(k) with $k > 1$ is problematic in view of the requirement that arbitrary valid prefixes of sentential forms must be parseable.

Certain languages have context conditions which are not definable within the current definition language. The scope and visibility analysis cannot handle features like elliptical record references in PL/1 or FORWARD procedure declarations in PASCAL (which will lead to a 'double declaration' error). Within our framework - no declarations required before use - FORWARD declarations do not make sense anyway. The context analysis phase is unable to handle user-defined polymorphic or overloaded objects such as overloaded functions in ADA. We are currently working on a more powerful specification language for context conditions which will overcome these shortcomings. Finally, the semantics definition language is unable to handle any form of parallelism.

The performance of PSG environments has not yet reached production quality, as far as context analysis and program execution are concerned. For the context analysis, this is primarily a problem of the current implementation, which is merely a prototype. We expect that a more sophisticated implementation will result in a time speedup factor of at least five. However, the intrinsic complexity of the method is greater than that of e.g. attributed grammars: For an abstract syntax tree containing n nodes the Reps/Teitelbaum algorithm will perform with $O(n)$, whereas our method requires $O(n*ln(n))$.

The performance difficulties concerning program execution are of a slightly different nature, as we have difficulties to see how to speed up the interpreter simply by improving its implementation. The interpreter is much faster than that of SIS. However, it is not fast enough for production programs, as is also noted by Pleban for PSP ([Ple84]). We think that these shortcomings may be overcome by compilation of the metalanguage terms [Bah84b], utilizing techniques like data flow analysis, elimination of unneccesary call-by-name and delayed evaluation, and elimination of tail recursion and linear recursion.

4. Conclusion

We presented the PSG programming system generator, which generates powerful interactive programming environments from formal language definitions. The pros and cons of using a formal, entirely nonprocedural language definition language have been discussed. It turned out that use of a formal definition language allows very simple and rapid generation of reliable and powerful environments. On the other hand, certain strange and complicated features of certain languages are not definable with the currently implemented definition language, and the performance of the generated environments has not yet reached production quality. Nevertheless, we believe that the use of formal language definitions is an appropriate tool, and that the shortcomings in performance will be captured by more sophisticated implementations, which are still under way.

5. Acknowlegdements

I thank the other members of the project team, namely R. Bahlke, W. Henhapl, M. Hunkel, H. Jäger and T. Letschert for their valuable comments during the development of this paper.

6. References

[Bah84a] Bahlke, R. and Snelting, G.: Programmiersystemgenerator. Arbeitsbericht 1984. Bericht PU2R2/84, Fachgebiet Programmiersprachen und Übersetzer II, Technische Hochschule Darmstadt, Februar 1984.

[Bah84b] Bahlke, R. and Letschert, T.: Ausführbare denotationale Semantik. Proc. 4. GI-Fachgespräch Implementierung von Programmiersprachen, Zürich, März 1984.

[Bjo78] Bjørner, D. and Jones, C.B. (eds.): The Vienna Development Method: The metalanguage. LNCS 61, Springer Verlag 1978.

[Cel78] Celentano, A.: Incremental LR parsers. Acta Informatica 10 (1978), 307-321.

[Fei84] Kaiser, G.E. and Feiler, P.: Generation of language-oriented editors. Proc. Programmierungsübungen und Compiler, Berichte des German Chapter