SPECIALIZED LANGUAGES:

AN APPLICATIONS METHODOLOGY

by RICHARD H. BIGELOW, NORTON R. GREENFIELD, PETER SZOLOVITS, and FREDERICK B. THOMPSON

California Institute of Technology
Pasadena, CA

One objective of the information processing community is to aid the problem-solving activities of its clients. In this paper we will discuss a methodology for serving the needs of the "user", that is, the end-user: the manager running an organization, the accountant understanding the financial condition of a company, the anthropologist studying a culture, the engineer designing some equipment, or the meteorologist predicting the weather. Each of these users has his own particular, idiosyncratic problems. The computer should be an effective tool for him in dealing with these problems. Our methodology is designed to provide each of these users with an appropriate interface to the computer, with a language which is natural to his view of reality.

In this paper we examine the nature of today's ubiquitous applications packages, discuss our notion of applications languages and present some of our experience with the REL system, which has been designed to incorporate our views on specialized user languages.

This research has been supported by:
Office of Naval Research contract #N00014-67-A-0094-0024
National Science Foundation grant #GH-31573
Rome Air Development Center contract #F30602-72-C-0249

APPLICATIONS PACKAGES

The bare hardware of a computer, from the point of view of the user, is impotent. An operating system augmented by a few language processors, e.g. FORTRAN and COBOL, is hardly more useful. Indeed, when constructing complex applications for an end user, the standard programming technique is to first build a set of data structures and utility routines which then become the user's environment and make the computer habitable. We will call any consistent set of such structures and routines an *applications package.*

The need for many such applications packages is clearly demonstrated by their existence and wide usage. Examples are standard programs for payroll and inventory control in business, the SPSS[1] package for statistical analysis, subroutine libraries and languages such as NAPSS[2] for numerical analysis, and APT[3] for machine tool control. Hundreds of other illustrations may easily be found.[4,5]

All these systems have a common property: they provide operators which perform meaningful unit operations needed by their users. Their primitives draw up payrolls, compute correlations and solve differential equations. We wish to examine how the user invokes these primitive operations to fulfill his requirements.

In the most unsophisticated system, the user invokes a unitary operator: e.g. "produce the payroll". This is a complete program, perhaps operating on large bodies of data, with well-understood, structurally constant results. This type of applications package supports only a single aspect of its users' reality. Being optimized around a single task, it is not easily modifiable to meet even simple contingencies and it often quickly becomes inadequate. Problems change in structure as well as in data. Because the user has no concept of the computer representation of sub-parts of the whole problem, he is left with the use of a partly obsolete operator, or he becomes dependent on his programmers to make even the most minor structural changes in his problem solutions.

Concerns of typical computer users consist not of single, all-encompassing operations, but of a number of lower level tasks which in combination allow the solution of a range of related problems. For example, the accountant does not care merely to be able to produce his quarterly financial report. He has to be able to investigate data on various aspects of the company's fiscal status to understand his problems. His products are not just periodic reports, but also the tax and cash flow calculations, projections, special briefings, etc. Similarly, a physicist manipulating his experimental data looks not for a single answer, but for a multitude of indications and partial results which may help him understand the processes he is studying.

A computer system which supports such investigations must embody a number of different primitive operators, to correspond to the variously complex conceptual units of the user. It must also allow the hierarchical combination of these primitives to build up the operations which match higher-level user concepts.

The common production of standardized subroutine libraries in many fields attests to the widespread acceptance of this view. Such libraries, along with the standard algebraic computer languages, allow the construction of hierarchically-composed calls on the primitives, offering flexibility and power. What, then, are the inadequacies of these sophisticated applications packages to the user?

On the one hand, a computer system for a particular user must embody a large set of the conceptual primitives of his problem area in order to be useful to him. In addition, however, that system must also exclude the incursions of as many as possible of those computer concepts unrelated to the problem area. All of today's generally available programming languages have a strong bias in their syntactic and semantic capabilities to fit the needs of their designers, namely computer scientists. Their natural primitives include the control of storage, input and output, the declaration of procedures, data types, etc. Every one of these concepts is foreign to the problem area in which the nonprogrammer user is working. Thus, although the subroutines in a library may well represent valid primitives to our user, the irrelevant concepts of program control, procedure calling, and data management intrude upon and disrupt his problem solving.[6]

Current users struggle with this disruption in different ways: the accountant must work through a programmer, removing himself from direct contact with his data; the physicist often *becomes* a programmer, sacrificing his productivity as a scientist to develop competence in a field of only incidental interest to his work.

The information science community can provide better technical solutions and more viable tools.

APPLICATIONS LANGUAGES

To be most effectively utilizable, the computer must metamorphose to be each user's own conceptual machine. It must embody exactly those primitive notions which the user finds fundamental; it must support that structuring of complex problems which the user finds natural. And because the user must be able to communicate easily with his machine, it must provide for communication in a language which embodies the user's conceptual primitives and the means of composing them clearly and concisely.

It is not generality that the language must provide. Indeed it is exactly in its ability to reflect the biases, limits, and idiosyncratic representation of the user's reality that a specialized language finds its greatest strengths. The user brings with him a host of presuppositions, the knowledge of his field, of which he is only peripherally aware, but whose logic underlies all of his problem-solving activities. General languages *know* nearly nothing about the problem domain. All checks, all limits, all structures must be explicitly expressed by the user. In any high-level application, the amount of knowledge which the user has about his data is enormous. To enter it as explicit instructions to the computer and to probe his data in a system which recognizes none of his tacit knowledge is unconscionably tedious.

The implicit inclusion of the tacit knowledge of a specialized problem domain is the advantage which gives the applications language both expressive conciseness and computational efficiency in the problem-solving tasks of the particular end user.[7] With such a language the user can concentrate on his problem instead of the programming details. There is no intrusion of foreign concepts from the implementation - the user manipulates structures and operators that are familiar and relevant. The power of the language opens new options and capabilities in his use of the computer, and the naturalness of the language allows him to exploit those capabilities himself, bringing his own implicit knowledge and intuition to bear without having to work through a programmer.

At the same time, the embodiment of the user's presuppositions implicitly in the prior programming of the primitive operators results in increased computational efficiency. It is often erroneously assumed that higher-level, user-oriented languages entail increased computing times as well as excessive implementation costs. Quite the contrary. The existence of specialized knowledge of the field of application allows more global optimization of the basic primitives. And once programmed, these primitives can be composed in the solution of wide-ranging problems, being reused a multitude of times without involving any new programming tasks. One can appreciate the extent of such savings by considering the compaction of records and optimization of access to peripheral storage which the programming of specialized primitives can embody, savings in ultimate computer time which can amount to orders of magnitude for large data bases.[8]

The fear has been expressed that the widespread development of such languages would lead to a large number of small user communities, each with its own highly specialized language, each unable to communicate data and methods of solution to the others. Consequently, the argument goes, we should concentrate upon standardizing our languages rather than specializing them, to allow the easy exchange of data, algorithms and personnel.

We find two related answers to this line of argument. First, we do not believe that our current experience with sharing data or programs justifies the requirement of adherence to rigid standards on the part of all computer users. Specialized languages already tend to arise in response to natural divisions which exist among groups of users. Hence, between groups isolated by specialized languages, it is already unlikely that they would profit from sharing of common technique and common data. Second, the increased capabilities provided a group by a specialized language may well justify accepting the cost of relative isolation. It is the user community's responsibility to regulate language development to achieve an economic balance between specialized capability and communication. Between groups where communication and sharing of data is desirable, their various specialized languages can explicitly facilitate precisely such common access and cross-talk.

Currently, the economic factors underlying the decision of whether or not to create a specialized language are dominated by implementation cost. Technical advances of the sort we will describe can reduce this cost sufficiently to allow that decision to be made on the grounds discussed above. We now examine the task of implementing specialized languages.

METALANGUAGES

From the implementor's viewpoint, a computer language consists of a set of procedures containing the semantic primitives of the language, the set of data structures to which these are to be applied, and a syntax which allows the user to compose his operations and to apply them to his data. The task of the language implementor is to analyze the natural requirements of the user in these three areas and to design and code the procedures, data structures and syntactic processor to realize the language.

We can examine the language writer's problem just as we looked earlier at the end-user's problem. We note that current programming languages do not have operators and data structures in their semantics which specifically support language implementation. Because their facilities are much more primitive and detailed, the construction of applications languages is difficult and costly. The language implementor needs a specialized language, just as the user does. The primitive concepts of this language must be parsing, storage management, permanent and temporary data base management, semantic compositions, etc. Again we emphasize that this implementor's language is not a generalized language. Not all implementors will want the same parser nor the same data base management scheme. However, for particular classes of language implementors, those implementing similarly-structured user or *object* languages, a particular implementors' or *meta-* language is useful.

A metalanguage structures and supports the task of the applications language implementor in the same way that the applications language structures and supports the task of the user. It allows the implementor to concentrate on the problems of designing his language and supports its implementation. For example, provision within the metalanguage of an efficient parsing algorithm coupled with a simple means of expressing syntax rules will allow the programmer to utilize a natural syntax in his language. He is no longer forced to a simple syntax by the high cost of implementing anew a complex parser.. The metalanguage can embody much of the tacit knowledge of the language implementor about the internal structure of the language. For instance, a rigid coupling between rules of grammar of the object language and the invocation of the associated semantic primitive routines allows the metalanguage to know the calling and return structures of these semantic routines, and to use this knowledge to allow a more concise description of the routines and to perform error checking or optimization on the object language. The metalanguage also directs the attention of the language implementor to the central issues of his task: the construction of the operators and data structures that are significant to the user and a natural syntax for combining them.

REL - THE RAPIDLY EXTENSIBLE LANGUAGE SYSTEM

The REL System has been developed to give concrete realization to the ideas presented in this paper and allow us to get actual experience with the use of such a system. We will not further describe REL here, but will only enlarge upon those aspects which relate to ideas discussed in this paper. For a more complete description of REL, see references [9,10,11].

The REL System provides a metalanguage for the implementation of sentence-driven, syntax-directed, interpretive and extensible applications languages[12]. Within the REL environment, a language is represented by a set of general rewrite grammar rules, their corresponding processing routines, and the data structures of the associated data items. The grammar rules structure the operation of the language, define the valid syntactic constructs which the user may employ, cause invocation of the syntactic and semantic processing routines, and define which data types may be related in the language. As an example, consider the following grammar rule:

<class;relation_image> => <relation> 'OF' <class>

This may be a rule of grammar of a language which expresses aspects of a relation calculus. The rule, written exactly as shown here, states to the REL system that:

- the syntactic construct "name of a relation" followed by the word "of" followed by "name of a class" is valid.
- such a construct represents another data item of the type "class",
- this new item may be computed by applying the program named "relation_image" to the two old data items.

Notice how this metalanguage forces the implementor's attention to exactly the problems which should concern him: the primitive entities of his language, e.g. "class" and "relation"; the primitive operations of the language, e.g. "relation_image"; and the syntax by which these can be combined, e.g. "regions of salesmen", "vendors of components".

We have emphasized that the user's language should fit his needs naturally. That means that he must often be able to define new operations on the basis of his previously existing operations to express new tasks and methods of solution. REL provides the applications programmer with a powerful tool to implement this ability for his language. Using whatever external syntax he finds natural for his users, the programmer can invoke an REL system utility which will add to the user language's grammar new rules which express the desired definitions of the user.

REL APPLICATIONS LANGUAGES

We have used REL to implement a variety of languages and have found it to be very supportive of them. Indeed, even if we had wanted to develop only one fairly complex language, we would have found it desirable to separate the REL system and general language processor facilities from the syntax and semantics of the particular language. Doing so has given us a framework in which to design our languages that has been at least as important as the support we have gotten to actually write the code.

The languages that have been implemented under REL to date include REL English,[13,14,15] the Animated Film Language (AFL),[16] a language for solving ordinary differential equations, [17] and a discrete simulation language.[18] We present a few examples from the first.

REL English is a technical English question-answering language for the analysis of complex sets of highly interrelated data. Its primitive operations are based on the data and semantics of a relational algebra. Thus the language was designed with a view to serving users with messy, large-scale data-related models. REL English's current users include a cultural anthropologist, a research hospital, and elements of a military staff.

The syntax of REL English is a complex, quite natural, deep case grammar which provides our users with powerful but concise statement and query capabilities. The primitive data entities of the language are individuals, classes, and binary relations. REL English has all the common notions of sentence structure, time, function words like "all", "what", and "the". It does not include any particular vocabulary but provides the ability for the user to introduce new words which denote individuals, classes and relations from his own problem domain. Further, it has the capability for defining new verbs in terms of relations and the verbs of being, and it provides the ability to extend itself by new syntactic forms which represent composed operations of the language, as specified by the user.

Note that this much REL English is common to a wide variety of users. Relating to the earlier discussion of the cost of implementing specialized languages, we remark that to this point the cost of adding yet another English-based language to REL is merely the effort of deciding that the relational data structure and an English statement and query capability are natural to the user's problem area. To specialize to the requirements of a particular user, the extension facilities of REL English are used to introduce the relevant user concepts to the language.

Our example will be the familiar one of the personnel data base. The initial preparation of the language consists of acquiring a copy of REL English and adding appropriate terms:

```
employee := class
department := relation
immediate supervisor := relation
salary := number relation
```

We can then include all of the basic data on each person, usually taking it from some fixed-format file. At this point, the personnel manager can ask the usual questions:

```
What is Sue Jones' salary?
When was John Smith Bob Jones' immediate supervisor?
How many departments have employees whose salary is over 20000?
```

The manager will soon extend this simple language with meaningful and useful terms:

```
def: senior employee: employee whose salary is at least 18000
def: subordinate: converse of immediate supervisor

Are all managers senior employees?
What proportion of senior employees are female?
Which managers have more than five subordinates?
```

The user can, of course, produce reports. The statement:

```
What is the ratio of male employees to female employees in each department?
```

produces a columnar listing of the departments and their male female ratios. Other involved conceptualizations can be expressed by verbs:

```
earn := verb (<object> is the salary of <agent>)

Does some employee earn more than his immediate supervisor?
```

The capabilities represented here allow the user to efficiently explore the interrelationships which are meaningful to his task.

The above is a small illustration of the type of applications language which we have implemented in the REL system. Each of the other languages mentioned have quite different syntax and semantics. Although our experience to date is limited, these applications have been found to be directly and effectively usable by their intended users and inexpensive to implement.

CONCLUSION

The continued development of more sophisticated software and better, less expensive hardware should lead to a great increase in the number of computer users. As a tool for organizing and managing large, complex human problems, specialized computer languages promise to increase our effectiveness in handling a complicated world. Indeed, only by the support of specially tailored "natural" languages will the large group of new computer users have the ability to effectively deal with this growing resource. Whenever possible, the burden of making man-machine communication tractable should fall on the machine, where the burden is manageable through the use of specially designed metalanguages and applications languages.

Our experience with REL gives us confidence that the notion of a metalanguage for the implementation of whole classes of applications languages is legitimate and valuable. We intend to continue exploring the wide range of end-user oriented languages which find a natural home within our system, and we envision the future construction of other metalanguages (or programming systems) for different classes of applications languages.

We would like to make a few final comments about the impact of the above ideas on the computer professions. We expect a redefinition of the relation between systems programmer, applications programmer, and user. The user has problems to solve, which he can state in some language specialized to his universe of discourse. The task of the applications programmer, in our view, is to provide the user not with solutions to individual problems, but with computer languages and capabilities to allow the user to pursue the solutions of his problems in terms of concepts which are natural to his problem domain. The task of the systems programmer is to build efficient language processing systems and their associated metalanguages so that the applications programmer can concentrate on the structuring of the data, preparation of the processing algorithms, and specification of the syntax natural to his user. The current work in computer systems such as REL will facilitate the task of the applications programmer.

Finally, the power (and thus the responsibility) of the applications language implementor is great. As our everyday language affects our thoughts, our computer languages guide and limit our work. An appropriate and flexible applications language can greatly enhance the work of a user; a poor and rigid one can impoverish it. The future of our ability to effectively use one of our most powerful tools, indeed, of our ability to cope with an informationally overwhelming world, is at issue.

ACKNOWLEDGEMENT

The authors acknowledge the assistance of Ms. Sara Gomberg in the preparation of this paper.

REFERENCES

1. Nie, N.H., Bent, D.H., and Hull, C.H., *SPSS: Statistical Package for the Social Sciences.* McGraw-Hill, New York, 1970.
2. Symes, L.R. and Roman, R.V. Structure of a language for a numerical analysis problem solving system. In *Interactive Systems for Experimental Applied Mathematics,* Klerer, M. and Reinfelds, J. (Eds.), Academic Press, NY, 1968, 67-78.
3. *APT Part Programming.* McGraw-Hill, New York,1967.
4. Sammet, J.E. *Programming Languages: History and Fundamentals.* Prentice-Hall, Englewood Cliffs, NJ, 1969.
5. Sammet, J.E. Roster of programming languages, 1972. *Computers and Automation* 21, 6B (1972 Aug 30), 123-132.
6. Dmytryshak, C.A. The universal consulting language - alias the investment analysis language, *Proc. Fall Joint Comput. Conf. 1972,* **41**, Part I, 525-535.
7. Thompson, F.B. and Dostert, B.H. The future of specialized languages, *Proc. Spring Joint Comput. Conf. 1972,* **40**, 313-319.
8. Greenfeld, N.R. *Computer System Support for Data Analysis.* Doctoral Dissertation, REL Project Report No. 4, Calif. Inst. Tech., Pasadena, CA, 1972.
9. Thompson, F. B., Lockemann, P.C., Dostert, B.H. and Deverill, R.S. *REL: A Rapidly Extensible Language System.* Proc. 24th ACM Natl. Conf., 1969, 399-417.
10. Dostert, B.H. *REL - An Information System for a Dynamic Environment.* REL Project Report No. 3, Calif. Inst. Tech., Pasadena, CA 1971.
11. Thompson, F.B. and Dostert, B.H. *The REL System,* Fourth International Symposium on Computer and Information Sciences (COINS - 72), Miami, FL, 1972 Dec.
12. Szolovits, P. *The REL Language Writer's Language: A Metalanguage for Implementing Specialized Applications Languages.* Doctoral Dissertation, Calif. Inst. Tech., Pasadena, CA, forthcoming.
13. Dostert, B.H. and Thompson, F.B. *The Syntax of REL English.* REL Report No. 1, Calif. Inst. Tech., Pasadena, CA, 1971.
14. Dostert, B.H. and Thompson, F.B., "Syntactic Analysis in REL English: a computational case grammar", *Statistical Methods in Linguistics,* 8(1972), 5-38.
15. Dostert, B.H. and Thompson, F.B. Verb semantics in a relational data base system. *Proc. ONR Symp. on Text Processing and Scientific Research,* Pasadena, CA, 1972 Nov.
16. Bigelow, R.H. Greenfeld, N.R., Szolovits, P. and Thompson, F.B. *The REL Animated Film Language.* REL Project Report, forthcoming, Calif. Inst. Tech., Pasadena, CA.
17. Bigelow, R.H. *Computer Languages for Numerical Engineering Problems.* Doctoral Dissertation, REL Project Report No. 5, Calif. Inst. Tech., Pasadena, CA, 1973.
18. Nicolaides, P.L. *RELSIM - An On-Line Language for Discrete Simulation in Social Science Research.* Doctoral Dissertation, Calif. Inst. Tech., Pasadena, CA, forthcoming.