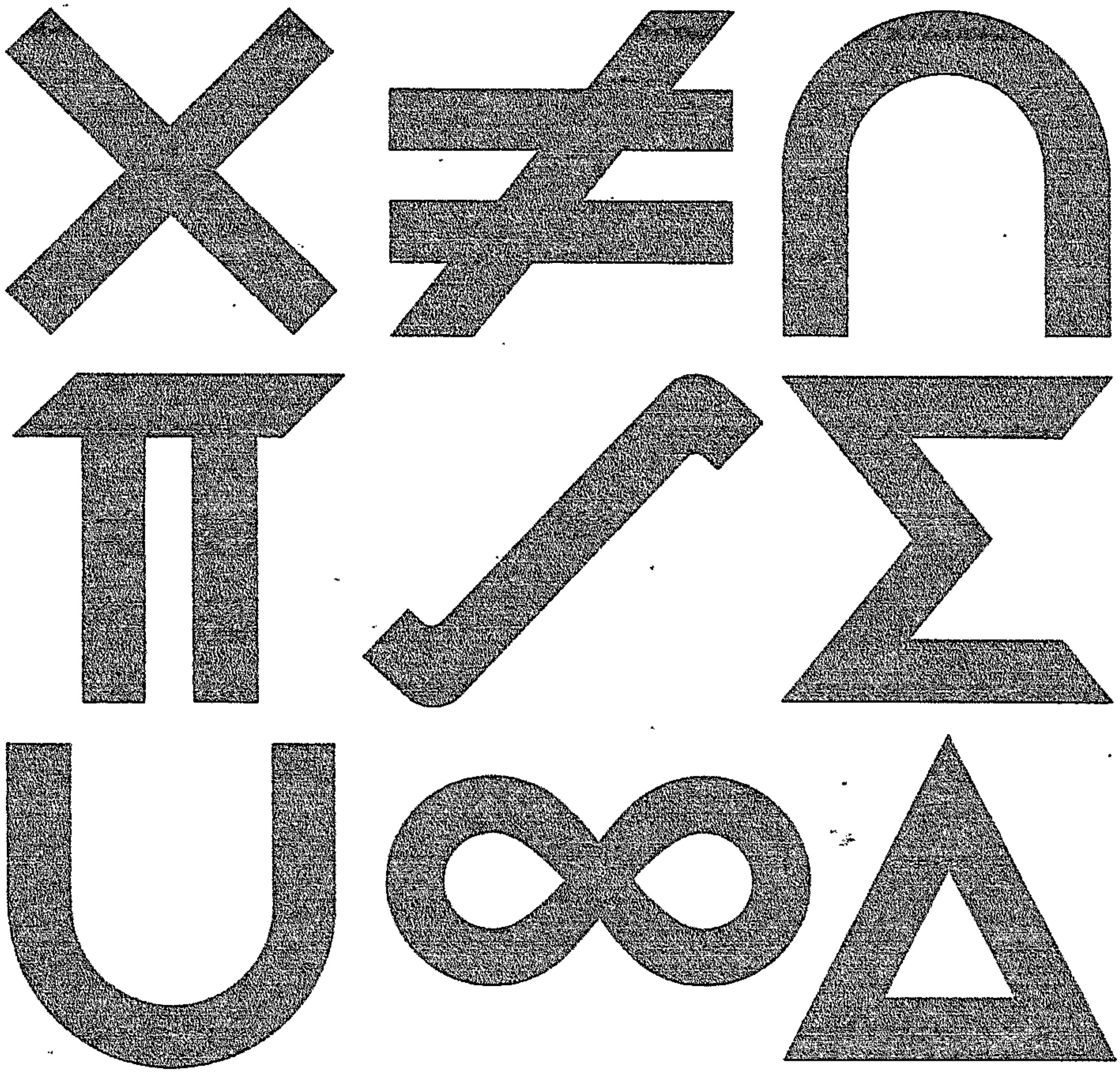
G320-2739, October 1984

MODULE AND APPLICATION GENERATION VIA DECLARATIVE MODULES

P. S. NEWMAN



IBM Los Angeles Scientific Center Report No. G320-2739

October 1984

Module and Application Generation via Declarative Modules

Paula S. Newman

IBM Corporation
Los Angeles Scientific Center
11601 Wilshire Boulevard
Los Angeles, California 90025

Abstract

This paper discusses a method for allowing declarative specification of some application components, such as screen input/output, together with procedural specification of others, with less disjointedness than is usually the case with complete or partial application generators. The method involves the use of programming units called "declarative modules", containing linear-form data collections with generator-specific schemas. Equivalent data collections are passed by the compiler to generators, which return source program units in a base language.

1. Introduction

The term "application generator" has been used to connote many different kinds of processors. In this paper use of the term is restricted to processors which require a relatively small, primarily declarative specification to produce all or part of an application. It does not include vehicles sometimes called "fourth generation languages" which generally include substantial procedural capabilities.

The application generators of concern here usually fall into one of two classes: those intended to produce complete or almost complete applications, and those intended to produce specific application components, most frequently screen management components.

While both types of generators are extremely helpful, improvements are needed in the area of their relationship to programming languages. The larger generators usually make provision for invoking procedural code (via "exits"), and the smaller generators are intended for use with such code. Yet the connections between generated and programmed application components are not smooth, and represent one source of the fragmentation which plagues programming environments.

Two kinds of discontinuities are involved, relating to specification and communication respectively. The specification problem lies in the fact that generator inputs must be specified and stored separately from associated procedural code. This affects application comprehensibility. One cannot, for example, examine a screen specification together with the functions which process the data obtained from the screen.

The communication problem is one of interfaces; interfaces between generated and procedural code are designed to be both language and application independent, and, consequently, are at a rather low level. Currently this is a problem of usability. However, in the context of newer, strongly-typed languages such as ADA™[1] (DOD 1983), such interfaces are not only inconvenient, but are also incompatible with the intent of those languages.

There have been some efforts to solve these problems, particularly for screen specification. Dialogue managers, such as the ISPF Dialogue Facility (1983), are explicitly designed to control interactions between generated screens and user-coded functions, thus allowing reasonable interaction between coded and generated components. However, they force control flow to be expressed in the rather stylized form required by the screen manager, and do not allow merging of generator input with procedural code.

Another approach in this area is that of language extensions for screens. LaFuente and Gries (1978) define a set of extensions to PASCAL for screen definition, and both Rowe and Schoens (1983), and Horowitz and Kemper

[1] ADA is a registered trade mark of the U. S. Government (Ada Joint Program Office).

(1984), elaborate on this proposal in the context of extensions to other languages.

The latter approach does not suffer from the control or source proximity problems posed by dialogue managers. However, the advisability of extending popular general-purpose languages in this way is questionable. It requires the selection of one kind of specification, for one kind of device interaction, in a era of rapid change in interactive i/o devices.

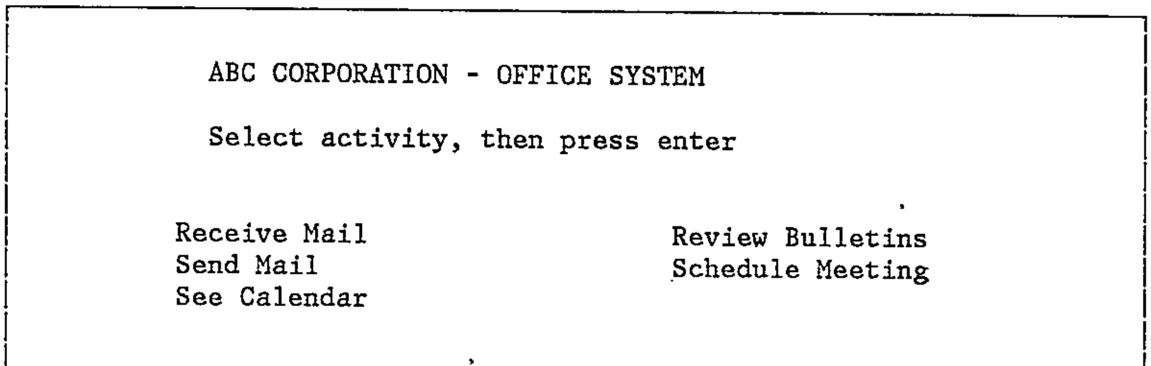
A more general approach to application generation, suggested by Ruiz-Herrera (1983), is the use of compilers designed for extensibility, i.e., designed to allow the addition of special-purpose syntax by users. A parser generator is assumed in the proposal, and the compiler extension procedure consists of extending the syntax of the language, and adding code-generation functions corresponding to the new parse-tree node types. One drawback of the approach is that its use is probably limited to professionals trained in compilation techniques; it requires, for example, the ability to deal with the problems of obtaining non-ambiguous grammars. Another difficulty of the approach is that it renders programs non-portable, without sufficiently isolating the non-portable aspects.

In this report, a technique is proposed which is more general than screen-oriented language extension, but does not have the problems associated with ad-hoc compiler extension. Special programming constructs, called "declarative modules" are used as containers for generator inputs of varying types. A declarative module is delimited similarly to other programming units in some chosen base language but, in some way, identifies a generator in its header. The primary content of the module is a linear form data collection, whose schema is generator-specific. When the compiler of the base language finds such a module, it loads a shareable repository with the specified content and makes it available to the specified generator. The generator accesses the data base to produce a module in the base language.

Building or changing a generator using this approach requires only application programming skills, namely knowledge of the base language and a data manipulation language, rather than compiler development skills. The approach also compares favorably with compiler extension in terms of portability. Not only are those sections of the code dependent on generators clearly delimited, but also the generators, written in the base language, and generating code in the base language, are themselves portable.

In the body of the paper, a concrete adaptation of the declarative module concept is developed, using ADA as the base language, and the relational model for the linear-form data collections. To begin, the use of external data collections for generator input is illustrated, by example. Then a linear form for relational data is specified, and used as the core of a declarative module, which can be nested within a larger program. Declarative modules are then extended to allow the embedding of procedural code.

After introducing the basic approach, several applications are suggested. The final section of the paper reformulates the approach as a special case of embedded language definition, with the latter being as general as compiler extension.



```
              ABC CORPORATION - OFFICE SYSTEM

              Select activity, then press enter


           Receive Mail                     Review Bulletins
           Send Mail                        Schedule Meeting
           See Calendar
```

Figure 1. Initial Screen

2. Data Collections for Application Specification

While the main focus of the paper is on generator inputs within source programs, for expository purposes it is convenient to begin by considering the use of external data collections for generator input.

Consider an application using a series of screens, the first of which is shown in Figure 1. This screen might be described to a screen generator using, for example, a data base containing a single relation, "Screens", as shown in Figure 2.

This input presumes a generator capable of designing detailed screen arrangements, given relatively sketchy specifications (as suggested by Rowe and Schoens (1983)). Thus the input indicates only that the screen, called "Act," is to contain two centered strings, followed by a menu, arranged in two columns.[2]

The relation might be loaded using standard data base access facilities or, to allow interactive screen design, via a generator-specific utility. After loading, the generator would be invoked to a file of programs in a chosen base language which can be used to produce the desired screen interaction.

The generator output file might contain one function in the base language for every screen described. The screen-associated function would be responsible for the display of that screen, and might return a record structure containing one field for each value to be obtained from the application user via that screen.

[2] The purpose of this report is to discuss an approach to application generation rather than a specific generator. Therefore the functions of the generator used in the examples are not described in great detail.

Screens	Screen	Seq	Type	Value
	'Act'	1	'justify'	'center'
	'Act'	2	'const'	'ABC CORPORATION - OFFICE SYSTEM'
	'Act'	3	'const'	'Select activity, then press enter'
	'Act'	4	'menugrp'	'Activity'
	'Act'	4	'cols'	'2'
	'Act'	4	'label'	'Null'
	'Act'	5	'sel'	'Receive Mail'
	'Act'	6	'sel'	'Send Mail'
	'Act'	7	'sel'	'See Calendar'
	'Act'	8	'sel'	'Review Bulletins'
	'Act'	9	'sel'	'Schedule Meeting'
	'Act'	10	'endgrp'	'Activity'

Figure 2. Screen Description Data Base

While almost any language might be used as a base, one having data abstraction capabilities is preferable. This choice allows the generated code to be seen not as a collection of separate programs, but, rather, as one encapsulated programming unit providing many operations. This is especially useful where the generated code must include definitions of shared data types and/or data.

For example, if the selected base language were ADA, the screen generator assumed above might produce a package specification[3] and package body providing one function for each screen. The package specification produced might be as shown in Figure 3. Note that the package contains not only the declaration of the function produced by the generator, but also the declaration of the record type used to return the results of the interaction.

The generated function might be used in a statement such as:

```
case Office.Act().Activity
   when 1 ==> .... ;
   when 2 ==> .... ;
   ....
   end case;
```

which decides on subsequent processing based on the "Activity" field of the record returned by Office.Act.

[3] An ADA package specification describes the externally visible aspects of the package, e.g., type definitions and definitions of interfaces to externally accessible programming units contained in the package. The package body contains code for the latter, plus additional type definitions and programming units not accessible from the outside.

```
package Office is

   type Act_Record is
        record
          Activity: Integer 1..5
        end record;

   function Act returns Act_Record

   end Office;
```

Figure 3. Package Specification Generated from Office Data Base

The important point here is the smooth interface to the generated code. It is (a) simple to use, (b) tailored to the needs of the application, and (c) at the level of the base language. Moreover, it does not subvert the typing constraints of the base language.

The body of the generated package might actually be very short, containing only some brief procedures communicating with larger, "canned" programs in a library associated with the generator. The latter communication might in fact involve awkward, general purpose interfaces, but the more complex interfaces would be used by the generated code, not by the application programmer.

3. Declarative Modules

The use of a data base for generator input is acceptable for the generation of complete applications, or for large parts of applications. However, for the generation of small components, such as individual screens, it is important to allow generator input to be embedded within procedural source code.

"Declarative modules" are proposed for this purpose. Declarative modules consist of data collections expressed in linear form, and surrounded by program-unit boundaries consistent with those of the chosen base language.

To process declarative modules, compilers must be extended to parse the linear form data collections, and to pass their content to generators, which return program units in the base language for subsequent compilation.

(Note that some generators might themselves return declarative modules as output. This would be the case if generators were tailored and specialized by layering. For example, a screen generator with simple input requirements but constrained capabilities might be layered on more general facilities requiring more complex inputs.)

```
A. Non-Factored Representation

Screens  ['Act'  1   'just'    'center'],
         ['Act'  2   'const'  'ABC CORPORATION - OFFICE SYSTEM'],
         ['Act'  3   'const'  'Select activity, then press enter'],
         ['Act'  4   'menugrp''Activity'],
         ['Act'  4   'cols'   '2'       ],
          ......

B. Factored Representation

Screens ['Act'  [1   'just'    'center'],
                [2   'const'  'ABC CORPORATION - OFFICE SYSTEM'],
                [3   'const'  'Select activity, then press enter'],
                [4   ['menugrp''Activity' ], ['cols' '2' ]],
                .......
```

Figure 4.   Linear Form Representations of Office Data Base

The precise syntax chosen for declarative modules depends, in part, both on the data model used for the generator inputs, and on the language. Some examples of a possible syntax, given the continued assumption of the relational model and ADA, are given in this section. The next section deals with some processing questions.

To express a collection of relations in linear form, each relation can be specified as a list of bracketed row specifications, as shown in Figure 4. Part A of the figure is a full linear representation of a relation. Part B is an alternative representation which is factored on duplicate left-most attribute values.[4]

To surround the data representation by a programming unit boundary, assuming ADA as a base language, an alternative form of package specification can be used, shown in Figure 5. The header keyword "For" identifies the presence of a declarative module to the compiler. The use of a package variant seems appropriate as (a) packages are the standard ADA method of

[4] The specification of data base content in linear form is not new. The best known example is probably PSL (Teichroew et.al., 1974), in which a linear form is used for the specification of the content of a data base containing application requirements. In the PSL case, the linear form requires a phrase-specific mapping to the underlying data relationships. More regular methods, in which the linear form is a direct expression of the desired data base content, are given by Falkenburg (1976), and by Newman (1980), in conjunction with data-base-oriented associative networks. In all these cases, the emphasis is on using the linear form as a means of expressing data content in a concise and readable way.

```
Package Office For GenScreens is

Database
Screens   ['Act' [1    'just'    'center'],
                 [2    'const'  'ABC CORPORATION - OFFICE SYSTEM'],
                 [3    'const'  'Select Activity, then press enter'],
                 [4   ['menugrp''Activity'], ['cols' '2' ]],
                 [5    'sel'    'Receive Mail'     ],
                 [6    'sel'    'Send Mail'        ],
                 [7    'sel'    'See Calendar'     ],
                 [8    'sel'    'Review Bulletins'],
                 [9    'sel'    'Schedule Meeting'],
                 [10  'endgrp' 'Activity'         ]]

    End Office;
```

Figure 5.   Declarative Package

encapsulating information, and (b) it allows one form of package to be used in generating another.

```
                ABC CORPORATION - OFFICE SYSTEM

                    Automated Calendar

              Fill in information, then press enter


       Date                        Person _________________

          Day   __
          Month __                 Format
          Year  __
                                         Full Month
                                         Full Week
                                         One  Day
```

Figure 6.   Calendar Request Screen

This section is completed by another example, the screen shown in Figure 6. This screen might be obtained by the declarative package shown in Figure 7, producing the generated package specification shown in Figure 8.

```
Package Calendar For GenScreens is

 Database

Screens ['Cal' [1  'just'  'center'],
               [2   'const' 'ABC CORPORATION - OFFICE SYSTEM']  ,
               [3   'const' 'Automated Calendar'],
               [4   'const' 'Fill in information, then press enter'],
               [5  ['bgngrp'  'Info'  ], ['cols' '2']],
               [6   'fldgrp' 'Date'  ],
               [7  ['infld'  'Day'   ], ['fmt' '2D']],
               [8  ['infld'  'Month' ], ['fmt' '2D']],
               [9  ['infld'  'Year'  ], ['fmt' '2D']],
               [10  'endgrp' 'Date'         ],
               [11  'infld'  'Person'       ],
               [12  'menugrp''Format'       ],
               [13  'sel'    'Full Month'   ],
               [14  'sel'    'Full Week'    ],
               [15  'sel'    'One Day'      ],
               [16  'endgrp' 'Format'       ]]

    End Calendar
```

Figure 7. Calendar Screen Specification

4. Declarative Module Processing

As discussed above, the desired effect of the generation process is the replacement of the declarative module by a procedural module in the base language, for subsequent compilation. The general structure of the needed approach is shown below:

```
           Data                           Data
          ------> Shared Data Structure ------->
Compiler                                           Generator
          <-----  Shared Data Structure <------
          Source                          Source
```

In other words, the compiler is to parse the declarative section containing the generator input data, place the data into a shared structure, and pass control to the generator. The generator, in turn, is to return the source code for the generated module.

This outline raises many questions, such as

- What declarative module structures should be used

- What kinds of shared data structures should be used to for communication of the results

```
    Package Calendar For GenScreens

       Type Cal_Record is
            record
               Day:    Integer range 1..99
               Month:  Integer range 1..99
               Year:   Integer range 1..99
               Person: String(1..15)
               Format: Integer(1..3)
             end record;

        Function Cal Returns Cal_Record

        end Calendar;
```

Figure 8.   Generated Package for Calendar Screen

- How should transfers of control between the compiler and the generator be handled.

Specific answers are best arrived at in the context of particular programming languages and operating environments, but some guidelines can be suggested.

The general criterion for selecting declarative module implementations is the maintenance of independence between the compiler and the generators. It should be possible to add new generators at any time, without change to the compiler (and with minimal other adjustments). Thus, first of all, only one set of changes to the compiler should be necessary, adequate for processing all declarative sections. This implies that all declarative sections should be describable using the same grammar, and should be processable, to point of transferring their content to a shared data structure, by the same adjunct semantic processes.

This, in turn, implies that the data model used for the input, and its linear-form expression, be chosen to allow this single grammar and processing. The relational model works well in this regard. The grammar of the linear form can consist, for example, of a specification of a list of rows, each row consisting of a list of fields, and each field either a number or a quoted string. Semantic checking by the compiler can be limited to ensuring that all rows of the same relation have the same number of fields, and all values in the same column are of the same type, with more extensive validation left to the generator. (Alternatively, a compiler might do more detailed checking against a data definition in an accessible data-dictionary-like structure).

The choice of a data model does not determine the choice of shared data collection. For example, the use of linear form relations within the declarative modules does not require the use of a relational data base; their content can be passed via sequential files. Like the choice of data model, the choice of data collection should be chosen with compiler/generator independence in mind. Thus, for example, the use of shared structures requiring detailed, generator-specific type definitions in the compiler should be avoided.

The handling of transfers of control between the compiler and the generators is very dependent on the choices available within the operating environment. Given an adequate command (shell) language, the transfers can be handled by interactive command sequences which alternately invoke the compiler and generators specified by compiler return values. (This requires, of course, the saving of compiler intermediate data between invocations.)

5. Embedding Procedures in Declarative Modules

The above approach allows the generation of application components as long as they can be specified by purely declarative inputs. However, most application generators accept non-declarative specifications as well. For example, screen generators usually provide facilities for validating input values and for generating new values.

The procedural capabilities of the base language can be used for such computations. To allow this usage to be well structured requires that hand-coded modules be included within declarative modules, following the declarative section.

To extend declarative modules in this way requires an adjustment to the assumed division of labor between the generator and the compiler, because the generator can no longer return a complete programming unit. Instead, it can only return a partial unit, consisting of the declarations and the code generated for the declarative section, to be combined by the compiler with the nested programming units. In the ADA case the generator would return a complete package specification (i.e., of the externally visible parts of the package) together with an incomplete package body.

As an example of how this might be used, the example of Figure 7 might be augmented in the following ways.

- The definition of each input field might be accompanied by an indication of the existence of a validation procedure (e.g., having the same name as that of the field).

- The definition of the screen might be augmented by the specification of an area explicitly devoted to error messages.

- The validation procedures might be included in the declarative module following the data section. The generator would be responsible for generating calls to those procedures.

- The validation procedures would access a systematically-named record local to the generated package to obtain the values to be validated. They would return an error message if any problem existed, and a null message if not.

6. Cross-Component Referencing

This section deals with a few additional aspects of the compiler/generator relationship, related to references between coded and generated components.

The first such aspect is the possibility of providing compiler assistance to generators to help them produce references to types and variables of scopes accessible from generated modules. It is true that all information needed by a generator to produce such references could be supplied via user-coded generator inputs. However, to avoid the need for redundant user specifications, and to foster the correctness of the generated code, it is preferable for a compiler to automatically supply some global declarative information. To allow this, a representation of the base-language declarative information in the chosen data model must be formulated. Then the declarative information could be passed to generators along with the generator-specific inputs.

Then, for example, a global variable name might be used, in a generator input module, to describe the content of a field of a screen. The generator could determine details of the field (e.g., string length) from type information for the variable, and generate correct code to obtain the value of the field.

Note that the global declarative information which could be supplied to a generator in this way is limited to that available given the necessary order of compilation/generation for the language involved. This brings us to the second subject to be mentioned here - namely the additional provisions needed to allow the intermixing of compilation and generation for languages with (explicitly or implicitly) exported types and variables, such as ADA. In such cases, careful attention must be paid to the order of generation and compilation, to ensure that references to generated exported objects are not processed before information about those objects is available. In ADA, for example, this requires that generator input packages be expanded before package specifications referencing the generated packages are processed.

A final consideration in this area relates to error handling. In general, the compiler should specially flag errors it finds when compiling generated code, to assist in tracing the errors to their sources. In some cases, the errors will stem from faulty generators. In other cases, relating to cross-component referencing, the errors might stem from incorrect generator inputs not detectable by the generator.

For example, the declarative part of a generator input module might reference a non-existent embedded module, resulting in an erroneous reference in the generated code. To assist in this area, the compiler might consult the responsible generator to determine the kind of user error which might have been involved, so as to provide an appropriate error message.

7. Applications

A major advantage of general purpose declarative modules is that they are not restricted to screen specifications. Another important potential application of declarative modules is in the definition of program units representing data bases. The application is appropriate, both in the sense that data bases are generally defined declaratively, and in the sense that data base accessing usually consists of accessing a program (the DBMS) which owns an encapsulated physical representation of the data base.

The application is useful in two ways. First, it would allow data base like structures to be defined, and accessed, as local data. (Data base models are useful independently of whether the data is persistent or shared.) Second, in object-oriented systems, it would allow the specification of data base objects in a manner consistent with program objects.

Another potential application of the declarative module approach is in connection with parser generators, to reduce the logistic complexity of their use in producing compilers. Using a parser generator generally requires the following steps: (a) submitting the grammar to the generator, to obtain code which declares and initializes parse tables, and then (b) compiling that code together with the parser skeleton to produce the parsing phase of the compiler. Using the declarative module approach, the grammar would be specified in a declarative module embedded in the skeleton code, and the parsing phase would be produced in a single compilation step.

Input to the parser generator might consist of a data base containing a relation with rows representing rules of the grammar, each rule containing the attributes:

```
LHS RHS1 RHS2 ........ RHSn
```

(with suitable prefixes or other means of distinguishing terminal from non-terminal strings, and null values for rules containing less than n elements on the right hand side). The size limitation for rules is not a serious problem, as most computer languages use short rules, and longer rules can be subdivided.

The result of the generation might be two generated packages:

- A package containing both the generated parse tables, and a procedure taking a string as input, and using the parse tables to build a parse tree.

- A package representing a parse tree abstraction, usable after parse by later compiler phases.

8. Declarative Modules as a Special Case

In the introduction to this report it was mentioned that declarative modules could be seen as a special case of a more general approach to language mixing. In this section the more general approach is sketched, and declarative module processing is reformulated to conform to this view.

In general, language mixing can be accomplished by associating a named generator with each of the languages involved, with the exception of a base language. Each such generator can be specified as a sequence of phases to be invoked by the compiler of the base language. Some of the phases, such as the parser, might be common (but operate on different parse tables). The generators would produce programming units in the base language, and implement a set of predetermined inter-language communication protocols.[5]

In this framework, generators accepting declarative modules can be seen as the simplest to produce, because they use large amounts of common code. Each such generator would have the following phases:

- A common parser, using a common parse table.
- A parse tree processor, producing a set of relations. This could be implemented as a common processor, executed interpretively to allow for the varying relational output schemas.
- A generator-specific phase taking the relations as input, and producing base-language code.

Placing the declarative module approach in this framework suggests interesting combinations. For example, one might construct an attractive screen generation facility by using linear-form declarations of screens together with simple correctness rules for fields, rather than requiring correctness checking functions in the base language. The augmented screen generator would use the common declarative parser and relation creation mechanisms to process the declarative parts of the generator input.

9. Concluding Remarks

Declarative-module-based application generators, which represent the central focus of the proposal, have the following advantages.

- They are susceptible to development and modification by users
- They generate modifiable output code

[5] One important use of language mixing is to allow the embedding of code in languages complementary to traditional procedural languages, such as logic programming languages.

- They relate well to procedural languages, both in terms of input text proximity, and output module communication.

The declarative module approach is not quite as well suited to any particular purpose as the use of specialized program structures, either supplied with a language, or defined via extensible compilers. However, it avoids the tendency to language divergence implicit in either of the latter approaches.

10. Acknowledgments

I thank Gene Sakamoto, Alex Hurwitz, Nan Shu, and John Sowa for their helpful comments on earlier versions of this report.

11. References

1. E. Falkenberg, "Significations: The Key to Unify Data Base Management", Information Systems 2, Pergamon Press (1976) 19-29

2. E. Horowitz, A. Kemper, "High-Level Input/Output Facilities in a Database Programming Language", Technical Report TR-84-307, Dept. of Computer Science, University of Southern California (June 1984)

3. "Interactive System Productivity Facility, Dialog Management Services". IBM Publication SC34-2088-1 (May 1983)

4. J.M. Lafuente, D. Gries, "Language Facilities for Programming User-Computer Dialogues", IBM J. Res. Develop, 22,2 (March 1978) 145-158

5. P. Lucas, "On the Structure of Application Programs", Lecture Notes in Computer Science 86: Abstract Software Specification, Springer Verlag (1980) 390-438

6. P.S. Newman, "An Atomic Network Programming Language", IBM Scientific Center Report G320-2704 (June 1980)

7. L.A. Rowe, K.A. Shoens, "Programming Language Constructs for Screen Definition", IEEE Transactions on Software Engineering, SE-9,1 (January 1983)

8. G. Ruiz Herrera, "The Programmed Compiler", IEEE Computer (March 1983) 135-139)

9. D. Teichroew, et.al., "An Introduction to PSL/PSA", ISDOS Working Paper No. 86, Dept. of Ind. Eng., U. of Michigan, (March 1974)

10. U.S. Department of Defense, "Reference Manual for the ADA Programming Language", MIL-STD 1815A (February 1983)

SCIENTIFIC CENTER REPORT INDEXING INFORMATION

1. AUTHOR(S) : Paula S. Newman	9. SUBJECT INDEX TERMS Application Generators Screen Managers Programming Language
2. TITLE : Module and Application Generation via Declarative Modules	
3. ORIGINATING DEPARTMENT 60G	
4. REPORT NUMBER G320-2739	
5a. NUMBER OF PAGES 14 — 5b. NUMBER OF REFERENCES 10	

6a. DATE COMPLETED August 1984	6b. DATE OF INITIAL PRINTING November 1984	6c. DATE OF LAST PRINTING

7. ABSTRACT :

This paper discusses a method for allowing declarative specification of some application components, such as screen input/output, together with procedural specification of others, with less disjointedness than is usually the case with complete or partial application generators. The method involves the use of programming units called "declarative modules", containing linear-form data collections with generator-specific schemas. Equivalent data collections are passed by the compiler to generators, which return source program units in a base language.

8. REMARKS :

FOR THE YEAR 1982

F. W. BALL, J. G. SAKAMOTO, "Supporting Business Systems Planning Studies with the DB/DC Data Dictionary", IBM Systems Journal, Vol. 21, No. 1 (1982).

F. W. BALL, J. G. SAKAMOTO, "Supporting Business Systems Planning Studies with the DB/DC Data Dictionary", Database Management, November-December issue, Auerbach Information Management Series (1982).

S. A. JUROVICS, "Daylight, Glazing, and Building Energy Minimization", ASHRAE Transactions, Vol. 88, Part 1 (1982).

S. A. JUROVICS, "The Impact of Daylight Utilization on Thermal and Lighting Loads: A Case Study", presented at ASHRAE Annual Meeting, Toronto, June 1982.

J. M. MARSHALL, "Systems Architect Apprentice System (SARA)", Proceedings of Corporate Symposium on Structured Logic, Yorktown, November (1982).

P. S. NEWMAN, "Towards an Integrated Development Environment", IBM Systems Journal, Vol. 21, No. 1 (1982).

M. M. PARKER, "Enterprise Information Analysis: Cost-Benefit and the Data Managed System", IBM Systems Journal, Vol. 21, No. 1 (1982).

FOR THE YEAR 1982 (Continued)

M. M. PARKER, "Toward Cost-Benefit Analysis of Data Administration", SHARE Proceedings, Vol. 1, August 1982.

J. G. SAKAMOTO, "Use of DB/DC Data Dictionary to Support Business Systems Planning Studies: An Approach", The Economics of Information Processing, Vol. 1, Publisher: John Wiley & Sons Publishing Co. (1982).

J. G. SAKAMOTO, "Use of DB/DC Data Dictionary to Support Business Systems Planning Studies: An Approach", Symposium on The Economics of Information Processing, Sponsored by the IBM Systems Research Institute (1982).

N. C. SHU, "Specifications of Forms Processing and Business Procedures for Office Automation", IEEE Transactions on Software Engineering, Vol. SE-8, No. 5 (1982).

N. C. SHU, D. M. CHOY, V. Y. LUM, "CPAS: An Office Procedure Automation System", IBM Systems Journal, Vol. 21, No. 3 (1982).

R. C. SUMMERS, "Computer Security", chapter of Computer Handbook, Publisher: Van Naustrand Reinhold Publishing (1982).

FOR THE YEAR 1983

C. WOOD, E. B. FERNANDEZ, "Minimization of Demand Paging for the LRU Stack Model of Program Behavior", Information Processing Letters, Vol. 16, No. 2 (1983).