THE REL LANGUAGE WRITER'S LANGUAGE:
A METALANGUAGE FOR IMPLEMENTING
SPECIALIZED APPLICATION LANGUAGES

Thesis by

Peter Szolovits

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1975

(Submitted October 3, 1974)

ACKNOWLEDGEMENTS

I wish to thank Dr. Frederick B. Thompson, my teacher, adviser and friend, for his guidance and inspiration. My perceptions of what problems are of interest and my intuitions about likely possible solutions owe much to his teaching. I have also felt privileged to work with Dr. Norton Greenfeld and Dr. Richard Bigelow, former colleagues whose experiences have contributed much to my understanding, and Dr. Giorgio Ingargiola, whose helpful comments on earlier drafts of this thesis have been of great value.

My graduate studies have been generously supported by the Fannie and John Hertz Foundation, for whose help I am grateful. The languages described in this thesis have been partially implemented in the Rapidly Extensible Language (REL) System, which is being developed under the leadership of Dr. Thompson and Dr. Bozena H. Dostert. The REL Project is supported in part by the following grant and contracts:

National Science Foundation grant #GH-31573

Rome Air Development Center contract #F30602-72-C-0249

Office of Naval Research contract #N00014-67-A-0094-0024

The Computing Center of the California Institute of Technology has provided the computer facilities which I have used to prepare and edit this thesis.

ABSTRACT

This thesis is an investigation into the task of implementing specialized computer application languages. It contains a discussion of the conceptual issues which make the development of specialized languages useful, and it motivates the selection of a scheme of syntax directed interpretation as the framework on which specialized languages are to be implemented. The thesis includes a description of the REL Language Writer's Language, in which the semantically primitive data types and operations and the extended syntax of object languages are to be specified. The definition of an illustrative object language for the storage and retrieval of personal bibliographic information is given. Also discussed is the relationship between this manner of language implementation and various alternative technologies.

TABLE OF CONTENTS

CHAPTER I

INTRODUCTION

> . . . to imagine a language means to imagine a form of life.
>
> -- L. Wittgenstein [1958, p. 8e]

How is man to use computers? Before entering a discussion of technical matters which will be presented in this thesis, it seems appropriate, if not obligatory, to examine the more global and philosophical issues which motivate research and debate among those who devote their attentions to applying computers for human use.

Computing is a very young practice. It has been barely twenty-five years since the development of general purpose stored program computers, and that quarter century has seen a dramatic, unprecedented and chaotic growth of uses of the computer. The fantastic diversification of current and intended applications has created the image of a field in constant flux and torrential change. Yet, in a

deeper sense, it is all too apparent that developments of the past decade in the technological areas have not been matched by the evolution of organizing concepts which make the physically available computer intellectually available to the vast potential number of people with complex, large-scale problems which could be amenable to solution using the computer as an important tool.

Current computer applications have been attained at a very high cost. Only the richest organizations have been able to allocate the resources required to implement computer systems that precisely satisfy their requirements. Even for these organizations, the slightest changes in their applications have made their systems inappropriate and have required a continuing supplemental infusion of money to keep their computer operations useful. For the user with a much smaller financial base, the high cost of tailored computer systems has forced on him a compromise between utility and cost. He has had to accept cut-rate systems which never "fit" quite right, but which at least provide some form of useful computing capability at a tolerable cost.

It should be the role of computer science research today to develop systems which will prove to be comfortable and valuable tools for investigators in various areas of human endeavor. We should move away from the customary view of the computer as the super-expensive, super-complicated general machine which only large businesses and major scientific research projects can have the resources -- both financial

and intellectual -- to apply effectively. The cost of hardware is decreasing rapidly enough that we can foresee computers owned by small businesses and even non-professional individuals. The major problem, however, will be: Of what use is the computer to the businessman or the homeowner unless he can use it naturally and effectively in pursuing his interests in an inventory, a budget, or perhaps a hobby?

The responsibility to answer this question must not be shunned. The coming universal availability of computer power can have tremendous liberating consequences, or it can merely add to the social forces which currently encourage uniformity and abstention from creative growth. The extreme cost of developing new software for computers dictates uniform, inflexible applications. The customary view of computer usage, if extended into the low price range, suggests that the individual computer will come with a turnkey program to allow its user to balance his checkbook, order his groceries, and read the AP or UPI news releases as they come off the wire. Overlooked will be the possibility that the individual may have a personal, uniquely interesting use of his powerful tool in mind -- a use he cannot realize because the tool is powerful only through the specialized knowledge of programming. Without careful nurture and a significantly increased productivity of technical support, the potential for individual creativity and diversity may be cut short by an economically encouraged conformity.

We now proceed to lay a conceptual framework for computer use,

in linguistic terms, on which to build a technology for bringing the computer's power directly and conveniently to the potential end-user, providing an interaction with him that is natural and specialized to his interests.

1.1 LANGUAGE AND COMMUNITIES OF INTEREST

Strong arguments have been set forth for the thesis that the thought and behavior of an individual or of a community are greatly influenced by their language. Indeed, language has been suggested as the embodiment of all the tacitly assumed views, knowledge, limitations and culture of a community.

As Benjamin Whorf, in his best known example, states the case [1956, p. 57]:

> . . . The Hopi language contains no reference to 'time,' either explicit or implicit.
>
> At the same time, the Hopi language is capable of accounting for and describing correctly, in a pragmatic or operational sense, all observable phenomena of the universe. . . . Just as it is possible to have any number of geometries other than Euclidean which give an equally perfect account of space configurations, so it is possible to have descriptions of the universe, all equally valid, that do not contain our familiar contrasts of time and space. The relativity viewpoint of modern physics is one such view, conceived in mathematical terms, and the Hopi Weltanschauung is another and quite different one, nonmathematical and linguistic.
>
> Thus, the Hopi language and culture conceals a METAPHYSICS, such as our so-called naive view of space and time does, or as the relativity theory does; yet it is a different metaphysics from either. In order to describe the structure of the universe according to the Hopi, it is necessary to attempt -- insofar as it is possible -- to make explicit this metaphysics, properly describable only in the Hopi language.

Thomas Kuhn, applying a similar analysis to the revolutionary change of scientific theories, characterizes the operation of "normal science" as the fleshing out of scientific theories, based upon the seemingly immutable bedrock of the current paradigm. Throughout his discussion, he assumes a coupling between an established paradigm and the language in which its content is stated [1970, p. 136].

> As the source of authority, I have in mind principally textbooks of science together with both the popularizations and the philosophical works modeled on them. . . . They address themselves to an already articulated body of problems, data, and theory, most often to the particular set of paradigms to which the scientific community is committed at the time they are written. Textbooks themselves aim to communicate the vocabulary and syntax of a contemporary scientific language.

Kuhn is interested mostly in the changes of paradigm which signal scientific revolutions. We will be content with considering particular languages which are closely tied to a current paradigm. Thus, the methodology explored below is in support of "normal", not "revolutionary" science.

We will be interested in examining the utility of specialized languages in the service of limited groups. It is useful to consider what such a group is. Kuhn, in a postscript to his well-known discussion of scientific change [1970, p. 176], recognizes the circularity inherent in identifying a paradigm as the set of fundamental (and unquestioned) assumptions of a normal-science community, and a community as a group whose members live by the same paradigm. He

attempts to eliminate the circular dependence by recourse to sociological research to determine the boundaries of groups; yet, perhaps it is exactly the adherence of a group to a commonly developed or accepted paradigm which is the fundamental character of the group and the paradigm. In any case, it is important to recognize the interdependence of scientific communities with their paradigms.

In an argument recalling information theoretic studies of communication, Frederick B. Thompson [1966a, p. 64] expresses this view:

> What happens in a research team, working intimately, intensively together? The commonality of experience, the familiarity bred by common environment, common action, and common goals, presses down into more and more discriminating levels the intersection . . . of their disparate languages. Their sparse syntax and crisp jargon rests on a deep fund of tacit understanding.

The size of a community, in the above sense, may vary greatly. Typically, the specificity of its language, the totality of its common pool of tacit assumptions, will vary inversely with the community's size. Communities are far from disjoint. One might imagine, for example, the community of physicists, all of whom share a large body of specialized knowledge and beliefs. Yet within their paradigm, the sub-communities formed by, say, low-temperature physics and astrophysics may have sets of paradigms, corresponding to technical jargons, so different that technical discussion between members of the two fields may be nearly impossible. Their communication is meaningful only at the more general level of their shared background and understanding.

The internal structure of a community and, correspondingly, the structure of the paradigm or language of the community are hierarchical. The higher in this structure that a conversation takes place, the wider is its scope and the lower is its precision. The lower it takes place, the more precise are its distinctions, the more efficient is its capability for expression, and the narrower is its scope of applicability.

Whereas one might identify the total hierarchy of such languages as a natural language, say English, we will concern ourselves with only specific languages, of rather narrow scope, serving fairly small communities of interest, and reacting poorly to revolutionary change. Specifically, we are interested in providing computer languages which successfully subsume a large fund of tacit knowledge, so that their interaction with a computer user may occur at a level of specialization appropriate to discourse with a colleague. If the language is of extremely limited use or even incomprehensible in a different context, that is a price we are willing to pay. The individual computer language is not intended as a system in which a complete human world view may be represented, and its inability to react well to major shifts in interest or conceptual structure need not be regretted. For sufficiently differing areas of interest or points of view, various distinct specialized languages can be developed, possibly with built-in methods of coordination.

The fragmentation of intellectual worlds implied by this discussion is a bitter pill for both humanists and scientists to swallow. Some, such as Gunnar Myrdal [1969], argue that "objectivity," namely the ability of each person to understand and assess the work of each other, is to be achieved by making explicit the assumptions under which we pursue thought and research. Yet, to hope for that is unrealistic, if not irrelevant. We cannot continually be conscious of all our assumptions; thus, we depend on our language to filter experiences in a way that is consistent with the fundamental decisions we have already (consciously or unconsciously) accepted about the world, so that they do not constantly clamor for our attention and interfere with our thinking.

Of what relevance is this discussion to the use of computers? Simply, that we propose to mirror in the computer the specialized linguistic capabilities of small communities of interest, by developing computer languages which naturally express the jargon and embody the paradigm of a specialized field.

1.2 THE ROLE OF THE COMPUTER

The "manufacturers' dilemma" has become a well-recognized problem in upcoming computing practice. It is that the cost of a unit operation on the computer is decreasing approximately exponentially,

with no short-term end of the trend in sight. This means that in order to remain profitable, the computer manufacturing and distribution community must devise techniques which make ever-greater and more sophisticated use of computers and attract larger and larger numbers of users to the computer. With a touch less hint of compulsion, Frank T. Cary, formerly president of IBM Corporation, put it this way [CACM July 1972, p. 511]:

> Continuing improvements in large-scale integration, semiconductor memory, and magnetics, let us put more information on line to the user. . . . Application programs plus new programming languages must be tailored more closely to the way people think, communicate, solve problems with their own minds. They will be needed to bring the computer's information more easily and quickly to the end user, who will not be a computer professional and will not have the time, or the inclination, to become familiar with either the computer's intricacies or the programs. He will want to talk and communicate with the machine in much the same way he talks and communicates with you or me -- without learning special languages to manipulate the machine. He will want that machine to be a convenient tool for improving the profitability and productivity of his business operations, his professional activities, his daily performance in whatever he does.

Thompson [1972, p. 315] has argued that the use of specialized languages secures significant economic advantages to the eventual user, as well as to the manufacturer. To the user, these come about through the manpower saved by allowing him to communicate in a language natural to his problem domain, and through the language implementor's ability to make use of the tacit knowledge of an application field to increase the efficiency of problem solving in that field. To the manufacturer, the obvious advantage lies in the expanded pool of potential customers who

will be attracted to the computer by the simple and powerful tools provided by the specialized languages.

These complementary pressures, to widen and deepen the computer user community, on the one hand, and to capture the sophisticated conceptualization of specialized problem domains, on the other, will lead to the widespread implementation and use of specialized computer languages. This thesis is concerned with techniques which make the implementation of such specialized languages convenient and efficient.

To avoid confusion, a few terms should be clarified here: By "computer user", or "user", or "end user", we mean the same person whom Cary describes above, not the traditional user of the computer, namely the programmer. By "language", we will generally mean "specialized language", of the sort one might use in investigating a user's problems, rather than today's normal programming languages. These distinctions are, of course, somewhat arbitrary, but the emphasis we wish to place is on the contact between the "person with a real-world problem" and a computer language tailored for his benefit.

In this thesis, we will explore the concept of specialized languages and one technology for making possible their cost-effective implementation. In Chapter II, specialized languages are related to notions now under development under titles like "Very High Level Languages" and "Artificial Intelligence", and an examination of current

practice in application programming leads into the presentation of syntax directed interpretation as a methodology for implementing specialized languages. Chapter III presents an informal description of BIBLIO, a specialized language for the storage and retrieval of a simple, personalized bibliographic information data base. The description of BIBLIO is intended to show the type of specialized language whose creation is addressed in this thesis, and examples to illustrate features of the Language Writer's Language (LWL) will be drawn from BIBLIO.

Chapters IV and V discuss the interesting features of LWL. Chapter IV deals with a PASCAL-like language for specifying data types and primitive semantic operation, and Chapter V presents the rich syntactic mechanisms by the use of which a specialized language can be built to express the fundamental semantic capabilities in a form natural to the user. Concluding remarks appear in Chapter VI, concerning alternative technologies of specialized language implementation and possible extensions to the techniques presented here. In addition, some comments on the social impact of language diversification tie links back to the introductory discussion of the first chapters. Two appendices are also included: Appendix A is a brief, semi-formal description of LWL; Appendix B contains the linguistic definition of BIBLIO in LWL.

CHAPTER II

HIGH LEVEL, SPECIALIZED LANGUAGES

> We dissect nature along lines laid down by our native languages.
>
> -- B. Whorf [1956, p. 213]

The specialization of computer languages, suggested in the discussion of the last chapter, is of value to the applications user because a specialized language is able to deal naturally with its user's problems in a restricted context. It is able to assume a great deal of implicit knowledge of data and techniques, and it dissects nature along the lines which its user requires (in Whorfian terminology). To an extent, these are the goals of much of the research which has recently been reported under the title "very high level languages." In a review article, Leavenworth and Sammet [1974] introduce the field in this way:

> . . . Consider the following list of terms, each of which should be followed by the word 'languages':

very high level
nonprocedural
less procedural
goal oriented
problem oriented
pattern directed
declarative
functional
relational
problem statement
problem definition
problem description
systems analysis
specification
result specification
task description

In addition to these terms, consider also the following:

automatic programming
artificial intelligence

This list is not necessarily a complete set of all the terms now being used by one or more groups of people to convey an intuitive notion of languages which in some sense are 'higher' than FORTRAN, COBOL, PL/1, etc. The most common term used for this concept has been nonprocedural, and the most common phrase has been 'what' rather than 'how'.

Of course, it would be a mistake to assume that each of the above terms refers to equivalent efforts and aims. Chapter I of this thesis strongly suggests that specialized languages will often be very high level languages, because by implicitly assuming much knowledge that is special to a particular domain, a great deal of the "how" can be built into the mechanisms of the language itself, leaving the user to deal mainly with the "what." It does not follow, however, that high level languages need be specialized, or that specialized languages are always high level. For example, languages with primitive data types

which might be identified with sets, sequences and n-ary relations are widely recognized as high level. Yet, they promote their capabilities as general computational languages, e.g., [Schwartz 1973a]. To the end user, in any field outside computer science or mathematics, they are hardly more capable of understanding "what" than are any of the older generation of lower level languages (a term, interestingly enough, that is not enthusiastically embraced by anyone). Conversely, important languages like COMIT [Yngve 1966] and IPL-V [Newell 1960], though rather specialized in their application domains, are hardly high level.

2.1 LOW LEVEL LANGUAGES

Perhaps the key issue, both technical and conceptual, in applying computers for human use, is the determination of the degree of "knowledge" we impute to the hardware-software combination which is attacking some posed task. Any student of computers can easily conjure up a picture of John von Neumann or Commander Grace Hopper leaning over an early vacuum-tube behemoth, setting the bit patterns needed to open the right electrical gates, to perform the earliest calculations on complicated trajectories. In this picture, the meaning of what is being computed, in the sense that one wants to label the resulting numerical values with some descriptors tying them to a world of objects and relationships, exists completely outside the operations of the computer. Only the programmer is aware of what significance the input and output

values have, and only the programmer realizes that the sequence of operations set up in the order code of the machine actually accomplishes the solution of a differential equation. This picture is titled the low-level programming approach.

Primitive operations which are semantically insignificant to a user characterize low level programming. In the war-time trajectory example, the significant operation is to achieve a model of an anti-aircraft defense system. This will normally reduce to seeking a solution to some set of fixed differential equations, which will satisfy a certain set of required conditions. In more detail, one might say that the problem is to determine a set of parameters for the anticipated functional form which will minimize some measure of deviation from the ideal solution. More concretely, the problem may be stated as an algorithm which will compute those desired parameters.

It is at these levels of generality that the person who poses the problem and the applied mathematician who devises a solution scheme perform their analyses. The task of storage allocation to parameters and auxiliary variables, the translation of algorithmic descriptions to sequences of machine order code detailed to the level of "add" and "jump if zero", and the creation of commands to cause input parameters to be accepted and results to be displayed are unseen -- thus insignificant -- at the level where the problem and its desired solution are described.

To illustrate further, note that in low level programming, the problem as understood by its formulator cannot even be recognized. Without extensive documentation and profuse commentary, a programmer other than the program's originator cannot grasp the essence of what is being computed. The familiar problem of "de-compiling" machine code into comprehensible algorithms, for instance, has no general solution, because at the operational machine level, those organizing concepts which give meaning to the problem solution being attempted are no longer explicitly available. The best de-compiler can yield merely a symbolic and somewhat compacted version of the code. The user's problem is irrecoverable.

It is not meant here to disparage low level programming as such; when the problems of interest are indeed low level -- e.g. in optimizing an algorithm to operate efficiently in terms of the number of unit operations performed -- then that is the appropriate level; the basic operations are significant and meaningful. What is criticized is the use of such low level techniques for an approach to problems whose scope is sufficiently broad to push the low level considerations into obscurity.

The keynote of the low level programming approach is its concentration on the symbol manipulating aspects of computation, as opposed to meaning at the user's conceptual level. An analogy may be drawn with mathematical reasoning: consider the axiomatic derivation of

arithmetic from the set theory. From our childhood, we understand the meaning of simple arithmetic statements and operations, and only in much later life do some become concerned with interpreting this meaning in a rigorous, formal structure. When we do arithmetic, we do not really perform the set theoretic operations which define arithmetic. In the use of arithmetic, its axiomatic definition is irrelevant -- a requisite foundation, to be ignored after it is successfully concluded. Similarly, when we compute a trajectory, the machine code which has expressed the detailed manipulations we have needed to come to a result is no longer of significance.

Thus far, we have argued that in the use of the computer to solve human problems, two separate levels of thought can be recognized: a conceptual understanding of the problem with some strategies for its solution, and a detailed cognizance of the techniques required to implement a solution. Further, we have argued that these two levels are generally quite dissimilar. We may also claim with relative safety that it is the former which is of central interest to those seeking a solution.

2.2 HIGH LEVEL LANGUAGES

The development of higher level programming languages has been a response to recognizing the above split; the higher level language

attempts to provide facilities which make the expression of the problem of interest central and the specification of the symbol manipulating operations peripheral or even unnecessary. It is much more difficult to bring to mind a picture of what high level programming is than to envision primitive low level computing; most of us have extensive experience with the latter approach, and very few have enjoyed much contact with "intelligent" or "knowing" computer systems.

Much of the "Artificial Intelligence" (AI) community, especially those engaged in work on automatic programming, see themselves involved in the creation of these high level tools. The view expressed is that a computer user should be able to approach the computer in English, or at least some good, powerful interlingua. He should be able to discourse about a problem he is attempting to solve, and the computer system should be capable of generating a model of the user's problem, finding the unclear aspects of it, interrogating the user further, making some "intelligent guesses" to complete the model, and eventually producing and performing an algorithmic solution to the problem (e.g., [Balzer 1973]).

With such a system, one could recast the scenario for solving the trajectory problem: instead of choosing a reasonable numerical technique for solving a differential equation to which the problem has been manually reduced, the investigator might simply begin by discussing his interest in the interception of aircraft by projectiles, the effects

of wind and shape on acceleration, the necessary proximity of shell to aircraft for a successful intercept and maybe the disposition of enemy pilots to perform particular maneuvers. The computer's model builder may in turn request some information on maneuverability of the airplane. After an extended conversation of this sort, the system would then produce a model of the anti-aircraft attack situation and a program to compute the required aiming parameters of the gun. Presumably, the user would also be able to interrogate the model, to determine in what way the pilot's probabilistic behavior, for instance, had been taken into account.

The above is, of course, highly speculative as a general capability, but it points in the direction that high-level techniques will aim. As a more immediate goal, a system which has a fair number of built-in model templates and some pattern matching capability to recognize one of its models as a generalization of a particular described situation promises to exhibit some of the same sophisticated behavior. The idea here is to create a high level language which exhibits sophisticated behavior across a wide range of problem domains. Because specialization appears to be the best way of building high level responsiveness, this approach uses the technique of creating a <u>high level</u> language by selecting for it the appropriate one of a set of already existing <u>specialized</u> languages, so that the innate structure of the particular selected pattern can guide further interaction and interpretation.

Whatever philosophical position one may hold on the ability of machines to possess knowledge, it is obvious that in the above paradigm, what is of interest is the meaning of the user's problem, not the manipulations required to solve it. Independent of the physical realization of this computerized "knowledge", the user is dealing with a system which appears to understand his problems at very much the same level as he does. This is the characteristic of high level languages: that their semantic primitives are coincident with the meaningful concepts of the user's problem domain.

In fact, it is meaningful to discuss an ordering of languages between the extremes we have identified. The placement of a language according to this ordering will correspond to the degree to which its primitives mirror the usage of its users. The scale is certainly not well-ordered, as level depends on intended application, but in general this thesis argues that the ideal is to move toward the higher level languages.

2.3 THE USE OF GENERAL PROGRAMMING LANGUAGES

The rest of this chapter, indeed the greatest part of this thesis, will be devoted to examining how specialized high level languages can be achieved. Thus far, the most widespread and most successful attempts at dealing with this problem have been made in the

development of a succession of more and more sophisticated general programming languages.

The strongest tendency in the history of programming language design has been to incorporate as primitives into new languages those often-used operations which had required tedious expression in previous environments. This development has generally increased the size and complexity of our programming tools, and has made it easier to express certain complex matters of data structuring and computation.

Each new programming language development increases the expressiveness and sophistication of computer languages, but does it make the computer any more natural a tool for the original man with the problem? We could answer "yes" only if he were the one who benefitted from using the newer capabilities of languages. But, typically, he does not.*

Today's principal user of a programming language is the professional programmer, not his employer. New developments in programming language design have succeeded not in bringing the computer's power directly to bear on the end-user's problems, but in easing the task of the intermediate, the programmer, who often does the

- - - - - - - - - -

*One school of thought holds that as more and more people will acquire sufficient programming skills, new programming languages will be of direct benefit to the applications user. There is no convincing evidence that any such trend exists for the very large user audience envisioned by planners like Cary (see Introduction).

work of translating a solution scheme into a running program. The newest high level languages have become high level for him, because they recognize and explicitly support his customary usage of the computer. They provide a meaningful semantics for discussing the complex symbol manipulations involved in computing. They do not, however, recognize the semantics of the original, motivating problem.

In the earliest publications on FORTRAN, that revolutionary development of the 1950's was described as the FORTRAN Automatic Coding System [Backus 1957]. Without belittling the enormous contribution of that early work, one must recognize that neither FORTRAN nor its near-contemporaries COBOL and ALGOL can be considered high level programming languages in the sense discussed above. Certainly, arrays in FORTRAN and pictures in COBOL are important conveniences, but neither truly embodies the useful concepts of its ultimate beneficiary. A profit and loss statement is not a COBOL record, just as a stress tensor is not a FORTRAN array. These language constructs still deal with the manipulations of arriving at a solution rather than the description of a problem. They rise far above their machine language predecessors: indexing operations or subfield selection can be subsumed under general notions; still, this falls far short of comprehending the semantic character of the real-world entities they model.

Current programming development deals with extremely flexible control structures, concurrent operation, contextual interpretation of

data and operators, privileged access to resources, automatic backtracking and error recovery, goal-directed procedure invocation, and myriad other similarly complex issues (e.g., [Bobrow 1973]). Because these concepts, once incorporated in a programming language, make the task of a programmer who chooses to use such techniques easier, the developers of these new complex languages have appropriated the names "high level" and "very high level" to their creations. It has indeed been suggested that the measure of "level" of a programming language should be the proportion of some exhaustive list of features which it includes [Schwartz 1973b].

The inclinations expressed here run counter to that belief. It may be true that each of the above features finds an area of natural applicability, but it is impossible to argue that they are all useful in expressing the semantically primitive notions of many problem domains. The criticism takes two forms. Certainly not all of these features can be of any use to a particular discipline. Thus, the cost of their inclusion in those instances where they are not of use is heavy and uncompensated, not only from efficiency considerations, but also because the presence of these general features will prejudice the thinking of the language's user in directions he might consider unnatural.* Even of greater significance, most uses of a computer -- perhaps barring those

- - - - - - - - - -

*For example, consider the difficulties involved in trying to teach merely a subset of PL/1, so that a student need not be bothered by the complexity of the full language.

special AI research applications for which many of these techniques were developed -- in fact do not find a sympathetic semantic expression in terms of these generalized primitives. A network of goals and theorems is not an insight into management practices. The meaning of a financial analyst's questions about the relationships among performance measures of certain types of companies may well be translated into a concurrent, backtracking relation-discovering procedure operating over a ring-structured network, but that is definitely not how the analyst understands it.

Unless the programming system is capable of communicating with the analyst in a language based on his own logic and terminology, he will become resigned to doing only what his programmers have foreseen for him, or to depending on his programmers to mediate between him and the computer system in a manner reminiscent of the shaman interceding for a person before his gods. The modern techniques which have been called high level are the tools of the medicine man; they support a high level organization of the manipulations of problem solving, but they do not support the ultimate human use of computers in an understanding way.

2.4 SPECIALIZED LANGUAGES

If the increased power and generality of the newest programming languages appears not to address the problem of naturalness and

sophistication of computer use for a wide variety of end users, what alternatives are available? Specialized languages, whose syntax and semantics are tailored carefully to the specific needs of their projected users, have been strongly proposed (e.g., [Bigelow 1973]).

The typical computer end user currently accesses the computer through the interface of an application program or an application programmer. For a relatively static interaction, such as the production of a payroll or the maintenance of a savings deposit record, single, complex programs which mirror the realities of their problem domains simply and fairly effectively are the rule. The bank teller may be interacting with a very sophisticated computer system which is capable, on detailed command, of solving problems beyond the teller's imagination; yet, to the teller, the system and his specially designed terminal are an obedient tool, faithfully recording deposits, withdrawals, computing interest, and supporting the many detailed requirements of a bank teller.

In situations which are less well defined and more subject to change, the unitary application program is less and less adequate. When the computer user is interested in discovering rather than recording information, his understanding must range over a wider latitude of detail and generalization. A bank's officer attempting to predict the future level of demand deposits must have the ability to look into the details of his bank's operation at the same level as the teller, to

develop data and insight into specific types of transactions, but he must also be able to abstract his view to the level where he can consider the impact of Federal Reserve policy and tax structure on his largest clients. Further, he does not understand at the beginning of his investigation the exact nature of his task, or even the questions he will eventually answer. The designer of an application program would need to be prescient to support the needs of such a user by a single, all-encompassing program.

The investigative computer user ordinarily has an application programmer at hand to answer questions for him by writing new programs as the need arises. The programmer-computer system is generally a high level tool of the sort discussed above. The experienced programmer can understand the essential aspects of his boss's questions, and can produce answers well suited to his employer's needs. The major difficulties with this tool are, however, serious. Its response time is inordinately long. If the figure of seven lines of finished code per day per programmer is within an order of magnitude of the good application programmer's productivity, even moderately complex demands on this "tool" introduce enormous delays in an investigation. The resulting hesitancy of the investigator to develop long lines of questions, where the nature of the next depends heavily on the results of the previous, entails a serious deficiency in effective analysis. In addition, the destruction of an intimacy between the investigator and

his data by the required intercession of the programmer can seriously reduce the value of the data to the investigator. Nevertheless, this is the current paradigm of user-programmer-computer interaction, and we should be able to profit from its analysis.

Let us examine the structure of the "tool" we are discussing, the programmer-computer system team. What makes it high level is that it has detailed knowledge of the problem domain of its user. Not only is the programmer familiar with the idiosyncratic views and usage of the investigator, but he also develops within the computer system a primitive "understanding" of the basic concepts with which he deals, in terms of a set of data structures, functions, and a program library. When faced with an individual problem, he need not start from scratch to build a particular solution, but can assemble parts of his prepared environment and specialize it to a sufficient extent to solve the problem. The measure of the value of this "tool", as of all other high level tools, is the degree to which the process of assembly and specialization corresponds to the way in which the user structures his understanding of the problem domain; the closer the correspondence, the easier and more effective it is to use the tool.

Several of the successful projects in natural language understanding have satisfied this model. Winograd, in his natural language system, selected a limited world of blocks on a table, built a set of functions which could manipulate this world, and provided a

strong linguistic component to allow his "user" to assemble the semantic operations of this world in English [Winograd 1972]. The LUNAR System of Woods uses a different technology, but in an essentially similar way, to provide its users access to lunar rocks data [Woods 1972]. With yet another approach, the REL English language has made an anthropologist's large data base available for use in a convenient and personalizable language [Dostert 1970].

Even a casual examination of the problems encountered by "Management Information Systems" (MIS), for example, is convincing proof that the development of tools of this nature is a difficult task. The larger the scope of development, the more difficult it becomes. Where the bank officer and his personal programming staff may be slow, they are likely to be satisfied and effective. An MIS for the company's leadership is bound to be less personalized and will undoubtedly be inadequate and unresponsive to many individual needs, and a company-wide MIS, to encompass uniformly the operations of all level from nuts and bolts to executive planning, is doomed to be a cumbersome, unmitigated disaster. General techniques, even if high level, tend to be less appropriate to the needs of an individual user than ones specialized for him.

It is important to seek ways in which the individualized use of computing can be aided and improved. As with all new technological developments, it is not even possible to foresee the innovative uses to

which a new technique, freed from its oppressive economic bonds, could be applied. Today, the cost of specialized languages, computer systems and extra programmers is so high that it precludes their utilization except in outstanding cases. If a considerable fraction of the task of the human part of the programmer-computer team can be shifted to the computer, a sufficient decrease in cost and increase in responsiveness is achieved to open the possibility of individualized, high level computing to many.

2.5 SYNTAX DIRECTED COMPUTING

The designer and implementor of a specialized computer application language faces a complicated and difficult task. He must discover and mirror in the computer the fundamental concepts of the application domain, provide facilities which allow these fundamentals to combine in the natural ways that a user may desire, and invent and implement a formal language which will be simple and natural enough to encourage its acceptance by the user and yet complex enough to allow its sophisticated application. At the same time, the implementor must not lose sight of the need for efficient ways of handling both the language he will provide and the fundamental algorithms which he will implement as part of the basic fabric (tacit knowledge) of the specialized language.

Syntax directed interpretation is an excellent general technique by which to structure and implement a specialized language. A general and powerful syntactic analyzer makes it possible for the language writer to implement rather sophisticated languages without excessive programming cost. In a well-designed language, the primitive interpretive routines will correspond to the semantically fundamental operations of the application domain, and the rules of grammar will express phrases whose meaning is a valid composition of primitive operations and data. The structure of the syntax directed interpreter is natural for the application language developer because it provides the above parallels to the manner in which the application programmer now works.

The REL Language Writer's Language (LWL) is a specialized programming language for the implementor of new specialized languages. Its "tacit knowledge" includes a model of syntax directed interpretation as performed by the REL System, and LWL is tailored to make it relatively easy for the language writer to specify the grammar of a new language and the data and functional primitives which represent fundamentally meaningful objects and operations of that language.

Languages implemented using LWL make rather strong demands on their computer environment for common services. This is characteristic of nearly all high level languages, because major aspects of computation expected to be shared by many programs are collected and abstracted

behind a few simple language constructs. Thus, "run-time packages" for nearly all languages provide standard input and output services; for the more sophisticated languages, the run-time environment includes garbage-collected heaps, dynamic error processing, synchronization of parallelism, etc. All languages that are implemented using LWL run in the REL System, which is essentially a large collection of services and facilities to bred by its specialized languages. It is a run-time enment which includes a great deal of tacit knowledge about syntactic analysis, language processing and data management; it is thus capable of providing high level abstractions which shorten and simplify the language implementor's task. The metalanguage may be viewed as the external representation or model of the REL System (to its user, the specialized language implementor).

The REL System is a complex syntax directed interpreter. It operates on the following cycle: wait for a sentence to be input by the user, perform a complete syntactic analysis in accordance with the grammar of the language currently in use, evaluate the meaning of the resulting structural analysis by using the data objects and functions of the language (composed as specified by the grammar), and (possibly) update the language's universe of discourse and output a reply to the user. This simple picture will later be complicated by the intrusion and handling of ambiguity, error and user extensions.

That part of LWL which expresses data structures and the

semantically primitive functions is discussed in Chapter IV, and the rather complex capabilities of the language processor are taken up in Chapter V.

CHAPTER III

BIBLIO: AN EXAMPLE

A book in the hand is worth ten in the library.

-- Anonymous

The discussion of the previous chapters has been concerned with a global view of the role of specialized application languages. At this point, we switch to a microscopic view, to consider a particular language, which will be of some use in showing how the principles of language implementation apply in a specific instance. Portions of the LWL programs which implement this language will appear throughout the thesis as examples of how various features of LWL are used.

The selection of a simple example to explicate a principle or methodology is never devoid of grave dangers. Languages for specialized domains range from the very simple, in both expression and power, to the extremely complex. To choose a simple one for tutorial purposes is tempting, but unrealistic -- it leaves untouched the complex issues

which guide the conceptual development of the methodology. Thus, something like a desk-calculator language, which is amenable to a straightforward implementation under a large number of systems, will not be adequate.

Much of the motivation for this research finds its roots in investigating the development of practical natural language processing techniques in a large data base, question answering context. The temptation is strong, therefore, to choose an example like REL English, which would illustrate a large proportion of the capabilities developed in the metalanguage. Unfortunately, such an exposition would be overly lengthy, and linguistic and semantic issues of such magnitude would arise from the example that they would threaten to swamp the exposition of meta- level ideas. Thus, we will choose a simpler language, and only occasionally refer to experience with REL English (which is partially described in [Dostert 1972; Greenfeld 1972; Thompson 1974a]).

Our example will be a language to provide a very highly specialized bibliographic reference file for an individual or small group of like-minded colleagues. The example is of more than mere academic interest, as anyone who has struggled with a three-by-five card file for such purposes can attest.

Of fundamental importance, intended to permeate this example, is the idea that it is to be a personal bibliographic filing system. It is

not to be considered for installation in place of the public library's card catalog, it is not necessarily appropriate even for a much smaller group of clients possessing fairly homogeneous interests; it is to be a private store of facts and organization, not intended for others' illumination or audit. Indeed, much of its power (and likability) will derive from its ability to capture idiosyncratic views of organization and information.

The language, which we will call BIBLIO, will deal with concepts like "book", "article", "report", "author", "editor", "publisher", "subject matter", "relevance to topic", "quality", etc. But what will properly form the fundamental building blocks of the language? Certainly, it should not contain as a fixed portion, some extensive "built-in" collection of references, because although the language intends to become a close reflection of its user's interests, a general (if somewhat hazy) distinction can be drawn between the underlying language and the information it contains about a particular universe of discourse. This is analogous to the distinction between FORTRAN and a FORTRAN program, or the lower predicate calculus and one of its models. Thus, while "publisher" is likely to play a meaningful role in the language, "MIT Press" is unlikely to deserve such special recognition.

The separation between innate parts of the language and data in its universe of discourse is not as clean-cut as the above distinctions imply. For example, since "subject matter" seems a meaningful concept

when discussing any book or article, it should be an innate language part. But what is the "calculus" of subject matters? Are we to assume that any general relationships, whatever, may hold among them, or do we identify subject matter with a common index, like the Library of Congress catalog number? This is typical of the questions which arise in the design and implementation of specialized languages, and we will look at it in some depth.

For the general library, a subject classification system like that used by the Library of Congress is necessary, because bibliographic information must be kept in a uniform, statically defined structure for the entire collection. Similarly, abstracting and reviewing services generally choose some fixed tree representation of their subject domain and maintain their data in that form; e.g., the categories of Computing Reviews [CR Jan. 1974, p. 43].

For a personal bibliographic system, the general tree-like character of the standard systems is often a reasonable organization, though the categories and their relationships will be quite different. Even considering a relatively specialized index like CR, the decisions of its editors on questions of emphasis and grouping will often be inappropriate to an individual. Working in the programming languages area, a researcher may well need to keep track of more references to "Procedure- and Problem-Oriented Languages" (CR category 4.22) than he ever encounters on other whole major branches of the subject tree.

Further, the general classification cannot afford to ramify its categories to a sufficient depth to provide adequate resolution for the specialist.

Not only is the general tree of subjects too coarse, but in some instances it must cut inappropriately across subtrees which may be quite useful in a certain conceptual view. For example, a long report on the design and implementation of a special purpose computer to support a particular language used for maintaining a large file might well bring together such categories as 6.22, 4.22, 4.33, 4.34, 4.35, 4.12, 3.73 and 3.74 (generally, areas in "hardware", "software", and "applications"). By the CR index, the report's parts should be scattered across the various categories most appropriate to each, and the general concept which unifies the development of this (hypothetical) system becomes invisible.

Notice the tradeoff that we are finding: Implementing a bibliographic language with a built-in index makes its initial use easier (if the index is appropriate to the user's field of interest), but it may make the language inflexible to later change. Alternatively, leaving the user a freedom of choice in deciding his indexing scheme requires of him more effort to define and maintain the index scheme, and requires that he have a better understanding of his problem and the manner in which the computer will operate on it, i.e., the "how". This tradeoff is characteristic of all specialization, and we find it again and again in discussing specialized languages.

3.1 BIBLIO STATEMENTS

Let us turn now to the informal definition of the BIBLIO language, considering first the universe of discourse that underlies it. We will assume that there are three classes of fundamental bibliographic entities. They are:

publications, by which we will mean books, articles, journals, published proceedings, reports, manuals, dissertations, collections, etc.
authors, which will include editors, collectors, translators, annotators, etc., and
subjects, which will themselves be categories of the user's choosing, describing the subject matter of the publication.

In addition to the three fundamental classes of entities, we will want to represent peripheral (though often highly useful) information about each. Thus, publications will typically have the name of the publisher and the date of publication, the name of a disseminator (e.g., the National Technical Information Service), the user's local source (e.g., the Comp. Center Library, or Fred's office), and an overall rating of quality. This peripheral information will be treated as attributes of the entities.

We distinguish between fundamental and peripheral information by anticipating the queries we are to answer. One might well ask "What are publications by Quine?" or "What articles are about syntax directed compiling?" or "To what subjects is Word and Object relevant?" but we do not anticipate "What books were published by Addison Wesley?" or "What

was published in 1960?" Therefore, we make authors, publications and subjects primary, and publishers, publication dates, etc., secondary.

The heart of the manipulable information in the data base serving as the universe of discourse for BIBLIO will be the relationships which hold between the major entities of this universe. Thus, we want to be able to represent the relationships between authors (used in the generic sense described above) and publications, and the specific type of such a relationship. For instance, Benjamin L. Whorf is the author of the book Language, Thought and Reality, and Saul Rosen is the editor of the collection Programming Systems and Languages. Other major relationships exist between publications and subjects (Aspects of the Theory of Syntax is about linguistics), between publications ("Two Dogmas of Empiricism" appeared in From a Logical Point of View), and between subjects (transformational grammar is part of syntax).

This brief description defines the underlying logic of the data base on which BIBLIO rests. The language in which the user and the computer communicate will have to reflect this logic. Because none of the data is predefined, BIBLIO must include ways to introduce the names of new publications, authors and subjects, and specify the relationships among them. We want the language to be easy to learn and use, and concise enough to be congenial; thus compound statements which will define and relate several new entities will be useful. We may want, for instance,

> Whorf, Benjamin L. is the author of the book Language, Thought and Reality, which was published by MIT Press in February 1956, and is extremely relevant to linguistic philosophy and also highly relevant to semantic models; it is an excellent work.

To avoid the difficulties involved in handling an English sufficient to process the above, we will settle for a somewhat less natural, though similar linguistic form. (Simple extension of the upcoming language by allowing innocent "noise words" and other trivial techniques can improve its appearance considerably, but we will not discuss that here.) We will state the above as:

> Author: Whorf, Benjamin L.; book: Language, Thought and Reality; publisher: MIT Press; February 1956, A-subject: linguistic philosophy; B-subject: semantic model; A

Here, we have translated "extremely relevant" to an "A-" modification on the subject, "highly relevant" to "B-", and "excellent work" to "A". Notice that the "February 1956" is implicitly known to be the publication date, just as the "A" at the end can only be the overall assessment of quality.

Actually, the language becomes more concise as more data are present. For instance, after the above statement, and another which has introduced Quine, Willard vanOrman as an author, the statement

> Quine, book: Word and Object; MIT Press, 1960, A-linguistic philosophy, A

carries quite a bit of information in rather few symbols. Since Quine, MIT Press and linguistic philosophy are already known entities, they need not be reintroduced, and the relations implied among them are

straightforward consequences of the types of these entities.* Notice that the specialized nature of the language allows a rather concise, high level presentation of information.

The statement of the sort presented above serves to introduce new authors, subjects, publishers, etc., and states the author-publication and publication-subject relationships. It will also express the publication-publication relation ("appeared in"), but it is not

- - - - - - - - - - -

*Several questions about ambiguity should assault the careful reader here. For instance, in the unlikely event that "MIT Press" is introduced as the title of a new book (perhaps a retrospective of the Cambridge weight lifting championships), a later use may find the term ambiguous. Usually, the ambiguity will be resolvable by syntactic means. In this case, if we require that a statement must refer to exactly one publication, we can always settle the sense in which "MIT Press" is used, and select the appropriate interpretation. One of the major advantages of the REL metalanguage scheme of implementation is that this disambiguation will be almost totally an automatic byproduct of the way language is defined, therefore very easy for the language implementor to provide.

The problem is tougher if our perverse "worst case analysis" suggests that the user may become interested in "MIT Press" also as a subject matter. We would find it overly restrictive to require the appearance of a publisher, and a relevance rating for each subject mentioned in a statement. Therefore, in something like

Quine, Word and Object, MIT Press

the "MIT Press" can be either the publisher or the subject. In that case, we require the user to disambiguate, e.g.,

Quine, Word and Object, publisher: MIT Press;

Similar comments can be made about other cases of ambiguity. For example, though "Quine" will be sufficient to identify Willard vanOrman Quine, if both Terry and Shmuel are in the data base, "Winograd, Terry" is required for an unambiguous reference.

sufficient to introduce the subject-subject relationships. As discussed before, this relationship could take several forms. We choose to define it as a partial order on subjects, where the order is "is part of." This is more general than the tree-like organizations considered before, as lattice-like structures can be represented. For instance, "Grammar is part of syntax directed interpretation," "Grammar is part of compiling," and "Syntax directed interpretation and compiling are parts of language processing." To express this relation, another kind of statement will be used:

subject: model generation; is part of subject: automatic programming;

or, assuming the previous introduction of "computer aided education",

model generation is part of computer aided education

3.2 BIBLIO QUESTIONS

The purpose of questions in BIBLIO will be to retrieve information that exists (either explicitly or implicitly) in the data base. The simplest queries will retrieve the basic information about the primary entities. Thus:

Q: Quine?
A: Quine, Willard vanOrman

Q: Language, Thought and Reality?
A: Whorf, Benjamin L., Language, Thought and Reality, MIT Press, February 1956.

Q: language processing?
A: language processing, syntax directed interpretation, compiling, grammar

These queries merely return more complete information about the requested entity; the first yields the full name, the second, a complete bibliographic reference, and the third, a list of all sub-categories of the subject. This usage reflects the type of operation to which a normal card catalog is customarily put.

The use of the interconnections in the data base makes BIBLIO truly powerful. By recursively combining applications of derivative computations, a large and wide net can be cast throughout the data base to search for the desired information. The following computations can be combined, where each "function" is a map from lists of one primary type to lists of another:

AUTHOR OF <publication>	-> <author>
WORKS BY <author>	-> <publication>
TOPIC OF <publication>	-> <subject>
GENERALIZATION OF <subject>	-> <subject>
WORKS ABOUT <subject>	-> <publication>
WORKS <rating>-RELEVANT TO <subject>	-> <publication>
<rating> QUALITY <publication>	-> <publication>

These allow us to pose questions like:

Works by Quine?
Author of A quality works B-relevant to linguistic philosophy?
Works about generalization of topic of works by author of Word and Object?
. . .

Notice that with these primitive functional operations, the amount of information retrieved by a complex query might be very large, because conjunction is not provided. At least for "WORKS", conjunction and dition may be very useful constructs, permitting constructions like:

Works by Quine and about logic?
Works about compilers or interpreters and by Wegbreit?

In addition, we would like a definition facility in BIBLIO, to allow the introduction of new concepts in terms of existing ones. Thus, we want the ability to say

Define linguistic philosophers: author of A quality works B-relevant to linguistic philosophy

Then, questions like

Works by linguistic philosophers about computational linguistics?

become convenient. This provides not only a shorthand form of expression (which is itself quite valuable), but a simple manner of creating and using new concepts. After the above definition, the user will ask questions involving "linguistic philosophers" without recalling the detailed meaning he has assigned to that term. Thus, the term acquires a life of its own, with a compatible meaning to both user and computer (assuming the definition was made well). Here, new high level concepts, easily introduced by the user, are immediately available for further use. This is a very important capability of any good high level language, and is strongly supported by the REL system.

This completes a rather informal descripof the BIBLIO language, which will be used as an example to illustrate techniques of language implementation. Undoubtedly, one could discover some aspects of bibliographic reference not adequately treated in this elementary language, or treated differently from one's own preferences. To

objections on the basis of preference, we can only point to the comments about specialization made at the beginning of this discussion -- BIBLIO is not a general language to please everyone. Other languages, to some extent similar, but making different choices in design, are possible and desirable for other users, and to the extent that they are similar, they will share implementations with BIBLIO. Omission of further detailed capabilities is (mostly) justifiable by considerations of space.

This specialized language allows communication at a high level and provides its user with a convenient, reasonably flexible and powerful facility for keeping track of his personal bibliography. It is illustrative of the notions we have presented for specialized languages, and should give a flavor of what they may be like. We will now present the computer system and metalanguage in which such specialized application languages can be implemented.

CHAPTER IV

THE METALANGUAGE: UNIVERSE OF DISCOURSE

> Specifying the universe of a theory makes sense only relative to some background theory, and only relative to some choice of a manual of translation of one theory into the other.
>
> -- W. V. Quine [1969, p. 54]

If we may substitute "language" for "theory" in Quine's relativistic prescription, the above is an outline for the program of the next two chapters of this thesis. We are interested in the implementation of new, specialized languages, and we will present the "manual of translation" into the REL Language Writer's Language, which will serve as our "background" language, or metalanguage.

Within the framework developed in Chapter II, a language consists of two parts: a data representation which defines what types of objects can exist in the language's universe of discourse (called data structures) and what operations may be performed on them (called

functions);* and an extended syntax which defines the meaning of each phrase of the object language in terms of the data representation.

To specify data structures and functions, LWL uses a style drawn from familiar programming languages, and its syntax for statements and type declarations follows, wherever possible, Wirth's definition of the language PASCAL [1973], extended and modified where necessary to our needs. This forms the subject of the current chapter. PASCAL provides no help concerning the definition of languages; thus, LWL introduces its own capabilities there, using the flavor of the Backus-Naur formalism (BNF), greatly enhanced. The related LWL facilities are presented in Chapter V.

The discussion below is concerned with those areas in which LWL differs from PASCAL, and considerable attention is devoted to motivating and discussing LWL's novel features. Therefore, this presentation is topical rather than complete, and for the purposes of exposition it assumes the reader's familiarity with the PASCAL language. When not stated otherwise, LWL includes exactly the corresponding features from PASCAL. A complete syntactic description of LWL appears in Appendix A.

- - - - - - - - - -

*Note that the data representation contains both data structures and their operations. Thus, it includes most of what is in a normal programming language.

4.1 DECLARATIONS

Data are represented by constants and the values of variables, each of which must appear in a variable declaration which associates a data structure (data type) with the variable. The data values are constants, where each data type defines the set of constants which variables of that type may have as values.

4.1.1 BASIC DATA TYPES

The basic data types are the scalar types. Their definition indicates an ordered set of values, each of which is (implicitly) defined to be a constant of that type. In addition, there exist four standard scalar types: Boolean, with the constant values False and True; char, with the constant values corresponding to the host computer's character set; integer, with constant values corresponding to the allowed range of integer arithmetic; and real, with the constant values corresponding to the allowable values of floating point arithmetic on the host computer.

PASCAL includes a subrange type, which defines a type whose constant values are a consecutive subset of the values of some other type. The use of subrange as a type has been severely criticized by Habermann [1973, p. 50]. LWL retains the form of the subrange, but merely as an expressive mechanism by which the language writer can

concisely indicate not only the basic type but also a range of values within that type for a variable. Thus, if we say a variable is of type 1 .. 256, we will really mean that it is of type integer, and needs storage only sufficient to distinguish 256 values. No commitment is implied to enforcing the bounds at run-time.*

4.1.2 STRUCTURED DATA TYPES

Structured types are defined by describing the types of their components and by indicating a structuring method. LWL supports three of the PASCAL structured types, array, record and set,** with some major differences involving the REL System's conventions on data space allocation and persistence. Principally, a record type definition may

- - - - - - - - - - -

*This does not satisfy Habermann's objection in full, since it leaves unanswered the same question that is raised by the PL/1 program segment

```
DECLARE I FIXED BINARY(31),
        J FIXED BINARY(15);
. . .
J=I;
```

What is the value of J if I was greater than 2^15-1? The question is sidestepped by most implementations, which is exactly the strategy we pursue here. Admittedly, this does violence to Wirth's desire for clarity and transparency, but the run-time discipline required to enforce value bounds is a cost we are unwilling to assume. Notice, however, that a host computer of the Burroughs 1700 type, with variable word length, would find such bound checking (at least to the nearest power of two) quite straightforward. Also, a "debugging" feature like PL/1's SUBSCRIPTRANGE or STRINGRANGE could be defined for explicitly requesting a check on subrange boundaries.

**The file structure type is not implemented, as justified in the section on data handling and paging, below.

include a storage class specification, indicating how this structured type may be represented in the REL underlying data structures. The three storage classes are stack, corresponding to an ALGOL-like dynamic run time stack; list, referring to a rather specialized garbage collected heap; and page, which corresponds to space in a random access file system managed as a virtual memory.

Pointer types play a relatively more important role in LWL than in PASCAL, because nearly any non-temporary datum (i.e., anything which persists in the universe of discourse of the language) must be referenced through pointers. The pointer is an object with strict limitations in LWL. The target of a pointer must be a record type which has an associated storage class specification. The pointer is said to be bound to its target type. The pointer can also be identified with the storage class of its target, and we will refer to list pointers, stack pointers, and page pointers. A pointer variable may also have the value nil, which is a valid constant of all pointer types.

Partly in recognition of the above limitation on pointers, we introduce another abbreviative convenience like subrange, the subtype pointer. For example, we allow

```
type publication_page = page record
    publication_data : array [1..num_pubs_per_page] of
        record
        title : title_string;
        num_auth : 0 .. max_num_auth;
        auth : array[1..max_num_auth] of author;
        . . .
        end
    end
```

```
category publication =
    @ publication_page.publication_data[*];
```

The second definition makes publication a subtype, defining it as a pointer to one of the elements of publication_data in a publication_page.* Notice that this mechanism, in essence, returns to LWL the flexibility in the use of pointers which is apparently removed by the strict limitation on pointer targets introduced above. Thus, any data type may be the target of a subtype pointer, but only if it is contained in a storage classed record type. This guarantees that the compiler has sufficient information about the target of a pointer so that it may generate the proper code to access the data which is referenced. This is especially useful when a collection of objects (e.g., the above publication_data structures) are to be treated both as individuals and as a group.

4.1.3 TYPE CHECKING

Type checking is very strong in LWL. The strict enforcement of typing hierarchy is carried to rigorous extremes. Every appearance of any anonymous type is a unique type. Though there is some doubt about Wirth's definition of the equivalence or compatibility of types, PASCAL

- - - - - - - - - - -

*The traditional alternative to this scheme would be to represent a publication by a pair, [publ_page_pointer, publ_index], in which case a reference to a publication would appear as: publ_page_pointer@.publication_data[publ_index]. The subtype pointer merely incorporates the index as part of the pointer.

generally assumes that any two types defined identically are the same type. Thus, the program segment

```
var a : array [1..50] of char;
    b : array [1..50] of char;
. . .
a := b;
```

defines one anonymous type (array [1..50] of char) and considers a and b to be variables of the same type.

LWL attempts to force the language writer to structure his data definitions hierarchically; therefore it considers two anonymously defined types to be distinct, and the assignment a := b, above, is a type fault error. To achieve the intended effect, one would code

```
var a, b : array [1..50] of char;
. . .
a := b;
```

In even better style, one would name the defined type, which indeed stands for a meaningful concept -- in this case, "string of characters in a title":

```
type title_string = array [1..50] of char;
var a, b : title_string;
. . .
a := b;
```

This preferred style gives an identifier, title_string, to the data type being used, so that it can be referred to elsewhere in the program without any need to know its exact definition.

This rigid typing convention means that every type which is used must be explicitly named and declared. This should improve program

correctness and understandability, because the translation of a given type into the LWL data structures can be defined in only one place. In the above example, the programmer is prevented from being sloppy by reflecting his knowledge of the physical structure of title_string in every place that he defines a variable of that type. To be consistent with this point of view, even the common stylistic aberration

```
const title_string_length = 50;
type title_string_selector_index = 1..50;
type title_string = array [1..50] of char;
```

should be written instead as

```
const title_string_length = 50;
type title_string_selector_index =
    1..title_string_length;
type title_string =
    array [title_string_selector_index] of char;
```

We encourage, by this limitation, the extensive use of constant and type definitions, which are part of a general and powerful definition facility for LWL.

4.2 DEFINITIONS

The user of a computer language often needs to introduce new notations and concepts into the language for his own convenience. These may range in complexity from wanting to write "pi" for "3.14159" to the desire to introduce fairly complex new functions under a new syntax.*

- - - - - - - - - -

*Note that this process is orthogonal to the language writer's basic task, to define the syntax and semantics of an object language. We address here only extensions which the language writer makes to his metalanguage for his own use. These may well have little apparent

The definition of new data types is such an extension, of intermediate complexity. LWL provides a powerful, general definition facility, which is appropriate for the PASCAL-like definition of new constant values and data types, but also allows the introduction of new syntactic constructs of the metalanguage.

4.2.1 THE define STATEMENT

LWL's general definition mechanism is of the form

define definiendum = definiens;

where the definiens is a meaningful phrase of LWL,* and the definiendum is an arbitrary string of characters not including the "equal sign" (=). Loosely speaking, this statement defines the definiendum to be a valid paraphrase of the definiens. For example, in its simplest use, definition may become a mechanism for introducing abbreviations:

```
define (! = begin;
define !) = end;
```

These definitions introduce the abbreviation which permits compound statements like

- - - - - - - - - -

relation to features which he is building into the object language. The relation of metalanguage extension to object language construction will be taken up again in the final chapter.

*What is a meaningful phrase of LWL, is determined by the grammar by which LWL is defined (see Appendix A); this is also the manner in which the PASCAL Report defines PASCAL. Thus, strings like "array [1..8] of" or "b :=" are not valid, as they have no meaning in LWL, whereas "a+3" is a <simple_expression> and "var a:integer" is a <variable_declaration>. This is an indication that the definitional extension mechanism of LWL is context free.

(! s1; s2; . . . sk !)

This is the most trivial, string-replacement type of definition.

The processing of definitions is, however, different from macro-style text replacement. The definiendum is bound to the structural analysis of the definiens as it is parsed at the time of the definition. This makes LWL definitions more like the macro evaluation facilities of the Vienna Definition Language [Wegner 1972] than like the operations of a general string processor. Thus, the meaning of a definition needs to be analyzed only at the time of the definition, potentially saving extensive parsing for commonly used defined constructs. On the other hand, definitions are not completely "compiled", since the meaning of other defined terms appearing in the definiens and the binding of metavariables (see below) are not determined until each use of a definition. This latter allows chains of definitions which adjust correctly when some definition in the chain is altered.

Both definiendum and definiens may contain metavariables, which are of the form

"phrase"

where phrase is a meaningful phrase of LWL. The metavariable stands for an arbitrary entity of the metalanguage which is of the same kind as the phrase between the quotation marks. For example, "5" is a metavariable for <unsigned_constant>s, "a < 8" is a metavariable for <expression>s, and if publication_type has been defined by

```
type publication_type = (book, article, journal, report,
    proceedings, manual, dissertation, collection);
```

then "publication_type" is a metavariable for scalar types.

The above technique allows these metavariables to be defined by example. The phrase between the quotes (") is parsed according to the grammar of LWL (Appendix A), and the first non-terminal which spans the whole phrase is taken as the part of speech of the metavariable.

A metavariable has a part of speech, and it may also have a range. This allows the use of metavariables to stand for selected subcategories of the LWL parts of speech. For example, given the above BIBLIO definition for publication_type and a declaration

```
var which : publication_type;
```

the metavariable "which" has part of speech <variable>, and range publication_type. Thus, a definition involving "which" will be applied only under the constraint that the actual LWL variable to be bound to "which" will have been declared as a variable for publication_type. This is the only use of range made by LWL; the general facility may be important in the implementation of object languages, and is described fully in Chapter V, in the section VARIABLES AND BINDING.

Alternatively, constructions like "<constant>" or "<expression>" allow the more knowledgeable language writer to use his understanding of the LWL grammar to define metavariables explicitly, with less chance of error and confusion. Metavariable definition by example is, however, usually sufficient to achieve the intended result.

The metavariables of the definiendum are bound to the corresponding metavariables of the definiens, as in a lambda calculus. They are variables in the logical sense, having a type but no specific value. They are bound at each use of the definition.

4.2.2 CONSTANT, TYPE AND CATEGORY DEFINITIONS

For consistency with PASCAL, the const and type definitions are retained as special cases of define. They must not contain the use of metavariables and must in fact define constants and data types, respectively. For example,

```
const title_string_length = 50;
type title_string_selector_index =
    1..title_string_length;
type title_string =
    array [title_string_selector_index] of char;
```

could be written equally well as

```
define title_string_length = 50;
define title_string_selector_index =
    1..title_string_length;
define title_string =
    array [title_string_selector_index] of char;
```

An additional form for data type definition exists. The sentence

category <identifier> = <type>

declares not only a new data type of the LWL, but also identifies this data type as the representation of a semantic category of the object

language being defined. The identifier is then used to name the three associated things: the LWL data type, the object language semantic category, and the corresponding object language syntactic category (part of speech). For example, to define the category subject of the BIBLIO language, we write

```
type subject_data = page record
    subject_name : subject_string;
    . . .
    end;
category subject = @ subject_data;
```

This is the method by which the object language's parts of speech are introduced.

4.2.3 METALANGUAGE EXTENSIONS

The general form of the define statement gives the language writer considerable power to tailor LWL to his needs. In this section, we take up two difficult but common questions of extensible languages: the use of parameterized types and functional extensions of the metalanguage.

The use of metavariables is handy in aggregating LWL constructions which are not identical, but differ only in detail. In the PASCAL Report, for instance, string is defined [p. 9] as

```
packed array [1..n] of char
```

The upper limit of this array dimension, n, appears to be an arbitrary

number, and in fact no further mention is made of it. Wirth holds back from actually claiming that string is a type, for in PASCAL n would then have to be fixed for all strings. In fact, hidden additional mechanisms of the compiler permit the use of string constants so that n comes out appropriate for each instance, but this is much more complex than the Report indicates. Certainly, string is not like any other type definable in PASCAL. In LWL, one states instead

```
define string("7") = array [1.."7"] of char;
```

This definition is not prettier,* and it does not solve all of the problems, but at least it faces up to the fact that PASCAL (as well as LWL) does not have facilities for arbitrary length structures, and thus there can be no real type string. One might use the above definition, instead, to isolate the various string-like types that will be introduced, from the detail that they will be implemented as arrays of characters. Thus, we might rewrite the definitions of title_string and subject_string as

```
type title_string = string(title_string_length);
type subject_string = string(subject_string_length);
```

The lack of varying length structures is one of PASCAL's most controversial features, because it fails to make full use of the dynamic

- - - - - - - - - -

*The syntax "string(8)" is arbitrary. It could equally well have been "string.8", "8-string", "stri..8..ng", etc. One of the advantages of a general definition scheme is that it will accomodate rather peculiar syntactic forms, when desired.

storage allocation mechanism included in its definition. Further, it necessitates the arbitrary imposition of maximum array sizes, set at compilation rather than execution time. This is a drawback routinely overcome in contemporary programming languages, and it is annoying and overly rigid. This limitation also exists in LWL, but it is a less arbitrary restriction because a priori limitations on the maximum size of contiguous data, imposed by the REL System's data management schemes, already effectively exclude the possibility of arbitrary sized data structures.

The apparent incompatibility between strong type checking and variable structural parameters is a common one, which LWL does not escape. In the PASCAL Report [p. 37], Wirth writes a deceptively nice piece of code, for the function Max, whose arguments are a vector and an integer (the length). The implication is that Max is indeed a general function to compute the maximum value of a vector of reals. Yet, because the type vector has had to be declared as an array with some particular length, under strong type checking the function Max will only work for arguments of type vector; thus only for real arrays of that particular length.

Languages which do allow variable length structures do not completely overcome this difficulty either. For instance, a related problem plagues EL1, which defines string [Wegbreit 1970, p. 180] quite generally and legitimately as

```
DECL string : mode;
string <- ROW(CHAR);
```

The length of this row of characters is not bound, and string is said to be length unresolved. This leads to some difficulty, because the similar mode string8, defined by

```
DECL string8 : mode;
string8 <- ROW(8,CHAR);
```

is not equal to string. Apparently, the declaration

```
DECL a : string;
```

is not permitted without a modifier that fixes the string length. Typically, this will be the BYVAL modifier which declares a to be the argument of a PROC, in which case the mode is length resolved by the actual parameter. However, since the length resolution will generally not occur until the PROC is invoked, the code segment

```
DECL a : string;
DECL b : string8;
. . .
a <- b;
```

leaves unclear whether an implicit type conversion will be required. It will generally result in a type fault failure, unless the programmer has had the foresight to define conversion routines among strings of arbitrary lengths.

It is possible to overcome these problems by loosening the requirements of type checking to exclude length as a type differentiator. This, however, forces recourse to runtime checking of array bounds, and violates some intuitive sense that arrays of different

lengths cannot be the same type. Perhaps this difficulty can be resolved by making explicit the hierarchical relationships among types and the structure-determining "hidden bindings" across procedure calls, as suggested by Ingargiola [1974]. In any case, LWL does not implement varying length arrays.

Notice that the definition facility is more widely useful than in type definition. For example, it subsumes part of the operator extension facility which is typically included in extensible languages. To define a simple new operator on constants, for instance, one codes

```
define "1" % "2" = ("1"+"2")*("1"-"2");
```

Later, this could be used in

```
const min_bound = 1;  param = 5;
const max_bound = param % min_bound;
```

Similarly, given the above definition of title_string, we can write

```
var a, b : title_string;
define ::"a" =
    max(ord("a"[1]), ord("a"[title_string_length]));
```

which defines a rather odd metric on title_strings, the maximum ordinal of their first and last characters. This newly defined construct is then immediately available for further use, and

```
while ::a = ::b do . . .
```

becomes a legitimate statement of LWL.

Further sophisticated uses of definitions might be considered, but they may often surpass the limits of what are desirable extensions

of LWL. In the implementation of the BIBLIO language, some data type which represents a list of publications will be useful in computing answers to queries relating to publications. Ordinarily, when we code functions which compute with such a type, the syntactic forms of reference employed to access elements of the list predetermine, to a large extent, the actual data types. This means that the functions are difficult to change in significant ways, because many matters related to data access are scattered throughout the code. Given a typical definition of such a list,

```
type list_of_publications =
    array [lop_index] of publication;
```

a variable of type list_of_publications is probably referenced in many places in the program as "a[i]". If it should become desirable to change the representation of list_of_publications to, say, a true linked-list structure, either many of the functions referring to variables of that type would have to be rewritten, or we must extend the metalanguage to accept the array form of reference for lists of publications which are no longer represented in that way. One may be tempted to consider something like the following:

```
type list_of_publications = @ pub_list;
type pub_list = list record
        link : list_of_publications;
        pub : publication
        end;
```

```
var a : list_of_publications;  i : integer;

define "a"["i"] = if "i" = 1
        then "a"@.pub
        else begin
            var x : list_of_publications;
            var j : integer;
            j := "i";  x := "a";
            while j>0 & x@.link <> nil do begin
                j := j-1;
                x := x@.link
                end;
            x@.pub
            end;
```

This would not be at all a satisfactory solution. Note that in the code compiled for a function with references to an "a[i]" such as above, this definition would be expanded for every occurrence, and the defined program fragment would be inserted at each use. Even if the compiler were sufficiently clever to aggregate each of the expansions in a common subroutine, the algorithm is still costly. This reflects a common cost disadvantage of extensible programming. Further, every phrase of LWL would have to be able to return a value (including a so-called "left value", as the target of an assignment), to allow the functional expression of definitions, as above. Many languages take recourse to such a policy: In LISP, everything is an S-expression with a value; in EL1, everything is a FORM with a value. For us, such a decision would wreak havoc on the grammar of LWL -- every compound statement would need to carry additional syntactic information about the type of its value, and we would become mired in the treatment of generic mechanisms and related type-dependent processes.

A better technique for avoiding this costly alteration is to view the meaning and responsibility for computing "the i'th publication on a list" as a part of the object language, where it can be implemented once. Any change in list_of_publications will then reflect back to the LWL only in the data types and functions which implement that particular part of the object language. Thus, we write*

```
define "a"["i"] = publication_selector("a","i");

function publication_selector
    (listp : list_of_publications, count : integer)
    : publication;
    begin
        var counter : integer;  counter := count;
        var list : list_of_publications;  list := listp;

        while counter > 1 & list@.link <> nil do begin
            counter := counter - 1;
            list := list@.link
            end;
        return list@.pub
        end;
```

Note that although this function may be equally useful and available to both the language writer and the eventual language user, the above definition commits only the language writer to this method of invoking the function. The object language construct for which this function is the semantics may be of the form

<number>th <publication>

as in

15th work by Quine about logic

- - - - - - - - - -

*Recall that a and i are as above. Thus, the syntax defined here will only apply when array-type selection is attempted on a list_of_publications.

(if, for instance, list_of_publications were chronologically ordered), or the function may not be available in the object language at all (for example, BIBLIO does not contain this particular construct). In general, the proper structuring of the object language's implementation will place fewer demands on the extension mechanism of the LWL, and will greatly improve the efficiency of the compiled code.

Definitions extend and specialize LWL by introducing concepts and terminology which are relevant to implementing the specialized language under consideration. Recall that the definitions do not become part of the new language; they merely extend the metalanguage for the convenience of the language writer. Definition is, however, a very powerful mechanism, one of LWL's major strengths.

4.3 PROGRAMS

LWL does not support the creation of programs, as such. In its top level structure, it is heavily influenced by the requirements of the REL language processor which supports the specialized languages created by LWL. The most notable omission in LWL is the lack of statements to declare or invoke procedures; the most radical change is in the syntax and semantics of function invocation.

4.3.1 MISCELLANEOUS DIFFERENCES FROM PASCAL

In the denotation of variables, intermediate field identifiers and pointer references may be omitted when no ambiguity is introduced thereby. For instance, given the following definition of BIBLIO's author and q_author categories,

```
category author = @ author_entry;
type author_entry = page record
    name : name_string;
    num_pub : 0 .. max_num_pub;
    pub : array[1..max_num_pub] of publication
    end;
category q_author = @ author_list;
type author_list = list record
    next : q_author;
    this : author
    end;
```

and

```
var x : q_author
```

then the variable designator

```
x.pub[1]
```

is an abbreviation for

```
x@.this@.pub[1]
```

This form of abbreviation is especially useful when referring to some part of the complex list structures maintained by the language processor.

For the convenience of the language writer, the for statement has been expanded and modified. PASCAL's to and downto constructions need not be distinguished, and a new form exists for allowing an

iteration increment other than one. New forms also exist to express iteration over all values of a scalar type and all elements of a list. Thus,

```
for type = all publication_type do . . .
```

is equivalent to

```
for type = book .. collection do . . .
```

given the definition of publication_type introduced earlier. Also, if v2 is a pointer to a record which includes another pointer to that record type (typical in a singly linked list), then

```
for v1 := all v2 do S
```

is equivalent to

```
v1 := v2;
while v1 <> nil do
   with v1 @ do begin
       S;
       v1 := <the next pointer value for v1>
       end
```

In the specification of relational operators, a minor change is introduced. Knuth has pointed out the desirability of performing numerical comparisons on pointers, because an arbitrary (but consistent) ordering allows certain sophisticated algorithms to be more efficient than if only equality or inequality between pointers were determinable [1973, p. 2 of the letter to Hoare]. In response to this observation, the relational operators <, >, <=, and >= also apply between pointers of the same type (i.e., the same target type).

4.3.2 FUNCTIONS

The fundamental program unit to be created in the metalanguage is the function. Inputs to a function are called its constituents, and its result is called its value; the types of both are determined from the rules of grammar which mention the function as their semantic function. Functions are normally evaluated by the language processor in response to its analysis of a user's sentence; thus, the constituents and value of a function are represented in a structure called a phrase marker, which is a tree representation of linguistic and computational information developed by the language processor.

The elimination of procedures and the redefinition of functions both result from the discipline imposed on evaluation by the REL System language processor. In a syntax directed interpreter, the user's sentence is analyzed according to the grammar of the object language, and interpretive functions corresponding to each of the applied grammar rules are invoked to compute the "meaning" of the sentence. According to the linguistic model which is the basis of the REL language processor, every rule of grammar represents a meaningful operation of the object language, and its corresponding function defines the matching computation on the language's universe of discourse. Functions are composed by the composed application of rules of grammar in the analysis of the user's sentences. Functions may also be called by other functions, to permit a hierarchical composition of algorithmic tasks.

In that case, we permit a slightly altered syntax from PASCAL's for the function reference.

In addition to the forms which correspond to PASCAL's function designation, we also allow a function to appear as a selector on its first constituent. For example,

```
f(a,b,c)
```

is equivalent to

```
a.f(b,c)
```

This is done so that the distinction between reference to an item in a structure and the functional computation of an attribute of a record type object may be deliberately blurred. For instance, if we define

```
type complex = record rp, ip : real end;
var cx : complex;
function norm (c:complex; real); . . .
```

then LWL will allow the references

```
cx.rp
```

and

```
cx.norm
```

to appear identical. This is desirable, because in referring to the norm of a complex number, there is no need to distinguish in the form of the reference between the above method of computing the norm at every reference and the alternative strategy of storing its value in the representation of the type:

```
type complex = record rp, ip, norm : real end;
```

This syntactic usage recognizes that a selector is a function just as any other operation which computes a value from a record, and it unifies the syntax of all function references. This is a minor part of SIMULA 67's class concept [Dahl 1968].

4.3.3 THE PHRASE MARKER

The phrase marker, mentioned above, is the universal structure of the REL language processor. Space considerations prevent a complete description here, and the reader is referred to [Thompson 1974b] for a full treatment. For the purpose of describing LWL, the following condensation is presented.

The phrase marker, as its name implies, is the semantic marker developed by the language processor for each phrase recognized in the user's sentence. The phrase marker is a tree of interlinked phrase and phrase_information (abbreviated pi) records, all of the storage class list. The phrase record contains the part of speech and features of its phrase and a pointer to its pi record. The pi can represent either "data" or various structures which determine the way in which the phrase's value can be computed. If the pi is a "data" type, it either is or points to an object of the type defined with the phrase's part of speech in a category definition.

A phrase may have a pi of the cases rou or gen, in which case it

specifies the address of a function to be invoked to compute the phrase's value, a pointer to the first of a list of phrase markers which represent the constituents of the phrase, and a variable list which specifies the binding of metavariables which needs to occur when the function is invoked. The effect of evaluating a phrase is to replace its "structure" pi by the "data" pi of its value.

The pi also has cases def, amb, var and out. A def pi includes a pointer to the phrase marker which represents the value of the defined phrase and a variable list which specifies any bindings of metavariables between the definiendum and definiens. An amb pi represents a phrase whose value is ambiguous, and contains a pointer to the first of the various phrase markers which represent its ambiguous meanings. A var pi identifies its phrase to be a metavariable; it includes the metavariable's name and a phrase marker which defines its range. The out pi is a message to be output to the user; it is ordinarily the value of a <sentence> phrase.

The phrase marker of a sentence is a recursive tree of phrases and their pi's, where the leaves of the tree contain only "data" pi's or pi's which invoke functions with no arguments.

The passing of parameters between functions and the language processor and among functions is by reference to the phrase marker whose rou or gen pi specifies the function invoked. Within the invoked

function, the built in function constituent returns a pointer to the root phrase of that phrase marker. Although the above implies that the parameter of every function is in principle the same, in fact one (ordinarily) knows at compilation time the parts of speech of each constituent phrase in the phrase marker, and therefore the type of its "data" pi. The non-terminal phrases of the grammar rule which selects a function as its semantic representation determine the types of the function's constituents, or these types are specified in an explicit declaration.

In the case that a function is defined with an explicit parameter list, the parameters must be declared with types that are categories, and the given names then refer to the corresponding actual constituent phrases when the function is invoked. If no explicit names are given to the parameters, then the category name (as used in the rule statement) may be used as a function on constituent to select a pointer to the appropriate phrase. For example,*

```
constituent.subject
```

In either manner of naming constituent phrases, the language processor guarantees that formal and actual parameters will match in type. Usually, this is possible to check at compilation, but for functions invoked with parameters in the wrong order, or from rules involving

- - - - - - - - - -

*If the function has several <subject> constituents, they are called constituent.subject(1), constituent.subject(2), . . . Any non-existing constituents yield the pointer nil.

transformations or from more than one rule, run-time binding is often necessary.

LWL does not require that the names of variables be fully specified; even referenced variables may have omitted intermediate names. Therefore, the language writer can refer to the value of a phrase without detailing the path through the phrase marker which must be selected to reach the value. For instance, the variable

constituent.publication.title[1]

is the first character of the title of the first <publication> constituent of the current function.

LWL has functions and prefix functions, corresponding to rou and gen pi's. Before a function is invoked, its constituents are evaluated, but no such evaluation occurs before invocation of a prefix function. Thus, prefix functions must refer only to structural components of their phrase markers, or use the built in function evaluate to evaluate any of their constituents, as desired. The tag gen for prefix functions is indicative of their most common use, as generators over metavariables.

4.3.4 THE SCOPE OF VARIABLES

One further significant difference between LWL and PASCAL is in the scope of variables. LWL has no notion corresponding to PASCAL's program, because the overall organization of "program" execution is

determined dynamically by the grammatical interpretation of the user's statements. The usual nesting of procedures and functions which determines the structure of a program in PASCAL gives way to a more variable structure of function interactions imposed by the grammar of the object language. Therefore, a deep nesting of function declarations is unusual, warranted only when subproblems of a function's implementation are identified in the implementation as separate functions, but are not in the grammar as meaningful primitive operations.

LWL's rules of scope reflect the strong modularity implied above. All identifiers (and, more generally, all definitions) defined or declared in a function declaration are local to that function, but not to any functions declared within them. Thus, the scope of identifiers does not extend to contained functions, as in PASCAL. It is possible to declare variables or to define various constructs outside any function declaration, in what is called the global environment. The scope of such global declarations and definitions is universal; i.e., they apply both in the global environment and within every function declaration. Side effects of functions are, therefore, limited to the manipulation of variables declared globally.

4.4 DATA ALLOCATION, PERSISTENCE AND ACCESS

The semantics of a programming language are primarily based on its innate capabilities for handling data in various forms. As foreshadowed in the above discussion of storage classes, LWL has some fairly peculiar innate capabilities and limitations in its data allocation and access. These are the results of decisions adopted during the development of the REL System and they are currently so deeply ingrained in the design and objectives of languages operating in or contemplated for the system that LWL retains them even in the face of some considerations to the contrary.

4.4.1 THE USE OF REL

The following discussion will be clarified if we begin by outlining the manner of use of application languages in the REL System. REL is, itself, a semi-permanent entity consisting of a load-module which contains the resident parts of the language processor, utilities and an interface to the operating system of the host IBM 360 or 370 computer, and several data sets which represent the total virtual memory resources of the REL System. Such a system is initially generated and then exists indefinitely. A terminal session is the total processing performed during a single execution of the REL System, which corresponds to the contiguous time that a single user spends at a terminal,

interacting with REL. A user, when beginning a terminal session, is connected to the REL Command Language [Gomberg 1973], in which he can create, delete or invoke for use (enter) those versions to which he is allowed the proper access.

A version is a particular instance of a specialized language, along with all the data which have been incorporated into it. Thus, our example, BIBLIO, is a language, but an instance of the BIBLIO language to which bibliographic information about parsing and related subjects has been added (perhaps called PARSING BIBLIO) is a version, which we say is based on the language BIBLIO.* A version occupies a certain amount of the REL-controlled disk space allocated to it, as needed, by the REL System. The source programs which define a version in LWL, the compiled machine code which implements the language's functions, and any data assimilated by those functions into the permanent data base are all part of the version.** A typical terminal session will appear like this:

```
REL - LOGON PLEASE
>pete
REL COMMAND LANGUAGE - PROCEED
>enter parsing biblio
PROCEED
```

- - - - - - - - - -

*Actually, a language and a version are identical, from the system's viewpoint. In use, often a language is a version which has no particular data associated with it; it serves as a base version, from which multiple versions may be created.

**Currently, the process of basing a version on another involves a full, actual copy. This is not always necessary, and the source and compiled code, and even some common basic set of data might be shared among versions if the necessary protection and access mechanisms were added to the REL System. For instance, in BIBLIO, several colleagues may want independent versions of the language, but all starting with a shared common set of references.

```
>articles by Wegbreit?
. . .
>exit
REL COMMAND LANGUAGE - PROCEED
>exit
THANK YOU
```

The segment of interaction between "enter . . ." and the succeeding "exit" is called a session. (Note that, therefore, a single terminal session may contain several sessions, if several different versions are entered or even if the same version is entered several times.) Each version persists until it is specifically deleted. Thus, any data base or context accumulated during interaction with the user may carry over into succeeding sessions, if the language writer has provided for that. Typically, the data base will be permanently kept with a version, allowing the user to exit a session and continue without major difficulty at some later time, but information about the local context of interaction (e.g., the referents of anaphoric references) is dropped at the end of each session.

This style of interaction naturally introduces three time scales of data persistence, which correlate with the three storage classes of LWL. Of these, the simplest and most temporary is the stack storage class, which provides data objects which are created at the entry and destroyed at the exit of each function. The most permanent is the page storage class (and the associated version common area), which persists for the duration of the version. Intermediate to these is the list storage class (and the associated version global area) whose persistence

is the duration of a session. The list and page storage classes need further discussion, following.

4.4.2 LIST PROCESSING

The list storage class consists of data items allocated by the built-in function new and recovered by a garbage collector. Any list structure is guaranteed to persist as long as any reference is made to it by a pointer variable in the scope of the function being evaluated or its ancestors (i.e., any stack variable in the current or invoking functions, including the parsing graph, and any variable in the session global area).

The form of list data items (elements) is rigidly constrained by two conventions: every list element is of fixed length, and the first byte of each list element must be a variable of type char, the value of which defines the structure of the remainder of the record. This information is used by the garbage collector to determine which fields need to be "chased." If the language writer's definition of a list record type does not include the necessary initial char variable, it is added by default and initialized, on allocation, to a value appropriate to the structure of the record.

4.4.3 PAGING

Considerable evidence has accumulated indicating that natural language performance involves an interaction with data in the language's universe of discourse. We take this view of specialized computer languages as well: that the universe of discourse is an integral part of a language. This implies that data is not something kept in a "file structure" and processed by an "input/output system" to make it available to an operation, but is innately and intimately tied to the linguistic and operational constructs of the language.

Consider the implications of the above for "programming" in a specialized application language like BIBLIO. In a typical programming language, say FORTRAN, the user-introduced symbols of an instance of the language (namely, a program) are all variables, with the responsibility for associating meanings to these variables by input or computation resting with the user. In BIBLIO, by contrast, essentially every "identifier" introduced by the user acquires a meaning at the time of its original appearance, either as a reference to a particular object of the user's universe of discourse (e.g., MIT Press), or as a method of determining some objects (e.g., the definition of "linguistic philosopher"). LWL's mechanisms for data manipulation are specifically designed to reflect this style.

LWL supports no input and output operations, per se. All

communication between a specialized language running under the REL System and the external world is through the language processor (described in the next chapter). Instead of access to files, LWL provides a very large, directly addressable virtual memory, which is a permanent part of every version.*

Several requirements and constraints have colored the development of REL's and LWL's handling of virtual memory, and the resulting mechanism has some unusual characteristics. The conventional virtual memory systems, like BBN's TENEX [Bobrow 1972], rely on a hardware-assisted operating system which automatically translates addresses from a relatively large virtual space into the actual hardware address where the appropriate part of the virtual space resides in main memory. Of course, to make this mapping possible, the operating system must also at times read from a backing store into main memory some part of the virtual space that is newly referenced; it may, at the same time, need to overlay, or write back to the backing store, parts of the virtual space not likely to be accessed soon -- this requires the adoption of a replacement policy, most often some form of the Least Recently Used (LRU) or Working Set (WS) strategies. The transfer of information between main memory and backing storage is in units of

- - - - - - - - - -

*The MULTICS system [Saltzer 1974], which supports access to files, implements that access through its virtual memory management. In REL, each version can access a full 32-bit address space (4,294,967,296 bytes), subject, of course, to the availability of that much direct acces secondary storage.

pages, which are ordinarily large compared to a single basic data item of the computer. (In the REL System, a page is 2048 bytes.)

In principle, the virtual memory mechanism is completely transparent to a program; it gives the appearance that the actual computer is totally dedicated to the single program, with real main storage resources equal to the size of the virtual address space. This is, of course, a handy fiction which works well in the general case, but has serious negative impact on the processing of programs which actually attempt to exploit a significant portion of this resource. The simple exchange of row for column processing of FORTRAN arrays can save as much as a factor of the order of the matrix in the number of page faults generated to compute large matrix problems [Moler 1972]. A colleague has found examples in the processing of relational data primitives where three or more orders of magnitude may be lost in elapsed time by unknowingly running in a virtual memory [Greenfeld 1972]. Similar conclusions result from a study of sorting algorithms [Brawn 1970].

What is of great significance is that in general, the more that an algorithm attempts to take advantage of information it may have about its host computer, the worse is its degradation when that host turns out to be implemented virtually.

The intimations of horrible performance deducible from calculations like those mentioned above have convinced us of the need

for paging, or virtual memory management, which allows rather strict control by the language writer. This control is exercised through the three standard functions lock, release and unlocked, and an extension of PASCAL's with statement. Lock and release both take arguments which are page pointers (pointers to records of storage class page), and return them unchanged, with the side-effect that a page which is locked is guaranteed not to be replaced by the paging system until a release is executed.* Unlocked is a function of no arguments, whose value is the number of actual page frames which are currently available for use and not locked. Further, access to a page record may be made only inside a with statement which controls that record.

Virtual memory management as undertaken by IBM's VS2 operating system postdates the development of REL's paging design and implementation, and fails to satisfy our requirements on two accounts. It is committed to a 24-bit virtual addressing space, which must be shared among all multiprogramming tasks (in the current release), and which is volatile at the termination of every task. We have felt that the non-persistence of the paging space from session to session with a language and the relatively small (and unpredictable because of the presence of other tasks) size of the virtual memory pose serious problems if we were to base our paging on it. Further, the variant of the working set strategy employed in VS2 does not support our

- - - - - - - - - -

*With the exception of intervening error and interruption processing.

requirements for the unlocked, lock and release functions without significant (and impractical) subversion of VS2. Therefore, we have chosen to implement the paging services provided by the REL System as a software layer, using the IBM operating system's standard direct access I/O facilities, and ignoring the paging services provided by VS2.

The decision to implement paging by software has been a very expensive one in design, implementation and operation, because of the need to efficiently invoke the paging services to have all referenced pages in main memory when needed and to mark pages for rewriting to the backing store when they are modified. At the level of machine code, the software processing of paging requires giving up a strict demand paging strategy and incorporating some aspect of paging prediction. To stay with demand paging strictly, the machine code would have to generate extra "instructions" (in fact, subroutine calls) around each data access instruction of the real machine, to insure the presence of the required data. Thus, load would become

```
load_page; load
```

and store would become

```
load_and_mark_page; store
```

This is clearly too inefficient to adopt, since the load_page "instruction," even if no page fault occurs, is one hundred times slower than the load. Thus, something like a flow analysis of the program is required to determine those points at which paging instructions should be placed to efficiently simulate the true demand paging policy.

Fortunately, the control structures (without goto) implemented by PASCAL and LWL are essentially simple flow graphs, and the needed analysis is a direct result of the structure of the language. However, the inefficiencies achievable by paging are so great, that LWL forces the language writer to devote considerable attention to paging by permitting reference to the components of a page record structure only within the context of a with statement, which is extended to perform the required paging operations. This makes with a syntactic marker to indicate the scope of access to the virtual memory.*

Consider the BIBLIO semantic function which relates authors to their publications, based on a user's input sentence. Given the definitions of publication and q_author from earlier examples, the basic input function is:

- - - - - - - - - - -

*Note that this mechanism could also become the basis for an implementation of critical region, if REL were to support the simultaneous use of one version by several people. Such use is currently not supported.

```
function basic_input begin
    var auth_ptr : q_author;
    with publication @ do
        for auth_ptr:=all constituent.q_author do begin
            if num_auth > max_num_auth
                then error 'Too many authors.'
                else begin
                    num_auth := num_auth + 1;
                    auth[num_auth] := this
                    end;
            with this @ do
                if num_pub > max_num_pub
                    then error
                    'Too many publications for author.'
                    else begin
                        num_pub := num_pub + 1;
                        pub[num_pub] := publication
                        end
            end
end basic_input;
```

The explicit use of lock and release supplements the control exercised through the use of with by allowing data paging control not tied to the program's hierarchic structure. In computations where highly efficient interleaved processing of two sets of pages is required (for instance, a merge), these functions allow the simulation of efficient coroutine behavior which is not supported by LWL. Greenfeld's thesis includes several algorithms in which these functions are particularly useful [1972].

This concludes the discussion of that part of LWL which allows the language writer to define the primitive data types of the object language and the functions which operate on them. In this, LWL is very similar to PASCAL. The major semantic differences result from special

considerations introduced by the host REL System's memory management mechanisms and the fundamental framework of the syntax directed language processor. Other differences result from a desire to provide a concise notation to the language writer for dealing with commonly encountered tasks and from the powerful definitional capability which allows the extension of LWL itself.

CHAPTER V

THE METALANGUAGE: LANGUAGE PROCESSING

> . . . a theory of linguistic structure that aims for [descriptive] adequacy must contain*
>
> (i) a universal phonetic theory that defines the notion "possible sentence"
>
> (ii) a definition of "structural description"
>
> (iii) a definition of "generative grammar"
>
> (iv) a method for determining the structural description of a sentence, given a grammar
>
> -- N. Chomsky [1965, p. 31]

Language is now commonly understood as a set of sentences, recursively enumerated by a formal system which determines what strings of symbols are valid in the language; the formalism assigns a structure to each acceptable string and represents its meaning in terms of that structure. The use of this conceptual framework for describing artificial languages has become no more unusual than its application to the linguistic analysis of natural languages (e.g., [Naur 1963]), and considerable success has even been realized in its use for implementing some languages (e.g., [Irons 1961]).

- - - - - - - - - -

*In the section quoted, Chomsky is actually discussing explanatory rather than descriptive adequacy; he includes another requirement, "a way of evaluating alternative proposed grammars." Later [p. 34], he drops that requirement for a descriptive theory. It should be noted that Chomsky is interested in analyzing the linguistic competence of a "native speaker," whereas our interest lies only in using linguistic theory to guide the design of computer languages.

The REL System's Language Writer's Language supports the application of these linguistic techniques to the implementation of specialized application languages. As described in the previous chapter, the REL System is a generalized syntax directed interpreter. In the following sections, we will discuss how it performs the tasks alluded to in Chomsky's list, and how LWL allows the language writer to define the syntax and semantics (the extended syntax) of a particular language.

The basic idea of syntax directed interpretation is that a sentence is decomposed into its constituent phrases, recursively, according to the formal rules of a grammar. The meaning of the sentence is computed by composing the actions of functions which correspond to each grammar rule, as it is applied. This is the typical manner of application of syntax directed techniques, for instance, in many compilers, where the effect of the functions is to generate code which implements the intent of the statement under consideration.

An extension of this idea, presented in [Thompson 1966b] and explored in theoretical terms in [Benson 1968] and [Randall 1970], forms the basis of syntactic and semantic evaluation in the REL System.

5.1 SYNTAX AND SEMANTICS

Perhaps the best description begins with an example. In BIBLIO, we would like to ask the question

Generalization of grammar?

to find the list of subjects which have been stated to immediately include the subject grammar in their extent. (By the examples of Chapter III, these would be syntax directed interpretation and compiling.) Clearly, we would find a language implementation based strictly on simple patterns rather strained, so the rules of grammar will be sufficiently general to accept not only relatively simple queries like the above, but much more complex, composed queries. Below is a portion of the syntax of BIBLIO.

The distinguished symbol of BIBLIO, as of every language defined using LWL, is <sentence>. Thus, one rule will be

<sentence> ::= <query> ?

which specifies that one valid kind of sentence of BIBLIO is a <query> followed by a question mark. Since the answers to queries will be lists of authors, lists of publications or lists of subjects, we need corresponding syntactic rules. For the above query, we require only

<query> ::= <q_subject>

where <q_subject> is a list of subjects. Then, to allow a simple query like*

- - - - - - - - - -

*Recall from Chapter III that a query merely naming a subject asks for a list of all subjects covered by it.

Grammar?

we will have a rule

<q_subject> ::= <subject>

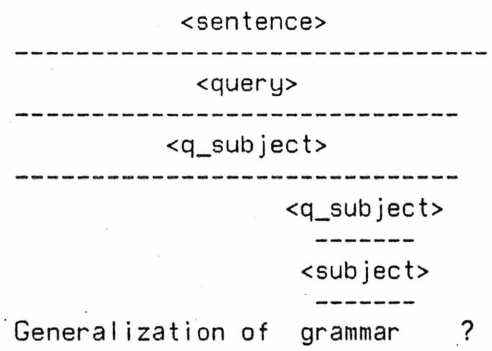
Finally, to define the "generalization" query, we need

<q_subject> ::= generalization of <q_subject>

With the above fragmentary grammar, and assuming a lexical rule corresponding to

<subject> ::= grammar

our example sentence is analyzed in the following form:



Notice that the generality we have introduced creates some extra levels of analysis in the grammar; however, it immediately allows more complex queries like

Generalization of generalization of grammar?
Generalization of generalization of generalization of grammar?
. . .

Although the grammar above determines the structure of our sentence, it says nothing about how its meaning (in this case, a reply to the user) is to be computed. We rectify this by associating with

each rule the name of a function which will carry out the computation implied by the syntactic transformation. The resulting grammar fragment, as actually written in LWL, is below.

```
query_forming rule <sentence> ::= <query> '?' : (print)
subject_query rule <query> ::= <q_subject> : (format_subjects)
generalization rule <q_subject> ::= 'generalization of ' <q_subject>
    : (generalize)
primitive_subject rule <q_subject> ::= <subject> : (single_subject)
```

Each rule is named, so that it may later be referenced for modification or debugging. Text that is literally mentioned in the left or right hand side of a rule (i.e., terminal symbols) is expressed by quoted strings, since spaces in the object language are significant but spaces in LWL are not. The information following the ":" is the semantic specification; in this case, the names of the appropriate functions. These rules have the side effect of specifying the result types and constituent types of the functions they mention. In general rewrite rules, one semantic specification appears for each non-terminal phrase in the left hand side. An omitted semantic specification implies the identity semantic function.

With the above grammar, the meaning of the question

Generalization of grammar?

is

```
print( produce_subjects( generalize( single_subject(grammar) ) ) );
```

The phrase marker which represents this computation is produced by the language processor as a result of the syntactic analysis before any

actual evaluation is attempted; thus, syntactic and semantic processing do not ordinarily proceed in parallel. This may save considerable semantic computation if spurious partial parses can be rejected for purely syntactic reasons before any semantic computation takes place. Also, this convention minimizes the problem of "undoing" which can haunt many syntax directed compilers; having to undo falsely hypothesized actions based on spurious parses is usually avoided by adopting very simple bounded context grammars. As we shall see below, the processing of semantics synchronously with syntax is possible, though not mandatory.

Let us return again to Chomsky's list, and take up the discussion more specifically. His first question, in our context, asks, "What are the strings of symbols to be considered?" The terminal symbols of every object language are the printable characters of the host computer (the EBCDIC printable characters), augmented by the symbols <string_begin>, <string_end>, <input_terminator> and <carriage_return>. <string_begin> and <string_end> are automatically appended around the input string, for the convenience of syntactic analysis. <input_terminator> is a REL-recognized symbol by which the user indicates the end of his input sentence, and <carriage_return> is the end of line.* The complete vocabulary of every language is encoded

- - - - - - - - - -

*Carriage return is ordinarily a normal input character, and a special <input_terminator> is required to initiate processing of an input sentence. In TSO, this is (control)-S. Note that most EBCDIC devices do not have lower case characters or some of the special symbols used in the description of LWL. For LWL, a transliteration similar to that

as objects of the LWL type char; thus, the non-terminal vocabulary is made to correspond to unprintable characters of EBCDIC. These do not overlap the codes for the four characters defined above or the char constants described in the section on List Processing in Chapter IV.

Turning to Chomsky's second question, we understand "structural description" by the description introducing functions in the previous chapter. The structural description is the phrase marker, representing both the sentence's syntactic analysis and the composition of functions required to compute its meaning.

Corresponding to Chomsky's third question, about "generative grammar", we need to define our complete notion of "analytical grammar". The fundamentals of the rule statement appear above; in the succeeding sections, we elaborate the definition of rules by introducing useful extensions.

5.2 FEATURES

The LWL definition of language associates with each non-terminal part of speech (category) a data type. Conceptually, the association is symmetric, and excellent justification exists, in fact, to treat the data type as fundamentally important and the part of speech as a mere representative of the abstract category defined by the data type.

- - - - - - - - - -

proposed by Wirth for PASCAL is followed. In BIBLIO, although example sentences in this thesis use both upper and lower case characters, the grammar is written assuming that all characters in use are upper case.

Current techniques in the syntactic description of programming languages often rely on the invention of new parts of speech to assure the unambiguous parsing of certain phrases. For instance, PASCAL guarantees that

```
a + b * c
```

cannot be interpreted as

```
(a + b) * c
```

by introducing parts of speech <factor>, <term>, <simple expression> and <expression>, each of which may represent objects of the same type. This practice clutters the connection between syntax and semantics.

A similar difficulty arises in BIBLIO. Just as we introduced the category <q_subject> above to represent a list of subjects involved in a query, we need a category <q_publication>. As we defined BIBLIO, the manner of expressing <q_publication>s must be far more flexible than that for <q_subjects>. Indeed, the ability to handle queries like

```
Works by Wegbreit or about parsing and C-relevant to extensible
    languages?
```

requires a reasonably complex syntax. We begin with

```
<query> ::= <q_publication>
```

which is obvious. Since the above query must be treated as an expression, with the possibility of "operators" of different precedence, a grammar like the following is plausible:

```
<q_factor> ::= 'by ' <q_author>
<q_factor> ::= 'about ' <q_subject>
<q_factor> ::= <rating> '-relevant to ' <q_subject>
```

```
<q_term> ::= <q_factor> | <q_term> ' and ' <q_factor>

<q_expression> ::= <q_term> | <q_expression> ' or ' <q_term>

<q_publication> ::= 'works ' <q_expression>
```

It is never good practice to hide knowledge that the programmer uses and depends on; yet, the above does exactly that, by failing to show that each of the phrases actually represents a list of publications. It does violence to a notion of structuring that identifies syntactic with semantic structures.

Features provide a syntactic subcategorization of the categories introduced by the language writer. Each feature is a binary flag which qualifies the part of speech of a phrase. Features may be tested for presence or absence on phrases in the right hand side of a rule; they may be carried over, set, reset or reversed on phrases of the left hand side. Using three features to subcategorize <q_publication>, conjuncted, disjuncted and completed, we rewrite the above rules as

```
<q_publication> ::= 'by ' <q_author> : (works_by)
<q_publication> ::= 'about ' <q_publication> : (works_about)
<q_publication> ::= <rating> '-relevant to ' <q_subject> :
    (works_relevant)

<q_publication,+conjuncted> ::= <q_publication,-disjuncted-
    completed> ' and ' <q_publication,-disjuncted-conjuncted-
    completed> : (q_and)

<q_publication,+disjuncted> ::= <q_publication,-completed> ' or '
    <q_publication,-disjuncted-completed> : (q_or)

<q_publication,+completed> ::= 'works ' <q_publication,-completed> :
    (q_identity)

<query> ::= <q_publication,+completed> : (format_publications)
```

Features to be carried over are represented by a constituent number in the lhs (e.g., +1 means to copy over all features now on the first rhs non-terminal phrase). Checking for the presence of a feature and setting it are indicated by the + sign (e.g., +completed). Checking for absence and resetting are shown by the - sign, and reversing the setting of a feature in the lhs is shown by a *.

If the verbosity of such long descriptions is an objection, the LWL definition mechanism may be used to provide a convenient shorthand. For instance,

```
define q_term =
    q_publication,+conjuncted-disjuncted-completed
```

With such definitions, a language writer committed to a syntax like the first proposed above can write it in exactly that way; however, it is preferred to retain the visible correspondence between syntactic category and semantic data type that is possible through the use of features.

The above example shows how features may be used to simulate a precedence mechanism in the parser. However, features are also useful in many other instances. In natural language processing, for example, they have been used to record essentially semantic distinctions in the syntax, to aid disambiguation by the grammar (e.g., the animate feature, discussed in [Dostert 1972]). Features are somewhat like the mechanism Knuth describes as the semantics of context free languages [Knuth 1968].

They are not as general, because their values are restricted to one Boolean, and their dependence is expressible only in a bottom-to-top direction.

5.3 TRANSFORMATIONS

Formal systems tend to be concise, rigid and minimal. Considerations of human engineering often complicate and extend the structure of a formalism as it is put to use, even though the new complications and extensions add nothing essential to the formal system. The inclusion of transformations in the rules of grammar by which we define specialized languages is an extension of convenience, not principle. Below, we present several examples of the use of transformations, ranging in complexity from a simple identity transformation to a set of complex local transformations which are useful in the parsing of English sentences.

As a first example, consider the rule

```
<q_publication,+completed> ::=
   'works ' <q_publication,-completed> : (q_identity)
```

which was presented above. The semantic intent of this rule is merely to pass the value of the right hand side on as the value of the left hand side, without change. Clearly, the trivial function q_identity can be written to perform this service. However, it seems inappropriate to require the invention of trivial semantic functions to serve purely

syntactic purposes. Of course, such extra functions, where they exist, also contribute an unnecessary increment to the time spent in evaluating their phrases. The need for the function q_identity can be completely eliminated by specifying transformational semantics for the rule in question. We write

```
<q_publication,+completed> ::=
   'works ' <q_publication,-completed> : (1)
```

The (1) means that, at syntactic processing time, the semantics of the left hand side non-terminal is replaced by the semantics of the first right hand side non-terminal. Clearly, both must have the same part of speech. The transformation is, in general, a transformation on the phrase marker, as it is being developed in the process of parsing.

Consider now the syntax by which bibliographic data are input to BIBLIO. In the basic input statement, we may have a number of authors associated with a single publication. Therefore, we need a rule like

```
<input> ::= <author> {, <author>}, <publication>
```

Instead of using such a repetition (which LWL does not support), we write the rules

```
basic_input rule <input> ::= <q_author,+explicit> ', ' <publication>
    : (basic_input);
single_author rule <q_author,+explicit> ::= <author> :
    (build_author_list);
multiple_authors rule <q_author,+explicit> ::= <q_author,+explicit>
    ', ' <author> ; (<1,*><2>,build_author_list)
```

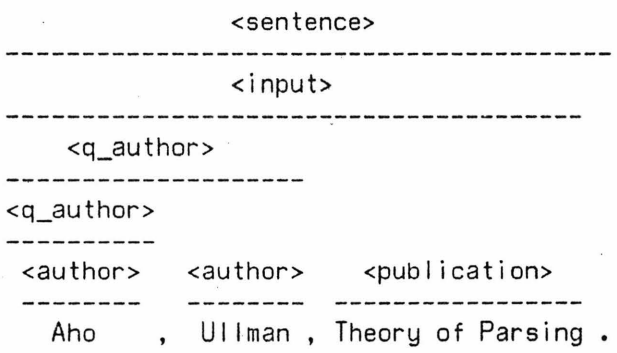
The transformation part of the last rule's semantic specification, <1,*><2>, means that the semantic constituents of the left hand side

should be: all the constituents of the first non-terminal in the right hand side, and the second non-terminal on the right. The function build_author_list will be the semantic transformation computing the value of the resulting <q_author> phrase, and each of its constituents will be an <author>.

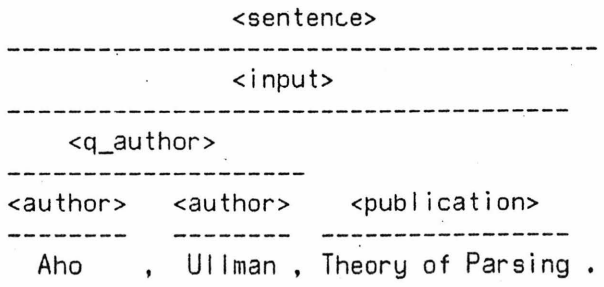
Without the above transformation, the syntactic analysis of the sentence

Aho, Ullman, Theory of Parsing.

would be



By applying the transformation, we get



This transformation not only simplifies the syntactic structure of the sentence, but also permits the unification into one function of the task

of building a list of authors (<q_author>). Without the transformation, the semantic functions of the single_author and multiple_authors rules could not be the same because of the different types of their constituents, and the collection task would be spread between two functions.

The parser, in its normal operation, determines that each non-terminal of the lhs has a phrase marker which computes its value. Each non-terminal of the rhs is a constituent of these phrase markers. The transformations specify a replacement of the constituents in the phrase marker. The identity transformation has been discussed above. In all other cases, the language writer writes a list of selectors which specify the actual constituents to be used and their proper order. A selector of the form <i> selects the i'th rhs non-terminal. <i,*> selects each constituent of the i'th rhs non-terminal. Further forms of the selector permit the selection of only particular categories of constituents of the i'th rhs non-terminal. These forms of selection actually flatten the phrase marker, and remove from it calls on a function which has been referenced by a previously applied rule.

No single transformation can flatten the phrase marker by more than a single level. This is a limitation on the generality of the transformation mechanism, but practical observation (e.g., the implementation of REL English) indicates that transformations which attempt to "reach back" into the structures of their constituent phrases

to a depth of two or more rule applications are too complex to use. They would need to anticipate so much of the structure of the grammar that went into building up those phrases that they would destroy any modularization of syntactic structure gained by the use of a grammar to express it.

Transformational grammar is ordinarily thought of as a mechanism for general tree transformations on the syntactic marker of a sentence. One of the few designs of a computer system with such capabilities is presented by Keyser and Petrick [1967]. Their scheme is to represent the syntactic transformation by a pattern transformation, mapping a given cut across the phrase marker tree into another one. This is a global view of transformations, necessitating an elaborate and expensive pattern matching and control facility to interface the application of ordinary grammar rules to the use of pattern transformations. Although the transformation facilities discussed above have been motivated by concerns similar to Keyser and Petrick's, the emphasis has been on local transformations.

The application of transformational rules is coupled to the application of ordinary rules of grammar. Indeed, as the above examples show, transformations may be of use in instances which fall outside the domain of traditional transformational grammar. The source of many of our examples, the BIBLIO language, has a grammar that is not sufficiently sophisticated to demand the presence of "genuine"

linguistic transformations. To exemplify such a use, consider some small part of a grammar for English.

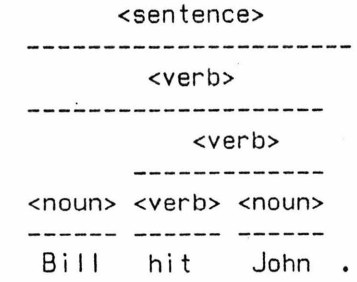
REL English bases its syntactic analysis on the theory of verb cases [Dostert 1971, 1973]. In this approach, each part of the sentence is related to the central verb according to one of the verb's case positions. In the simple example below, we use the order of noun phrases in the phrase marker of the verb phrase to distinguish their cases. In the sentence

Bill hit John.

"Bill" is the agent (the first noun constituent), "John" is the object (the second noun constituent), and "hit" is the verb. A very fragmentary English grammar to parse the above sentence is

```
<sentence> ::= <verb,+has_object+has_agent> '.' :
    (declarative_statement)
<verb,+has_agent+2> ::= <noun> ' ' <verb,-has_agent+has_object> :
    (<2,verb><1><2,noun>)
<verb,+has_object> ::= <verb,-has_object> ' ' <noun>
```

Assuming lexical rules for the nouns "bill" and "john" and the verb "hit", the above sentence parses with the syntactic structure



The effect of the transformation is to eliminate the surface details,

like the order in which the agent and object were incorporated into the verb phrase. The transformed syntactic structure appears as

```
          <sentence>
     ---------------------
           <verb>
     --------------------
     <noun> <verb> <noun>
     ------ ------ ------
      Bill   hit    John  .
```

and it is the order of constituents in the verb phrase that actually determines the sentence's meaning. The deep structure, represented by the phrase marker of the sentence, is:

```
(*)    <sentence>
        <verb>
            <verb> -- "hit"
            <noun> -- "Bill"
            <noun> -- "John"
```

Now, consider the equivalent sentence

John was hit by Bill.

We extend the above grammar fragment by rules to handle the passive case.

```
<verb,+has_agent+passive> ::= 'was ' <verb> ' by ' <noun>;
<verb,+has_object+1> ::= <noun> ' ' <verb,+passive-has_object> :
    (<2,*><1>);
```

With this grammar, the untransformed syntactic analysis is

```
             <sentence>
     ----------------------------
              <verb>
     -------------------------
                  <verb>
            -------------------
     <noun>      <verb>    <noun>
     ------      ------    ------
      John  was   hit  by  Bill  .
```

but the phrase marker is, nevertheless, the same as in (*) above.

By contrast, Keyser and Petrick write the rule [1967, p. 8]

```
(OPT (NP AUX V X NP X BY PASS)
     (5 2 (BE EN 3) 4 0 6 7 1) ()
     PASSIVE)
```

This specifies the optional replacement of a pattern fitting the first template by the corresponding pattern specified in the second template. This scheme has the advantage that it unifies the operation of the passive transformation in a single rule. However, the control mechanisms needed to guide the application of pattern replacements in a reasonable, non-exploding order, and the searches involved in matching patterns which allow arbitrary constituents (like X, above) introduce both conceptual and practical difficulties.

The simple fragment of a case grammar for English, described above, is clearly insufficient for an actual grammar of English. However, the strategy of combining very local, simple transformations with the application of general rewrite grammar rules is an effective technique of processing transformational grammars. It is especially useful because it is very efficiently implemented, as part of the parser's application of each rule of grammar.

5.4 PARSING AND CONDITION FUNCTIONS

Chomsky's fourth demand was for "a method of determining the structural description of a sentence, given a grammar." The REL System includes a general purpose parser, employing an algorithm similar to Martin Kay's "powerful parser" [1967]. The unique advantage of this parsing algorithm is that it parses general rewrite rule grammars and finds every valid parse of an input sentence once and only once. The current implementation of the parser is due to Frederick B. Thompson, and is described in detail in [Thompson 1974b].

The parser operates on a structure called the parsing graph. It is a directed, acyclic graph which is initially a single strand of arcs, each labelled with one phrase (character) of the input sentence to be analyzed. As each rule is applied, the rule's left hand side is added as an arc (string of arcs in the general rewrite rule case) which defines an alternate path to the existing right hand side. Each newly added arc is labelled by a phrase marker which includes a reference to each non-terminal (constituent) in the rule's rhs. The parser operates from bottom to top and right to left. When no more rules of grammar apply, the parse is completed, and every arc labelled by <sentence> which spans from a <string_begin> to a <string_end> arc is a valid parse of the input sentence. If no such arc exists, the string is not a sentence, and the input is syntactically meaningless; if more than one such arc exists, the sentence is syntactically ambiguous. A sentence can also be

semantically meaningless, if in the evaluation of each of its possible meanings some semantic function executes the standard function fail. Semantic ambiguity is also possible, and the whole subject of ambiguity is taken up below.

To give the language writer an additional degree of control over the operation of the parser, each LWL rule may specify a condition function, to be evaluated when the parser is about to apply the associated grammar rule. The checking and setting of features and any transformations specified in the rule are performed before the condition function is invoked. The constituents of the condition function are: the list of phrases constructed by the parser which would have become the labels of arcs inserted into the parsing graph, and the recursion stack of the parser, from which its total environment is accessible. The value of a condition function is a list of phrases, which are added to the parsing graph in the normal way, or nil, in which case the current rule fails and nothing is added to the parsing graph.

The name of the condition function is written between the two colons of the "production arrow" (::=). For example, if we wished to evaluate <subject>s to <q_subject>s during the parsing process, we would write the primitive_subject rule as

```
primitive_subject rule <q_subject> :evaluate:= <subject> :
    (single_subject)
```

Then, the semantic function single_subject would be evaluated every time

that this grammar rule was applied by the parser, and the <q_subject> phrase added to the parsing graph would be the evaluated list of subjects. Note that in contrast to semantic functions, one of which must exist for each non-terminal of the lhs, each rule has but a single condition function.

Typical uses of condition functions are to make more complex feature checks than those allowed by the rule specification (e.g., requiring a certain phrase with either one or another feature set), to compute a set of features for a lhs phrase that is not specifiable in the rule (e.g., depending on the carried over features of more than one rhs phrase), or to perform transformations of greater depth or complexity than those allowed by the rule. The capabilities of the rule statement are all constrained by the design requirement that they be implemented very efficiently. When more complex syntactic processing is required, the condition function is used.

Some investigators have made much of the ability to evaluate semantic information while the syntax analysis is proceeding, to reduce syntactic ambiguity (e.g., [Winograd 1972]). Although this is often an expensive strategy when semantic operations are lengthy, one possible use of the condition function (demonstrated above) is to evaluate some or all phrases before the syntactic analysis is complete.

Because the condition function has access to the complete

environment of the parser, it can introduce arbitrary side effects. However, this is generally not useful, and it is very harmful to the notion of linguistic structure imposed by an explicit grammar. Efficiency considerations sometimes dictate such a use of condition functions: scanning a string of digits and converting them to a <number>, and a simple lexical analysis of the input string are two common uses of condition functions with significant side effects.

5.5 METAVARIABLES AND BINDING

In the syntax directed interpretation scheme so far described, every phrase specifies, by its evaluation, a specific object of the universe. Because these objects are, in fact, arbitrary data types of LWL, they may be designed to represent collections or sequences of "objects" at some lower conceptual level. However, that forced change of view is antithetical to the linguistic point of view expressed in the introductory chapters.

Consider, for instance, the BIBLIO category q_author, which represents a collection of authors. In BIBLIO, as we have described it in Chapter III, viewing a collection of authors as a simple object of the object language is an acceptable strategy, because the authors in such a collection need not be explicitly distinguished. However, it is easy to conceive of a more expressive BIBLIO, in which we could ask a question like

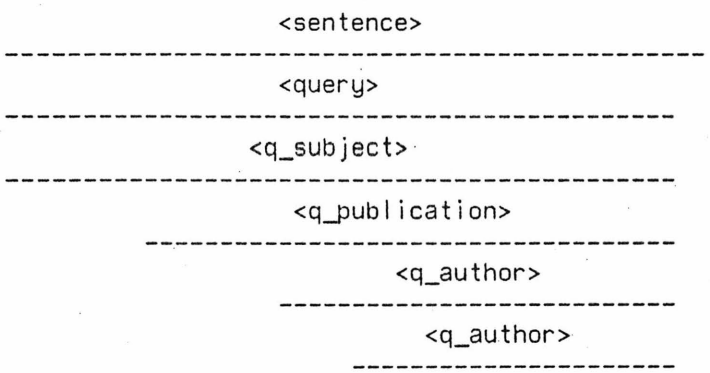
Topic of works by each linguistic philosopher?

In our actual BIBLIO, the similar query

Topic of works by linguistic philosophers?

evokes a list of subjects, undifferentiated by affiliation with the various linguistic philosophers from which they were derived.* The more demanding first query requests exactly such an affiliation.

We will not change BIBLIO to include the handling of quantifiers, but it is illustrative to see how that would be done. Within the interpretation scheme presented above, every phrase must have a unique meaning, represented by an object. Presuming that the query parses according to the following structure,



Topic of works by each linguistic philosopher ?

the q_author phrase "each linguistic philosopher" must differ sufficiently from "linguistic philosopher" that subsequent semantic functions which compute the topics of works by these authors keep

- - - - - - - - - -

*Recall that "linguistic philosophers" has been introduced by

define linguistic philosophers = author of A-quality works B-relevant to linguistic philosophy

separate accounts for each author. That is a significant complication of both the data types and semantic functions which implement these notions. In BIBLIO, the required effort for these capabilities is unwarranted.*

Alternately, we would like to view the phrase "each linguistic philosopher" as a metavariable for <q_author>, and then evaluate the whole query in turn for each individual "linguistic philosopher". The metavariable, as we discussed in Chapter IV, has a part of speech, in this case <author>, and a range, in this case all linguistic philosophers. The metavariable acts as a placeholder for its part of speech in the syntactic analysis, and its range defines the set of semantic values to which it may be bound.

Metavariables act much like variables of the lambda calculus, raising corresponding issues of binding. Metavariables are created by grammar rules which invoke the standard condition function metavar. In our hypothetical extension to BIBLIO, we would include the rule

variable_author rule <q_author> :metavar:= 'each ' <q_author>

The grammar rule must have exactly one non-terminal in the lhs and at most one in the rhs. The part of speech of the lhs non-terminal becomes the part of speech of the created metavariable, and the rhs phrase

- - - - - - - - - -

*By contrast, such a quantification capability is so important in REL English that the suggested complication is incorporated. From efficiency considerations, the quantifier scheme using metavariables (below) is unacceptable in REL English [Greenfeld 1972].

becomes its range. If the rhs consists of all terminals, the range is nil. The metavariable is also given a name, to allow several metavariables with the same parts of speech and range to be distinguished.* The name contains the literal characters which make up the rhs.

In LWL, the use of metavariables is not predetermined by the language processor. However, in any meaningful use, no metavariables may be free in a <sentence>. Every phrase, some of whose constituents either are metavariables or have unbound metavariables, is said to have those metavariables free in it. The parser maintains, for each phrase, a variable list of its free metavariables. The standard function metabind binds all free metavariables in its lhs phrase; thus, the expanded BIBLIO would have the rule

```
subject_query rule <query> :metabind:= <q_subject> :
    (.format_all_subjects)
```

which would bind the metavariables created by every "each" to the function which computes the meaning of <query>. In a semantic function, the standard function n_bound tells the number of metavariables bound in this phrase, and the function metabound(i) returns pointers to the n_bound variable phrases in the variable list. The semantics of this rule, format_all_subjects, is a prefix function. It would evaluate its

- - - - - - - - - -

*LWL's notion of metavariables is implemented in terms of the above metavariable capabilities of the language processor, although by a different condition function than metavar. The name permits the "1" and "2" to be distinguished in define "1" % "2" = ("1" + "2") * ("1" - "2")

<q_subject> constituent phrase for all possible combinations of values from each of its n_bound metavariables. Format_all_subjects would use the standard function evaluate, with an additional parameter that would be a list of phrases to be substituted for the various metavariables bound in the phrase.

The evaluation mechanism, including the binding and instantiation of metavariables, is complex. Because only the creation and binding of metavariables is known to the language processor and their use is left to the individual language implementor, metavariables serve a large variety of functions. The above example of quantification is one, and the use of metavariables in the LWL definitional mechanism is another. A complete discussion is beyond the scope of this presentation; for more detail, refer to [Thompson 1974b].

5.6 AMBIGUITY

The notion ambiguity has been both blessed and cursed for its role in language. Innate ambiguities appear to interact with context-resolving mechanisms in natural languages to provide conceptual generality and expressive conciseness. In the computerized processing of languages, ambiguity has been viewed as the anathema of any practical methods of parsing and analysis. Especially when employing general rewrite rule grammars, parsers have been subject to uncontrolled combinatorial growth of ambiguous interpretations.

Language implementation in LWL does not attempt to suppress the potential problems which arise from the tolerance of ambiguity, but it does provide some methods of controlling those problems without resorting to the customary prohibition against any use of ambiguity.

The REL language processor includes automatic mechanisms for allowing the ambiguous interpretation of sentences. If, for example, an input sentence can be assigned several different structural analyses, each phrase marker will be evaluated by the language processor, and if their values differ, all will be output, with an indication that these are ambiguous responses.

Ambiguity may be introduced either through the syntax or semantics of an object language. In the syntax, the case above is typical; that is, a sentence can parse in more than one way. In the semantics, it is possible for a semantic function to compute more than one value for its phrase, in which case the phrase becomes ambiguous. The standard function ambig acts similarly to return, except that it does not actually return control from the function. All values specified by each call on ambig and the final call on return from a single invocation of a function become the ambiguous resulting values of the phrase.

In most cases, the processing of ambiguity is hidden and automatic. In the process of semantic evaluation (the application of

evaluate), the presence of ambiguity, whether syntactic or semantic, may be ignored except in prefix functions. When the evaluator is about to invoke a semantic function with ambiguous arguments, it invokes the function a number of times instead, once with each combination of unambiguous arguments formable from all the ambiguous constituents. All values returned by the function for its various constituents are collected and form the ambiguous value of the resulting phrase. Of course, the blind application of this strategy is what can lead to combinatorial explosion not only in parsing but also in semantic evaluation.

The above techniques merely introduce ambiguity, but do nothing to control it. The resolution of ambiguity is most generally handled by the ability of a function to fail (a standard function call). If the phrase for which a value is being computed is unambiguous and the semantic function fails, or if each of the phrase's ambiguous interpretations fail, the phrase is meaningless and every phrase of which it is a constituent also becomes meaningless, without the need for further semantic processing. Therefore, one such failure can eliminate a proposed parse of a sentence, or several of its ambiguous alternatives. The function fail has an optional argument, which is an output message to the user, to be displayed if this function's failure makes the whole sentence meaningless. If some other evaluation succeeds, however, the failing phrase and message are discarded.

In Chapter III, we noted that BIBLIO would accept the name Winograd as a synonym for both authors Terry and Shmuel, if both were known in the data base. In practice, this occurs because the <author> phrase produced by the lexical rule

```
<author> ::= 'Winograd'
```

is assigned an ambiguous meaning, referring to both alternatives. With a syntax for inquiring about authors that is similar to that for subjects,

```
author_query rule <query> ::= <q_author> : (format_authors)
primitive_author rule <q_author> ::= <author> : (single_author)
```

the evaluation of the query

```
Winograd?
```

will apply single_author in turn to Terry and Shmuel, producing an ambiguous <q_author> phrase. Then, format_authors will apply twice, to give an ambiguous <query>, then print twice, for an ambiguous <sentence>, and the reply will be

```
AMBIGUOUS:
(1) Winograd, Terry
(2) Winograd, Shmuel
```

Of course, if any of the semantic functions thus called had failed (not likely in this example), the ambiguity would have been resolved.

The elimination of ambiguity by semantic failure is a general, but often expensive approach. To allow greater control to the language writer, the prefix function may be used. Because a prefix function is invoked before its constituents are evaluated, the automatic propagation

described above does not come into play, and the prefix function can itself manipulate its ambiguous constituents. To aid this, two standard functions are provided. The function n_amb returns the number of ambiguous alternatives for its constituent phrase. For instance, in the function single_author in the previous example, if it were made a prefix function, the value of

```
n_amb(constituent.author)
```

would be 2. To select one of these ambiguous phrases, the standard function amb, with a phrase and an integer argument returns the selected ambiguous phrase of the given phrase. For example,

```
constituent.author.amb(2)
```

yields the second ambiguous value of the <author> phrase. The ability to deal with ambiguity explicitly in an object language is of special importance when some innate construction of the object language naturally expresses a reasonable interpretation of the ambiguity. In the BIBLIO case, the desired effect of "Winograd"'s ambiguity is to treat the synonym as standing indistinguishably for both its values. But the category <q_author> is exactly a type which exhibits that required behavior. Thus, the natural interpretation of the ambiguity of an <author> phrase is to convert it to an unambiguous <q_author> phrase which lists the alternative meanings. The single_author function is the appropriate place for that conversion.

If ambiguity can be successfully mapped into one of the

primitive concepts of the object language, then the overhead and possible uncontrolled growth of ambiguity processing is eliminated. The ability to effect this shift, from general evaluation mechanism to a specific function of the object language, is an important opportunity for the language writer.

5.7 LANGUAGE EXTENSION

We operate under the assumption that no language designer or implementor has the foresight to create an object language that will perfectly fit a user's evolving needs. Therefore, LWL supports the provision of extensibility in any object language. It is useful to distinguish between two forms of extension: one, by the language writer, to add significant new features to a language; the other, by the user, to add to his language new objects and concepts of his universe, in terms of the fundamentals of the object language. The hypothetical addition of "each" to BIBLIO, discussed above, would be an extension of the first kind, requiring intervention by the language writer. The definition of "linguistic philosopher", also mentioned before, is an extension introduced by the user, for which all required facilities already exist in the object language. These two forms of extension seem to be what Wegbreit calls metaphrase and paraphrase extension [1970, pp. 124-125].

The introduction of fundamental new capabilities into a language is accomplished by its further development in LWL. The single reserved sentence, metalanguage, of every object language switches control of the session from the object language to the particular instance of LWL in which the object language has been implemented. At that time, all the facilities of LWL for adding or deleting grammar rules, data types and functions are available, and the language writer is given the same free hand in altering the object ianguage as he had in implementing it. Ordinarily, the language user is not expected to modify his language in this way.

The way in which an object language is extended depends very much on the particular language. Therefore, LWL makes available to the language writer a standard function extend, which provides all of those features available in the rule statement. Thus, any object language may include functions which extend the grammar of the language itself. The form and meaning of such capabilities are completely determined by the language writer, when he writes the syntactic rules and the condition and semantic functions of the object language which implement its abilities for extension. The BIBLIO define statement is an example of such an extension capability.

Although many languages implemented in the REL System include sophisticated extension mechanisms like BIBLIO's define, or even the more complex define of LWL, that is by no means the only use of

extensibility. In BIBLIO, for instance, the names of new authors, subjects, publications and publishers must be added when they are first introduced. For example, to satisfy the specifications in Chapter III, if the phrase

Author: Whorf, Benjamin L.;

appears in an input sentence, it must have the effect of adding the two lexical rules

<author> ::= 'Whorf, Benjamin L.'

and

<author> ::= 'Whorf'

to the language. In addition, it must create an object of the category author, initialize it and use it as the "data" semantics of the above rules. The portion of the BIBLIO grammar dealing with this is:

```
<item,+author-complete> ::= 'Author: '
<item,+1> ::= <item,-complete> <letter> : (<1,*><2>)
<item,+1> :getchar:= <item,-complete> ' '
<item,+1> :getchar:= <item,-complete> ','
<item,+1> :getchar:= <item,-complete> '.'
<item,+complete+1> ::= <item,-complete> ';' : (1)
<author> ::= <item,+author+complete> : (create_author)
```

The condition function getchar must append the blank, comma or period as a further constituent of the <item>.* A similar treatment is given for the other new items.

```
<item,+subject-complete> ::= 'subject: '
<subject> ::= <item,+subject+complete> : (create_subject)
```

- - - - - - - - - -

*In fact, to avoid rules of grammar to create <letter>s and the resulting parsing overhead, an actual implementation of BIBLIO would employ a condition function on the first <item> rule. Its side effect would be to collect each character up to a semicolon as constituents of the <item>.

```
<item,+publisher-complete> ::= 'publisher: '
<publisher> ::= <item,+publisher+complete> : (create_publisher)

<item,+publication+book-complete> ::= 'book: ' <publication> ::=
    <item,+publication+complete> : (create_publication)

. . .
```

The create_publication function determines, on the basis of the features of its constituent <item>, whether a book, article, etc., is being introduced.

The generality of LWL's extend function can be used by the language writer to program the object language to reflect accurately the needs of its users.

In this chapter, the language definition capabilities of LWL have been discussed. The basic statement with which we were concerned is the rule statement. It allows the specification of general rewrite rules of grammar and their associated semantic functions. In addition, features, local transformations and condition functions were introduced to enhance the usefulness and conciseness of grammar rules. Finally, we discussed the handling and use of metavariables and ambiguity, and the capabilities for extension of the object language.

The above discussion of LWL's specification of the extended syntax of an object language, together with Chapter IV's presentation of how LWL is used to specify the object language's universe of discourse, completes the topical definition of the REL Language Writer's Language. Its formal definition is in Appendix A.

CHAPTER VI

IN RETROSPECT

> And the earth was of one tongue, and of the same speech. . . .
>
> And the Lord came down to see the city and the tower, which the children of Adam were building. And he said . . . let us go down and there confound their tongue, that they may not understand one another's speech. . . . And therefore the name thereof was called Babel.
>
> -- Genesis, 11:1-9.

When introducing a new computer language, its author has a dual responsibility: to justify the need for yet another addition to the programmers' Babel of languages, and also to consider how well his language meets the criteria which motivated it and how it will affect the world which spawned it. In this concluding chapter, we will cover three topics: a comparison of LWL to other possible methods for implementing specialized languages, some thoughts on why LWL falls short of meeting the challenges posed in the introduction and some possible remedies, and a few disturbing questions about the diversification of the myriad new languages made possible by our approach.

6.1 ALTERNATIVE TECHNOLOGIES

From the discussion of various design points of LWL scattered throughout the body of this thesis, some of the reasons why other existing technologies for language implementation were not considered adequate should be apparent. In this section, we examine this question with respect to the available capabilities of extensible programming languages, compiler generators and the recent crop of semantically powerful languages developed for the support of AI research (which we call semantic languages, for lack of an accepted term).

The principal issue which distinguishes these three techniques of language implementation is: What should be the relationship between the metalanguage and the object language? The extensible language answer is that the object language should be built by adding to the base language any new capabilities needed. Thus, the object language would contain within it the metalanguage. A language implemented by a compiler generator is completely divorced from the compiler generator (metacompiler) when it is completed. The semantic languages take an ambiguous position, since their current users, the AI researchers, use them as object languages, but they contain very strong extension mechanisms that make it possible to use them as metalanguages for creating application languages.

Each of the approaches has some great attractions and

shortcomings. We take up each, in turn, and discuss the debts which LWL owes to each, as well as the compromises which have been made in LWL to avoid some of the worst problems with each of these other technologies.

6.1.1 EXTENSIBLE PROGRAMMING LANGUAGES

The fundamental premise of extensible programming languages is simple and elegant: if it is possible to invent a basic computer language with a very few primitive operations and some very simple and extremely powerful techniques for composing them, then perhaps any desired programming language can easily be extended from that base language. The early major attempts in this direction succeeded in recreating from a stripped-down base language most of the features then current in complex programming languages, and the above premise was assumed proven. To single out two particular projects, Standish's PPL demonstrated the feasibility of data structure extension from a basic polymorphic set of data types by a very few simple operators. Wegbreit's ECL made a serious attempt to tackle the most complex problems of extensibility, including considerations of efficiency and language contraction. Yet, against this background of apparent success, interest in extensible languages has waned and their actual use is minimal. Even Standish now writes articles with titles like "After Extensible Languages, What Next?"

According to an old folk saying, "The fruit doesn't fall far from the tree," and so it is with the languages created by extension. The examples foreseen by the base language's designer all work out well, but many truly unforeseen extensions are just not possible with any reasonable effort. As an example of this problem, we turn to Standish's PPL.

PPL is an extensible APL-like language, with only a limited basic set of data types and operators. The expectation is that the user who needs other data types and other operations can add them to the language using its extension capabilities. In those instances which conform closely to the incremental augmentation strategy (where the desired language is "close" to the base language), the resulting language can be quite reasonable. The ubiquitous "COMPLEX" example fits well with the base language, and one can conceive a similar extension strategy to build PPL up to a full APL or similar language.

To use PPL-style extension for the creation of significantly different languages, however, is problematical. Incorporating a facility for symbolic manipulation is a well-known (e.g. FORMAC) technique for improving the usefulness of an algebraic language (to users who are concerned with symbolic manipulation, of course). Any extensible language is capable of expressing such an addition, but not necessarily in a useful way. Consider the symbolic differentiation example [Taft 1971, p. 51].

PPL is extended to include data types FORMula, Binary Formula, Unary Formula and ATOM, in terms of structures and the primitive types:

```
$FORM = UF ! BF ! ATOM
$BF = [LO:FORM, OP:CHAR, RO:FORM]
$UF = [OP:CHAR, RO:FORM]
$ATOM = STRING ! CHAR ! REAL ! INT ! DBL
```

Then the formulas F and G are defined by

```
F <- BF('X, '+, 3)
G <- UF('-, BF('X, '*, F))
```

After a PPL function DERIV(F,X) is defined, we are shown

```
   DERIV(F, 'X)
[LO:1,OP:+,RO:0]
   DERIV(G, 'X)
[OP:-,RO:[LO:[LO:X,OP:*,RO:[LO:1,OP:+,RO:0]],OP:+,
RO:[LO:1,OP:*,RO:[LO:X,OP:+,RO:3]]]]
```

With the addition of a special print routine, this becomes

```
   PF(DERIV(F, 'X))
(1+0)
   PF(G)
(-(X*(X+3)))
   PF(DERIV(G, 'X))
(-((X*(1+0))+(1*(X+3))))
```

One might, as suggested, add the simplification algorithms needed to make symbolic differentiation look better, but the problem with this "new" language is deeper than that. As long as formulas need to be entered in the unnatural manner shown above, this language will not be congenial to a real user. And note that the input routines are not accessible for "extension" to deal with FORMs. Further, some simple and potentially desirable capabilities are impossible to provide. For instance, given the above definitions of F and DERIV, one might like to evaluate

```
   X <- 5
   X*DERIV(F, 'X)
65
```

But this is not possible, because the result of DERIV is a FORM, which is not a proper argument to TIMES. So, the symbolic differentiation is in some peculiar sense unavailable to the rest of the language. The difficulty is that the new language is implemented by representing formulas as structures of symbols in the base language. This makes their manipulation quite simple, but their evaluation impossible. The failure results from not recognizing the relationship between the implicit formulas of the PPL language and the FORMs of the new language. A formula in PPL may be evaluated by the built-in language processor, but a FORM is merely a structure of symbols on which the DERIV and PF operations are valid.

This particular difficulty is not universal. In LISP, for instance, the notion of "formula", or S-expression, is built in, and would be a natural representation for FORM. However, in every extensible language there are some extensions that just lie outside the scope of what can be handled naturally by the provided language processor, and these can not be reached effectively by extension. The biggest difficulties arise whenever the base language is extended to a new capability which may conflict or have harmful interactions with an old feature of the base language.

Symbolic differentiation added to an algebraic language is

hardly a radical extension. Yet, in PPL in particular, it cannot be effectively accomplished. Because the specialized application languages we consider involve operations more closely mirroring external realities (presumably even farther removed from the basic operations of an extensible language), an extensible language seems like an inappropriate choice for the language writer.

A related but distinct trouble with extensible languages that has surfaced concerns their often severe inefficiency. At the root of this trouble is the extreme generality of primitives and extension mechanisms which is demanded by the fundamental premise. For any particular problem domain, a specially written language can take advantage of the peculiarities of the specific case to outperform the general techniques. If this were the only difficulty, it would be unimportant, because the savings in implementation would quite often offset this disadvantage. The serious problem results from the fact that many such inefficiencies compound as layers of extension are built on layers of extension to form the desired object language.

Consider the possibility of a language which verifies tautologies in the sentential calculus, which has been written in a list-processing language, which has in turn been written as an extension of ECL. Now, in the design of the list processing language, its primitives are realized by data structures and operations of ECL. These design decisions greatly influence the efficiency with which that

language will perform certain operations, which will in turn have a significant effect on the tautology checker. Yet, to expect the implementor of the tautology checker to know each layer of language above which he works so that he might optimize the performance of his routines is unreasonable; thus, decisions over which he has lost control introduce inefficiencies which will hurt him. For instance, Wegbreit illustrates ECL's data type extension by introducing a new type, RBUF, a FIFO buffer of characters [1974]. Now, it might be that for the implementor of the list processing language we are imagining, such an RBUF is an attractive implementation of atom names. Then, the implementor of the tautology verifier will be using a rather complex buffering mechanism, most likely unknown to him, even if he should decide that all sentence symbols in his language are to be single characters.

The group working on the ECL system has started an attack on the above problem. They have proposed incorporating very good optimizing techniques in their compiler and have suggested the application of program verifying methods so that the compiler could automatically or by interacting with a programmer discover and eliminate inefficiencies introduced in the extension process [Cheatham 1972]. Certainly, this attempt is necessary if extensibility will ever compete effectively with specially built problem-oriented languages and programs. However, the current state of understanding of programming language semantics and

especially of the relationship between the semantics of a real-world problem domain and the semantics of the implementing computer language is sufficiently muddy that automatic techniques of transferring knowledge from one to the other are unlikely.

One final criticism of extension in extensible programming languages centers on the common assumption that no matter how the base language may be modified, the desired mechanisms of extension remain essentially unchanged. That this is unsatisfactory from the viewpoint of this thesis should be apparent from considering four different problems of language use and extension which are discernible in the creation and use of an object language:

1) extension of the metalanguage itself, by the language writer, to introduce his favorite abbreviations and convenient functions,
2) the language writer's use of the metalanguage to create and modify the object language,
3) the end user's use of his basic application language, as prepared by the language writer, and
4) the end user's extension of his language, using mechanisms put into the object language by the language writer.

Considering the major differences between the task of the language writer and the language user, there is no reason to expect that their languages will be at all similar, or that any part of the metalanguage will be of use to the user. For example, essentially nothing of LWL is of any direct use to the user of BIBLIO. In exactly the same vein, there is no reason to assume that the manner in which the language writer and the language user want to extend their languages

will be at all similar. For example, both the syntax and semantics of LWL and BIBLIO differ greatly for the introduction to LWL of a new data type compared to the addition to BIBLIO of a new publication. In an extensible language implementation, the same mechanism would have to serve for both.*

The goals of LWL are much more modest than those expounded for extensible languages. Neither LWL nor any object languages created by it aim for extreme simplicity or elegance. And, most important, in no sense is a tower of languages, all built on top of each other, envisioned. In fact, with the semantic primitives supported by LWL and the REL System, the implementation of object languages which in turn are able to create other new languages is very difficult.** Because only a single full level of "extension" is desired from LWL to the particular application language, and because the extension actually builds a completely new language rather than adding new capabilities to a base, the severe problems of inefficiency, the disturbing interference between old and new features, and the confusion of different modes of language use and extension are avoided.

- - - - - - - - - -

*If a knowledgeable language user insists on extending LWL in a manner not supported by the language writer (for instance, to include the addresses of publishers), he steps outside the role of language user and employs the facilities of LWL as a language writer.

**This accounts, in part, for many of the practical difficulties encountered in implementing LWL, which runs in the REL System almost as an ordinary object language.

6.1.2 COMPILER GENERATORS

The earliest appearance of syntax directed techniques for language implementation probably occurred in the development of syntax directed compilers for algebraic languages, like the Brooker-Morris Compiler Compiler [Rosen 1964] and Irons' ALGOL compiler [Irons 1961]. Unfortunately, almost no significant improvements over these methods have been introduced in metacompilers since then, and few recent systems have even reached the degree of sophistication of the Brooker-Morris system.

The major attraction of the compiler generator (we use compiler generator and metacompiler interchangeably) is its ability to transform a language definition into a rather efficient compiler for that language. The metacompiler view is that the definition of an object language is translated into an independently existing new language. Once a language has been so created, its connection to the metacompiler is completely broken. If we define, say, FORTRAN, by this technique, the result will be a FORTRAN compiler, with none of the capabilities of the system which was used to create it. Although its syntax may have been defined in BNF, the new language will have no such mechanisms.

The use of a metacompiler involves the following steps (as outlined in a problem suggestion in [McKeeman 1970, p. 241]):

1. Invent a language . . .
2. Use BNF, ANALYZER, and English descriptive text to describe your language.

3. Implement and thoroughly test a syntax checker for your language.
4. Invent an ideal machine to execute your language. Implement and document an interpretive simulator for your machine.
5. Add code emitters to the syntax checker, and incorporate the interpreter . . .
6. Test your language thoroughly . . .

A metacompiler like XPL [McKeeman 1970] aids this process at steps two through five. Steps two and three create a simple, unambiguous grammar for the desired language, and XPL's ANALYZER helps by checking the grammar and eventually translating it to a set of tables to be used by the skeletal syntactic analyzer. Steps four and five couple interpreters and code emitters to the productions of the syntax, to create the finished language processor.

The semantic routines which a language writer implements in a metacompiler are used typically to emit machine language instructions. (XPL, for example, provides built in routines like EMITRX to produce machine instructions.) The basic facilities of the metacompiler best support this usage.

The major shortcoming of metacompilers used for application language implementation is that the problem of application language processing is fundamentally not a compilation problem. In a normal programming language, the semantically meaningful primitives are at such a low level that very many of them must be invoked to perform a significant computation. Thus, in implementing such a new programming

language, the most important tasks are to be able to analyze statements of the new language as rapidly as possible and to translate them into efficient and efficiently connected code segments. Metacompilers, therefore, provide parser generators which work for restricted classes of grammars so that the generated parser can be very fast, and they support the language writer's work by providing a standard set of functions for code generation, the allocation of temporaries and register optimization.

In the processing of application languages, neither the speed of parsing nor the amount of language processor overhead in composing semantic primitives is important compared to the cost of performing the basic data manipulations meaningful in the application field. In answering a complex query, BIBLIO will not apply more than twenty rules of grammar in analyzing the sentence, but the resulting retrieval may search hundreds of pages of the data base. Thus, the extra time spent by a powerful parser and a sophisticated semantic evaluator like those provided by LWL can improve the usability of the language without significant additional cost. Also, because semantic primitives are large and self-contained, little advantage would result from recompiling them in the appropriate combinations for each sentence. Instead, LWL supports a language for defining each semantic primitive independently, and a compiler to turn those definitions into efficient implementations, just once. Then, the functions act as the interpretive routines of a syntax directed interpreter.

Concisely put, a program in an application language is likely to be a single sentence. Thus, the services provided by typical metacompilers are often irrelevant, and the metacompiler is not a very appropriate tool for the implementation of specialized application languages.

6.1.3 SEMANTIC LANGUAGES

Based on the experience of AI researchers into problems of natural language understanding, robotics, vision and goal-directed problem solving, a large number of semantically very powerful new languages have been designed and at least partly implemented. These languages have been described elsewhere [Bobrow 1973], but a few comments about them as application language implementation tools are in order.

The great attraction of languages like PLANNER, QLISP, CONNIVER and their relatives is that they include a rich repertoire of basic semantic capabilities which have been found useful by years of experience in AI research. General pattern matching, pattern directed procedure invocation, high level data primitives like sets and sequences, flexible control structures, and computations in dynamically switchable contexts are all built into some of these languages. With these features, many of the traditional AI algorithms reduce to short

programs, and the user of one of these languages gets a great deal of computation per statement. In that the above mentioned capabilities are likely to be useful in implementing the semantics of application problem domains, these languages might be excellent metalanguages.

The main reason why application languages have not yet been implemented in these languages is that they tend to be inordinately inefficient. The richness of their primitives easily leads to immoderate application and exorbitant cost. Even the primitive operations are sufficiently complex to use a significant amount of computing time, and if any sort of combinatorial growth is allowed to occur in the use of these primitives, the languages become totally impractical. This view is supported by current experience, which shows that only languages with very small universes of discourse can be implemented at all.

It seems that the freedom granted by the semantic languages is too much. They impose so little structure on their users that they fail to provide guidelines for what will be practical. A language writer using one of these has no "language processor" already defined for him, no definition of what an object language is or of how to implement it. In some sense, the environment is too poor: if a context free parser is needed, it must be implemented. On the other hand, it is too rich: The language writer may be tempted to use pattern matching as a fundamental access mechanism to his data base because it is so easily available; if he does, its cost will make his language impractical.

Perhaps what is needed is a somewhat restrictive language processor imposed on top of this environment, to insure that at least those aspects of language processing which are clearly understood will be implemented efficiently. This idea will be taken up again in the next section, considering future extensions of LWL. Without such additional structure on the general capabilities of these languages, however, they do not provide a sufficiently well defined formalism for application language implementation.

6.2 ANOTHER LOOK AT METALANGUAGES

Metalanguages, in general, and LWL, in particular, share many of the advantages and disadvantages of the above techniques for language implementation, because of the many similarities and shared capabilities. In this section, we consider some deficiencies of the present metalanguage approach and some possibilities for improvement.

6.2.1 MANY POSSIBLE METALANGUAGES

Like an extensible language, a metalanguage can be the base of only a limited class of object languages. The more appropriate that a metalanguage is for defining a particular object language, the less effective will it be for implementing other, different languages. Clearly, the simplest metalanguage for implementing a particular object

language is one which already contains all of the primitives of the desired language. But that metalanguage will include many features unique to the particular problem domain it best supports, and will therefore be peculiar to use for implementing languages in remote fields.

The choice of how specialized a metalanguage should be to a particular domain is a choice similar to how specialized a particular object language should be. The more specialized, the easier it is to use, but the fewer opportunities will arise to use it. If it is anticipated that many application languages will need to be built, all including a certain special set of capabilities, it makes economic sense to support those capabilities in a fairly specialized metalanguage. The dividing line between what responsibility belongs to the implementor of a particular application language and what to the metalanguage builder then becomes a question determined by economic considerations.

The metalanguage described in this thesis, LWL, has developed as the result of many years of experience with the implementation of specialized languages. The problem domains of the individual languages have ranged from the production of abstract motion graphics [Thompson 1974c] to natural language question answering (REL English). LWL is most appropriate for implementing a language whose primitives are complex computations reflecting meaningful operations in an application field, whose grammar is complex and benefits from the availability of a

powerful syntactic analyzer, and whose efficiency depends on the availability of efficient data access and manipulation facilities. Ambiguity tolerance and user-introduced extension is also well supported. Languages which permit the user to interact with large, persisting data bases in a complex manner are particularly well suited to implementation in LWL. The implementation of a radically different language (e.g., a variant of JOSS) is possible but receives relatively little appropriate support from LWL.

Because of the likely future increase in the use of specialized computer application languages, it is reasonable to suggest and expect that many metalanguages will come into existence, with distinct, large, but limited domains of applicability. Existing system implementation languages also suggest that metalanguages need not be implemented from an assembly language base, and one may expect to see a hierarchy of metalanguages each of which allows the implementation of other metalanguages or object languages.* Because each of these languages would be complete and self-contained, it might be possible to avoid the compounding of inefficiencies which accrues in a similar use of an extensible language to build a hierarchy of languages. If not, then it

- - - - - - - - - -

*LWL has been partially implemented in assembly language, using the same system facilities and standard functions which are available to the object language implementor. LWL does not meet various of the criteria which an object language must meet to receive proper support from this environment (e.g., most of LWL's semantic primitives are the low level primitives of programming languages). Therefore, its implementation might have been easier in a different metalanguage, had an appropriate one existed.

will still be possible to build at least a few useful metalanguages like LWL, that will be able to support the implementation of large classes of useful specialized application languages.

6.2.2 LANGUAGE IMPLEMENTATION IS DIFFICULT

LWL has been designed to make it easy to implement languages like BIBLIO. Has it been successful? Yes and no! The linguistic framework provided by LWL is a valuable tool for structuring BIBLIO's implementation, the REL language processor is powerful enough to make the design of BIBLIO's syntax quite straightforward, and the PASCAL-like LWL facilities for defining data structures and storage and retrieval algorithms are convenient. Certainly, compared to an implementation in assembly language, or even in a higher level language like PL/1, the LWL implementation of BIBLIO is simpler and easier.

However, LWL itself is a large, complicated language, not easy for a language writer to learn to use fully and effectively. Some of the complications (e.g., the inflexibility of the list storage class) are incidental, the result of historical accidents. The complexity of the rule statement, on the other hand, is not a result of poor design -- each of its capabilities and nuances has at some time been quite useful in implementing a portion of various languages.

The task of language implementation is undoubtedly complex, and

we know of no significant way of making it easy. The only successful models of language description existing today all pose the task in terms of defining a syntax according to some formal grammar and the semantics according to another formal system (e.g., a programming language), and connecting the two. The most optimistic researchers in automatic programming may envision other possible approaches, in which the programming system can elicit from the intended end user enough information to construct his desired language without forcing him to resort to formal descriptions, but only much more research and experience will decide if this optimism is justified.

A more promising direction is to specialize the metalanguage even more than LWL is specialized, and to add to it very complete models of a particular application domain. A metalanguage like that might already contain facilities which support features sufficiently close to the desired object language of a particular user that he could specify, in terms natural to the field, what his object language is to be like. This is essentially the approach taken by Martin [1974] in building a "metalanguage" in which to implement inventory control systems.

6.2.3 METALANGUAGES WITH HIGHER LEVEL PRIMITIVES

Concepts and experience from extensible languages and compiler generators have obviously contributed to the metalanguage scheme

discussed above. Contributions from the semantic languages have not yet been included. LWL now provides the language writer strong linguistic tools with which to shape an object language syntactically, but the primitive semantic capabilities of LWL from which the object language's semantics must be built are relatively low level.

For the development of specialized languages which exhibit semantically sophisticated behavior as well as a flexible syntax, features from the semantic languages could be incorporated into LWL. With careful regard for efficiency (without which an application language is useless), some pattern matching and deduction primitives and the addition of a set of basic relational data file manipulation primitives could be added to the LWL programming language. To provide truly convenient, flexible and natural application languages, semantic capabilities based on such techniques will very likely be necessary. This augmentation of LWL would make it easier for the language writer to implement application languages which use such complex primitive semantics.

The incorporation of notions of program and data structuring which are now being developed in an attempt to implement some of Dijkstra's ideas on structured programming [1973] would be a useful addition to LWL. Especially in the construction of very complex semantic primitives, the class and class concatenation concepts from SIMULA would be quite important [Dahl 1973]. This would permit the

introduction of a level of structure which falls below that imposed by the object language's rules of grammar.

Finally, the inclusion of highly interactive, powerful debugging aids might well be a more significant improvement in LWL than any other additional feature. This thesis has not addressed the problems of language testing and debugging, but obviously the addition of facilities with a sophistication similar to Interlisp's break, undo and DWIM [Teitelman 1974], but aimed at the particular language building environment of LWL would be extremely useful.

This concludes the discussion of possible extensions to LWL which promise to make it more effective as a language implementation tool. The three extensions proposed have been: the development of a hierarchy of metalanguages, the specialization of a metalanguage by the incorporation of models for very limited problem domains, and the addition of very high level semantic primitives. Each suggestion is intended to explore a way of making metalanguages more powerful and easier to use. However, as with all challenging human tasks, no fully automatic techniques can replace the necessity of hard thought and inventive design to create a good language.

6.3 THE PROLIFERATION OF LANGUAGES

Sammet, in one of her recent surveys, reported the existence of nearly two hundred different computer languages in use in the U.S. [1972]. The pressures for creating specialized languages discussed in the introduction and the increasing availability of tools like LWL which make language implementation easier will encourage an increase in that already large number of languages, perhaps by orders of magnitude. The possible creation of many incompatible, idiosyncratic languages will exacerbate the serious problems of incompatibility which we already face and will make the interchange of data in a meaningful form among different users nearly impossible.

Potentially, each investigator (be he scientist or businessman) will have the ability to have developed for him specialized computer tools which impose his own structural views on the phenomena he is studying. Because a specialized language can be changed arbitrarily to fit the peculiarities of its user's special views, two investigators starting from the same data and similar interests might develop their thoughts and their analyses in such radically different directions that, at a later time, their results may no longer be meaningful to each other. If each investigator has absolute freedom in developing his studies and his specialized tools, they might well become incommensurable with the work of others.

Of course, there are now natural checks which prevent most individuals from retreating into personal worlds, and we should presume these to continue to operate. Thus, a scientist will keep in touch with the paradigm of his field because he does not want to lose the ability to communicate his ideas to colleagues who are only accessible through that paradigm. In a business, no department will be allowed to develop accounting techniques, for instance, which are irreconcilable with those used elsewhere in the company, no matter how attractive that may be to the department.

Whereas the present cost of providing specialized computer tools acts as the primary deterrent to specialization, after the introduction of inexpensive language implementation techniques, it will be the information cost that will discourage the abandoning of common standards. The advantages gained by providing particularly appropriate, specialized tools to each member of an organization will have to come into equilibrium with the disadvantages which result from the communication breakdown caused by incompatible specializations. This equilibrium is a better one on which to rely for decisions in computer system design than the current one, which is so heavily dominated by considerations of the cost of implementing different computer systems and languages.

The metalanguage methodology helps to eliminate the controlling factor of excessive cost. It does not, however, take a stand on whether

common or unique application languages are appropriate. In fact, a nearly continuous range of possibilities is supported, since two object languages implemented in LWL may sharehere from all to none of their syntax, data structures and semantic functions. One attractive compromise between common and unique languages is to require some common core, but to let each of several users have special facilities of his own beyond that core. Another is to require the same data structures and some common set of semantic functions, but to allow each user different syntactic access to a commonly structured data base. The methodology of application language implementation in a metalanguage supports all of these possibilities.

6.4 CONCLUSION

This dissertation has presented a point of view favoring the creation of specialized languages as a means of giving natural access to the computer's organizing and problem solving capabilities to large numbers of users. A discussion of the role of specialized languages and current application computing practice led to the presentation of a methodology for aiding the implementation of specialized application languages. The interesting features of the REL Language Writer's Language were presented, with examples from the BIBLIO language used to illustrate the application of LWL. Finally, LWL's approach to supporting language implementation was compared with other possible techniques.

The use of metalanguages is effective in easing the task of the language implementor; therefore, it will make it more economical to implement specialized languages. This will lead to the availability of natural computer languages to aid specialized areas of human endeavor, and thus will bring the computer within the reach of many.

LIST OF REFERENCES

Backus, J. W., R. J. Beeber, S. Best, R Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes, and R. Nutt 1957. The FORTRAN Automatic Coding System. Proc. Western Joint Computer Conference 11:188-198. Reprinted in Rosen, Saul, ed., Programming Systems and Languages. New York: McGraw-Hill, 1967.

Balzer, Robert M. 1973. "A Global View of Automatic Programming," Proc. 3IJCAI (20-23 Aug. 1973). Menlo Park, Ca.: Stanford Research Institute.

Benson, David B. 1968. Formal Languages, Part Theory, and Change. Ph.D. Dissertation, Calif. Inst. of Tech., Pasadena, Ca.

Bigelow, R. H., N. R. Greenfeld, P. Szolovits, and F. B. Thompson 1973. "Specialized Languages: An Applications Methodology," Proc. AFIPS National Computer Conference 42(1973):M49-M53.

Bobrow, D. G., J. O. Burchfield, D. L. Murphy, and R. S. Tomlinson 1972. "TENEX, a Paged Time Sharing System for the PDP-10," Comm. ACM 15 (March 1972).

Bobrow, D. G. and B. Raphael 1973. "New Programming Languages for AI Research," presented at 3IJCAI (Aug. 20, 1973). Menlo Park, Ca.: Stanford Research Institute.

Brawn, B. S., F. G. Gustavson, and E. S. Mankin 1970. "Sorting in a Paging Environment," Comm. ACM 13(Aug. 1970):483.

CACM (Communications of the ACM) July 1972. "As the Industry Sees It," Comm. ACM 15:506-517.

Cheatham, T. E. and Ben Wegbreit 1972. "A Laboratory for the Study of Automatic Programming," Proc. AFIPS SJCC 40(Sept. 1972):11-21.

Chomsky, N. 1965. Aspects of the Theory of Syntax. Cambridge: MIT Press.

CR (ACM Computing Reviews) Jan. 1974. "Categories of the Computing Sciences," ACM Computing Reviews 15(Jan. 1974):43-44.

Dahl, O.-J., B. Myhrhaug, and K. Nygaard 1968. The SIMULA 67 Common Base Language. Norwegian Computing Centre, Forskningsveien 1B, Oslo 3.

Dahl, O.-J. 1972. "Hierarchical Program Structures," in Structured Programming, O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, eds. London: Academic Press.

Dijkstra, E. W. 1972. "Notes on Structured Programming," in Structured Programming, O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, eds. London: Academic Press.

Dostert, Bozena H. 1970. REL -- An Information System for a Dynamic Environment. REL Project Report No. 3. Pasadena, Ca.: Calif. Inst. of Tech., Dec. 1970.

Dostert, B. H. and F. B. Thompson 1971. The Syntax of REL English. REL Project Report No. 1. Pasadena, Ca.: Calif. Inst. of Tech., 1971.

Dostert, B. H. and F. B. Thompson 1972. "Syntactic Analysis in REL English: A Computational Case Grammar," Statistical Methods in Linguistics 8(1972):5-38.

Dostert, B. H. and F. B. Thompson 1973. Verb Semantics in a Relational Data System, REL Project Report No. 6. Pasadena, Ca.: Calif. Inst. of Tech., 1973.

Gomberg, Sara 1973. The REL Command Language. REL Project Report No. 8. Pasadena, Ca.: Calif. Inst. of Tech., 1973.

Greenfeld, N. R. 1972. Computer System Support for Data Analysis. Ph.D. Dissertation, Calif. Inst. of Tech., Pasadena, Ca.

Habermann, A. N. 1973. "Critical Comments on the Programming Language Pascal," Acta Informatica 3(1973):47-57.

Ingargiola, Giorgio P. 1974. "Hierarchies and Relations Among Data Types," to appear in Proc. ACM National Conference (Nov. 1974).

Irons, E. T. 1961. "A Syntax Directed Compiler for Algol 60," Comm. ACM 4(Jan 1961):51:55.

Keyser, S. J. and S. R. Petrick 1967. Syntactic Analysis, AFCRL-67-0305. Bedford, Mass.: L. G. Hanscom Field, Air Force Cambridge Research Laboratories.

Kay, Martin 1967. Experiments With a Powerful Parser, RM-5452-PR. Santa Monica, Ca.: Rand Corporation.

Knuth, Donald E. 1968. "The Semantics of Context Free Languages," Mathematical Systems Theory 2(June 1968):127-146.

_____, 1973. A Review of "Structured Programming," STAN-CS-73-371. Stanford, Ca.: Stanford University, Computer Science Dept., June 1973.

Kuhn, Thomas S. 1970. The Structure of Scientific Revolutions, Second Edition, International Encyclopedia of Unified Science, Vol 2, No. 2. Chicago: University of Chicago Press, 1970.

Leavenworth, Burt M. and Jean E. Sammet 1974. "An Overview of Nonprocedural Languages," (Proc. Symp. on Very High Level Languages) ACM SIGPLAN Notices 9(April 1974):1-12.

McKeeman, W. M., J. J. Horning, and D. B. Wortman 1970. A Compiler Generator. Englewood Cliffs, N.J.: Prentice-Hall.

Martin, William A. 1974. "OWL: A System for Building Expert Problem Solving Systems Involving Verbal Reasoning," manuscript. Mass. Inst. of Tech.

Moler, Cleve B. 1972. "Matrix Computations with Fortran and Paging," Comm. ACM 15(Apr. 1972):268-270.

Myrdal, Gunnar 1969. Objectivity in Social Research. New York: Random House, Pantheon Books.

Naur, Peter, ed. 1963. "Revised Report on the Algorithmic Language Algol 60," Comm. ACM 6(Jan. 1963):1-17.

Newell, A., F. M. Tonge, E. A. Feigenbaum, G. H. Mealy, N. Saber, B. F. Green, Jr., and A. K. Wolf 1960. Information Processing Language V Manual. Santa Monica, Ca.: Rand Corporation.

Quine, W. V. 1969. Ontological Relativity and Other Essays. New York: Columbia University Press.

Randall, David L. 1970. Formal Methods in the Foundations of Science. Ph.D. Dissertation, Calif. Inst. of Tech., Pasadena, Ca.

Rosen, S. 1964. "A Compiler-Building System Developed by Brooker and Morris," Comm. ACM 7(July 1964):403-414.

Saltzer, Jerome H. 1974. "Protection and the Control of Information in Multics," Comm. ACM 17(July 1974):388-402.

Sammet, Jean 1972. "Programming Languages: History and Future," Comm. ACM 15(July 1972):601-610.

Schwartz, J. T. 1973a. On Programming: An Interim Report on the SETL Project; Installment 1: Generalities. Courant Institute of Mathematical Sciences, New York University.

_____, 1973b. "Central Technical Issues in the Semantic Design of Procedural Programming Languages," manuscript. Courant Institute of Mathematical Sciences, New York University.

Taft, E. A. 1971. PPL User's Manual. Cambridge: Harvard University, Harvard Extensible Language Project, January 1971.

Teitelman, Warren 1974. Interlisp Reference Manual. Palo Alto: Xerox Palo Alto Research Center.

Thompson, F. B. 1966a. "Man-machine Communication," in Seminar on Computational Linguistics, A. W. Pratt, A. H. Roberts, and K. Lewis, eds. U.S. Department of Health, Education, and Welfare, Public Health Service Publication No. 1716.

_____, 1966b. "English for the Computer," Proc. AFIPS FJCC 29(1966):349-356.

Thompson, F. B. and B. H. Dostert 1972. "The Future of Specialized Languages," Proc. AFIPS SJCC 40(1972):313-319.

Thompson, F. B. and B. H. Dostert 1974a. Practical Natural Language Processing: The REL System as Prototype. REL Project Report No. 13. Pasadena, Ca.: Calif. Inst. of Tech., Jan. 1974. To appear in Advances in Computers, Vol. 13, ed. M. C. Yovits and M. Rubinoff. New York: Academic Press.

Thompson, F. B. 1974b. The REL Language Processor. REL Project Report No. 11. Pasadena, Ca.: Calif. Inst. of Tech., 1974.

Thompson, F. B., R. H. Bigelow, N. R. Greenfeld, J. R. Odden, D. Reece, and P. Szolovits 1974c. The REL Animated Film Language. REL Project Report No. 12. Pasadena, Ca.: Calif. Inst. of Tech., 1974. To appear in Computers and Graphics Vol. 1(1974).

Wegbreit, Ben 1970. Studies in Extensible Programming Languages. ESD-TR-70-297. Bedford, Mass.: L. G. Hanscom Field,Directorate of Systems Design and Development, May 1970.

Wegbreit, Ben 1974. "The Treatment of Data Types in EL1," Comm. ACM 17 (May 1974):251-264.

Wegner, Peter 1972. "The Vienna Definition Language," ACM Computing Surveys 4(March 1972):5-63.

Whorf, Benjamin L. 1956. Language, Thought, and Reality. Cambridge: MIT Press.

Winograd, Terry 1972. Understanding Natural Language. New York: Academic Press.

Wirth, N. 1973. The Programming Language Pascal (Revised Report). Berichtige de Fachgruppe Computer-Wissenschaften, ETH, No. 5. Zurich, Switzerland: Technical University, July 1973.

Wittgenstein, Ludwig 1958. Philosophical Investigations, Third Edition. New York: Macmillan.

Woods, W. A., R. M. Kaplan, and B. Nash-Webber, "The Lunar Sciences Natural Language Information System: Final Report." BBN Report 2378. Cambridge, Mass.: Bolt, Beranek, and Newman, June 1972.

APPENDIX A

THE SYNTAX OF LWL

{1} - Introduction

To explicate the discussion of various constructs of the REL Language Writer's Language in this thesis, this appendix presents an informal syntax of LWL. The level of formality desired in this presentation is that of Wirth's PASCAL Report [1973]. This means that the language description is accurate, but does not comprehend in detail the actual implementation.

The language writer who uses LWL's sophisticated definitional capabilities may need to refer to the following syntactic description of the language, to understand the meaning assigned to his paraphrase definitions, and especially to understand the ontology of his metavariables.* For the less daring language writer, the operations of the LWL are rather straightforward, and its grammar will rarely introduce difficulties.

Because LWL is intended to parallel PASCAL wherever that

- - - - - - - - - -

*C.f. Chapter IV, Section 4.2.1.

language has been adequate to our task, its syntactic definition will take advantage of the excellent and widely available definition of the PASCAL language [Wirth, 1973]. The discussion below will follow the organization of the PASCAL Report and will explicitly define only those features of LWL which differ significantly from PASCAL (except where clarity demands greater completeness). Section headings are numbered by corresponding section numbers from the Report. Section 10, on procedure declarations, is essentially omitted, and Section 12, on the rule statement, is added.

{2} - Summary of the Language

The REL Language Writer's Language is a metalanguage in which new, specialized object languages may be defined for the REL System. An object language consists of: its extended syntax, the rules of grammar by which the REL language processor interprets utterances of the language; and the data representation, the declaration of the types of objects in the language's universe of discourse (data structures) and the operations which may be performed upon them (functions).

In addition to the standard scalar types, Boolean, integer, char and real, and the user-defined scalar types, LWL introduces the scalar type function, which is an arbitrarily (but consistently) ordered set of values, each of which is a primitive or user-defined function. This

permits the construction and manipulation of data types which include functions as components. Two "special" types of data items required in the processing of grammar rules, the part_of_speech and features are in fact represented in terms of the above scalar types. Parts of speech are made equivalent to the non-printing values of the data type char, and the features used to subcategorize each particular part of speech form a scalar type, whose name is

<pos>.features

Subrange is retained as an abbreviation for the scalar type and concomitant advice to the compiler about the required amount of storage needed for a variable. The values of a variable declared of type subrange, however, do not differ from the values of the type on which the subrange is based. Further, subranges may not be formed of the scalar types real and function.

The PASCAL structured data types array, record and set are provided. For the record type, a storage class is an additional possible attribute. The three storage classes are stack, list and page, and correspond to allocating the record's corresponding storage on the ALGOL-like stack, in a garbage-collected list space or in the paged virtual memory.* The set structure may not have a base type of real or

- - - - - - - - - -

*Certain specific limitations and defaults exist on the maximum size and organization of record types of the various storage classes: The maximum size of a page record is the REL System virtual memory page size (currently 2048 bytes); list records are each of fixed length (currently 12 bytes) and the first field of every list record type must be of type char (required by the garbage collector). If it is not explicitly

function, and should ordinarily not be based on integer. There is no file structure.

Variables of pointer type may be used to dynamically reference storage classed record variables (not arbitrary variables, as in PASCAL). Pointer variables may have the value nil or any record of the class and type to which the pointer variable is bound. A pointer may also be bound to a subtype of a defined type, namely some pre-selected part of a larger data type.

In the declaration of variables, the issues of scope and persistence arise. In PASCAL, the scope of all locally declared variables is the procedure or function in which they are declared, and every variable persists throughout its scope, with the exception of those unnamed, dynamically created variables which may be referenced only through pointers.

In LWL, all record variables of the list or page storage class are of that latter, dynamic kind. No variable may be declared with a type that is a page or list storage classed record -- only the function new may be used to create such variables dynamically, and they may be referenced only through pointer variables bound to their type. Variables of the page class persists until they are either explicitly

- - - - - - - - - -

declared so, the declaration is augmented to satisfy this requirement, and the newly introduced anonymous variable is assigned an initial value consistent (for the garbage collector) with the structure of the rest of the record.

removed by the function free or implicitly freed by the deletion of the complete version. Variables of the list class persist until either no more pointer variables reference them or the user session is terminated.*

All static variables must be declared with data types with storage classes other than page or list. Ordinarily, their persistence is co-terminous with the invocation of the function in which they were declared, and their scope is exactly that function, excluding other functions called by it. The special declarations common and global give such a variable universal scope; global insures the same persistence as the current session, and common the same as the version.**

Assignment and the basic operators are as in PASCAL, except that the ordering relations also apply to pointer and function types. There are no program or procedure declarations, the functional parameter and evaluation mechanism is quite different*** and there are extensions of the for and while statements.

The extended syntax of the object language is introduced by the use of the rule statement. Its syntax is defined in Section 12, below.

- - - - - - - - - -

*C.f. chapter IV, Section 4.4.1.

**Note that the value of a pointer variable declared common and bound to a list record type becomes meaningless after the end of a session, and that dynamic variables of the page class may be permanently lost to a version at the end of a session unless they are pointed to by a variable declared common.

***C.f. Chapter IV, Section 4.3.3.

{3} - Notation, terminology, and vocabulary

The syntactic description of LWL will use an extended Backus-Naur form. The semantic part of the rule definitions will be left out, and the semantics will be described informally or left implicit if obvious. The LWL syntax uses features to keep track of the source from which a particular phrase derives. For instance, <scalar_type,+real> and <scalar_type,+integer> are both instances of <scalar_type>, both recognized using the rule

<scalar_type> ::= (<identifier> {,<identifier>}) | integer | real | Boolean | char | function

but with the further requirement that it was the real or integer alternate of the scalar_type rule which applied to them. This provides a systematic syntactic means for taking into account grammatical information not explicitly specified by the PASCAL grammar rules. Unless otherwise specified, assume that each part of speech is marked by features indicating its actual derivation, as in the above example.

The optional presence and repetition (zero or more times) of an item in the syntax will be represented by enclosing it in a pair of the metabrackets { and }. The vertical bar, | , will indicate alternatives, and the metasymbol, ::= , will be the production symbol of BNF.

It must be noted that because LWL is itself an extensible language, its syntax as presented here is only an accurate statement before the language writer has commenced. For example, the definition

of a new <record_type> adds to the LWL syntax grammar rules which recognize the newly defined type identifier and the various field identifiers. The PASCAL definition's ubiquitous use of the syntactic category <identifier> is avoided by identifying newly introduced identifiers with their syntactic type in a new rule of LWL.

The vocabulary consists of the basic symbols

```
<letter> ::= A | B | C | D | E | F | G | H | I | J | K | L | M | N |
    O | P | Q | R | S | T | U | V | W | X | Y | Z | a | b | c | d |
    e | f | g | h | i | j | k | l | m | n | o | p | q | r | s | t |
    u | v | w | x | y | z | _

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<special symbol> ::= + | - | * | / | ! | & | > | < | ( | ) | [ | ] |
    { | } | := | . | , | ; | : | ' | @ | ^
```

The words div, mod, not, nil, in, if, then, else, case, of, repeat, until, while, do, for, to, begin, end, with, const, var, type, category, define, array, record, set, page, list, stack, function, rule, etc., although they are standard words of the language, are not considered reserved or quoted. It is the language writers responsibility to avoid usage that would cause ambiguity. A comment, delimited by the symbols { and }, may appear in the text anywhere that a blank may appear (except inside strings) and has the same meaning.

{4} - Identifiers, Numbers, and Strings

Identifiers denote constants, types, variables, functions, rules, parts_of_speech and features. An identifier may denote several of these at the same time, so long as its use in any particular instance is unambiguous. Identifiers are defined as in PASCAL. Numbers are expressed in the normal decimal notation, as in PASCAL, but with the caret (^) used to show the scale factor instead of E.

Sequences of characters enclosed in quote marks (') are called strings, as in PASCAL. Strings may appear only as <simple_expression>s.

{5} - Definitions

In PASCAL, constant definition and data type definition are treated as distinct. LWL subsumes both under the more general operation of definition. Syntactically, this is provided by the define statement

<definition> ::= define <definiendum> = <definiens> |
<constant_definition> | <type_definition>

The definiendum is a sequence of metavariables and characters other than =. Metavariables are enclosed in " marks, and either identify the LWL part of speech for which they stand explicitly or provide an example from which the part of speech is determined. Thus,

<metavariable> ::= "<LWL_part_of_speech>" | "any_phrase_of_LWL"

The alternative "<LWL_part_of_speech>" is actually written with the

angle brackets, and refers to the left hand side of any of the grammar rules of LWL, e.g.

"<expression>"

The definiens is an LWL phrase, possibly including metavariables. The meaning of this rule is similar to adding to the grammar of LWL the general rewrite rule

definiens ::= definiendum

It is similar rather than equivalent, because the rule is not entered as stated; if the definiens parses to a part of speech <ps>, then the rule added is

<ps> ::= definiendum

with a def semantics which contains the phrase marker of the definiens. Note that the requirement that the definiens must parse before the rule is added eliminates the possibility of general rewrite rule and recursive definitions.

In detail, the operation of the definition rule is rather complicated. For example, it is necessary to determine that parse of the definiens which is minimally sufficient to span it, and it is necessary to identify the metavariables which are to participate in the definition. This is more difficult than might first appear, since so little is assumed about the form of the definiendum. Thus, a definition like

define "x".."y" = sin("y")

is meaningful, but here the "x" are merely symbols of the defined phrase, and do not serve as a metavariable. The actual defining process works as follows:

I. Find all parses of the definiens. If there are none, the definition is meaningless and fails. Eliminate any parses in which the last rule applied was of the form
<non_terminal> ::= <non_terminal>
to get the minimal parses of the definiens. If there is more than one such parse, the definition is ambiguous and fails.

II. Consider all possible sequences of terminal characters and metavariables of the definiendum. E.g., for the above example, the possible sequences are
"x"..<metavariable for variables like y>
<metavariable for variables like x>.."y"
<metavariable for variables like x> .. <metavariable for variables like y>
"x".."y"
where the first is the correct interpretation.

III. Find all pairs of <sequence from II, parse from I> which have correspondingly the same sets of metavariables. If there is more than one such pair, the definition is ambiguous and fails. If there are no such pairs, the definition is meaningless and fails. If there is exactly one such pair, then a new grammar rule is added to LWL, where the right had side is the sequence found in II., and the left hand side is the part of speech of the parse from I., with a semantics which is the phrase of that parse and with the corresponding metavariables bound to each other.

{5.1} - Constant Definition

Constant definition is a special case of <definition>, in which the definiendum is an identifier and the definiens is a constant expression. A constant definition introduces an identifier as a synonym for a constant. Any expression which may be evaluated to a constant

value at the time of constant definition may appear in the definition, allowing constants to be defined in terms of other constants. This includes the application of standard functions, which return constant values given constant constituents.

<constant definition> ::= <identifier> = <expression,+constant>

A constant definition, like

const limit = 8;

has the effect of adding to the LWL grammar the rule

<constant> ::= limit

with the meaning that limit evaluates to 8.

In addition, numbers and the null pointer are constants:

<constant> ::= <unsigned_number> | nil

{6} - Data Type Definitions

A data type identifies a semantic category of objects. It thus determines the set of values which variables of that type may assume. A data type definition extends LWL by introducing an identifier whose meaning is the newly defined type. In addition, the category data type definition identifies a semantic category of the object language, and therefore, a syntactic part of speech.*

- - - - - - - - - -

*The relationships among the type, define and category definitions are discussed in Chapter IV, Section 4.2.2.

```
<type_definition> ::= type <identifier> = <type> |
    category <identifier> = <type>

<type> ::= <simple_type> | <pointer_type> | <structured_type>
```

{6.1} - Simple types

The simple types are scalars, both standard and user defined, and the restricted ranges of these, called the subrange types. The counting types form the valid base types for sets and array indices.

```
<simple_type> ::= <scalar_type> | <subrange_type>
<scalar_type> ::= (<identifier> {,<identifier>}) | integer | real |
    Boolean | char
<subrange_type> ::=
    <constant,-real-function> .. <constant,-real-function>
<counting_type> ::= <scalar_type,-real-function> | <subrange_type>
```

{6.2} - Structured types

Structured types are aggregations of components, co-ordinated by a structuring method and perhaps specifying a storage class. The structuring methods are array, set and record, and the storage classes are page, list and stack.

```
<structured_type> ::= <array_type> | <set_type> | <record_type> |
    <subtype>

<array_type> ::=
    array [<counting_type> {,<counting_type>}] of <type>

<set_type> ::= set of <counting_type>

<record_type> ::= <unclassed_record_type> | <classed_record_type>
<classed_record_type> ::= <storage_class> <unclassed_record_type>
<storage class> ::= page | list | stack
```

```
<unclassed_record_type> ::= record <field_list> end
<field_list> ::= <fixed_part> | <fixed_part>;<variant_part> |
    <variant_part>
<fixed_part> ::= <record_section> {;<record_section>}
<record_section> ::= <identifier> {,<identifier>} : <type>
<variant_part> ::= case <identifier> : <counting_type,+identifier>
    of <variant> {;<variant>}
<variant> ::= <case_label_list> : (<field_list>) |
    <case_label_list> :
<case_label_list> ::=
    <constant,-real-function> {,<constant,-real-function>}
```

A further structured type, the subtype, is a selected component of a structured type, often used as the target of pointer types where the desired referent is known to be some element of a larger known structure. The subtype is syntactically represented much like a variable denotation, but with the arbitrary * as the only array index permitted. The feature mechanism is used to assure that only classed record types are subject to subtype definition, and the subtype acquires the storage class of its base type.

```
<subtype> ::= <subtype_head> | <subtype_component>
<subtype_head> ::= <structured_type,+identifier+classed_record> .
    <field_identifier>
<subtype_component> ::= <indexed_subtype> | <subtype_field>
<indexed_component> ::= <subtype,+array> [* {,*}]
<subtype_field> ::= <subtype,+record> . <field_identifier>
```

As in constant definition, the application of many of these rules has as side effect the extension of LWL's grammar by the addition of new lexical rules. The scalar type rule adds rules of the form

```
<constant> ::= identifier
```

for each constant of the new user-defined scalar type. The record section and variant part rules add other rules of the form

<field_identifier> ::= identifier

for the tag and other fields of a structure.

{6.3} - Pointer types

Pointer types may have as their targets only storage-classed record types or subtypes, and the target type must be named by an identifier.

<pointer_type> ::= @ <structured_type,+identifier+classed_record>

{7} - Declaration and Denotation of Variables

Variables are declared and referenced in the same manner as in PASCAL, with the exceptions that intermediate field identifiers may be omitted where no ambiguity can result, and pointer references may also be omitted, in which case the denotation is assumed to represent the shortest path in a breadth-first, left-to-right search of the connected data types. Field identifiers determining among variants of a case record may not be omitted.

{8} - Expressions

Expressions are very much as in PASCAL, with a few minor differences:

```
<factor> ::= <variable> | <constant> | <function designator> | <set>
    | (<expression>) | not <factor>
<set> ::= [<expression> {,<expression>}] | []
<term> ::= <factor> | <term> <multiplying_operator> <factor>
<simple_expression> ::= <term> |
    <simple_expression> <adding_operator> <term> |
    <adding_operator> <term> | <string>
<expression> ::= <simple_expression> |
    <simple_expression> <relational_operator> <simple_expression>
```

The relational operators for comparison are extended to operate as well on pointer variables with the same target types and on function variables. Features are used to mark whether an expression has all constant components and the type of the result of expression evaluation. An alternative form of function calls is also permitted:

```
<function_designator> ::= <function,-variable> |
    <function> (<constituent> {,<constituent>}) |
    <constituent> . <function,-variable> |
    <constituent> . <function,-variable>
        (<constituent> {,<constituent>})
```

{9} - Statements

Statements are executable segments of algorithms. The empty statement and assignment statement remain as simple statements, along with a limited version of PASCAL's <procedure statement>. The <goto statement> and its corresponding <label>s have been eliminated. Within compound statements, definition statements are treated as null statements, allowing definitions to be interspersed with other statements of a function declaration.

In the assignment statement, only the form

<assignment_statement> ::= <variable> := <expression>

is allowed, since functional values are to be returned by the standard functions return and ambig.

The structured statements <compound_statement>, <conditional_statement>, and the <while_statement> and <repeat_statement> are as in PASCAL. The for and while statements differ.

The for statement has the expanded form*

```
<for_statement> ::= for <entire_variable> := <for_list> do
    <statement>
<for_list> ::= <expression 1> to <expression 2> |
    <expression 1>,<expression 3>,...,<expression 2> |
    all <counting_type> | all <variable,+pointer>
```

The <entire variable> which receives the new values on each loop of the for is called the control variable. It and any value assigned to it must be of the same type.

The with statement is considerably modified, to provide a simple syntactic form for control over the REL System's software paging mechanism.

```
<with_statement> ::= with <record_binding_list> do <statement>
<record_binding_list> ::= <record_binding> {,<record_binding>}
<record_binding> ::= <record_variable> |
    <identifier> : <record_variable> | <array_variable,+pointer> |
    <identifier> : <array_variable,+pointer>
```

The first alternative for <record_binding> lies closest to the

- - - - - - - - - -

*C.f. Chapter IV, Section 4.3.1.

purely syntactic use of with chosen by PASCAL. In LWL, however, the with statement is executed, and if the <record variable> has the storage class page, the with surrounds its constituent statement with calls on the REL paging services to guarantee appropriate access to the record within the scope of the statement. The second alternative additionally introduces a local synonym for the record variable, so that if in the constituent statement a field reference must be disambiguated, the local identifier may be used in place of the (often longer) actual variable denotation.

The third alternative is a special construct to aid in the writing of functions which pay close attention to paging. It allows an array, each of whose elements is a pointer to a record structure, to appear in the binding list. Its effect is that every non-nil element of the array is used to identify a record which is made available. The fourth alternate is analogous to the second.

In addition, any field selection in a record binding which involves a variant of a case record implies that the case validity is verified once only, at the beginning of the with statement.

Some calls on standard functions may take the form of additional statement types by the omission of otherwise demanded parentheses. For instance,

```
return x;
```

takes the place of the <procedure_statement>

```
return (x);
```

{10} - Procedure Declarations

LWL does not provide for the declaration of user procedures.

{10.1} - Standard procedures

The following standard procedures exist:

return -- returns its single argument as the value of the function currently being executed.
ambig -- adds its single argument to the list of ambiguous values which will be returned by this function when return is invoked.
fail -- causes the current function to fail; this causes the current interpretation of its phrase to be abandoned.
error -- returns its single argument as the out-type value for the current function, causing the noted error to propagate to the top level of evaluation.
vacuous -- like error, but the output propagates only so long as no valid alternative interpretation of this phrase is encountered.*
message -- writes its argument immediately to the user.

{11} - Function Declarations

LWL supports functions and prefix functions, which differ in that the constituents of a prefix function are not evaluated when the function is invoked. Because the type information of a function's parameters and value can be deduced if the function appears as the

- - - - - - - - - -

*C.f. [Thompson, 1974b].

semantics of a rule of grammar, the function declaration need not include these type declarations for such functions.

```
<function_declaration> ::= <function_heading> <compound_statement>
<function_heading> ::= <function_word> <identifier> |
    <function_word> <identifier> <parameters>
<parameters> ::=
    (<formal_parameter_section> {,<formal_parameter_section>})
    : <type,+identifier> | : <type,+identifier>
<function_word> ::= function | prefix_function | subroutine
```

The evaluation of functions and prefix functions and the binding of parameters are performed by the REL language processor.* A subroutine is a restricted form of function, which may be invoked only directly from another function; it may not be invoked by the language processor in response to the application of a grammar rule and it may not fail or return an ambiguous value. The parameters of a subroutine are bound by value. This mechanism is provided to allow an efficient form of function-to-function calling without paying the overhead of the generality of the language processor's evaluation mechanism.

{11.1} - Standard Functions

In addition to the PASCAL standard functions, LWL provides the following:

- - - - - - - - - -

*C.f., Chapter IV, Sections 4.3.2 and 4.3.3, and [Thompson, 1974b].

Simple Extensions of the PASCAL Functions:

The function ord may take any scalar type but real, and returns the ordinal position of that value in the scalar type. Instead of the single function chr, the name of every (non-real, non-integer) scalar type is a function, and so, for instance, char(i) is the i-th character in the character representation.

Dynamic allocation is performed by the function new, which differs from the PASCAL procedure new since its first parameter is not a variable, but a type to be allocated, and its value is a pointer to the newly allocated variable.

Functions for Paging Control

unlocked -- a function of no arguments, it returns an integer which counts the number of yet available page frames for further paging operations.

lock(<page classed record variable>) -- returns its argument with the side effect that the page of virtual memory occupied by the selected record is not subject to automatic replacement until explicitly released or until a fail or error or vacuous statement is executed, or until processing of the current sentence is terminated.

release -- the release function for lock.

Functions for using the Language Processor

evaluate(<phrase>, <list of phrases>) -- calls the semantic evaluator on its first argument; the second argument, if present, is a list of phrases to be bound to the metavariables of the the phrase before it is evaluated.

constituent -- a function of no arguments, yielding the phrase which is currently being evaluated

For each part_of_speech, there is a function of two argument, the first a phrase, the second an optional integer, which selects the

constituent of that phrase with that part_of_speech. The integer, if present, requests selection of the i-th such constituent.

n-amb(<phrase>) -- the number of ambiguous phrases which represent the ambiguous values of the phrase.
amb(<phrase>,i) -- the phrase which represents the i-th of the ambiguous values of the input phrase.

metavar -- a condition function which creates new metavariables.
metabind -- a condition function which binds metavariables.
n_bound(<phrase>) -- the number of metavariables bound in the phrase
metabound(<phrase>,i) -- the i-th bound metavariable phrase in the input phrase.
range(<metavariable phrase>) -- the range of the metavariable, another phrase.
name(<metavariable phrase>) -- the name of the metavariable, a string (array of char).

extend(<list of phrases>, <list of phrases>, function, string) -- the arguments are, in turn, the phrases of the left hand side and right hand side of the new rule to be added, the condition function for the rule, and the name of the rule.

The interface with the language processor is further specified in a document on that subject, [Thompson, 1974b].

{12} - Rules

The object language to be implemented is defined in terms of a set of syntactic rules which allow the language processor to interpret sentences of the language in terms of its defined data objects and functions. the rule statement has the form:

<rule_statement> ::= <identifier> rule <left_hand_side> <production> <right_hand_side> : {<semantics>}
<production> ::= :(<function>):= | ::=

The identifier names the rule for future reference; the first form of

the <production> allows the specification of a condition function to be invoked when the parser is about to apply this rule.

The left and right hand sides of the rule are lists of parts_of_speech, including specification of feature matching and setting requirements.

```
<left_hand_side> ::= <part_of_speech_list>
<right_hand_side> ::= <part_of_speech_list>
<part_of_speech_list> ::= <terminal> | <non_terminal> |
    <part_of_speech_list> <terminal> |
    <part_of_speech_list> <non_terminal>
<terminal> ::= <string> | <string_begin> | <string_end> |
    <input_terminator> | <carriage_return>
<non_terminal> ::= < <part_of_speech> > |
    < <part_of_speech>,{<feature_check_or_set>} >
<feature_check_or_set> ::= <feature_sign> <feature> |
    <unsigned_integer>
<feature_sign> ::= + | - | *
```

The semantic specification of a rule mentions the semantic functions which are to compute the meaning of each non-terminal phrase of the rule's left hand side and the associated syntactic transformations which are to take place on the phrase marker which represents that value.

```
<semantics> ::= (<unsigned_integer>) | (<function>) |
    (.<prefix_function>) | ({<transformation>},<function>) |
    ({<transformation>},.<prefix_function>)
<transformation> ::= < <unsigned_integer> > |
    < <unsigned_integer>,* > |
    < <unsigned_integer>,<part_of_speech>{,<unsigned_integer>} > |
    < <unsigned_integer>,*-<part_of_speech>{,<unsigned_integer>} >
```

The meaning of the rule statement is discussed in Chapter V.

The features statement allows the definition of features to subcategorize parts of speech:

<features_statement> ::= features <part_of_speech> =
    [<identifier>{,<identifier>}]

APPENDIX II

THE SYNTAX OF BIBLIO

The BIBLIO language was presented informally in Chapter III, as an example of the kind of language which is easily implementable in LWL. This appendix defines the syntax of BIBLIO, in LWL.*

The basic data input statements are defined by the following syntax:

```
<SENTENCE> ::= <SUBJECT> ' IS PART OF ' <SUBJECT> '.' :
    (RELATE_SUBJECTS)

<SENTENCE> ::= <I_PUBLICATION> '.' : (SAY_OK)

<I_PUBLICATION> ::= <Q_AUTHOR,+EXPLICIT> ' ' <PUBLICATION> :
    (BASIC_INPUT)

<I_PUBLICATION> ::= <I_PUBLICATION> ' ' <SUBJECT> :
    (RELATE_TO_SUBJECT)

<Q_AUTHOR,+EXPLICIT> ::= <AUTHOR> : (BUILD_AUTHOR_LIST)

<AUTHOR> ::= <AUTHOR> ','

<PUBLICATION> ::= <PUBLICATION> ','

<SUBJECT> ::= <SUBJECT> ','
```

- - - - - - - - - -

*Note that the syntax examples drawn from this language for the presentation of Chapter V differ somewhat in detail from the syntax presented here.

The last three rules are merely to allow the inclusion of commas to separate parts of the input statement.

The following rules permit the introduction of new entities into BIBLIO:

```
<ITEM,+AUTHOR> :(COLLECT_ITEM):= 'AUTHOR: '
<ITEM,+SUBJECT> :(COLLECT_ITEM):= 'SUBJECT: '
<ITEM,+PUBLICATION> :(COLLECT_ITEM):= 'PUBLICATION: '
<AUTHOR> ::= <ITEM,+AUTHOR> : (CREATE_AUTHOR)
<SUBJECT> ::= <ITEM,+SUBJECT> : (CREATE_SUBJECT)
<PUBLICATION> ::= <ITEM,+PUBLICATION> : (CREATE_PUBLICATION)
```

Thus, the creation of new entities is a two_stage process: During parsing, the collect_item codition function collects the string of characters to the right of its constituents (up to a semicolon) into an <item>.* Then, the semantic functions like create_author actually create the specified entity and enter into the dictionary its name (and any desired aliases).

The following rules define the syntax of questions:

```
<SENTENCE> ::= <Q_AUTHOR> '?' : (PRINT_AUTHORS)
<SENTENCE> ::= <Q_SUBJECT> '?' : (PRINT_SUBJECTS)
<SENTENCE> ::= <Q_PUBLICATION> '?' : (PRINT_PUBLICATIONS)
```

- - - - - - - - - -

*It must also manipulate the parsing graph to show the <item> spanning the whole string, including the terminating semicolon.

The only way to get an author except by explicitly naming him is by using the rule

```
<Q_AUTHOR> ::= 'AUTHOR OF ' <Q_PUBLICATION,+COMPLETED> :
    (AUTHORS_OF)
```

To request a list of subjects, the following may be used:

```
<Q_SUBJECT> ::= 'TOPIC OF ' <Q_PUBLICATION> : (TOPICS_OF)

<Q_SUBJECT> ::= 'GENERALIZATION OF ' <Q_SUBJECT> : (GENERALIZATION)

<Q_SUBJECT> ::= 'PARTS OF ' <Q_SUBJECT> : (SPECIALIZATION)

<Q_SUBJECT> ::= <SUBJECT> : BUILD_SUBJECT_LIST

<Q_SUBJECT> ::= <Q_SUBJECT> ' ' <SUBJECT> : (BUILD_SUBJECT_LIST)
```

The computation of lists of publications is most complicated due to the need for conjuncted and disjuncted phrases. The simple requests for publications are accomplished by the following rules:

```
<Q_PUBLICATION+BY> ::= 'BY ' <Q_AUTHOR> : (WORKS_BY)

<Q_PUBLICATION+ABOUT> ::= 'ABOUT ' <Q_PUBLICATION> : (WORKS_ABOUT)

<Q_PUBLICATION,+RATED> ::= <RATING> '-RELEVANT TO ' <Q_SUBJECT> :
    (WORKS_RELEVANT)

<Q_PUBLICATION,+COMPLETED> ::= 'WORKS ' <Q_PUBLICATION,-COMPLETED> :
    (1)
```

Conjunctions and disjunctions where only "works" is shared (e.g., "works by Quine and about ontology") are simply handled by the following:

```
<Q_PUBLICATION,+CONJUNCTED> ::=
    <Q_PUBLICATION,-DISJUNCTED-COMPLETED> ' AND '
    <Q_PUBLICATION,-DISJUNCTED-CONJUNCTED-COMPLETED> : (Q_AND)

<Q_PUBLICATION,+DISJUNCTED> ::= <Q_PUBLICATION,-COMPLETED> ' OR '
    <Q_PUBLICATION,-DISJUNCTED-COMPLETED> : (Q_OR)
```

More elliptical constructions in which significant information must be distributed over the various constituents (e.g., "works B-relevant to extensible languages or natural language") are accepted by general rewrite rules:

```
<Q_PUBLICATION,+BY> ' AND BY ' <Q_AUTHOR> ::= <Q_PUBLICATION,+BY> '
    AND ' <Q_AUTHOR) : (1) (2)

<Q_PUBLICATION,+BY> ' OR BY ' <Q_AUTHOR> ::= <Q_PUBLICATION,+BY> '
    OR ' <Q_AUTHOR) : (1) (2)

<Q_PUBLICATION,+ABOUT> ' AND ABOUT ' <Q_SUBJECT> ::=
    <Q_PUBLICATION,+ABOUT> ' AND ' <Q_SUBJECT> : (1) (2)

<Q_PUBLICATION,+ABOUT> ' OR ABOUT ' <Q_SUBJECT> ::=
    <Q_PUBLICATION,+ABOUT> ' OR ' <Q_SUBJECT> : (1) (2)

<Q_PUBLICATION,+RATED> ' AND ' <RATING> '-RELEVANT TO ' <Q_SUBJECT>
    ::= <Q_PUBLICATION,+RATED> ' AND ' <Q_SUBJECT> : (1)
    (<1,<RATING>>) (2)

<Q_PUBLICATION,+RATED> ' OR ' <RATING> '-RELEVANT TO ' <Q_SUBJECT>
    ::= <Q_PUBLICATION,+RATED> ' OR ' <Q_SUBJECT> : (1)
    (<1,<RATING>>) (2)
```

Note the effective use of general rewrite rules and transformations.

This completes the BIBLIO syntax.