Partial compilation of REXX

by R. Y. Pinter
P. Vortman
Z. Weiss

A comprehensive set of compilation techniques for coping with various dynamic features of the REXX programming language are described. Among them are a novel symbol table structure, a multiple representation method for type-free objects, and a number of run-time acceleration techniques. Most of the work can be unified under the general principle of* delayed execution, *which is applicable in other situations as well. Significant performance gains were observed in an experimental setting, and these results led to the decision to develop IBM's recently announced REXX compiler product.

The Restructured Extended Executor language, called REXX,[1,2] is a personal programming language that was originally designed as a structured command language. It offers the programmer many degrees of freedom, deviating from many traditional language design principles, such as procedural abstraction, data typing, and structured control flow. The semantics of instruction sequencing is rather liberal. The language has the ability to interact with the environment and with external programs.

Until recently, the only language processors available for REXX were interpreters, in a variety of environments. The interpreters are well suited to REXX's highly dynamic semantics. The language can be interpreted straightforwardly, using an operational model induced directly by its semantics, which results in a considerable performance penalty. Programmers like the freedom permitted by the language mostly during initial program development and debugging. Many programs have been written in REXX and run on the various interpreters. The authors address the problem of how to compile REXX programs so as to improve their run-time performance when run in production mode. The study reported in this paper eventually led to the development of the recently announced REXX compiler product.[3]

The dynamic nature of REXX presents obstacles that inhibit the utilization of almost every common compiling technique.[4] The problems come from three major sources: run-time requests to change binding environments (both with respect to types and versions), indeterminate control flow, and the interaction with the external environment (coprocesses and the operating system). Some of these problems come up in some other languages, such as APL, LISP, and SNOBOL, but rarely is their presence so intensive under one framework as in REXX. The authors made a special effort to isolate them and provide orthogonal and general solutions. The new techniques described here to handle dynamic phenomena can be applied to similar situations in other modern languages.

REXX also poses several challenging syntactic problems. For example, incomplete constructs may not be flagged by the interpreter if an exit occurs before the scan is complete, and some se-

mantic activities need to be performed during scanning. These issues, however, are beyond the scope of this paper and are not addressed here.

The following section describes the difficulties that REXX presents to a compiler builder. Subsequent sections outline solutions to the major problems with emphasis on their generality, describe experimental results and possible extensions, and discuss relationships to other work. We conclude with an overall evaluation of this effort.

Why is REXX hard to compile?

REXX is an "open" language. This openness is manifested in more ways than one. For example, the semantics of instruction sequencing is rather liberal, there is no traditional block structure, there are no declarations or types, operations are applicable on the basis of instantaneous values, arithmetic is performed with dynamic precision, and variables can be shared with auxiliary processes (written in other languages). In this section we explain these features and analyze the effect that they have on the compilation process.

Undetermined scope. The language does not have a traditional block structure for procedure definitions and program calls. The CALL and RETURN statements merely transfer control (similar to "Branch and Link" at the assembly level), but do not hide the global environment from the local one. The PROCEDURE statement is an executable statement that takes effect only at run time; hence it does not provide a syntactic boundary to the definition of a procedure. The following example displays several of the key problems with REXX's scoping rules:

```
 1. if a = 1  then call p1
 2.           else call p2
 3. call p3
 4. if x = 3  then signal p3
 5.           else signal p2
 6. p1: procedure
 7. p2: x = a + 1
 8. if x = 2 then return
 9. p3: x = x + 1
10. return X
```

Since the execution of the PROCEDURE statement can be made conditional (upon the entry point or some predicate that is computed at run time, as in lines 7–9), one cannot use the information during compile time to resolve variable references. Moreover, one can "fall through" (from lines 4 and 5) into the body of a procedure from the code preceding its definition. Similarly, there is no explicit denotation to the end of a procedure; its logical end is the frontier of RETURN statements reachable from the entry point(s), but this depends again on run-time behavior (see lines 8, 10).

In addition to the lack of block structure, REXX has a unique sequencing semantics for statements. By using the SIGNAL statement, execution control can wander through various parts of the source text, including the body of procedures. More severely, there is a computed SIGNAL (using all the power of expression evaluation in the language) where the target of the jump is calculated at run time; this is stronger than, for example, a FORTRAN- or a PL/I-like computed GOTO construct, since we do not know the possible range of labels at compile time.

REXX is an open language.

Thus, even though the execution of a PROCEDURE statement automatically creates a local environment, the statements that follow may be executed in a different environment if reached through CALL or SIGNAL without executing the PROCEDURE statement. Consequently, it is often impossible to know in which context an instruction will be executed, and the amount of work required at run time to perform context resolution and switching may be prohibitive, especially for data-intensive programs.

Dynamic binding and creation of variables. The language is declaration free. Variables come and go (using the DROP statement) dynamically. One cannot predict whether a variable exists or not at a given point in a program without full-fledged flow analysis, which is very difficult as noted above. Hence no generation of symbol-table pointers as part of the code produced by the compiler seems feasible.

As in SNOBOL,[5] the binding of identifiers to variables is dynamically dependent on the most recent assignment to a variable with the given identifier. Unlike other languages, there are no declarations, either explicit or implicit. A variable is declared implicitly merely by assigning a value to it (again, similar to SNOBOL).

Another effect of the declaration-free language is that its variables do not have types.

If this is not enough, names of variables can be generated at run time, using all the power of expression evaluation of the language. As the names cannot be known at compile time, there is no way to arrange them in a static symbol table and use pointers to them in the generated instructions.

The major instance of this is compound variables, which are essentially associative arrays with arbitrary (numeric or string) keys. From the user's point of view, the meaning of A.I.J may be similar to that of A(I,J) in FORTRAN, but in REXX I and J can be either numeric values or strings. This means that the reference routine to derived names cannot be generated at compile time. Notice that dimensions, lengths, or other attributes cannot be used.

Since the PROCEDURE statement is an executable statement and not a declaration, the lifetime of variables is dynamic, depending on the calling sequence and the visibility, or exposure, of variables among procedures.[6] Moreover, the fact that a variable was exposed in a called procedure may cause "reverse-inheritance," namely that the variable appears in the caller only after the call, whereas it was not present in the first place.

No types. Another effect of the declaration-free language is that its variables do not have types. As a matter of fact, all variables are of type **string**, but since certain operations are permitted only if the operands have values in a given range (e.g., numeric or Boolean), there is a considerable penalty for checks, conversions, and simulations of operations in an alien domain that creates a computational burden on the interpreter. Naturally, one is tempted to try to deduce the type of a variable (in order to generate the code required to implement the various operations as well as symbol-table access mechanisms) at compile time, but this can be done only to a very limited extent due to the aforementioned problems with control-flow analysis.

Moreover, arithmetic itself is a troublesome issue. The language offers a dynamic precision setting which has an effect on every intermediate result of an expression. This makes the deductions pertaining to the applicability of operations to arithmetic data even more complicated.

The situation may seem reminiscent of the problems that come up when trying to compile APL programs, but unlike APL—where there are exactly two types and the conversions between them are explicit—here we must contend with the dynamic precision feature that practically generates an unbounded number of types. Moreover, the effect of a precision setting is total in the sense that it affects all arithmetic operations in the current environment, whereas the conversions in APL are "local," i.e., they happen per variable. Since the semantics of arithmetic operations is sensitive to the values involved and their relation to the current precision setting, the generation of efficient code to support dynamic precision is not an easy task. Here we can draw some inspiration from APL, because every implementation of APL that we know of, represents numeric data in one of three different forms, and conversions among them are done spontaneously.

An important effect of all objects semantically being strings of unbounded length is that memory must be allocated dynamically. This means that the run-time support must include a fairly heavy mechanism for maintaining all values, both temporary and specified variables alike.

All in all, REXX displays two important dynamic characteristics concerning variables and their values, (1) dynamic existence of variables and (2) dynamic types and attributes of objects.

Solutions

The solutions presented here deal only with the more difficult aspects of the language. We first define the compiler's target in terms of an abstract machine,[7,8] then outline some of the specific techniques used to overcome the problems described in the previous section. This section concludes with a discussion of the delayed execution principle that is used throughout this work.

The abstract machine. Since explicit code cannot be generated, due to all the uncertainties concerning object types, versions of variables, and so on, the authors have opted to describe the semantics of a program in an intermediate representation that is later interpreted during run time. To do this a rather simplistic $n + 1$-operand abstract machine is defined, where the opcode is an entry point in a threaded-code interpreter. The n operands (where n varies according to the opcode) are pointers to object descriptors or immediate operands, and the $n + 1$st field contains target information.

The data types of the machine are character string, numeric string, and binary (fixed-point) integers. There are specialized fetch and allocation services for literals, variables, and temporaries for these types. These access routines are organized in a table, and their invocation is triggered by the general-purpose descriptors stored in the operand fields of each instruction. To support operations on these data items, routines are provided to handle arithmetic (in binary representation) as well as string-oriented interpretation of arithmetic.

One could ask why the authors did not use something more sophisticated than simply an $n + 1$ abstract machine. There are two answers to this question: First, this seems to be the lowest level possible in order to achieve both reasonable efficiency as well as portability. Second, a stack machine was deemed inappropriate since no gains are to be made by pushing value descriptors on a stack, and pushing the values themselves will cause unnecessary movement of data in memory due to the dynamic nature of the sizes involved.

Variable binding management. In order to accelerate execution time, a unique symbol table organization was used to provide fast hiding of large environments upon the execution of a PROCEDURE statement. A *lazy* hiding policy is proposed for variables: instead of reallocating local variables, the method waits until the first assignment to such a variable before it is allocated.

Following the execution of a PROCEDURE statement, each variable should be initialized to its literal name value, unless it was EXPOSEd explicitly, in which case it is bound to its latest (i.e., closest in the nesting sequence) creation.

An activation record (AR) is maintained for each name. An AR consists of a static portion and a dynamic portion. The dynamic parts are pushed and popped on a stack during environment changes only upon demand, as explained shortly. The static part contains the identifier's literal name, which is also its initial value, per the REXX semantics; the dynamic part contains a pair ⟨**ver, ptr**⟩ where **ver** is the "version number" of the variable (or how deep it should be in a virtual run-time stack), and **ptr** points to the value corresponding to this version.

During run time, the following actions are taken:

1. Initially, the program sets up and then maintains a global version number corresponding to the nesting level. At each instant, the current version of a variable is the one whose version number is equal to the global version number.
2. Every time a PROCEDURE instruction is executed, the global version number is incremented. The version numbers of all EXPOSEd variables are incremented as well.
3. Every time a variable is being accessed, we check whether the version number in the **ver** field of the dynamic AR matches the global version number. If it does, we use the pointer (from the **ptr** field of the AR) as is; otherwise, there are two cases to be considered:
   a. If a new value is created, the pointer to the previous value is pushed on the stack and the involved variable is recorded in a log.
   b. If no value is created, then the literal name value is used (since it should be considered uninitialized).
4. Every time a RETURN instruction is executed and a PROCEDURE instruction was performed since the last CALL, we decrement the global version counter, and for each variable entered in the log, we recover its pointer from the stack.

This scheme distributes the work load in such a way that the method does not have to push anything unless a value is assigned to the same name in the new environment. Two comments are due:

- As was implied from Step 3a of the algorithm, a variable log is maintained for each internal procedure. This log is used to record all variables whose previous generation was pushed on the stack. The log is used to recover the variables on return to the caller.
- Recall that a variable in REXX is EXPOSEd if it was explicitly declared to be so in a given procedure; then its value in the calling sequence can be accessed from within the procedure. An additional log is maintained to keep track of the EXPOSEd variables of each internal procedure. This log is used to record the values of all exposed variables in order to assign their new values in the global environment upon return. Special care must be taken when a variable is assigned a value in an internal procedure before it is assigned a value in the enclosing environment.

In order to support the above algorithm, the following data structure is maintained. A symbol table entry (STE) is created at set-up time for each identifier. The initial AR is identical to this STE. The AR always reflects the current variable and is dynamically updated during the program's execution. Since the STE is being used as the frame of the current AR, the AR is physically static and can always be referenced using the same address.

Specifically, efficiency is gained by the following:

1. Only one STE is created for each name. By using only one STE we save dynamic allocations, overall storage, and the reinitialization of STEs in each new environment.
2. Once an STE is created, its location is determined and can be used at any moment for successive references.
3. Only the minimal required information is pushed and popped.
4. Pushing and popping of variables is reduced only to those which are actually active in an internal procedure. (As a result of our study of REXX programs, we found that internal procedures use a very small subset of the entire group of variables which are active in a REXX program.)
5. Hiding the global environment from the local one is automatically performed using the fact that the nesting level does not match the version number of the variables. This fact assists also on the return from the local environment, as reviving the variables is automatically performed.
6. The STEs are arranged in a balanced binary tree. This structure was found to be best as compared to other alternatives such as hashing.

The name of a compound variable depends on the values of its components at the instance of reference.

An assisting structure for compound variable access. Compound variables display many of the dynamic characteristics of REXX: their names, values, and environment are generated during run time and cannot be predicted. The name of a compound variable depends on the values of its components at the instance of reference. Consequently, STEs for compound variables are not generated during compile time, because (1) the names of the compound variables are not known at compile time, as they are later derived using the values of their components, and (2) it is difficult to know whether they are global or local variables.

In order to reduce the cost of name derivation and compound variable access, we define an assisting symbol table entry (ASTE). An ASTE contains the pointers to the constituent components of the compound variable's name. By the use of these pointers, the name derivation cost is reduced. Also, this structure is used to save a direct pointer to the associated compound variable which can be used on consecutive references, and hence save access time.

For example, the ASTE of A.I.J points to the STEs of the variables A, I, and J. In case all the components are literals, the name itself is stored in the proper STE and no derivation is needed. Thus, A.1.2 has an STE which holds the derived name as is, and no pointers are necessary at all.

In REXX, operations are applicable on the basis of instantaneous values.

The structure of an ASTE is similar to that of a regular STE, and all ASTEs are maintained in their own structure, separate from the main symbol table. The actual STEs for compound variables are created during execution, and are inserted into the main symbol table dynamically.

In addition, a base anchor for each cluster of compound variables is always created and has the same access mechanism as any other variable. The base anchor provides an initial point for searching entries of the compound variable, regardless of the search algorithm used. It also contains the value of the stem, if defined. The combination of the base anchor with the ASTE yields the desired run-time efficiency.

Multiple representations. Recall that in REXX, operations are applicable on the basis of instantaneous values, and they cannot rely on previously acquired knowledge about data types. Moreover, as the precision setting is dynamic, objects can have different values as a result of a change in the precision setting.

Often values stay in one domain type from one reference to another, while in other cases successive references require different data representations of the same object. The authors try to improve the run-time efficiency by keeping multiple representations of objects' values in such a way that unnecessary conversions are avoided.

In order to achieve this goal, a *value block* structure is defined, which potentially holds all possible representations of a value. A value block comprises a representation indication field to record which representations exist at any given time, followed by the values in the various representations themselves. All the valid representations reflect the same value in several different types. For example, the numeric value 100 can coexist as the character string '100', as the 8-bit integer 01100100, and as the floating-point number 1.00E2.

Using this multiple representation technique, an object can be in one of the following states:

1. Only one of the representations is valid.
2. Some but not all of the representations are valid.
3. All the representations are valid.

The rules governing the maintenance of value blocks are as follows:

- A value is stored in the same representation in which it was created or as it was fetched. This can save conversions for successive operations that stay in the same domain.
- Once a conversion is required from one data type to another, both representations are kept. The converted value is stored in the value block and recorded accordingly in the representation indication field.
- Sometimes arithmetic operations invalidate the last string value. In such cases, only one type which is the target type is kept. No attempt is made to revive other value types unless specifically required.

To exemplify these rules, examine the case of dual representation. The *value* of x is defined as a pair of possible representations, $\langle x_s, x_n \rangle$, where x_s is the string representation of x and x_n is its binary representation. A *bottom* denotes that a representation is not available.

Then the value of x is defined as follows:

$Val(X) = \langle x_s, x_n \rangle$ when both representations exist,
$Val(X) = \langle x_s, bottom \rangle$ when only the string representation exists, and
$Val(X) = \langle bottom, x_n \rangle$ when only the binary representation exists.

Now consider the program fragment:

```
(1) A = B + C
(2) A = A || B
(3) D = C + A
```

The initial values are:

$Val(B) = \langle b_s, bottom \rangle$

$Val(C) = \langle c_s, bottom \rangle$

In order to perform the first additions in (1), the following conversions are invoked implicitly by the fetch mechanism

$b_n \leftarrow$ convert_to_binary(b_s)

$c_n \leftarrow$ convert_to_binary(c_s) after which

$Val(B) = \langle b_s, b_n \rangle$

$Val(C) = \langle c_s, c_n \rangle$

and now the operations are interpreted as follows with one more conversion needed before each of the statements (2) and (3):

(1) $Val(A) \leftarrow \langle bottom, b_n + c_n \rangle = \langle bottom, a_n \rangle$
(2) $a_s \leftarrow$ convert_to_string(a_n) $Val(A) \leftarrow \langle a_s \| b_s, bottom \rangle = \langle a_s, bottom \rangle$
(3) $a_n \leftarrow$ convert_to_binary(a_s) $Val(D) \leftarrow \langle bottom, c_n + a_n \rangle = \langle bottom, d_n \rangle$

After (1), only the numeric representation of A becomes available, and the string representation is invalidated. In order to use A in a string operation (concatenation), in (2) it must be converted. Notice that B's representations are readily used; they are not changed because B is not set. In (3) again we can use the binary value of C which is kept from (1), but the new value of A must be converted to binary again.

All in all, keeping multiple representations saves conversions. Using this principle is similar in a way to common subexpression detection techniques for saving temporary results.

Delayed execution. As well as solving the problems which the dynamic features of the language present, the authors had to think about the cost of these solutions. Often it seemed profitable to adopt an optimistic approach and postpone some corrective operations until they were required. This attitude was carried out in several cases that follow and is referred to as the *delayed execution* principle.

1. Creation of the local environment for a procedure. Instead of pushing all the variables of the previous environment on each procedure's AR, the push operation is postponed until variables are assigned values. The push is performed on each variable specifically and acts therefore only on those local variables which are assigned new values in the local environment, while all other variables rest intact. The pop operation on return from the procedure is therefore performed only for those variables which were pushed.
2. Creation of symbol-table entries for compound variables. This activity is delayed until the corresponding variable comes into existence. In this way, only keys that are actually active cause space to be allocated. Since the key may comprise several variables, the reference routine is partially compiled (up to the actual key evaluation), leaving it necessary to look up only the values at run time.
3. Maintaining several representations. The cost of retrieving values for objects may increase. In order to decrease the cost, retrieval of values is performed with ascending cost using the fastest routine first, and the heavier routines only in case the first one failed. Operations which will need numeric values will check for binary representation to accelerate execution. Whenever possible, conversions will be postponed until required, and values will stay in the same domain unless a change of domain is needed.
4. Arithmetic operations performed on first trial using binary (fixed-point) integer values. With the hope that most arithmetic operations will succeed using binary values, we delay the heavier string arithmetic until we really need them. The same strategy is used for logical operations, which use only 0 and 1.

Results and extensions

The techniques described above were tested in an experimental setting. We observed an improvement in run time over the SPI interpreter of factors ranging between 4 to 10, depending on the nature of the source program. These results led to a decision to develop a general availability REXX compiler which has been recently announced,[3] using

(among others) the design and techniques described in this paper. In our work, no attempt was made to optimize the compiling routines themselves, therefore no measurements on compile times were made; these issues were addressed by the developers of the product.

There are three major areas in which further improvements can be expected.

1. In addition to saving the binary format of numeric values when appropriate, the floating point representation could also be saved, thus speeding up scientific application programs. (Such programs are being written in REXX!) Another validity test per access would be necessary, but the savings seem to justify it.
2. The range of keys for compound variables could be analyzed to provide a better idea on how to construct efficient referencing routines.
3. Flow analysis, as hard as it may be with REXX, could lead—for well-structured programs— to conclusions that may result in significant simplifications to the generated code.

Relation to other work

Where to draw the line between compilation and interpretation is a matter of semantics. Almost every compiler produces some code which is interpreted at run time (e.g., FORTRAN for the FORMAT statement, PL/I on some bit manipulation operations). For some languages, producing intermediate code that is interpreted during run time is the only way the languages can be compiled.

The authors looked at the problems presented by other dynamic languages, such as APL, LISP, and SNOBOL. Different attempts to compile these languages usually used threaded code for the compiled output code.[8,9] Other compilation efforts[10–13] have made changes to the languages so as to make them compilable to executable code. The most notable change is to enforce lexical scoping rules (as in the programming language, Scheme, vs pure LISP[10,13] and the changes suggested to APL in References 11 and 12), but other problematic issues have also been ignored. These changes enabled the application of data flow analysis and other types of deduction mechanisms.

Our effort was conducted under "real world" constraints. Language restrictions were not allowed (other than the insistence on complete constructs and not handling the INTERPRET instruction). Thus, most of the techniques that are enabled by linguistic and semantic changes were not applicable to the work discussed in this paper. Still, the authors were able to achieve the four- to ten-fold speedups reported above, which is commensurate with the aforementioned efforts.

Moreover, the authors claim that REXX encompasses a wider spectrum of dynamic features compared to the other languages. Most notable are the rather unconventional block structure, dynamic precision, and compound variables. As explained in the section on solutions, the method used indirect threaded code (in an $n + 1$ operand abstract machine), based on solutions already used for SNOBOL4 (SPITBOL)[14] and APL.[15,16] However, to solve the problems presented by some of the dynamic features of REXX, we had to seek additional solutions. Specifically, in APL and SNOBOL all identifiers' names can be deduced from the source code, while in REXX they may be computed only at run time. SNOBOL uses a way of declaring **arrays** and **tables** (arrays for integer indices, and tables for string indices), while REXX uses the unified notion of compound variables—generated dynamically during run time—for both purposes. Some special characteristics of the language, such as dynamic boundaries of procedures and dynamic generation of code, are unique to this language and required general solutions. (By "general" we mean that it will handle these features in the presence of the other dynamic features as well.)

The concept of delayed execution was intended to overcome some difficulties with minimal cost. Unlike SPITBOL and APL, we do not stack the environment on entry to a function (procedure), but postpone it until explicitly required. As variables that are not pushed on the stack need not be popped, we save the stacking of local variables. Our concept of multiple representation of values was formed to handle the difficulties with dynamic precision and variable data types. While SPITBOL and APL require conversions and validation of values, holding a multiple representation saves some of the conversions and results in a more efficient compiler. As data types are attributes of the values and not of the variables, it

allows another degree of freedom. The same applies to the service routines, such as fetch and allocate. As they are attributes of the variables and not part of the generated code, they can be changed dynamically or use different routines tables corresponding to the instantaneous instruction requirements.

Summary

This paper presents several newly developed techniques for compiling REXX programs, in face of the language's highly dynamic nature. These techniques, summarized as follows, can be applied to a number of similar features in other dynamic languages, such as APL, SNOBOL, ICON, and the UNIX® Shell.

- The multiple representation technique makes typeless objects feasible, saving unnecessary conversions.
- Using indirect pointers to values solves the problem of the dynamic length of variables.
- By keeping a tree-like symbol table during execution we allow changing, deleting, or adding symbols at run time.
- By using an access routines table, we can access variables efficiently while keeping their dynamic features.
- By delayed-execution techniques, we achieve good performance of the compiled code while sustaining dynamic boundaries to procedures as well as dynamic binding of variables.
- Finally, many traditional compilation techniques concerning the allocation of temporary values, compile time optimizations based on flow analysis, and threaded-code are used.

Acknowledgment

The authors would like to thank Micky Rodeh and Igal Golan for many helpful discussions.

UNIX is a registered trademark of UNIX Systems Laboratories, Inc.

Cited references and note

1. M. F. Cowlishaw, "The Design of the REXX Language," *IBM Systems Journal* **23**, No. 4, 326–335 (1984).
2. M. F. Cowlishaw, *The REXX Language*, Prentice-Hall, Inc., Englewood Cliffs, NJ (1985).
3. *CMS REXX Compiler User's Guide and Reference*, SH19-8120, IBM Corporation; available through IBM branch offices.
4. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers—Principles, Techniques, and Tools*, Addison-Wesley Publishing Co., Reading, MA (1986).
5. R. E. Griswold, J. F. Poage, and I. P. Polonsky, *The SNOBOL4 Programming Language*, second edition, Prentice-Hall, Inc., Englewood Cliffs, NJ (1971).
6. G. V. Cormack, "Extensions to Static Scoping," *Proceedings of the SIGPLAN'83 Symposium on Programming Language Issues in Software Systems* (June 1983), pp. 187–191.
7. J. R. Bell, "Threaded Code," *Communications of the ACM* **16**, No. 6 (June 1973), pp. 370–372.
8. R. B. K. Dewar, "Indirect Threaded Code," *Communications of the ACM* **18**, No. 6 (June 1975), pp. 330–331.
9. H. Glass, "Threaded Interpretive Systems and Functional Programming Environments," *SIGPLAN Notices* **20**, No. 4 (April 1985), pp. 24–32.
10. R. A. Brooks, R. P. Gabriel, and G. L. Steele, Jr., "An Optimizing Compiler for Lexically Scoped LISP," *Proceedings of the SIGPLAN'82 Symposium on Compiler Construction* (June 1982), pp. 261–275.
11. T. Budd, *An APL Compiler*, Springer-Verlag, Inc., NY (1988).
12. W.-M. Ching, "An APL/370 Compiler and Some Performance Comparisons with APL Interpreter and FORTRAN," *Proceedings of the APL'86 Conference*, ACM (July 1986), pp. 143–147.
13. D. Krantz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, and N. Adams, "ORBIT: An Optimizing Compiler for Scheme," *Proceedings of the SIGPLAN'86 Symposium on Compiler Construction* (June 1986), pp. 219–233.
14. SNOBOL is an acronym for StriNg Oriented and symBOlic Logic; SPITBOL is used to indicate speedy SNOBOL.
15. Z. Weiss, *An Adaptive APL Machine*, Doctoral dissertation, Technion, Israel Institute of Technology, Haifa, Israel (1978).
16. H. Saal and Z. Weiss, "Compile Time Syntax Analysis of APL," *Proceedings of the APL'81 Conference* (October 1981).

Ron Y. Pinter *IBM Science and Technology, Technion City, Haifa 32000, Israel.* Dr. Pinter received the B.Sc. degree in computer science from the Technion–Israel Institute of Technology, Haifa, in 1975, and the S.M. and Ph.D. degrees in electrical engineering and computer science from the Massachusetts Institute of Technology, Cambridge, in 1980 and 1982, respectively. During 1982–1983 he was a member of the technical staff in the Computing Sciences Research Center, AT&T Bell Laboratories, Murray Hill, New Jersey. In December 1983 he joined the IBM Israel Scientific Center, where he is currently the manager of the Programming Languages, Compilers, and Environments department. He is also a Senior Research Affiliate with the electrical engineering department at the Technion, he has taught at the Hebrew University, Jerusalem, and spent the academic year 1988–89 as a Visiting Scientist at the Department of Computer Science, Yale University, New Haven, Connecticut. His research interests include parallel programming techniques, code generation algorithms, and layout for integrated circuits. Dr. Pinter is a member of the Association for Computing Machinery, ACM SIGPLAN, and the IEEE Computer Society.

Pnina Vortman *IBM Science and Technology, Technion City, Haifa 32000, Israel.* Mrs. Vortman received the B.Sc. in mathematics and physics from the Hebrew University, Jerusalem, in 1969. She joined IBM Israel in 1970 as a systems analyst and later became a systems engineer. She specialized in database applications and database tuning, and subsequently became a specialist in large systems performance and tuning for VM and MVS. During 1984–86 Mrs. Vortman was on assignment with the IBM Israel Scientific Center, working in the areas of DASD simulation and REXX compilation. She is currently on leave of absence from IBM Israel, working in the editors area of the IBM Application Systems Division, Bethesda, Maryland. Her main interests remain the performance and storage management of applications and systems, as well as database management systems.

Zvi Weiss *IBM Research Division, Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, New York 10598.* Dr. Weiss received the B.Sc. degree in physics from the Technion–Israel Institute of Technology, Haifa, in 1968, and the M.Sc. and D.Sc. degrees in computer science from the Technion in 1972 and 1978, respectively. From 1974 to 1985 he was a research staff member with the IBM Israel Scientific Center, where he was the manager of the programming languages group in 1984–85. During that period he was also a Research Affiliate with the computer science department at the Technion. From 1985 to 1990 he was senior research staff member and Deputy Program Director of the Software Technology Program at MCC, Austin, Texas. In April 1990 he joined the IBM T. J. Watson Research Center in Hawthorne, New York. His research interests include optimization techniques for very high-level programming languages, cooperative debugging systems, and software engineering environments.

Reprint Order No. G321-5437.