both busy by allowing a number of users to share the resources of the system at one time.

Early conversational systems were an attempt to put some of the aesthetics of program development back into computing, and many had as customers novice programmers who, it was suspected, would be easily "turned off" by the plethora and complexity of control cards necessary to run a job on some systems. Some of these systems, notably JOSS [Ba 66] and BASIC [KK 66] involved languages which were relatively simple to learn although limited in the scope of their applications. The amazing observation about these systems is the amount of constraint of expression which was tolerated because of the other appealing aspects of the systems, in particular, the level of interaction which they provided as compared with their contemporary systems.

Interaction can be of value in three primary aspects of computing: program design and development, debugging, and program execution. A programmer can compose programs at his console, simply treating the terminal as a scratch pad for writing down parts of programs as they are developed. He can also alter and change the program as errors are detected. And, finally, he is able also to use the program interactively by having it request input from him and then displaying the results to him.

When compared to the first interactive systems, bare computers, current conversational programming has improved in a number of ways. First, higher level languages, which are easier and more natural for the user, and more directly applicable to his problems, have taken the place of primitive machine language. Structure, in terms of variables and programs, has replaced simple binary words in memory as the fundamental units of information. In place of switches, buttons, and lights on the computer, the user now has a typewriter device on which both he and the computer can type, in characters and numbers instead of just binary zeroes and ones. Such devices also have buttons which can be used to "interrupt" the computer and so allow the user to gain control of the conversation. A growing number of systems designed for interactive use take advantage of cathode ray tube displays and so-called "tablets" on which one may write with a stylus.

1B The Evolution of Interactive Programming

In the early days of computers, programs were typically written in machine language, loaded into the computer's memory, and run by the programmer himself, who then had full use and control of the machine. During such a period he might simply run a problem, take the answers which were printed, and depart. Of course, any reasonably complex program has errors in it initially. In such cases, the programmer could usually set switches or actually alter his program in memory to cause it to stop at some selected points during its execution. When it paused, the user could display memory locations to inspect directly intermediate results, values of variables, and so on. He was in some sense conversing with the computer. The catch was this: he was constrained to interact with the computer only in language directly understandable by it, binary words and manually set switches. The context intelligible to the computer was limited to the words and bits in an unstructured memory of limited size, and the user had to translate all requests and commands into this context. The transmission media, namely switches, buttons, and lights, were clumsy and slow to use. Nevertheless, the rate of interaction was reasonable, aside from the translation problem for the human being, and it was possible for the user to maintain his context (the program being tested) well enough to make progress.

Simon [Si 66] has pointed out that this became more and more wasteful of the computer's time as machines became bigger and faster, and partly, at least, prompted the first batch processing systems. During the era of batch systems (1954 to the present), developments were made in programming languages, loaders, program libraries, and so on, which increased the utility of the computer. Stacked batch systems allowed the computer to be used more efficiently, but, as Simon states [Si 66]:

> "The subsequent era of batch processing was distressful to programmers, because it did not eliminate the man-computer imbalance, but simply reversed its direction. Now the programmer had to wait for the response of the computer, and unless he could switch to another task while waiting, much of his time was idle."

The advent of time-sharing systems kept the programmer and computer

and less efficient than those accessed by compiled code in a more rigid environment. Indeed, for the important case of the interpreter/compiler spectrum, we will develop a method of the option (c) variety.

Interactive systems have so far been used, almost without exception, for relatively small tasks. One of our goals will be to design a systems methodology for constructing interactive systems of use for large tasks such as building operating systems, or in fact building an IPS.

1A3 Construction

The last part of the thesis deals with the construction of an IPS by bootstrapping it from a small set of primitives. The primitives are those generated by the preceding parts of the research and are few enough and logically simple enough to allow one to move a system built upon them from machine to machine by moving the base only.

These three options also reflect varying degrees of automation of the design process. Option (a) implies that the designer, using some unstated method, chose a single point of compromise (such as either a form of interpretation or compilation in the execution spectrum). Option (b) requires a strategy within the system for choosing the correct point for a given situation, and therefore has automated the choice function, given a set of discrete points; a good example of this would be the decision whether to represent a program in interpreted state or compiled state in a system with both an interpreter and a compiler. And, finally, (c) represents completely automated design, for it is able to pick any point out of the spectrum which corresponds to a given situation and use it. The scheme for a combined interpreter and compiler to be described in the thesis is of this latter form. Of course, the design of a scheme of the option (c) type is the designer's problem: we have really only removed the problem one level. Nevertheless, (b) and (c) can still provide a system which is responsive to changes in its operating environment, hopefully in an optimizing fashion.

This representation of the design problem can be called a spectral philosophy: whenever there is more than one instance of a structure or feature in an IPS, those instances will be considered as bands in a spectrum, and we will investigate and hopefully define its limits and possibly other lines of interest in that spectrum. For instance, the representations of data structures to be used in a highly interactive and changeable environment will be more general

into three categories: aesthetics, architecture, and construction.

1A1 Aesthetics

In order to know what features can make an IPS which really aids people's work and which they will want to use, we need to establish some evaluative and synthetic criteria for designing an interactive system and judging new features, however introduced. The latter part of this chapter will deal with this extremely important area of interactive systems.

1A2 Architecture

In designing an interactive system, we are faced with two apparently opposing constraints: flexibility for the user, and efficiency for the computer. They can be viewed as lines in a spectrum over program execution, extending from flexibility-inefficiency at one end to inflexibility-efficiency at the other. The options open to a designer in this perspective are the following (listed in order from least to most desirable):

(a) considering the design constraints, pick a single "point" in a spectrum and design an implementation for it;

(b) pick a number of distinct points in a spectrum and make the system able to select a single point depending on the state of the system when it is running; or

(c) find a method by which the system can move in a continuous manner along that spectrum, in response to changes in the state of the system.

For each spectrum (or dimension) which is of interest in a specific system design, one of these options will be chosen. The degree to which the system responds to changes in that spectrum will depend on the option used. If option (a) is used, the system will be able to handle only situations whose characteristics in that spectrum match the point chosen. When (b) can be used, the system will be able to treat a number of discrete characteristics well, but must still map each situation onto one of the chosen points. Option (c) will allow the system to respond to any situation along that spectrum by picking a point according to the characteristics of a given situation.

structures and program dynamics are, by themselves, distinctly non-trivial. By factoring these concerns out of the overall problem, we hope to gain insight into them, both individually, and jointly, as they affect one another. By treating them individually, we hope to gain an appreciation, with respect to name spaces, for instance, for the time and lexical scopes of names in programs, and suitable (implementable) models for them. By also dealing with these issues jointly, we will be able to view better the interfaces between them: e.g., how the representation of control mechanisms and name spaces influence the discussion of data structures since they are important instances of those structures.

The integration of these considerations and the interpreter/compiler methodology will yield a set of primitive operations and control which can act as the base of an IPS. The remainder, and a large part of the bulk of such a system is then interactively bootstrappable and modifiable. Moreover, by showing that those primitives are few and simple we hope that this base will be, if not highly mobile, at least moderately mobile from machine to machine. Motivation for this approach comes largely from the author's experience with systems in general, and a highly interactive system, LC^2 [MPV 68], in particular, as well as external influences such as the first and second NATO Conferences on Software [NB 69], various other research (such as [Ea 69], [EER 68], and [I1 68]), and much discussion with both peers and patrons.

1A Specific Goals

The aim of this research is to find ways to improve interactive programming systems. For our purposes, this can be subdivided

at almost no extra cost: the usually separate notions of interpretation and compilation are presented as a unified view of program execution.

Flexibility and efficiency are almost continuously singled out as motivation and constraints on much of the work in this thesis. For that reason we wish to make definitions for these two terms which are more specific than their normal English semantics. Flexibility is the ease with which the user of an interactive programming system (IPS) can manipulate the objects of interest to him; this universe of discourse includes programs, variables, data and control, in their various representations. Thus, a system in which lines of program text can be altered by the user would be deemed more flexible than one in which they cannot. Efficiency can be viewed as the ease and grace with which the system can perform a given task; alternatively, the amount of overhead (due to maintaining user flexibility) in an IPS is an inverse measure of that system's efficiency. Note that flexibility and efficiency actually reflect a symmetric situation: efficient "use" of the human is achieved by flexibility in the system and the system can be more efficient if the user is more flexible and can work in units which are easy for the machine to do. What renders the human-machine relation asymmetrical is that this author considers the human as the important element and the machine as a tool to accommodate him and his work.

A number of sub-problems which arise in this treatment concerning incremental parsing, name spaces, control mechanisms, data

1. Introduction

The presence of a human as an active control element in a computer system provides a great source of variability - one able to tax the talents of the designers of an interactive programming system (IPS). This research is an attempt at understanding the form and source of that variability, and using it as a guide and motivation for the study of the design and construction of conversational systems.

We will treat this study first by giving a survey of some existing interactive systems, and a development of the features of such systems which make them conversational. Some of these features are natural extensions of similar notions in non-interactive programming systems; others have arisen in research by the author and others [MPV 68, EER 68, Si 66] into interaction; and still others are the results of considerations of the psychology of humans in an interactive computer environment.

The remainder and bulk of the thesis deals with some synthetic tools for the building of interactive systems, although some of the techniques have direct applicability to other types of systems and to programming languages. Of central concern is the development of a scheme which combines the advantages of an interpreter (for flexibility) and a compiler (for efficiency) within one system in such a way that part of the user's programs may be partially interpreted or compiled depending on their use and constancy over some period of time. This interpreter/compiler scheme is done in such a way that only an interpreter is written. The compiler comes along at a large "discount,"

Chapter 4, Differential Data Structures and an Extended TFI 4-1

4A Names in an IPS 4A-1

4B Attributes of Variables 4B-1

4C The Tree-Factored Interpreter Generalized 4C-1

Chapter 5, Bootstrapping an IPS (With an Eye to Exportability) 5-1

5A History of Bootstrapping 5A-1

5B History of Exportability 5B-1

5C Bootstrapping an IPS using the CBM 5C-1

5D A Model of an IPS 5D-1

5E Design Considerations for a CBM 5E-1

Chapter 6, Conclusions, Future Research and Summary 6-1

6A Flexibility in Programming Systems 6-1

6B The "Spectrum" Philosophy 6B-1

6C Short Term Research Areas 6C-1

6D Long Range Research Goals 6D-1

6E Summary 6E-1

Appendix A: A Conversational Base Machine A-1

Bibliography

Table of Contents

Chapter 1, Introduction 1-1

1A Specific Goals 1-2

1B The Evolution of Interactive Programming 1B-1

1C Considerations and Criteria for an IPS 1C-1

1D The Interactive Criteria Applied to LC^2 1D-1

Chapter 2, Design Considerations for an IPS 2-1

2A Direct and Stored Statements 2-1

2B Accessing Other Data Structures 2-1

2C The Concept of the User as a Function 2-2

2D Coroutine Control for Interaction 2-3

2E Bootstrapping 2-5

2F Other Features 2-6

2G Monitoring 2-8

2H Interpretation and Compilation 2-9

Chapter 3, Interpretation and Compilation in an IPS 3-1

3A Simple Model of Interpretation as a Pseudo-Machine 3-1

3B Using a Parse Tree to Drive an Interpreter 3B-1

3C Factored Interpretation 3C-1

3D Extending an FI to Parse Tree Interpretation 3D-1

Appendix 3A

Appendix 3B

Acknowledgements

These pages contain few truly original ideas: most are the result of seeds planted by those older and wiser than myself as well as others younger and as keen. It is a pleasure to acknowledge those debts.

Professor Alan J. Perlis has been a source of constant surprise, insight, and enthusiasm. Most of the "important" ideas in this work were initially put forth by him and at first classified by me as impossible! Collaborating with Dr. Perlis on the LC^2 project I count as my most valuable education in these past four years.

There is a great debt due also to Professor Bill Wulf for his continual support, ideas and invaluable criticisms. His role as cheering section bears much of the responsibility for this work's completion.

Projects, like books, seem to possess lives of their own; the LC^2 project, which prompted most of this research, was like that. But it owes its lifeblood to the minds and energies of Hal Van Zoeren, Dave Wile, Joe Newcomer and Ed McCreight. It has been an honor to work and argue with them.

To my wife, Barbara, I owe much for her help, patience and encouragement. She has my thanks, but I think she knows that.

Abstract

This dissertation treats interactive programming systems at three levels:

(1) the design of an interactive programming system (IPS) from considerations of the psychological and intellectual needs of the user;

(2) the implementation of an IPS which is both flexible for the human and efficient of computer usage; and

(3) the construction of an IPS by bootstrapping.

The bulk of the work centers on the second point. In particular, a method is developed for building an interpreter for a language (to achieve flexibility) which also yields a compiler for that language as a (relatively cheap) byproduct. Programs "become" compiled in the process of being interpreted, but are able to respond to changes in the interactive environment as if they were being interpreted only.

With such an interpreter/compiler, bootstrapping becomes a viable means of building an IPS. This is especially valuable in making the interfaces between the human and the system easily malleable to suit the user's needs and abilities.

The Design and Construction of Flexible and Efficient Interactive Programming Systems

by

James G. Mitchell

Department of Computer Science
Carnegie-Mellon University
June, 1970

Submitted to Carnegie-Mellon University
in partial fulfillment of the requirements
for the degree of Doctor of Philosophy

This work was supported by the Advanced Research Projects Agency of the Office of the Secretary of Defense (F-44620-70-C-0107) and is monitored by the Air Force Office of Scientific Research. This document has been approved for public release and sale; its distribution is unlimited.