CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

THE HISTORY OF JOVIAL

A STUDY OF PROGRAMMING LANGUAGE CHANGE

A thesis submitted in partial satisfaction of the
requirements for the degree of Master of Science in

Computer Science

by

Michael Olin Hogan

May 1983

The Thesis of Michael Hogan is approved:

Dr. Diane Schwartz

Dr. David Schwartz

Dr. Russell J. Abbott, chair

California State University, Northridge

ACKNOWLEDGEMENT

I would like to express my appreciation to all my associates in the JOVIAL-Ada Users Group, especially Judy Bamberger, who gave so much of their time in reading the drafts of this paper. I would also like to thank Harvey Bratman, Chris Shaw, Sue Upshaw, Jim Felty, Terry Dunbar, and Dave Ketchum for allowing me to interview them and for lending me their personal documents, many of which are unpublished. Finally, I would like to thank my friend and advisor, Dr. Russ Abbott, without whose guidance this thesis would not have come about.

TABLE OF CONTENTS

		Page
	ACKNOWLEDGEMENT	iii
	LIST OF FIGURES	viii
	LIST OF TABLES	ix
	LIST OF ACRONYMS	x
	ABSTRACT	xiii
Chapter		
1	INTRODUCTION	1
1.1	Motivation for this Study	1
1.2	Motivation for Choosing JOVIAL	2
1.3	Overview of JOVIAL History	4
1.4	Organization of Paper	6
	PART I THE EARLY PERIOD	8
2	THE ORIGINS OF JOVIAL	9
2.1	SAGE	9
2.2	Origins of CLIP	13
2.3	ALGOL	14
2.4	CLIP Compiler Development	15
3	JOVIAL/J2	20
3.1	J2 Language Development	20
3.2	Origin of the Name	22
3.3	J2 Compiler Development	23
3.4	Use of J2	26
4	INFLUENCES ON J2 LANGUAGE DEVELOPMENT	27
4.1	ALGOL	27
4.2	CLIP	28
4.3	SAGE	29
4.4	Compilers Programmed in JOVIAL	33
4.5	Non-Technical Influences	35
4.5.1	Concurrent Language Development	35
4.5.2	Lack of Committees	36
4.5.3	Language Experience	36
4.5.4	Language Specifications	37
4.5.5	Schedule Presures	38
5	JOVIAL/J3	39
5.1	J3 Language Development	39
5.2	J3 Compiler Development	41
5.2.1	The Q-7 J3 Compiler	43
5.2.2	The Philco 2000 J3 Compiler	44
5.2.3	The CDC 1604 J3 Compiler	46

Chapter		Page
6	JOVIAL TIMESHARING COMPILERS	49
6.1	The FSQ-32 Compiler	49
6.2	ARPA TSS	52
6.3	The JTS and TINT JOVIAL Compilers	52
6.4	ADEPT-50	54
7	JOVIAL SUBSET COMPILERS	58
7.1	JOVIAL/JX.2	58
7.2	JOVIAL/JS	59
7.3	Basic JOVIAL	60
7.4	SDC's JOVIAL Standardization Efforts	61
8	JOVIAL/J4 and JOVIAL/J5	64
8.1	J4 Language and Compiler Development	64
8.2	J5 Language and Compiler Development	66
9	JOVIAL/J6	68
9.1	J6 Language Development	68
9.2	J6 Language Features	70
9.3	J6/SPL Compilers	75
10	JOVIAL COMPILERS AT IBM	77
10.1	Use of 7090 J2 Compiler	77
10.2	The IBM 9020 Compiler	79
10.3	The IBM 360 Compiler for AWACS	80
11	JOVIAL/J3 COMPILERS AT CSC	81
11.1	CSC JX.2 Compiler	81
11.2	GENESIS/SYMPL Compilers	81
11.3	Other J3 Compilers	83
11.4	J3 JOCIT Compiler	84
12	JOVIAL COMPILERS AT ABACUS	87
12.1	J3 Compilers for Hughes	87
12.2	JOVIAL/J70	88
	PART II THE LATER PERIOD	90
13	STANDARDIZATION OF J3	91
13.1	Air Force JOVIAL/J3 Standardization	91
13.2	USASI Standarization of JOVIAL	92
14	JOVIAL/J73	94
14.1	The Committee to Modernize JOVIAL	94
14.2	J73 Language Design	95

Chapter		Page
15	JOVIAL/J3B	99
15.1	J3B-0	99
15.2	J3B-1	102
15.3	J3B-2	104
15.4	J3B Compiler Design	106
15.5	IBM J3B-0 compiler and the SPS Program	107
15.6	IBM J3B-2 Compiler for B52 OAS	110
16	JOVIAL J73/I	111
16.1	The Subsetting of J73	111
16.2	The DAIS Program	115
16.3	MIL-STD-1589	117
16.4	Other CSC J73/I Compilers	118
16.5	SEA J73/I Compilers	119
17	STANDARDIZATION OF J73	122
17.1	DOD Standardization Efforts	122
17.2	Air Force Standardization Efforts	124
17.3	JOVIAL Control Structure	127
18	J73A LANGUAGE DEVELOPMENT	130
18.1	The J73/I Upgrade	130
19	J73A COMPILER DEVELOPMENT	137
19.1	The JOCIT Compiler	137
19.2	Other SofTech J73A Compilers	140
19.3	SEA J73A Compilers	142
20	JOVIAL/J73B	144
20.1	MIL-STD-1589B	144
20.2	J73B Compiler Development	147
20.3	SEA Compiler Design	148
20.4	Use of J73B Compilers	149
	PART III CONCLUSIONS	151
21	TECHNICAL INFLUENCES	152
21.1	Technical Basis	152
21.2	Problem Domain	155
21.3	Hardware Constraints	160
21.4	Support for Programming Methodology	162
21.5	Other Technical Influences	163

Chapter		Page
22	NON-TECHNICAL INFLUENCES	164
22.1	Standardization	165
22.2	Language Specifications	166
22.3	Experience of Designer	169
22.4	Schedule Pressures	170
22.5	Lack of Committees	172
23	SUCCESS OF THE LANGUAGE	174
23.1	Technical Influences on Success	175
23.1.1	Compiler Performance	175
23.1.2	Lack of I/O	177
23.1.3	Compiler Availability	178
23.2	Non-Technical Influences	179
23.2.1	Vendor Support	179
23.2.2	Standardization	180
23.2.3	JOVIAL Culture	181
24	INFLUENCES OF JOVIAL ON LANGUAGE DESIGN	183
24.1	Contributions to Other Languages	183
24.2	Contributions to Language Design	184
24.3	Contributions to Compiler Design	187
BIBLIOGRAPHY		189

Appendix		Page
A	JOVIAL TIME LINE	220
B	COMPILER TERMINOLOGY	229
C	TABLE OF JOVIAL J2, J3, and J4 FEATURES	235
D	TABLE OF JOVIAL J73 (MIL-STD-1589B) FEATURES	254

LIST OF FIGURES

Figure		Page
1	Overview of JOVIAL History	5
2	The Origins of JOVIAL	10
3	JOVIAL J2 and J3 (1959 -- 1962)	18
4	JOVIAL Subset and Timesharing Compilers (1962 -- 1965)	50
5	JOVIAL J4 and J5 (1965 -- 1968	55
6	JOVIAL J6 and CSC Compilers (1968 -- 1971)	69
7	JOVIAL J73 and J3B (1971 -- 1974)	96
8	J3B and J73/I (1974 -- 1977)	103
9	JOVIAL J73/I AND J73A (1977 -- 1980)	125
10	JOVIAL J73B (1980 -- 1983)	139

LIST OF TABLES

Table		Page
1	Sample J6/SPL Program Called INTERGUIDE	74
2	J73 Features Excluded from All Three Subsets	113
3	J73 Features Excluded from Specific Subsets	114

LIST OF ACRONYMS

ADC	Air Defense Command
ADEPT	Advanced Development Prototype
AED	ALGOL Extended for Design
AF	Air Force
AFB	Air Force Base
AFLC	Air Force Logistics Command
AFM	Air Force Manual
AFR	Air Force Regulation
AFSC	Air Force Systems Command
AFWAL	Air Force Wright Aeronautical Laboratories
AFWL	Air Force Weapons Laboratory
ANSI	American National Standards Institute
ARPA	Advanced Research Projects Agency
ASD	AFSC's Aeronautical Systems Division
ASP	Advanced Signal Processor
ATC	Air Traffic Control
AWACS	Airborne Warning and Control System
BAL	Basic Assembler Language for the IBM 360/370
BMEWS	Ballistic Missile Early Warning System
BMO	Ballistic Missile Office
BNF	Backus Naur Form
BUIC	Backup Interceptor Control
CACM	Communications of the Association for Computing Machinery
CAPS	Collins Adaptive Processing System
CDC	Control Data Corporation
CHOL	Communications High Order Language
CIL	IBM's Common Intermediate Language
CLIP	Compiler Language for Information Processing
COC	Combat Operations Center
COMPOOL	Communications Pool
CPIC	Computer Program Integration Contractor
CPSS	Common Programming Support System
CPU	Central Processing Unit
CRL	Command Research Laboratory
CRT	Cathode Ray Tube (video monitor)
CSC	Computer Sciences Corporation
CSDP	Communications Software Development Package
CUC	Computer Usage Corporation
CUSS	Compiler and Utility System for SACCS
DAIS	Digital Avionics Information System
DASA	Defense Atomic Support Agency
DCA	Defense Communications Agency
DCA	Designated Control Agent
DEC	Digital Equipment Corporation
DEW	Distant Early Warning

DIS	Digital Integrating Subsystem
DMSP	Defense Meteorological Satellite Program
DOD	Department Of Defense
DODD	Department Of Defense Directive
DODDAC	Department of Defense Damage Assessment Center
DODI	Department Of Defense Instruction
DNCCC	Defense National Communications Control Center
ECSPO	Embedded Computer Standardization Program Office
ECS	Embedded Computer Systems
ESD	AFSC's Electronic Systems Division
FAA	Federal Aviation Agency
FOCCLANT	Fleet Operations Control Center, Atlantic Fleet
FOCCPAC	Fleet Operations Control Center, Pacific Fleet
FSD	Federal Systems Division, a division of IBM
FTSC	Fault Tolerant Spaceborne Computer
GE	General Electric Corporation
GPS	Global Positioning System
GPDS	General Purpose Display System
GRC	General Research Corporation
HBC	Westinghouse Hot Bench Computer (AN/AYK-15)
HIS	Honeywell Information Systems
HOL	High Order Language
HQ	Headquarters
HSA	Hollands Signal Apparten
IAL	International Algebraic Language, later called ALGOL
IBM	International Business Machines
IEC	International Electric Corporation, division of ITT
IL	Intermediate Language
I/O	Input/Output
IPSS	Interactive Programming Support System
ISA	Instruction Set Architecture
ITT	International Telephone and Telegraph corporation
IUS	Inertial Upper Stage
JAVS	JOVIAL Automatic Verification System
JCVS	JOVIAL Compiler Validation System
JID	JOVIAL Interactive Debugger
JOCIT	JOVIAL (J3 and J73) Compiler Implementation Tool
JOVIAL	Jules' Own Version of the International Algebraic Language
JTRANS	JOVIAL J73/I to JOVIAL J73 Translator
JTS	JOVIAL Time-Sharing compiler
JUG	JOVIAL-Ada Users Group
KB	Kilo Bytes: 1000 bytes
LALR	Lookahead Left-Right, compiler parsing method
LCA	JOVIAL Language Control Agent
LCB	JOVIAL Language Control Board
LCF	JOVIAL Langauge Control Facility
LUCID	Language Used to Communicate Information-system Design
MGD	Midcourse Guidance Correction
MIT	Massachusetts Institute of Technology
MOL	Manned Orbiting Laboratory
MRASM	Medium Range Air to Surface Missile
MX	Missile X

NAFEC	National Aviation Facilities Experimental Center
NAS	National Airspace System
NAVSTAR	Navagation satellite used to implement GPS
NAVCOSSACT	NAVal COmmand System Support ACTivity
NAVSPASUR	NAVal SPAce SURveillance system
NELIAC	Naval Electronics Laboratory version of IAL
NMCSSC	National Military Command System Support Center
NORAD	North American Air Defense Command
NTDS	Naval Tactical Defense System
OAS	Offensive Avionics System for B-52
OFS	Offensive Flight Software
PACT	Project for Automatic Coding Techniques
PDMS	Program Development and Maintenance System
PPS	Program Production System
PSL	Program Support Library
PSS	Proprietary Software Systems
RADC	Rome Air Development Center, an ESD laboratory
RFP	Request For Proposal
RFS/ECMS	B-1 Radio Frequecy/Electronics Countermeasures Subsystem
SAC	Strategic Air Command
SACCS	Strategic Air Command Control System
SAGE	Semi-Automatic Ground Environment
SAMSO	Space And Missile Systems Organization old name for SD
SCC	SAGE Super Combat Center
SCF	Satellite Control Facility
SD	AFSC's Space Division
SDC	System Development Corporation
SDD	System Development Division
SDVS	Software Design and Verification System
SEA	Software Engineering Associates
SEMANOL	Semantics Oriented Language
SPARS	Space Precision Attitude Reference System
SPL	Space Programming Language
SPLIT	SPL Implementation Tool
SPO	System Program Office
SPS	Simplified Processing Station
SSD	Space Systems Division, later changed to SAMSO
SSRL	Systems Simulation Research Laboratory
STC	Satellite Test Center
STP	System Training Program
TDMS	Timeshared Data Management System
TINT	JOVIAL Teletype INTerpreter
TIPI	Tactical Intelligence Processing and Interpretation
TRW	Thompson Ramo Woolridge Corporation
TSS	TimeSharing System
UNCOL	Universal Computer Oriented Language
USASI	United States of America Standards Institute
WCC	Weapons Control Computer
WWMCCS	World Wide Military Command and Control System

ABSTRACT

HISTORY OF JOVIAL

A STUDY OF PROGRAMMING LANGUAGE CHANGE

by

Michael Olin Hogan

Master of Science in Computer Science

This paper describes research on the factors that influenced the development and subsequent change of the JOVIAL (Jules' Own Version of the Internationl Algebraic Language) programming language during the period from the creation of the first version of JOVIAL in 1959 to the production of compilers for the most recent version, JOVIAL/J73 (MIL-STD-1589B), in 1982. During this period numerous versions of JOVIAL were developed. This paper presents the history and analyzes the technical and non-technical influences resulting in the creation and change of each version. The influences of compiler development, application specific needs, and governmental/polical policy on the evolution of the language are emphasized. The factors that cause programming language change are summarized. It is concluded that non-technical influences are much more important than previously thought.

Chapter 1

INTRODUCTION

1.1 Motivation for This Study.

Natural languages have existed for thousands of years. By comparison, the development of programming languages is very recent, beginning about 1954 [METR80 p124]. The study of natural language evolution has been a primary concern of linguistics for many years. Interest in the development and change of programming languages is accordingly recent, beginning most notably with the International Research Conference on the History of Computing, in 1976 [METR80] and the ACM History of Programming Languages Conference, in 1978 [WEXE81]. For the most part, these conferences have focused on the early history of programming languages. Few, if any, in-depth studies have been made of a single language and over a sufficient period of time to provide an understanding of how programming languages evolve in the way that linguists have studied natural languages.

This paper presents a comprehensive study of the JOVIAL programming language, a language designed in 1958 for programming large military command-and-control systems and still in use today. The purpose of this study is to discover what factors influenced JOVIAL's development and subsequent evolution and, perhaps, to discover some more general principles affecting the development of computer programming languages.

The reference to programming languages is, appropriately, limited to so-called high order languages (HOLs) like JOVIAL. HOLs are more or less independent of the underlying computer hardware they run on. The same language can run on many different computers and remain unchanged over many years. HOLs are contrasted with low-level machine and assembly languages, which are closely tied to the architecture of a particular computer; thus, they do not evolve and change so much as die when the computer becomes obsolete and are of little interest for studying language change.

1.2 Motivation for Choosing JOVIAL.

The factors that influence the development and evolution of programming languages can be divided (although not so neatly) into technical and non-technical influences. Technical influences include things like previous languages the design is based on and the applications the language is designed to support. The non-technical influences are more diverse and less well understood. JOVIAL was chosen for the study into these influences, especially the non-technical influences, for several reasons.

First, there have probably been more versions of JOVIAL than any other high order language (see the overview of JOVIAL in the next section). In fact, it was this multitude of versions that first interested me in doing such a study. This study has shown that most of these versions have been the result of non-technical influences, which were found to be more important than previously realized.

Second, JOVIAL is production language that has been in use long enough to provide sufficient material for study of language evolution. Unlike some academic languages, such as Alphard, for which one or sometimes no compiler was ever built, there have been over 87 JOVIAL compilers used by hundreds of people since its creation in 1958. Only ALGOL and FORTRAN have been in use longer.

Third, JOVIAL has many unique language features, such as the TABLE and the COMPOOL, which have been ignored in the literature, which had concentrated on the more popular languages such as FORTRAN, ALGOL, COBOL, APL, LISP and SNOBOL [WASS80]. Many modern compiler-building techniques, such as implementing a compiler in its own language, came about during the development of JOVIAL. Thus, there is a need to study the contributions of JOVIAL to the field of language design.

Fourth, JOVIAL is one of the few major programming languages (CMS-2 and Ada are two others) to be controlled solely by a military agency. JOVIAL never achieved national standardization by the American National Standards Institute (ANSI), and the Air Force eventually became responsible for control of the language. A study of JOVIAL should yield insight into the effects of control by a military agency on a programming language.

Finally, despite initial enthusiasm and backing by the System Development Corporation (SDC) and the Air Force, JOVIAL failed to gain wide-spread use or acceptance. A study of JOVIAL should provide some understanding into what factors affect the success of a language. Such information would be of benefit in assuring the success of Ada, the Department of Defense's (DOD) new common programming language.

1.3 Overview of JOVIAL History.

The history of JOVIAL falls into two general periods: an early period and a late period (figure 1). The early period extends from the creation of JOVIAL in 1959 by Jules Schwartz at the System Development Corporation (SDC) to 1973. This period is marked by SDC's influence over the language and by the fact that all of the early versions of JOVIAL (J0, J1, J2, J3, JS, JX.2, J4, J5, J6 and J70) are related to J3. J3 was the de facto standard version of JOVIAL. It was used by SDC as a corporate programming language for many Air Force programs and was later adopted by the Air Force as a standard language [AFM67].

The later period began in 1973 when an extensively revised version of J3, called J73, was produced. This period is marked by a high level of Air Force influence over the language. All of the later versions of JOVIAL produced after 1973 (J73, J3B, J73/I, J73A and J73B) are related to J73, although J73 was never implemented. In 1977, a subset of J73, called J73/I, was chosen as the standard language (MIL-STD-1589) for programming all Air Force weapons

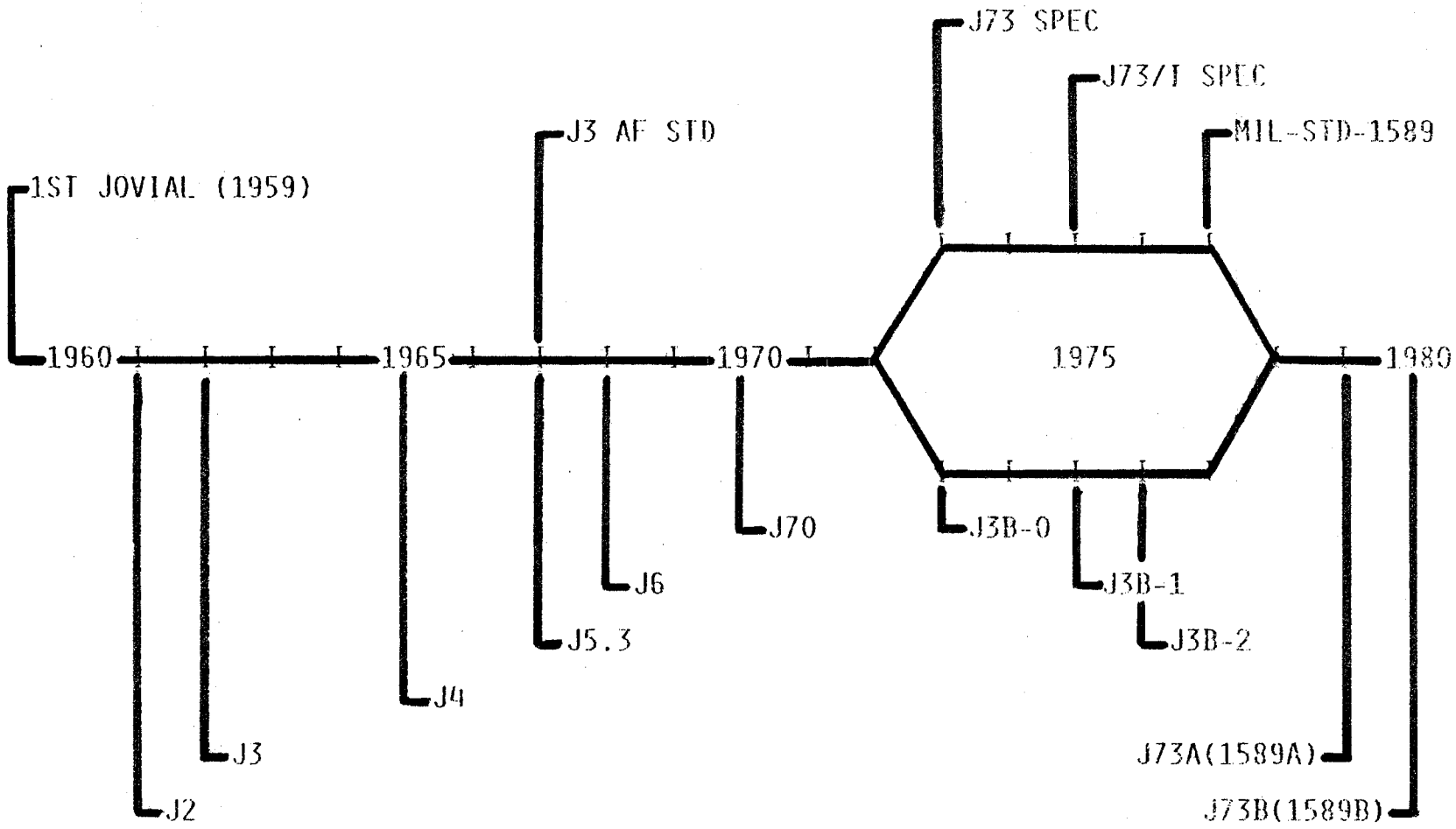


Figure 1. Overview of JOVIAL History

systems. In 1979, J3B and J73/I were merged to form J73A (MIL-STD-1589A), which was soon revised to J73B (MIL-STD-1589B). J73B is currently the Air Force standard language for programming embedded military computers (e.g. a flight control computer in an airplane or missile) until Ada is ready.

1.4 Organization of Paper.

This paper is divided into three parts. Part I covers the early period of JOVIAL. It describes those versions of JOVIAL that were based on J3 and that were developed during the period from approximately 1958 to 1973. Part II covers those versions developed from 1973 to the present. In general, each version of JOVIAL is described in a separate chapter. The influence of SDC is described in several of the chapters. In addition, separate chapters are devoted to other companies that had an important influence on JOVIAL.

Part III contains the conclusions, which are presented in four chapters. Chapters 21 and 22 summarize the technical and non-technical influences on JOVIAL. Chapter 23 examines some of the reasons for JOVIAL's lack of success, and chapter 24 shows how JOVIAL has influenced other languages.

The literature of JOVIAL is filled with acronyms. These are spelled out before each use, but are also summarized in a list following the table of contents. A bibliography is included. It contains the references made in the text as well as every document on JOVIAL that I could find. Many of the references are annotated.

Appendix A contains most of the figures in the text collected together to form a time line of JOVIAL events to help the reader see the relationship of the many dates given throughout the paper.

Appendix B is an explanation of the various terms used in describing high order languages and their compilers and should be read by those not familiar with compiler terminology.

Appendix C is a summary of JOVIAL/J3, condensed from [AFM67], [PERS66B], and [SHAW61B], to provide the reader with an understanding of specific J3 language features.

Appendix D is a similar overview of J73B summarized from [SOFT82]. J3 and J73 were summarized because they are the principal versions of the language. J3 was at one time regarded as the standard version of JOVIAL. Also J2, J3, J4 and J5 are so similar that a separate description of each would be redundant. J73B (MIL-STD-1589B) was chosen because it is the current language standard and because J73/I and J73A are very similar to J73B.

PART I

THE EARLY PERIOD

Chapter 2

THE ORIGIN OF JOVIAL

Primarily four events brought about the creation of the first version of JOVIAL: the SAGE program, Project CLIP, the publication of ALGOL, and the SACCS program (figure 2). This chapter describes how the first three events led to the decision to create JOVIAL. The next chapter discusses the influence of the SACCS program.

2.1 SAGE.

The influence of SAGE on JOVIAL cannot be underestimated. The company that developed JOVIAL, System Development Corporation (SDC), the people who were instrumental in developing JOVIAL, such as Jules Schwartz and Chris Shaw, and the technical constructs that went into JOVIAL, such as the TABLE and the COMPOOL (see chapter 4), were all products of the SAGE system.

The idea for SAGE began in 1950 when a committee, headed by MIT professor Dr. George E. Valley, recommended building an air defense system based on the new Whirlwind computer, developed by Jay W. Forrester for the Office of Naval Research [BAUM81]. But Whirlwind's 110 computations per second fell short of the 36,000 computations per second the proposed system would need. Further refinement of the concepts for a computer-based air defense system was required. In 1951, the Lincoln Laboratory was established at

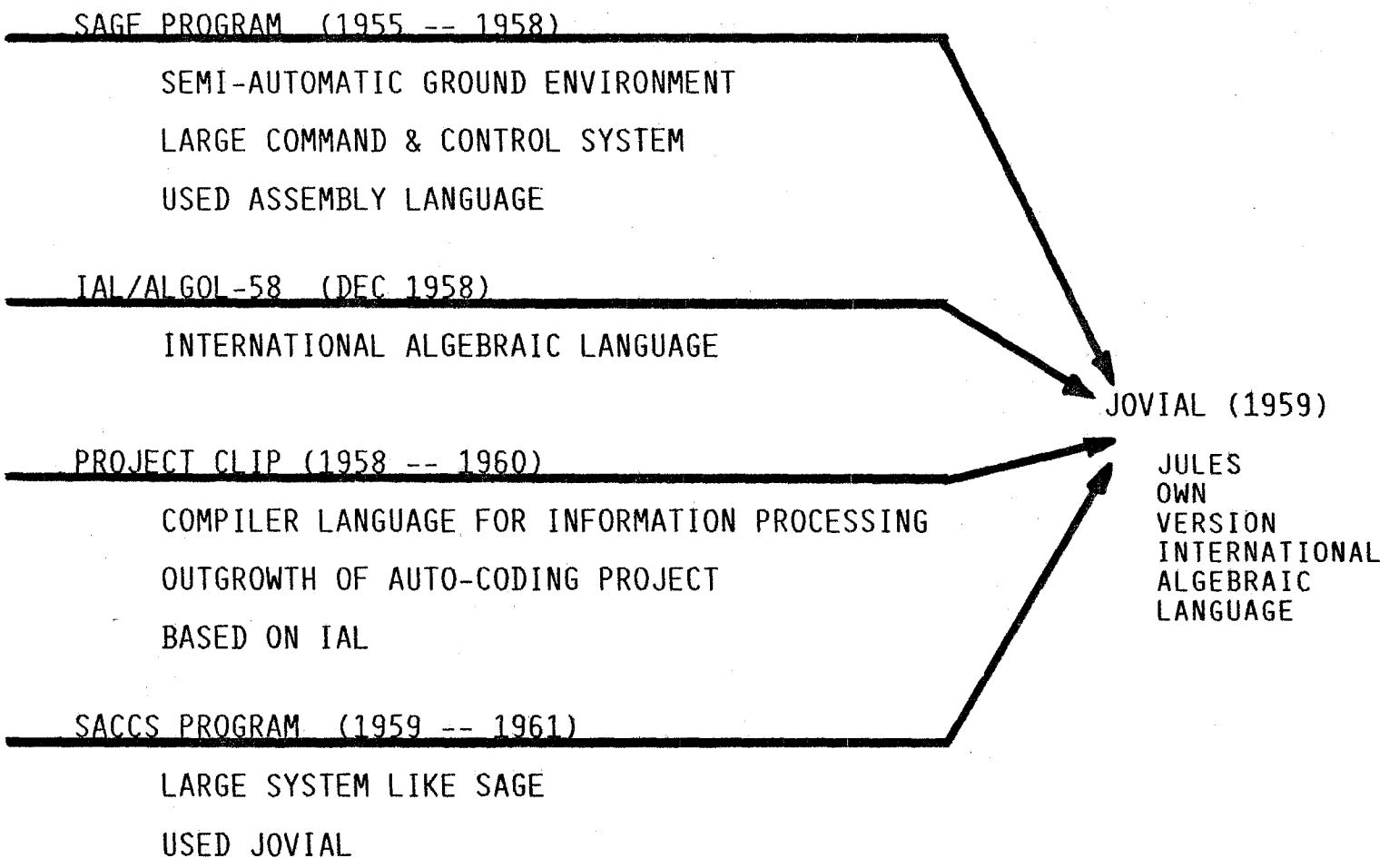


Figure 2. The Origins of JOVIAL

MIT in Lexington, MA, to research the air defense problem.

Lincoln Laboratory soon developed a prototype system, which used the XD-1 computer, the successor to Whirlwind, to process radar data from the Cape Cod, MA, radar site. The Cape Cod System proved the feasibility of a computer-based air defense system and, in 1954, the Air Defense Command awarded Western Electric a contract to build a full-scale version called the Semi-Automatic Ground Environment (SAGE) system.

SAGE was based on the 32-bit IBM AN/FSQ-7 (Q-7) and AN/FSQ-8 computers. The Q-7 was a production version of the XD-1. Under SAGE, the country was divided into divisions, each containing three to five sectors. Each sector collected radar data on incoming enemy aircraft and sent them to a direction center, where two Q-7s (one acting as a backup) summarized the data for the sector. This data was then transmitted to the combat center at division headquarters, where the Q-8 integrated the sector air data into the division air situation, which was then used by the North American Air Defense Command (NORAD) in Colorado Springs to monitor and direct the nation's air defense.

The biggest challenge of SAGE was developing the computer programs for the Q-7 and Q-8. Although chartered as a research center, Lincoln Laboratory began work on SAGE with the provision that another organization would take over final program development, installation, and maintenance. The Rand Corporation was chosen for this task.

Created by the Air Force in 1948 as a non-profit think tank to do defense research and development, Rand was experienced in both air-defense radar problems and computer-based systems. This experience derived from a contract to develop and install the System Training Program (STP) at the Air Defense Command's radar sites.

STP involved the use of simulated air defense situations to train radar crews. During preliminary development of STP, Rand created simulations of radar tracks of aircraft manually using punched cards. With the planned installation at 150 radar sites in the United States, this method was inadequate. So, Rand developed a computer-based system using the new IBM 701, which Rand had obtained by top government priority. In August 1954, the first STP was successfully installed at Boron, CA, and Rand began installation at the remaining sites [BAUM81].

It was at this time that the Air Force was looking for a company to take over development of the computer programs for SAGE. Rand was the obvious choice as a non-profit company with experience in both the air defense system and the new field of computer-based systems. Also, Rand employed over ten percent of the country's top programmers at the time. In July 1955, the first Rand programmers began work on SAGE in a joint effort with Lincoln Laboratory [BAUM81].

By December 1955, the work on SAGE and STP began to overwhelm the rest of Rand. SAGE and STP were split off into a separate division, the System Development Division, which became the System Development Corporation a year later. SDC had over a

thousand people, more than the rest of Rand put together. SDC began formal operations in December 1957 when SAGE and STP were officially transferred from Rand [BAUM81].

On 27 June 1958, the first SAGE system was delivered to McGuire AFB, NJ, followed rapidly by installation at other sectors until 1961, when all SAGE sites had been installed. SAGE became the model for all the military command-control systems to follow. It was the first large-scale, real-time system with almost 1000 people employed during its development and installation. SAGE had such state-of-the-art capabilities as on-line database retrieval, multiprogramming, timesharing, interactive graphic displays, and distributed processing.

Yet, despite all these firsts, SAGE was still programmed in assembly language. Development of large systems like SAGE clearly established the need for new high order languages to speed program development. It was to meet this need that JOVIAL was developed.

2.2 Origins of CLIP.

A significant source of technical ideas for the design of the JOVIAL language and compilers came from an earlier language called CLIP -- Compiler Language for Information Processing. CLIP slightly preceeded and continued in parallel with the development of JOVIAL.

The development of CLIP (and consequently JOVIAL) got started with an an article on Expression Analysis in the March 1958 Communications of the ACM [WOLP58]. This article was the first

exposure that those who would later develop JOVIAL would have to the subject of parsing mathematical and logical expressions [SCHW78]. Using the techniques in the article, some of the people who had just finished work on SAGE began experimenting with compiler-building techniques. This study began in June 1958 and was called the Auto-Coding Research Project [SHAW60C]. The members of this study were Erwin Book, Harvey Bratman, and Jules Schwartz. The development of a language was not planned at the time; simply being able to understand and parse complex expressions provided sufficient interest to motivate the effort [SCHW78].

2.3 ALGOL.

Not long after the Auto-Coding project began, an article appeared in the December 1958 Communications of the ACM [PERL58] describing the International Algebraic Language (IAL), which was later renamed ALGOL-58 (ALGOL-58 soon evolved into into ALGOL-60 [NAUR60], [NAUR63]). By this time, the Auto-Coding effort had shifted emphasis to defining a language that would be useful for writing compilers. This was the first actual contribution to the language effort that eventually led to JOVIAL.

The similarities between ALGOL and their work resulted in the group adopting much of the ALGOL language and notation for their language. ALGOL contained several features that made it useful for compiler writing. Furthermore, it seemed destined to have widespread use [BRAT59]. It was also realized that compiler writing was similar to other information processing applications [BANH59];

hence, they named their language CLIP: Compiler Language for Information Processing.

Just as the work on CLIP was starting, Jules Schwartz was transferred to SDC's New Jersey Division to work on the SACCS program where he headed the project to develop JOVIAL. The technical work Schwartz did in connection with the Auto-Coding project had an important influence on JOVIAL as discussed in chapters 3 and 4.

2.4 CLIP Compiler Development.

While Schwartz was working on JOVIAL in New Jersey, the development of the CLIP language and compiler was continuing at SDC's main offices in Santa Monica. CLIP was developed as a tool for research on the production of a language and compiler for information processing [BOOK60]. The work on CLIP influenced both the JOVIAL language and its compilers. The influence of CLIP on the JOVIAL language is discussed in chapter 4. The influence of the CLIP compiler is discussed here.

Several techniques used for the CLIP compiler were later incorporated into the early JOVIAL compilers. The most important technique was writing the CLIP compiler in CLIP. Although quite common today, this approach was quite new at the time and was later used for nearly all JOVIAL compilers as well as for many other languages.

As mentioned before, CLIP was based on ALGOL. By using CLIP to write its own compiler the designers were able to discover just what features needed to be added to ALGOL in order to program such information processing applications [BANH59]. Also by implementing the compiler in its own language, rehosting the compiler onto a different computer was simplified, since the compiler could compile itself.

Another technique borrowed from the CLIP compiler was dividing the compiler into a separate front end and back end (see appendix B). The front end translated CLIP into a machine-independent intermediate language (IL), which was further translated into machine language by the back end (code generator). This design originated with the UNCOL (UNiversal Computer Oriented Language) project [STEE58] and was used for many JOVIAL compilers. This approach further reduced the work in rehosting a compiler because only a new code generator had to be written for each new computer (appendix B).

Two other techniques developed for CLIP that were later used on the JOVIAL compilers were the ANCHOR algorithm for the analysis of algebraic and logical expressions [MANE59] and the use of hashing for performing symbol table look-up.

When CLIP began, the decision was made to first develop a pilot model of a compiler with a somewhat limited scope--CLIP-1. Experience gained in the production of CLIP-1 would later be applied to the design of the more complex CLIP-2 [BANH59].

The initial specifications [BAHN59] for the CLIP-1 language were finished in May 1959 (figure 3), and work began on the compiler front end, which was written in CLIP. Work on the front end was begun as soon as possible to discover what changes to the language would be necessary in order to express the compiler design. During the summer, changes had to be made in the TYPE and TABLE declarations to simplify parsing them, and the STRING declaration was added to simplify the use of string variables. By November 1959, the CLIP-1 language was completely specified, and the entire front end of the compiler had been coded in CLIP [BRAT81].

The front end was then hand compiled into 709 assembly code (called SCAT), which took four months to code and check out [BOOK60]. In April 1960, the hand-coded front end was combined with the code generator, which was implemented in SCAT, to produce an executable version of the CLIP compiler. This version was then used to compile the CLIP version of the front end. This self-compiled front end was then combined with the manually-coded code generator to produce the finished compiler. The compiler translated CLIP source code into SCAT, which was then input to the SDC SMASHT 709 assembler to produce the executable program [BOOK60].

But the CLIP-compiled front end ran three times slower than the hand-coded version [BOOK60]! By comparing the compiler generated code with the hand-written code, areas for improving the quality of the compiler were discovered and a modified front end was running about 60% faster by July. By August, the code generator was rewritten in CLIP, and the entire compiler was self-compiled.

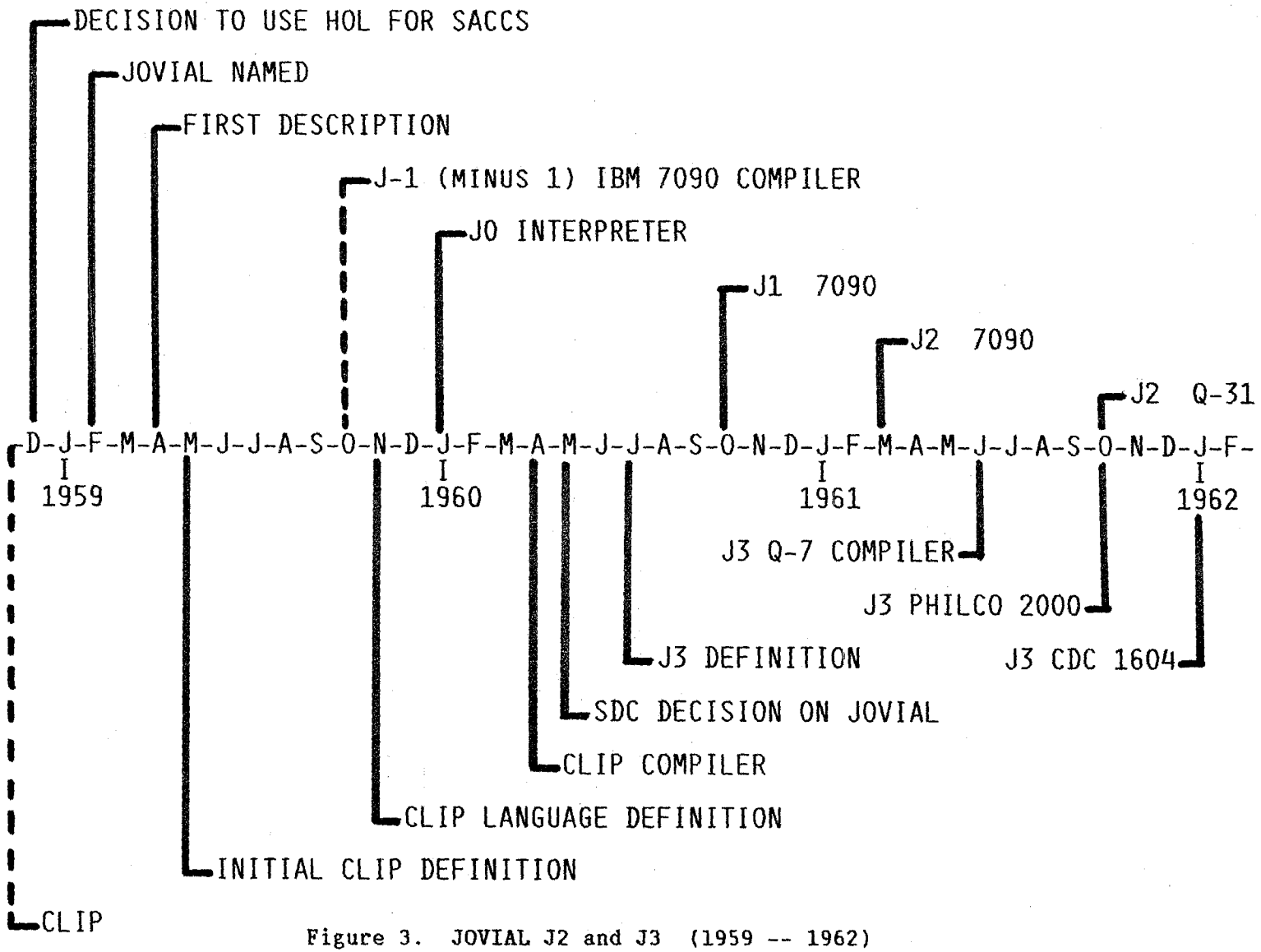


Figure 3. JOVIAL J2 and J3 (1959 -- 1962)

By this time, SDC had decided to adopt JOVIAL as corporate programming language (see chapter 5). CLIP was a useful vehicle for testing compiler writing techniques, but it was not as capable as JOVIAL for solving the kinds of problems SDC was interested in. During the last quarter of 1960, various optimizations and improvements were made to the CLIP-1 compiler. But work on a second CLIP compiler never began.

At its peak Project CLIP included Bob Bozak, Erwin Book, Harvey Bratman, Ellen Clark, Don Englund, Harold Isbitz, Howard Manelowitz, and Ellis Myers, most of whom later worked on JOVIAL compilers [SHAW60C]. The last use of CLIP was to write the first versions of the JOVIAL/J3 compilers in November 1960 (see chapter 5). But after the JOVIAL compilers were bootstrapped, there was no need for CLIP as a compiler writing language and CLIP eventually faded away.

An important result of the work with CLIP was the confirmation that HOLs were easier to use and cost effective. For examle, the CLIP version of the front end only required three weeks to check out as compared to six months for the assembly language version. Later, when the CLIP compiler was ready, the programmers chose to make modifications in CLIP rather than in the hand-coded version [BOOK60] because the changes could be done quicker and with less chance of error. This was a good proof of the effectiveness of HOLs, since these were professional programmers to whom machine language coding was not difficult.

Chapter 3

JOVIAL/J2

3.1 J2 Language Development.

JOVIAL actually got its start with the development of the SACCS program. In March 1958, the Strategic Air Command (SAC) approached SDC: General LeMay had seen SAGE and wanted a similar system to aid SAC commanders [BAUM81]. A month later, SDC began a six-month study to examine SAC operations and to develop a preliminary design for the Strategic Air Command Control System (SACCS). In August 1958, SAC issued a request for proposal (RFP) for SACCS. International Electric Corporation (IEC), a division of ITT, won the award and selected SDC as the software subcontractor.

Like SAGE, SACCS (also called program 465L) was another large command and control system. It consisted of a worldwide communications network linked to computers to provide SAC commanders with the latest information on the status of weapons and aircraft at each SAC base, location of SAC bombers, enemy troop and plane movements, and global weather conditions.

Development of a real-time system of this magnitude represented a formidable challenge for SDC [BAUM81]. SACCS was to be developed with new computers and a new operating system. The operating system for the SACCS main computer, the IBM AN/FSQ-31 (Q-31), would exceed a million instructions, over four times the size of SAGE. Another 300,000 instructions would be needed for

support programs on the IBM 709, IBM 1401, and IBM AN/FSQ-5 computers [BAUM81]. Clearly, the use of assembly language for SACCS was impractical. A new approach was needed.

That approach was provided by Jules Schwartz. In December 1958, just prior to his transfer to New Jersey, Schwartz made a bold recommendation. Influenced by his experience with SAGE, his language research work with CLIP, and his reading of the IAL description, Schwartz recommended the use of an HOL to program SACCS.

Schwartz' qualifications to recommend this approach were high. After graduating from Columbia University in 1953 where he studied statistics, Schwartz joined Rand where he worked on such early computers as the JOHNNIAC and the IBM 701. In the fall of 1955, Schwartz was one of the first Rand programmers to go to Lincoln Laboratory on SAGE where he helped implement the SAGE COMPOOL feature (see appendix C and chapter 3)[BAUM81]. On his return to Santa Monica, he started the Auto-Coding Project.

Given his exposure to language development, his familiarity with the COMPOOL concept that would be a central feature of the new language, and his knowledge of the language requirements for a command-and-control system like SAGE, Schwartz was convinced that using an HOL was a better way to code such a system than using assembly language [BAUM81].

Although the early work on CLIP had assured the practicality of HOLs, this was a risky apporach. No major system had yet been written in an HOL. Further, this language would have to be developed concurrently with the design of the system; programming

would have to be delayed until the language and its compilers were developed. Nevertheless, after extensive consulations, SDC president M.O. Kappler gave his approval, much to Schwartz' surprise [SCHW78] and Schwartz began designing the language that soon became JOVIAL (next chapter).

3.2 Origin of the Name.

The final recommendation for JOVIAL was written by Schwartz in December 1958 in New Jersey. It contained a brief description of some of the language constructs that could be implemented for the SACCS program [SCHW78] and was titled OVIAL: Our Version of the International Algebraic Language because it was based on ALGOL (IAL) [SHAW60C].

The subject of an acceptable name for the language was brought up during a meeting in January 1959 attended by SACCS and IEC personnel to discuss a number of technical issues [SCHW78]. The name that was originally recommended, OVIAL, had a connotation of the birth process that seemed unacceptable in the 1950's, so a new name was needed.

According to Schwartz own account [SCHW78], someone suggested the name JOVIAL, which seemed like a straightforward transition from OVIAL. The question was the meaning of the "J". Since Schwartz was at the front of the room conducting the meeting, somebody suggested that it be called "Jules' Own Version of the International Algebraic Language". This suggestion brought laughter to the group and the meeting ended with no final decision on the

name. According to Schwartz, he felt this was a joke and did not become too concerned about it [SCHW78].

But while Schwartz was away on a business trip soon after, his programming staff, along with several SAC and IEC officials, decided to put into the subcontract, which SDC signed with IEC in February 1959, an obligation for SDC to develop a language called "JOVIAL (Jules' Own Version of the International Algebraic Language) and compilers for this language for the IBM 709 and AN/FSQ-31". Thus, along with the dates for various deliveries, the name JOVIAL was officially written into the contract [SCHW78].

The group that was formed to develop this language and compilers was called CUSS (Compiler and Utility System for SACCS). In April 1959, SDC established the SACCS Division near IEC in Lodi, NJ. The CUSS group numbered 9 out of some 200 staff members at SACCS. A year later when SACCS was moved from Lodi to Paramus, NJ, the SACCS staff had grown to 500 [BAUM81], with 15 working on CUSS [SCHW78].

3.3 J2 Compiler Development.

Work offically began on the compilers with the signing of the IEC subcontract in February 1959. The SACCS system was built around the 48-bit Q-31 computer. Two JOVIAL compilers were to be delivered to SAC Headquarters at Offutt AFB in Omaha: one for the Q-31 and one for the 709. The Q-31 compiler was intended for the programming of the SACCS System; the IBM 709 for a variety of other on-going SAC functions including initial work on the JOVIAL

compilers and the Q-31 Development System until the Q-31 was installed later in 1959. The JOVIAL language was to be the same for both computers, and the compilers were to be programmed in JOVIAL.

According to the contract, the IBM 709 compiler was to be delivered in six months and the Q-31 compiler some months later. This schedule was unrealistic and was never met. The first formal delivery of the JOVIAL 709 compiler was not made to SAC until March 1961 (figure 3), which missed the original schedule by nearly two years [SCHW78]. Nevertheless, the impact of trying to meet this unrealistic schedule had lasting effects on the design of the language (see below).

Prior to the first formal delivery, a variety of intermediate JOVIAL compilers was produced. The first JOVIAL compiler was for a subset version of JOVIAL called J-1 (J minus one). It was available for the compiler developers in the fall of 1959. This subset compiler was first written in machine language, but was quickly reprogrammed in J-1. From then on, all versions of the compiler were programmed in some version of JOVIAL.

Except for the compiler builders, the first users of JOVIAL used an interpreter, not a compiler. This interpreter was for a very simple subset of JOVIAL called J0 [OLSO60]. It consisted of a front end and an interpreter for the intermediate language. It was running on the 709 by January 1960. About 8 man years of effort were expended before the J0 and J-1 versions began to be used.

The J-1 compiler was used to program the next version of the language, J1. In October 1960, a J1 compiler for the Q-31 was

running on the 709. It produced Q-31 programs that were interpretively executed on the 709 as the Q-31 had not yet been delivered by IBM [SHAW62C]. By December, a J1 compiler for the 709 was available [SHAW63B]. The first users of the J1 compiler outside of the compiler builders were the builders of the SACCS utility and operating system.

J2 soon replaced J1, which was used to program the J2 compiler. J2 was the final version of the language delivered to SAC. Usage of JOVIAL at SAC began with the delivery of the J2 compiler in March 1961. This compiler ran on the IBM 7090, which had replaced the 709 by that time and required about 25 man-years of effort to complete.

The J2 compiler for the Q-31 was not delivered until October 1961 [SHAW62C] but was ready in time for development of SACCS. As mentioned earlier, the design of both JOVIAL compilers was patterned after the CLIP compiler. Both were written in their own language and both used a machine-independent intermediate language.

The first official document on JOVIAL was published in April 1959 [SCHW59A] and superseded in May 1959 [SCHW59B]. These documents described a number of the concepts and details of the language. The next two documents [BOCK59] and [SCHW59C] appeared in September.

3.4 Use of J2.

About 95 percent of SACCS was programmed in JOVIAL, but, despite its larger size and greater complexity, SACCS required less than one-half the programming labor of SAGE [BAUM81]. This was an important early demonstration of the efficiency of using an HOL. The Q-31 compiler continued to be used for SACCS until about 1965 [DOBK68] and was also rehosted to the Q-32 computer by SDC for use on research programs (chapter 6).

Although the Q-31 compiler was mostly confined to the SACCS program, the 7090 J2 compiler saw more widespread use because of the popularity of the IBM 7090 computer. SDC released the 7090 compiler through SHARE, the IBM computer users group, to many other organizations: the FAA's National Aviation Facilities Experimental Center (see chapter 10), TRW, Cal Tech, Rand, MITRE, UCLA's Western Data Processing Center [SHAW62B], American Airlines, Burroughs, Computer Techniques Group, NASA/Goddard, Teledyne, the US Naval Ordance Test Station at China Lake, the US Naval Information Center, and the Lawrence Radiation Lab [GRAN64].

In April 1964, the SDC's Washington Division received a contract from the Naval Weapons Laboratory at Dahlgren, VA, to develop the Naval Space Surveillance System, NAVSPASUR, one of the earliest satellite tracking systems. They used the J2 7090 compiler to develop a timeshared system for the IBM 7094 that processed real-time orbit data in the foreground and satellite calculations in the background [BAUM81].

Chapter 4

INFLUENCES ON J2 LANGUAGE DEVELOPMENT

The development of the JOVIAL language proceeded in parallel with development of the compilers and was influenced by many factors, both technical and non-technical. The main technical influences were ALGOL, CLIP, SAGE, and the use of the language to write the compilers. The significant non-technical influences were the development of the language concurrently with the development of the compilers, the design of the language by one person rather than a committee, no formal language baseline during development, the lack of language design training on the part of the JOVIAL team, and the schedule pressures of the SACCS program.

4.1 ALGOL.

The basic structure of JOVIAL/J2 was taken from ALGOL-58. Schwartz chose ALGOL because "at the time it appeared as if it would become the commonly accepted standard for languages, and it seemed like a better technical basis for a new language than FORTRAN, which was the only other possibility" [SCHW78]. Although ALGOL was the starting point, no attempts were made to incorporate concepts from later versions of ALGOL, i.e. ALGOL-60.

The principal features taken from ALGOL were the compound statement (BEGIN/END), FOR loop statement, alternative statement (IFEITH/ORIF), RETURN statement, STOP statement, SWITCH statement,

multi-dimensional arrays and the logical, relational and arithmetic operators [PERL58]. Whereas ALGOL only provided floating-point and integer types, JOVIAL also had character (Hollerith and Transmission Code), Boolean, STATUS, and fixed point types (see appendix C).

The ALGOL switch statement was a numeric switch that provided for a series of branches based on the value of an integer item. In JOVIAL, this was not sufficient for such variables as STATUS items, which could take on symbolic values. Thus an ITEM switch was added which could cause conditional branching based on the value of not only STATUS items but also character strings (appendix C, section 4.10).

The limitation in ALGOL that arrays and procedures passed as subroutine parameters could not themselves contain actual parameters or subscripts (i.e. only the name could be passed) was eliminated. The ALGOL DO statement was also eliminated. It allowed sequences of statements already written down to be copied elsewhere in the program without rewriting. A macro-like substitution of identifiers in the reused statements was also permitted. The JOVIAL DEFINE capability provided a similiar power and was probably derived from the DO statement [BOOK60].

4.2 CLIP.

As mentioned earlier, CLIP was an important influence on JOVIAL. Many of the features that were added to ALGOL by the CLIP group found their way into JOVIAL. These features included Boolean, character, and signed integer types. In CLIP, the overlapping of

strings and items was permitted via the ORIGIN declaration [ENGL61]. A similiar capability was provided in JOVIAL via the OVERLAY declaration. Procedures could be parameterless, and parameters could be passed by reference as well as by value. I/O procedures such as Read, Write, Write end-of-file, and Rewind were provided [BANH59]. These were later added to J3.

One of the most obvious characteristics of JOVIAL (J2, J3 and J4) that came from CLIP was the use of the dollar sign "$" for the ALGOL statement terminator ";". However, JOVIAL also used ($ and $) for array indexes, unlike CLIP which used (/ and /) [BANH59]. Schwartz later noted that this choice for liberally used array brackets was unfortunate as it was very distracting [SCHW78].

Significant ALGOL features that were omitted from CLIP but not JOVIAL were the logical equivalence operator, exponentiation, floating-point variables, functions, and multi-dimensioned arrays.

The experience with CLIP also pointed out the lack of several important features which were added to JOVIAL. These included the ability to reference symbolically the number of entries in a table and the number of characters in a string, STATUS items, initial values for unsubscripted variables, nested FOR statements, and a COMPOOL capability [BRAT61A].

4.3 SAGE.

JOVIAL was designed for programming the SACCS system. Given that the language would be based on ALGOL, the central design question was what additions to ALGOL were needed to solve the

programming problems posed by SAGE, since it was assumed that SACCS would be similar to SAGE.

Systems like SAGE and SACCS were characterized by their large size, large number of programmers, need for reliability, and long maintenance phase. Other characteristics of these early systems were the need to store large amounts of data efficiently and to access machine registers directly. The JOVIAL designers attempted to achieve these goals by providing language features for modular program decomposition, data sharing among modules, efficient data storage, machine access at the HOL level, and simple input-output schemes.

Modular program decomposition and data sharing was achieved by addition of the COMPOOL (COMmunications POOL) feature to ALGOL. Although the COMPOOL is probably the most characteristic feature of JOVIAL, the concept was, in fact, borrowed from SAGE and was a design criterion for JOVIAL [SCHW78].

The COMPOOL provides a central data area that can be shared by separate modules (appendix C section 7.2 and appendix D section 7.3). The COMPOOL contains data name declarations that can be used to compile separate program modules to avoid compatibility and interfacing problems with accessing this central data. The compiler assures that separate programs using the same COMPOOL will use the same description of this data. The COMPOOL is particularly valuable for large command and control systems as it allows separate programs developed by separate programmers to communicate effectively [SAMM69].

JOVIAL's data types and data structuring capability were very much influenced by experience with SAGE and represented the most important departure from ALGOL. For example, the SAGE Q-7 computer had no floating-point arithmetic; everything was fixed point. Therefore, it was assumed that fixed-point arithmetic would be needed for SACCS as well even though the SACCS Q-31 had floating-point arithmetic [SCHW78].

The limited memory size of the computers in the late 1950's and requirements of the SACCS program required special techniques for packing as much data as possible into a small amount of memory. In SAGE, assembly language techniques were used to accomplish this. Clearly, any HOL for such systems had to be able to support similar techniques.

In JOVIAL this was accomplished by adding the TABLE data structure to ALGOL and allowing the programmer the ability to specify the length (in bits) of each data item as well as its layout in each table entry. Although similiar features had been implemented in CLIP, their source was clearly SAGE [ENGL61].

The JOVIAL TABLE, like the COMPOOL, is a hallmark of the language. Whereas ALGOL provided only for arrays of floating-point and integer values, the TABLE provided the ability to specify an entire data structure with multiple entries, each of which consisted of multiple ITEMs (the term came from SAGE). Each entry, like a record in Pascal, could contain items of any type

Items could occupy full or part words with the exact length in bits specified by the programmer in JOVIAL. Furthermore, the

exact placement of each item in a table entry could be completely specified by the programmer (appendix C section 2.4). Thus large amounts of data could be packed into a small amount of memory. For example, Boolean variables were implemented as bit strings, one bit long, so as many as 32 Boolean items could be packed into a single word on a 32-bit computer.

Another idea from SAGE that simplified the use partial-word items and packed tables was that the programmer should not have to know where an item was stored within a word, which word in a table entry it was in, or even what the size of the item was. Thus, a JOVIAL program could remain largely unaffected by changes in the data description.

One of the arguments used to justify creating JOVIAL was that JOVIAL could provide this data transparency while generating code as efficient as the machine code used for SAGE [SCHW78]. For example, to add two items and store the result in a third took one statement in JOVIAL:

```
AA = BB + CC $
```

The same operation in SAGE assembly code required a minimum of 10 instructions to provide for storage of temporary results required by the single-accumulator computer and for shifting and masking partial word items.

The omission from J2 of any input/output (I/O) capability was directly attributable to SAGE. This was a planned omission from the start. In SAGE, only the central control program did I/O. It was therefore assumed that few of the programs for SACCS would

require any I/O capability. This shortcoming was partially corrected in J3 [BOCK61] by the addition of some I/O subroutines. In fact, the addition of I/O is the major difference between J2 and J3.

Another effect of SAGE was that multi-dimensional arrays were not handled efficiently in the J2 compilers, although single dimensional arrays were handled quite well [SCHW78]. It was felt that systems like SAGE would require little use of multidimensional arrays. As a result, JOVIAL was not often used for programming multidimensional matrix problems.

4.4 Compilers Programmed in JOVIAL.

Besides SACCS, JOVIAL was designed for programming the JOVIAL compilers themselves. As a consequence, the language had to be able to handle low-level machine details without resort to assembly language. This requirement overlapped with the need to access parts of items required by SAGE and was provided for in JOVIAL by the introduction of the functional modifiers (appendix C section 6). Functional modifiers are almost unique to JOVIAL. They are one of the strongest features of the language and have been retained in nearly all later versions (compare appendices C and D).

Functional modifiers could have values assigned to them as well as return machine dependent values. For example the SIGN function could be used to read or set the sign of an item. The BIT and BYTE functions allowed access to the internal value of any item. The internal value of any item may be considered a string of

bits or, in the case of character items, of bytes. The BIT function allowed any substring of bits to be extracted or assigned to an item of any type. The BYTE function provided similiar access to strings of bytes. Thus, interrupt handling and other real-time operations could be accomplished, although JOVIAL did not provide directly for them. However, BIT and BYTE were not efficiently implemented in the first J2 compilers [SCHW78].

Although an important objective of JOVIAL was to avoid using assembly language, the designers also knew that machine language would be required on occasion. In the early versions of J2, machine language was required to implement even minimal I/O. The work with CLIP had also established the need to be able to insert machine language directly into programs [BOOK60].

Access to machine language in J2 and J3 is accomplished by enclosing the assembly code between the keywords DIRECT and JOVIAL. To provide the programmer with access to JOVIAL and COMPOOL items and TABLEs, the ASSIGN operator was used to move a value from an item to the accumulator or vice versa (appendix C, 8.2). The compiler automatically took care of shifting and masking the items as it normally would in JOVIAL statements [SCHW78].

Another effect of writing the compiler in JOVIAL was a reasonable degree of machine independence of the language. Having to write two compilers in JOVIAL for two totally different computers also helped to achieve this objective [WILK61].

4.5 Non-Technical Influences.

4.5.1 Concurrent Language Development.

One of the major problems with managing the JOVIAL effort was that language design proceeded in parallel with compiler development [SCHW78]. Since one of the major objectives of the language was the development of the compilers in JOVIAL, many of the ideas for language features came from those who were developing the compiler. New features were added as the need arose. Thus, language decisions had to be made quickly resulting in some awkward syntax (language elegance was ignored) and difficulties in implementing the compilers [SCHW78].

Major language features, such as the CLOSE subroutines and the ITEM switch, were developed in literally minutes. Schwartz recalls the hasty creation of the data-definition capability, which was poorly defined when the project started, "When we realized we couldn't postpone it any longer, we developed it in about thirty minutes. Henry Howell and I examined each possible data type and structure [HOWE60] and developed the syntax for them immediately" [SCHW78]. As a result, the data-definition capability was quite flexible and powerful, but syntactically it consisted of a string of hard-to-memorize characters, in a specific sequence (see appendix C). For example, the definition of long string constants (e.g. 21H(THIS IS A LONG STRING) was error-prone requiring laborious counting and recounting for changing [SCHW78].

4.5.2 Lack of Committees.

Except for STRING item, language features seldom required more than a day to design. STRING items were the exception because the compiler writers objected to their difficulty to implement. The STRING item provided for a rather complex combination of character string elements in a variety of formats within a TABLE entry (see Appendix C) and was the only committee decision in the early language [SCHW78]. Otherwise, there were no committees involved in JOVIAL language design; all the features were added by Schwartz personally or with his approval. Schwartz cites the lack of committee involvement as an advantage in the development of JOVIAL, "One couldn't define things in minutes with committees"[SCHW78].

4.5.3 Language Experience.

With the exception of Jules Schwartz, almost no one on the CUSS project had any experience with HOLs. Schwartz had worked on a language called PACT (Project for Automatic Coding Techniques) [MELA56] for the IBM 701 at Rand, but he felt that this experience was not particularly valuable as PACT had little in common with ALGOL type languages and the 701 had little in common with the 709 [SCHW78].

There are probably more compiler developers around today than there were programmers in the world at the time JOVIAL was developed. The people who worked on the CUSS project were primarily programmers, not language specialists [SCHW78]. Some people had experience with SAGE. Others had never programmed before; they were

hired and trained for this job. FORTRAN had been in use for a few years, but only one person on the project had used it. The others had only read the FORTRAN specifications.

An important consequence of this was that formality was ignored during the intial language development despite the example provided by the formal description of ALGOL. On the other hand, Schwartz felt that this approach contributed to JOVIAL's being very natural to use [SCHW78].

4.5.4 Language Specifications.

No complete language specification existed at the start of or during compiler construction. There were some basic concepts from ALGOL and SAGE and enough definition to get started. Later, informal documents were written which provided some baseline, but these early documents were essentially working papers. Changes were made almost daily and code was often produced based only on spoken commuication. No reasonably complete, final documents were published until the time of the first compiler delivery in March 1961, almost two and one-half years after the project began [SCHW78]. Later other formal language descriptions were written by those who designed J3 (see next chapter).

The lack of formal language descriptions was compounded by the lack of detailed plans and schedules for the project. Management of the JOVIAL language and compiler development was informal. According to Schwartz, management practices were "almost nonexistent. The main management technique was frequent interaction

among all personnel, where changes, techniques, and progress were discussed on an individual and group basis" [SCHW78].

4.5.5 Schedule Pressures.

The lack of sufficient time was a constant problem during development of the J2 language and compilers. The original schedule called for delivery of the 709 compiler about six months after the start of the project. Although the rapid fashion in which the language and compilers were implemented was mainly due to the schedule, Schwartz later attributed this hurrying as much to the personalities of the developers as to the pressure of the schedule [SCHW78]. The developers were typical programmers who were not very research oriented and liked to see things run.

A more long-term effect of the tight schedule was the lack of standardization with the J3 effort going on in Santa Monica (next chapter). Originally, the J2 7090 and Q-31 compilers were to be integrated with the J3 common front end. But, the schedule pressures of SACCS prevented delaying these compilers until completion of the J3 front end. Also, these developments were at opposite ends of the country, J2 in New Jersey and J3 in Santa Monica, and cross-continent attempts to eliminate source language differences were not very successful [SHAW62C]. Thus, the only compiler for the popular 7090 computer was for J2, which tended to perpetuate J2 and prevent J3 from becoming the standard version of JOVIAL.

Chapter 5

JOVIAL/J3

About June 1960 (figure 3), SDC began work on improvements to the language, even before the first J2 compilers were fully completed. The result of these improvements was JOVIAL/J3. The process of how J3 evolved from J2 clearly demonstrated the effects that non-technical decisions can have on language design.

5.1 J3 Language Development.

In early 1960, SDC was at its peak. The last of SAGE was being installed. Work on the SACCS JOVIAL compilers was in full swing. SDC was preparing for new contracts that would require programming three new computers. With the advantages of HOLs clearly established by the work done on CLIP and JOVIAL and a growing realization of the potential scope for applying HOLs, SDC decided in May 1960 to standardize on JOVIAL as a corporate language [SHAW60C].

Following this decision, SDC began development of three new JOVIAL compilers. Several groups were established under Bob Bosak, head of the SDC Programming Languages Committee, to develop the compilers. The CLIP research staff was recruited to do the job. Erwing Book headed what was then called the Common Generator project to develop a common front end. Each code generator was developed by a separate group.

Because these compilers would have to run on many different computers, it was desired to introduce the best features of CLIP into J2 in order to improve the language and make it more nearly computer-independent [SHAW62C]. The designers tried to stay as close as possible to J2 and still meet their perceived requirements [SHAW81]. The resulting language that finally emerged as the SDC corporate standard was called JOVIAL/J3 and eventually became the most accepted version of JOVIAL.

The design of J3 was appropriately given to the Common Generator group. Erwin Book and Chris Shaw were the chief contributors. Shaw was responsible for most of the early documentation of JOVIAL. His involment with languages at SDC began with writing user's manuals and teaching Q-7 assembly language for SAGE. As documentor of the new language, Shaw presented the user's point of view in the design discussions of J3.

Erwin Book was in charge of the actual compiler development and represented the implementor's point of view. During meetings to decide what features to put into the language, Shaw would propose a language construct that he felt would be useful and Book would say whether or not he could implement it. Bob Bozak, as head program manager, would arbitrate.

JOVIAL/J3 was first described by Shaw in [SHAW60A], [SHAW60B], and [SHAW64A]. The first formal presentation on JOVIAL/J3 was given at the International Symposium on Symbolic Languages in March of 1962 [SCHW62]. Appendix C contains a detailed description of JOVIAL/J3.

5.2 J3 Compiler Development.

Like the previous J2 and CLIP compilers, these new compilers were designed with a separate front end (the generator) and code generator (translator), although a different intermediate language was used to be more compatible with the computers for which the compilers were being developed [SHAW60C].

These compilers all shared a same common front end which was combined with different code generators to form a compiler for the different computers. The goals SDC hoped to achieve with this design are consistent with modern ideas on compiler design. Having only one front end meant that control over the form of the JOVIAL language was centralized, limiting the tendency toward divergent dialects. More importantly, the duplication of effort involved in writing a unique front end for each compiler was eliminated.

At first, the common front end was being written in JOVIAL. But, as the task progressed, it became apparent that not all the features offered by JOVIAL were being used to implement the front end. There were no floating point variables, no n-dimensional arrays, no exponentiation, etc. In fact, the set of JOVIAL features the compiler builders were using was nearly identical to CLIP. This was not at all suprising. The operations of a parser are pretty much independent of the language it handles. A new plan of action was initiated: modify the CLIP compiler to accept those forms of JOVIAL which were like CLIP and use it to produce and checkout a JOVIAL front end for the 709.

About August 1960, Harvey Bratman and Erwin Book made some minor modifications to the CLIP compiler to produce a special version that would accept a program written in the CLIP-like subset of JOVIAL. This version was used by the common front end project to produce a 709 version of the JOVIAL front end, thus bypassing the tedious task of hand-translation to SCAT 709 assembly code. This procedure allowed the JOVIAL front end to be ready in November 1960, months ahead of schedule [SHAW60C].

Meanwhile, the code generators were coded in JOVIAL and hand translated into CLIP. The modified CLIP compiler was again used to produce 709 versions of the Q-7 and Philco 2000 code generators as a first step in producing compilers for these computers [SHAW60C]. The code generators were then used to translate the IL versions of the front end and code generators into machine language versions for each machine [BOOK60]. These code generators were highly integrated, monothlithic programs. Of necessity they had references to CLIP language forms and did extensive checking of DIRECT assembly code statements that were passed along unparsed by the common front end [VAND69].

When the common front end project was formed and work began on upgrading the J2 language, projects to build code generators for the IBM Q-7, Q-31 and 7090 computers were established. These code generators were to be integrated with the J3 common front end. Later projects to build code generators for the Philco 2000 and the CDC 1604 were added. Working compilers for all five computers were scheduled for operation in the spring of 1961 [SHAW60C]. The J3

compilers for the IBM AN/FSQ-7, Philco 2000, and CDC 1604 were developed as planned, all sharing the same common front end and IL but, due to the SACCS schedule pressures, the 7090 and Q-31 compilers could not await completion of the front end and remained J2 compilers.

These compilers had 50,000--60,000 machine instructions and required about 5 man-years to develop [SHAW62C]. A central staff was established in Santa Monica under SDC's Information Processing Directorate [SHAW62C] to maintain the JOVIAL language, the common front end, the 7090, 2000, and 1604 compilers, and the documentation relating to them. The JOVIAL compiler staff consisted of about 16 people and was headed by Gene Gordon.

The development and use of each of these early J3 compilers is now discussed in more detail.

5.2.1 The Q-7 J3 Compiler.

In March 1960, even before the decision to standardize on JOVIAL, SDC began building a JOVIAL compiler for the IBM AN/FSQ-32 (Q-32) computer for use on the proposed SAGE Super Combat Center contract (see chapter 6). When that contract was cancelled in April 1960, the work on the Q-32 compiler effort was redirected to producing a compiler for the Q-7 and Q-8 computers to be used for upgrades to SAGE. By this time the common front end project had been started, so the Q-7 compiler was integrated with the common front end.

The IBM Q-7 code generator project was headed by Cal Jackson. Work began in July 1960 and was completed in June 1961 [SHAW62C]. Besides SAGE, the Q-7 compiler was used by SDC in Santa Monica for development programming by the Data Processing Department and by the Professional and Technical Training Staff of the Personnel Directorate [SHAW62B].

5.2.2 The Philco 2000 J3 Compiler.

The Philco 2000 computer was one of the first solid state computers. Because of its increased capability, it was used in several SDC projects. A code generator project for the Philco 2000, under Ellen Clark, was added to the JOVIAL front end project in October 1960 [SHAW60C]. The completed Philco 2000 compiler was ready in October 1961 [SHAW62C].

The newly-completed Philco 2000 J3 compiler was first hosted on the central computer of the Systems Simulation Research Laboratory (SSRL). The SSRL was designed for the study of man-machine interaction in computer-based systems as part of a new SDC research and development program in the systems sciences [BAUM81].

The Philco 2000 compiler was also a central component of two software development contracts SDC received in 1960 from NORAD: the Combat Operations Center (COC, Project 425L) and the Spacetrack System (Project 496L). The COC was designed to integrate defense data received from several sources, including SAGE and SACCS, into a status picture to provide the NORAD battle staff with real-time

displays of the aerospace threat and status of defense forces and operations. The COC was installed in the new NORAD underground Cheyenne Mountain complex in 1965 [BAUM81].

Spacetrack addressed the need for current space data as a result of the 1957 Soviet Sputnik launch. It was to detect, track, identify, and catalog all space objects and display this data to the NORAD staff. Spacetrack, which later became the Space Defense Center, was completed in 1963 [BAUM81].

Working with the Air Force Electronic Systems Division (ESD) and NORAD, SDC's Lexington, MA, staff decided to implement the two systems using three Philco 2000 computers: one for the COC, one for Spacetrack, and a third as a backup for both [BAUM81]. The operating systems for both contracts were written in JOVIAL/J3. The SDC later fitted the 2000 compiler into the Philco SYS operating system [SHAW62C].

Later in 1970, the Air Force recognized that increasing processing loads mandated an upgrade to the NORAD Cheyenne Mountain facilities and awarded SDC the prime contract for the Space Computation Center which was a segment of the Cheyenne Mountain Upgrade (program 427M). This program was also programmed in JOVIAL using a JOVIAL compiler for the Honeywell-6080 computer.

In October 1960, SDC's Washington, D.C. Division in Falls Church, VA, used the Philco 2000 JOVIAL compiler to automate the Defense National Communications Control Center (DNCCC) for the Defense Communications Agency (DCA). Using JOVIAL allowed SDC to design and produce the software in only 5 months [BAUM81]. In

addition to these uses at SDC, the 2000 J3 compiler was released to other users through the Philco computer users group, TUG.

5.2.3 The CDC 1604 J3 Compiler.

In December 1960, the Defense Atomic Support Agency (DASA) awarded a contract to SDC to design the software for the Department of Defense Damage Assessment Center (DODDAC), which was established to assess damage and fallout from nuclear detonations and to relay the data to designated military agencies [BAUM81].

To do this, SDC built a utility system and JOVIAL/J3 compiler for the 48-bit, Control Data Corporation CDC 1604 computer. Development of the 1604 code generator was added to the common front end project in December 1960. The 1604 compiler was operational in January 1962 [SHAW62C].

SDC used this compiler to convert the contamination and fallout programs from the IBM 709 to the CDC 1604. This compiler later formed the basis for the CDC 3800 used by SDC for the Satellite Control Facility (see chapter 8).

In early 1962, the Naval Command Systems Support Activity (NAVCOSSACT) adopted JOVIAL/J3 for its strategic command-control systems, rejecting two other command-type languages developed especially for the Navy. This decision was based on a study comparing JOVIAL with a number of other languages including machine language [SCHW78].

In 1965, NAVCOSSACT contracted SDC to build a computer program production system and related services to ensure the

effective use of JOVIAL for the Navy [BAUM81]. To do this, SDC modified the 1604 compiler to run on the NAVCOSSACT CDC 1604A computer, a slightly different model than the 1604. This CDC 1604A compiler ran under the NAVCOSSACT Master Control System and used the OASIS utility system, which contained I/O routines and math library [KLEI64]. This compiler was used at the Naval Fleet Operations Control Center for the Atlantic (FOCCLANT) and Pacific Fleets (FOCCPAC).

The 1604 compiler was released to other users through the CDC computer user's group, CO-OP, and was later transferred to the 1604B at RADC in 1968, where it was integrated into the Program Production System (PPS). PPS was developed by SDC for RADC as a batch-oriented utility system, which could perform several tasks within one job set-up, although it was initiated as a single task under the CO-OP Monitor Control System [BUDE69].

In 1968, the CDC 1604B J3 compiler was combined with the UNIVAC M555-hosted JOVIAL-MW compiler, which only accepted a small subset of JOVIAL, to produce a J3 compiler for the M555. This 1604/M555 compiler was built by writing a new code generator for the 1604B compiler to produce TRIM assembly code, which was assembled by the TRIM assembler in the the UNIVAC M555 compiler [VAND69].

The 1604B compiler was used similiarly by Informatics in 1970 to produce a limited test compiler that produced GE-635 GMAP assembly code [VERH70]. This compiler was developed as an

inexpensive means for constructing programs for the GE-635 for experiments in the area of automatic segmentation and reentrancy for an RADC research contract.

SDC also distributed copies of the 1604 compiler to companies such as Airborne Instruments Lab, Deerpark NY, CEIR Los Angeles Center, CDC, Cornell University, RCA, United Nuclear, USN Postgraduate School, Univ. of Minnesota, and the National Military Command System Support Center [GRAN64]

Chapter 6

JOVIAL TIMESHARING COMPILERS

As seen with the effect of SACCS on J2, one of the most important technical influences on a language is the expected application domain, which, or course, includes the expected hardware environment. The design of the early JOVIAL compilers and, to some extent, the language itself were influenced by the computers and the operating systems that these compilers ran on. All interaction with these early computers was via punched card input and printed output.

With the development of interactive timesharing operating systems, the demands on the language changed. An interactive JOVIAL interpreter, called TINT, and a timesharing compiler, called JTS, were developed to provide those features needed by an on-line user (see figure 4).

6.1 The FSQ-32 Compiler.

TSS came about through a series of events that began with cancellation of the SAGE Super Combat Center. Even before its installation was completed, SAGE was becoming obsolete. In particular, the SAGE direction centers were found to be highly vulnerable to a nuclear strike. The Air Force proposed building a series of blast-hardened underground Super Combat Centers, which would be based on a new computer, the solid-state IBM AN/FSQ-7A, later called the AN/FSQ-32 (Q-32).

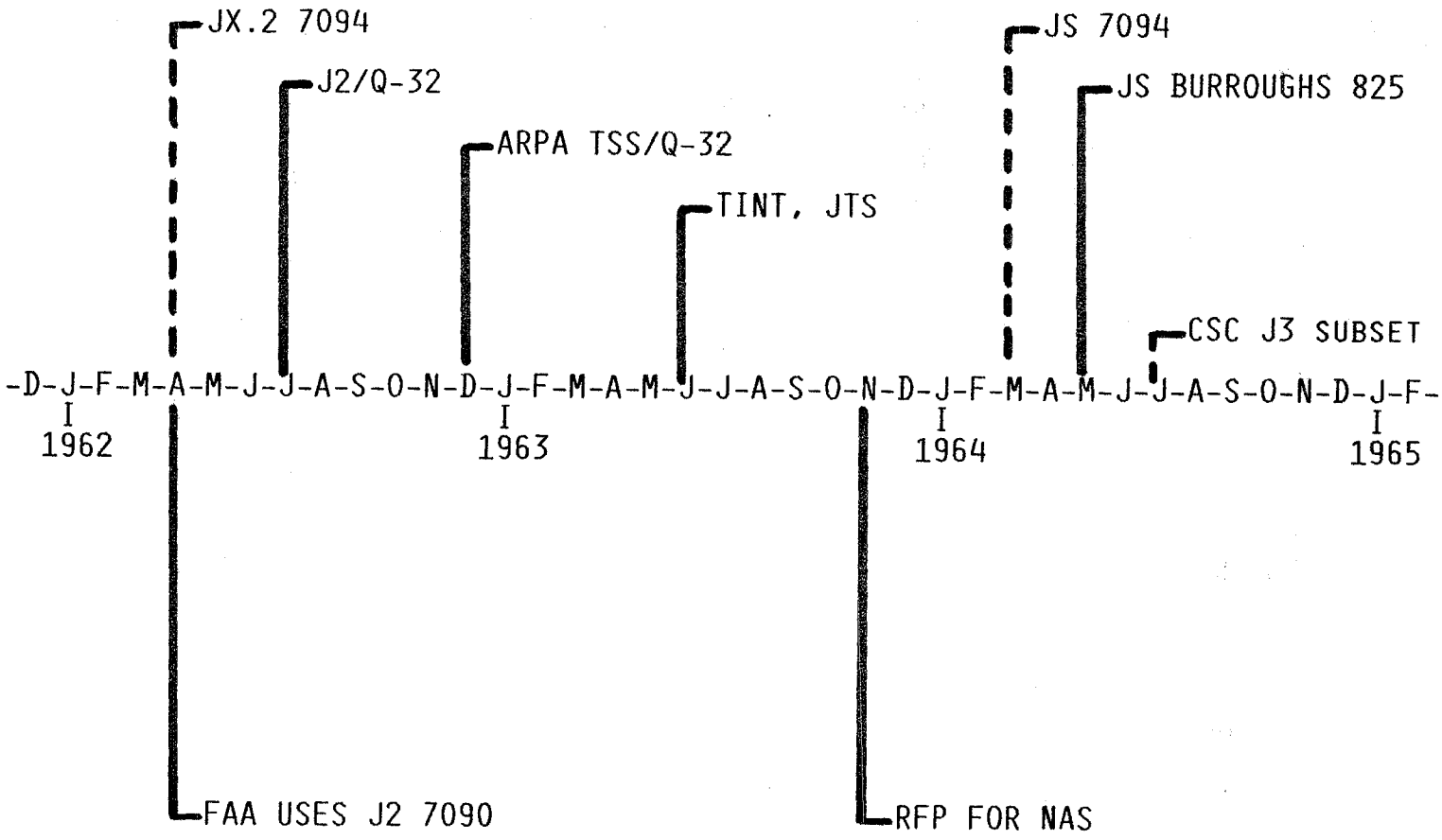


Figure 4. JOVIAL Subset and Timesharing Compilers (1962 -- 1965)

SDC was chosen as the software contractor. In March 1960, the Q-32 computer was ordered and work began on a Q-32 JOVIAL compiler [SHAW62C]. This was prior to SDC's decision to standardize on JOVIAL and added impetus to that decision. But the SCC contract was cancelled a month later after a presidential commission concluded that, although the Combat Centers themselves might not be vulnerable to nuclear attack, their extended communications lines were vulnerable, which would render them useless [BAUM81].

As already noted, work on the Q-32 compiler was switched over to development of a compiler for the Q-7 [SHAW62C]. But the shipment of the Q-32 computer went into limbo. This was remedied in September 1961 when SDC received a research contract from the Advanced Research Projects Agency (ARPA) of the Department of Defense (DOD). The contract called for using the Q-32 computer to support a new Command Research Laboratory (CRL) to conduct research on modern command-control systems [BAUM81].

In July 1961, the $7 million Q-32 was installed in the CRL, which occupied a building adjacent to its sister laboratory, the SSRL [BAUM81]. The Q-32 was a 48-bit computer with 64K of memory and an extremely powerful instruction set, including access to parts of words for loading, storing, and arithmetic [SCHW64]. The Q-32 was operational in March 1962 and was used by researchers from SDC, ARPA, and other ARPA contractors to create computerized models to study existing and proposed command centers [BAUM81].

6.2 ARPA TSS.

In mid 1962, Dr. J.C.R. Licklider of MIT, well-known for his interest in the novel concept of timesharing, became the technical monitor for SDC's ARPA contract. Soon thereafter, he redirected SDC's contract to focus on developing a timesharing system. In December 1962, a timesharing project was formed under the direction of Jules Schwartz [BAUM81] and, six-months later, the ARPA Time-Sharing System was running on the Q-32 (figure 4).

TSS was written in JOVIAL using a J3 compiler that was produced by rewriting the code generator of the Q-31 SACCS compiler and combining it with the J3 common front end to produce a compiler for the Q-32 [SHAW62C].

TSS allowed interactive use of the Q-32 computer via the use of Teletype terminals [KENN65]. A remote demonstration of TSS, involving a teletype linked to the Q-32 from MIT, was held in October 1963, and a year later, TSS demonstrated the first international use of timesharing from SDC in Santa Monica to Copenhagen, Denmark.

6.3 The JTS and TINT JOVIAL Compilers.

To make TSS a convenient vehicle for what Licklider called "man-computer symbiosis" [LICK60], SDC developed a variety of tutorial and conversational programs for TSS. One of these was the Teletype INTerpreter (TINT) for JOVIAL.

TINT was a two-pass interpretive program that operated upon a subset of JOVIAL. The first pass (front end) performed syntax checking and translated JOVIAL source code into a reverse Polish intermediate language. The front end was derived from the 7090 J2 compiler and modified to handle the dialect used by TINT [SCHW64]. The second pass was an interpreter instead of a code generator. It scaned the intermediate language version of the program for the prefix operators and their arguments and called the corresponding subroutine to perform the operation on these arguments. The operator subroutines and interpreter were developed specifically for TINT.

The TINT subset of JOVIAL was a subset of J2 and J3 with some unique extensions for I/O formatting. TINT included arithmetic, relational, and Boolean operators; procedure calls; table, array, and item (integer, floating point, and Hollerith) declarations; and the GOTO, IF, STOP, READ, and PRINT statements. The READ and PRINT statements were added to the language specifically for time-sharing operation. TINT programs were limited to a maximum of 150 lines [KENN65], although programs as long as 600 statements could be run by putting more than one statement per line.

Many on-line debugging and communication aids not available in a batch system were provided. TINT commands were preceeded by a "?" to distinguish them from JOVIAL statements and included the following: ?START, ?CONTINUE, ?EXECUTE, ?PRINT, ?INSERT, ?DELETE, ?COPY, ?RENUMBER, ?HALT, ?GO, ?GOTO, ?SAVE, ?LOAD, ?DEBUG, ?BREAK, ?TRACE, ?TAPE, ?EXPLAIN [KENN65] .

TINT allowed programs to be entered, edited, and executed on-line rather than waiting for keypunching, compiling, and execution as with a batch system. The TINT user could execute programs previously stored on tape or he could enter and execute programs directly from the Teletype. TINT programs could also be filed away on tape for compilation with the JOVIAL Time-Sharing (JTS) compiler.

JTS was a small, one-pass compiler that only recognized a very limited subset of J2 and J3, but it was still a larger subset than that of TINT. It was designed for batch-oriented use under TSS. In 1965, when SDC merged the two computer laboratories, the SSRL and the CRL, into a new Computer Center and replaced the Philco 2000 and Q-32 with the new, third-generation IBM 360 [BAUM81], both TINT and JTS were rehosted to the 360 (figure 5),

6.4 ADEPT-50.

Over the next four years, TSS was expanded into the areas of networking, security, and data management. The work on data management was especially important to SDC as systems designers came to see the problems in implementing command-control systems in terms of the underlying data management problem. Key SDC developments in this area were TSS, a conversational data management system called LUCID (Language Used to Communicate Information-system Design), and a technique of on-line display building called General Purpose Display System (GPDS) that allowed users to construct formats for interaction with on-line databases. In 1965, these elements were

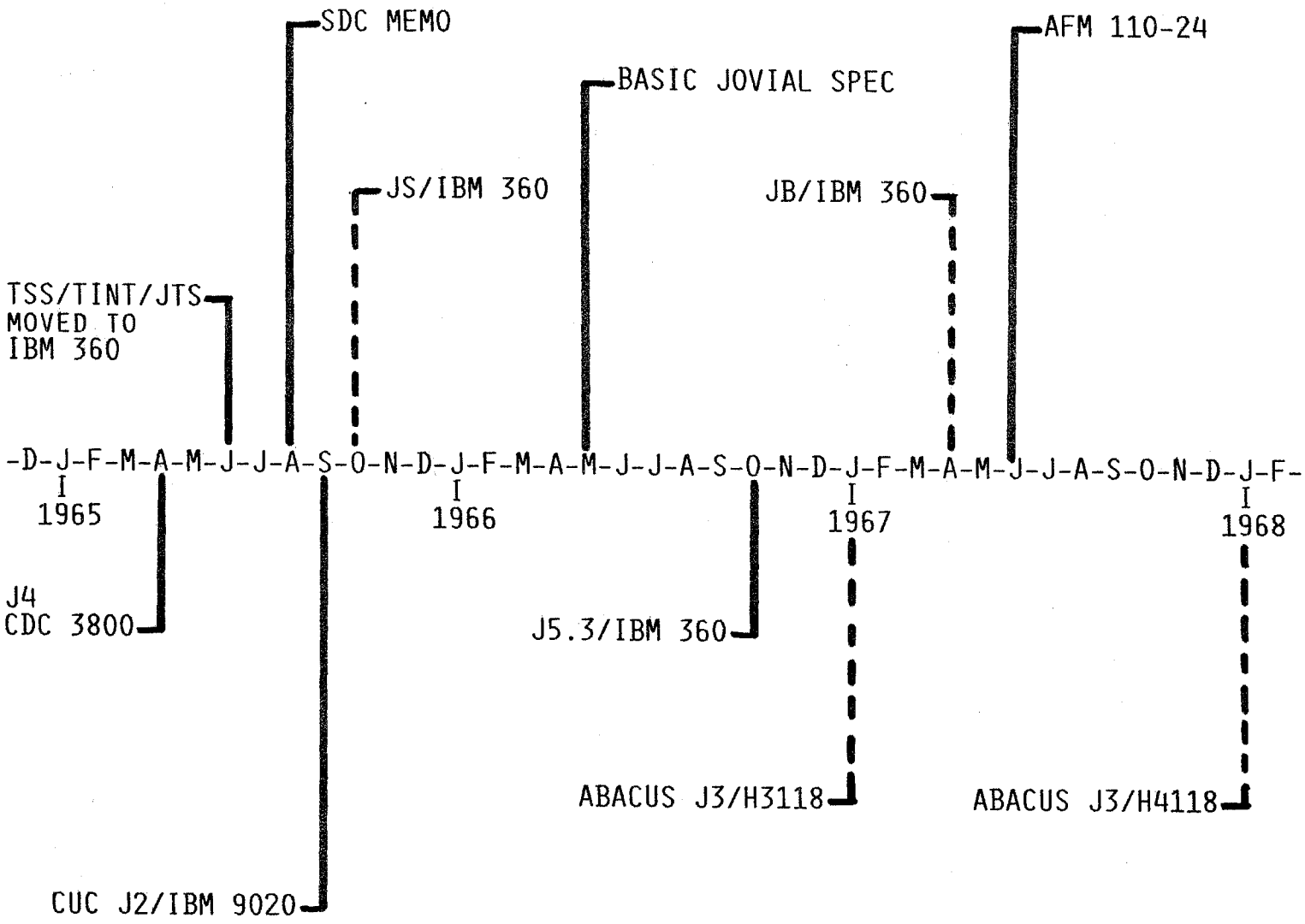


Figure 5. JOVIAL J4 and J5 (1965 -- 1968).

combined when LUCID and GPDS were reprogrammed to operate together under TSS on the Q-32. The culmination of this work was the pace-setting Advanced Development Prototype (ADEPT-50).

ADEPT came about when ARPA, eager to move this work on timeshared data management out of the laboratory and into operational settings, gave SDC a two-year contract to develop a new general-purpose timesharing system to demonstrate the feasibility of having on-line access to large databases without programmer intermediaries [BAUM81]. Work on ADEPT began in June 1967 under Technology Director, Jules Schwartz. By November, the first release was running on the IBM 360 Model 50 (360/50) with 256K bytes of core. The system was complete in March 1968 and was later transferred to the IBM 360/65.

ADEPT allowed either batch or on-line program execution. It consisted of 3 components: the time-sharing executive, the Timeshared Data Management System (TDMS), and the Interactive Programming Support System (IPSS). TDMS was an interactive data management system incorporating features of LUCID but having larger storage and more efficient data structures. It was built in 1966 when the IBM 360 replaced the Q-32 and was later incorporated into ADEPT [BAUM81].

IPSS was the first JOVIAL programming support environment. It combined the JTS JOVIAL compiler, a debugger, a text editor, a utility program, and the JOVIAL TINT interpreter into a single system. IPSS permitted the coding, editing, execution, and testing of JOVIAL programs to be done on-line as a single coordinated

activity centered around an interactive compiler [BRAT68]. It emphasized program editing via an interactive display. Work on IPSS began at SDC in 1965 and was later incorporated into ADEPT [SCHW69].

Trial installations of ADEPT were made at the National Military Command System Support Center (NMCSSC) in May 1968 and at the Air Force Command Post in August 1968 [SCHW69]. SDC's design for a Nuclear Weapons Status System for the Defense Atomic Support Agency was founded on ADEPT architecture, as was the Movements Requirements Generator, a strategic mobility model SDC developed for the Pentagon in 1968 [BAUM81]. Later, the Air Force contracted SDC to convert ADEPT from the IBM 360 to the GE 635.

In September 1971, ADEPT served as the basis of the Tactical Intelligence Processing and Interpretation (TIPI) system, a mobile battlefield information system. SDC had developed the prototype version of TIPI in JOVIAL using ADEPT and TDMS on the IBM 360. But the Air Force wanted the operational version to use the AN/UYK-7 computer. Therefore, SDC had to build a new operating system and a new J3 compiler for the AN/UYK-7 [DUNB81].

Chapter 7

JOVIAL SUBSET COMPILERS

After the first JOVIAL compilers were completed, the attention of compiler builders at SDC turned to improving the speed of JOVIAL compilers, which were very slow. Compilers for two JOVIAL subsets, X.2 and JS, were built with considerable speed improvement. Quite unexpectedly, these efforts led to the first efforts to standardize JOVIAL.

7.1 JOVIAL/JX.2.

The first effort to provide a faster JOVIAL compiler was the development by Harvey Bratman in 1962 (figure 4) of the JOVIAL/JX.2 experimental compiler hosted on the IBM 7090 and later on the faster 7094 [SHAW62C]. This compiler accepted only a limited subset of JOVIAL and produced absolute binary 7090 code. The idea was to create an extremely fast, one-pass JOVIAL compiler that still provided a useful subset of the language [BOOK63A]. Bratman initially wanted to called his language subset, TRIVIAL. But SDC management thought this too flippant, and he changed the name to JX.2, which had no meaning whatsoever, but it sounded good [BRAT81]. JX.2 was essentially a subset of J2 as it had features that were not in J3, but that were in J2 [CHRI66]. The JX.2 compiler rivaled other compilers in speed but did not implement enough of the language to be useful [SCHW78].

7.2 JOVIAL/JS.

About 1964, the work on JX.2 was expanded into the development of JOVIAL/JS (figure 4). JS was a subset of J3 and a superset of JX.2, except for the few features that were in J2 but not J3. SDC built JS compilers for the IBM 7094, the Burroughs D-825, and the IBM 360.

The first JS compiler was hosted on the IBM 7094, like the JX.2 compiler it was derived from. Like other JOVIAL compilers it consisted of a separate front end and code generator communicating via an intermediate language. It ran on the IBM 7094 either by itself or under the Common Programming Support System (CPSS) [BLEI64], which was an experimental SDC machine-independent, batch-oriented operating system. CPSS included the JS compiler, tape file maintenance programs, and comprehensive I/O functions. Over 95% of CPSS was written in the JOVIAL/JS subset [CHRI66] [KLEI64].

In June 1964, a JS compiler for the Burroughs D-825 [HIRS67A] was used to program the Backup Interceptor Control (BUIC) system. BUIC (program 416M) was a decentralized backup system for SAGE that came about when the previously proposed SAGE Super Combat Center was cancelled. BUIC used the new second-generation solid state (transistorized) Burroughs D-825 computer, also called the AN/GSA-51, which required less than a tenth the floor space of the twin SAGE Q-7's, yet performed as well [BAUM81].

To develop the D-825 JS compiler, SDC first developed a JS cross compiler on IBM 7094 by writing a new code generator for the existing 7094 JS compiler. When the cross compiler was working it was self-compiled to produce a compiler for the D-825. Most of the work was done on the 7094 because it had the fastest JOVIAL compiler [ZEIT81]. Software developed for a previous SDC program, the Hamilton Combat Center, was used to debug the D-825 compiler [CHRI66].

In 1965, when the IBM 360 was installed at SDC, a JS compiler was built for that machine (figure 5). First, a cross-compiler that operated on the 7094 and produced IBM 360 binary code, was built by combining a front end derived from the J3 common front end with a new code generator that was modeled after the JS 7094 code generator. This cross-compiler was then self-compiled to produce a compiler for the 360. The resulting JS/360 compiler ran on the 360/65 under either TSS or the OS/360 operating system. Thus, there were three JOVIAL compilers for the 360: TINT, JTS, and JS. A program for translating JTS programs to JS on IBM 360 was also developed [CHRI66].

7.3 Basic JOVIAL.

The work on JS led to the specification of BASIC JOVIAL, which became the cornerstone of SDC's JOVIAL standardization policy. Basic JOVIAL was essentially a formal specification of the JS subset; the name Basic JOVIAL seemed more appropriate than JS.

The specification for Basic JOVIAL was first published in May 1966 and later revised in 1968 [PERS68]. It was derived from the J3 specification, TM-555/002, by omitting sections that were deleted from J3 rather than by renumbering the document, thus making comparisons between Basic JOVIAL and J3 easier.

Because the definition of JS preceeded the specification for Basic JOVIAL, the JS compilers were not initially compatible with Basic JOVIAL. But, in 1967, the JS/360 compiler was modified to become the first Basic JOVIAL compiler [SAND67].

7.4 SDC's JOVIAL Standardization Efforts.

By 1965, the need for stepped up efforts at standardizing JOVIAL had become apparent. Differences among compilers, not to mention within versions of the same compiler, arose due to implementation pressure (e.g. the SACCS 7090 and Q-31 J2 compilers), computer idiosyncrasies, program specific needs, and individual choices of compiler implementers (TINT and JTS) [SCHW78].

Despite SDC's early attempts to implement all the JOVIAL compilers using a single, common front end, nearly every compiler implemented a different subset of the language. None implemented the full language [SHAW62B] as described in the official J3 standard [PERS66B], although the major aspects generally were implemented, and there was near-compatibility in most versions. The proliferation of official specifications of the language (See [PERS64] and [PERS66B]) did not help.

The following differences between the J2 7090 and the J3 1604A compiler was typical of other JOVIAL compilers. Line numbers could start in column 67 on the 7090 but only in column 73 for the 1604A compiler. Three JOVIAL operators were not implemented in the 1604A compiler. Arrays were not yet implemented. Output data from the 1604A compiler was written to tapes in 8 characters per word format. Furthermore, output was not in Hollerith code but in OPCON code. The 1604A compiler had some I/O capability unlike the 7090 J2 compiler where all I/O was done using 7090 machine code subroutines [KLEI64].

Although there were continuous attempts within SDC to control these problems, little real control was exercised over the language until 20 August 1965 when SDC Memo M14658 was issued. This memo mandated the standardization of JOVIAL compilers [PERS68].

SDC's new policy was that, as a minimum, any new JOVIAL compiler had to implement Basic JOVIAL (JB) [PERS68]. And if the new compiler provided capabilities included in J3, it must implement them according to SDC technical manual TM-555/002, The JOVIAL (J3) Grammar and Lexicon [PERS66B]. Furthermore any compiler that did not implement TM-555/002 exactly could not be designated J3. Thus, the emergence of the dialects J4 and J5 which were essentially still J3 compilers.

SDC recognized that some features, like double precision arithmetic, would be machine-dependent. Thus, slight deviations were allowed provided they were compatible with the specifications.

Any new JOVIAL compiler had to be fully documented with respect to those features of JOVIAL described as system-dependent and with respect to differences from J3 or Basic JOVIAL. The intent of the policy was that any program that used no more than Basic JOVIAL could be compiled on any new compiler at SDC.

Although this policy was limited to SDC, it was the first effort of any kind to control the JOVIAL language. Thus, the creation of JOVIAL subsets, which would be regarded today as a move away from language standardization, actually led to the creation of a language standardization policy within SDC.

Chapter 8

JOVIAL/J4 and JOVIAL/J5

The effect of SDC's new standardization policy was the creation of J4 and J5. Although J4 and J5 did not differ from J3 in more than a few places, SDC's JOVIAL policy required that any version of JOVIAL that did not conform the J3 specification could not be called J3. J4 had four input/output extensions that were non-standard. J5 was a version of basic JOVIAL with extensions. The effect of not being able to refer to these versions as J3, gave the impression of there being more versions of JOVIAL than actually existed.

8.1 J4 Language and Compiler Development.

JOVIAL/J4 was developed by SDC for programming the satellite tracking and control system at the Air Force Satellite Control Facility (SCF) in Sunnyvale, CA. The SCF was established in 1956 to support the Discoverer satellite program and consisted of a network of six world-wide tracking stations connected to the central Satellite Test Center (STC) in Sunnyvale, CA. The SCF is responsible for monitoring and controling all U.S. military satellites after they are in orbit.

SDC's involvement in the SCF began in October 1961 when SDC was selected as the SCF Computer Program Integration Contractor (CPIC) [BAUM81]. This came about when SDC's Space Systems staff

submitted an unsolicited proposal to the Air Force Space Systems Division (SSD), then located in Inglewood, CA, to develop a reconnaissance satellite system. This proposal stimulated SDD's interest in applying SDC's computer experience to satellite systems, and SDC was later chosen as the CPIC for the SCF. SDC's function as CPIC was to integrate the computer programs developed by various contractors for a particular satellite into the SCF's central tracking and control system.

About 1964, SDC began a modernization of the SCF data system. A new tracking and control systems was designed around the CDC 3600 computer, which was replaced by the faster CDC 3800 a year later before development of the new system had progressed very far.

To improve productivity in developing the many computer programs that would be required for each new satellite system, SDC programmed the new system in J3 instead of assembly language like the old system. SDC developed a new JOVIAL compiler for the CDC 3600 by modifying the CDC 1604A compiler SDC had used for DODDAC in 1962. Slight modifications were again required when the 3800 computer was introduced. The compiler was completed in April 1965 (figure 5).

The new 3800 compiler contained four added constructs: FORMIN, FORMOUT, DECODE, and ENCODE (appendix C, 4.14) to enhance the input/output capabilities of the 3800 with the new disk file system. Because these statements were an extension to J3, the compiler was designated J4 although it differed from J3 by only these 4 out of approximately 104 language constructs.

JOVIAL again played a role in the SCF in September 1980, when IBM Federal Systems Division, Gaitherburg, MD, began another modernization of the SCF data system using IBM 3033 and 4341 computers programmed in JOVIAL/J73B using the SEA IBM-hosted J73B compiler (see chapter 20).

8.2 J5 Language and Compiler Development.

Unlike J4, which was an extension to J3, JOVIAL/J5.3 was only a JOVIAL subset. JOVIAL/J5.3 came from a proposed version called prJ5.2, which was derived by adding extensions to the Basic JOVIAL subset [SYST67], [PERS67]. J5.3 was used by SDC to develop programs for the NMCSSC, which was created in 1963 out of the old DODDAC that SDC had programmed with CDC 1604 compiler (see section 5.2.3) [BAUM81].

The JOVIAL/J5.3 compiler was built in 1967 (Figure 5) by modifying the JS/360 compiler [DOBK68]. It ran on the IBM 360 under the NMCSSC operating system [SYST67], the TDMS timesharing system, or ADEPT [SAND69]. Under TDMS the user could invoke the compiler either on-line or submit compilations from his terminal for batch processing via the TDMS batch monitor.

The J5.3 language was based on the Basic JOVIAL subset with the following extensions:

1. Floating point and integer types were 32 bits.
2. External names could be 8 rather than only 6 characters long.
3. 6 additional characters implemented as Hollerith values in EBCDIC were added to the character set.
4. The fixed-point data type was added.
5. Floating point as well as integer exponents were permitted, but complex roots were undefined.
6. Referencing a statement label where execution is to be restarted by the operator following execution of a STOP statement (see Appendix C, 4.12) and allowing specification of the starting machine address for loading the compiled program were not permitted as all programs operated under control of an operating system.
7. Externally compiled procedures and functions could be referenced in a program, but a skeleton declaration was then required.

Note that item 6 above was due to the use of an operating system and again demonstrated how the hardware environment of compiler influenced the content of the language.

Chapter 9

JOVIAL/J6

JOVIAL/J6 was an extensively modified version of J3 that was developed in 1968 (figure 6) to solve the technical programming problems of a specific application and demonstrates again how important the influence of the application problem domain can be. J6 was developed for programming spaceborne flight-computers and came about due to the lack of adequate capabilities in existing languages to perform the matrix arithmetic and interrupt handling operations needed for space programming.

9.1 J6 Language Development.

J6 was the result of an SDC study [HIRS67A] on software development for space systems. The report recommended the use of a common, powerful HOL to replace the use of assembly languages for programming spaceborne flight computers. ALGOL, FORTRAN, JOVIAL/J3, NELIAC, and PL/I were evaluated as candidates for such a language. PL/I and JOVIAL/J3 were chosen for further evaluation. PL/I was found to be more expressive but overly complex with many features not well integrated into the language. Nearly all J3 features were useful for spaceborne software, but J3 lacked many necessary capabilities. Neither language was judged to be satisfactory without extensive modication. The study, therefore, recommended development of a new space programming language [HIRS67A].

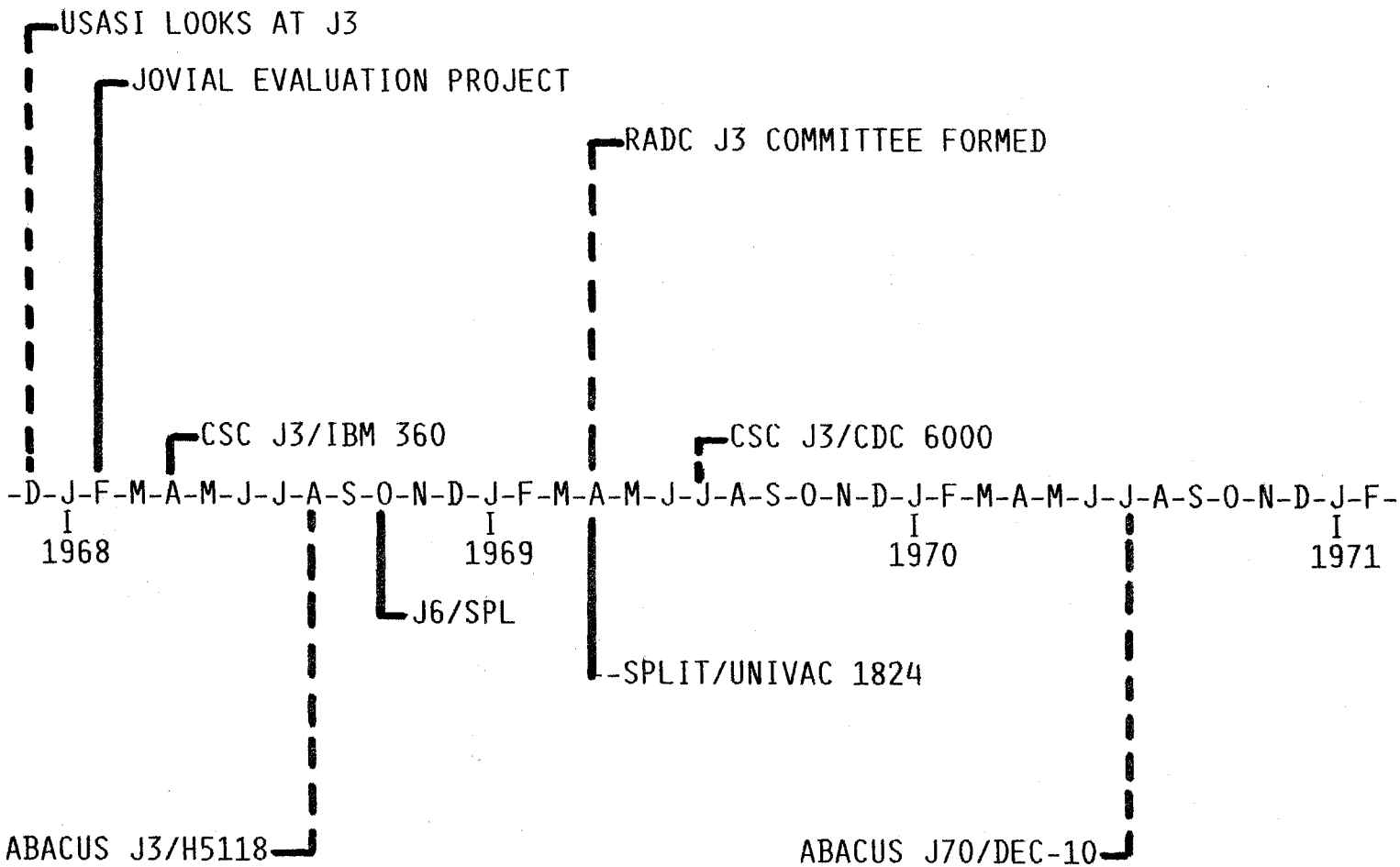


Figure 6. JOVIAL J6 and CSC Compilers (1968 -- 1971).

In December 1967, SDC was contracted by the Air Force Space and Missile Systems Organization (SAMSO), formerly SSD, to develop the proposed language. The language was delivered in October 1968 [HIRS68]. It was called Space Programming Language or JOVIAL 6 (SPL/J6).

SPL/J6 was based on J3 rather than created from scratch largely because of political considerations. By this time, J3 had come to be identified as an Air Force language (chapter 13). Since SPL/J6 was developed for Air Force space programs, compatibility with J3 was an objective.

SDC specified five subsets of SPL/J6: SPL/Mark I (also called CLASP) was the simplest and was implemented in 1970, SPL/MK II was designed for fixed-point spaceborne computers, SPL/MK III for floating-point spaceborne computers, SPL/MK IV for ground-based support computers, and SPL/MK V for ground-based multiprocessor computers.

9.2 J6 Language Features.

The most notable feature of J6 was the ability to perform matrix operations. Major differences between J6/MK II subset and J3 are as follows [NIME70] (compare with appendix C):

1. Language form. SPL eliminated the $ statement terminators as well as BEGIN END brackets. ENDs are used where many statements occur in a compound manner, such as in the IF or FOR statements.

2. Data Declarations. Many data declaration features were added. For example, implicit data attributes do not have to be specified, such as INTEGER, which means full word, signed and not rounded. Also, more than one data item may be declared in a declaration list.
3. Matrix operators. SPL allows for non-scalar expressions well as the usual arithmetic ones. The matrix operators are sum, product, transpose, inverse, scalar product, scalar sum, and scalar difference [HIRS68].
4. Real-Time Operations. SPL provides statements for interrupt handling as well as the usual DIRECT capability and the functional modifiers. The CHRONIC statement is executed only on the occurrence of its enabling condition, which is a Boolean formula that is automatically evaluated whenever its first operand, which could be a hardware device, acquires a new value. The SPL compiler generates all the code necessary for saving registers and returning control to the instruction following the location where the interrupt occurred. The DELAY statement allows waiting for a hardware or software condition to occur before proceeding.
5. Input/Output. The formatting of subfields within records is done implicitly during program execution. All I/O statements are machine-independent. The file declarations decribe what the subfields of the record look like to the external device and what type of conversion the operating system has to do before the data can be used.

6. Closed Subroutines. SPL provides for recursive, reentrant, multiple-entry, and multiple-output closed subroutines. No distinction is made between procedures and functions since procedures can return output values.

7. Directives. Compiler directives were added to allow for calculation of parameters during compilation, to permit optimization of a block of code for space or time, and to cause timing of portions of an object program.

8. Decision Logic. The loop statement may have a WHILE or UNTIL clause:

```
FOR I,J = 1,4 WHILE I=I+2 LS 100 and J=J+I LS 50
   . . .
END
```

Alternate forms of the loop statement omit the FOR clause and begin with WHILE or UNTIL. The IF statement may contain and ELSE clause and is followed by an END statement to resolve ambiguity of nested IF-THEN-ELSE statements.

9. Decision Tables. A unique feature of SPL was the decision table, which can be thought of as a type of CASE statement. The decision table decribes a series of rules for deciding what action is to be taken when certain condition are met. For example, the following SPL decision table is executed as follows:

```
CONDITIONS  A      EQ [   B   ,  19   ,  C/D  ]
            Q      LS [   R   ,  Z    ,  J*9  ]
            ZR     GR [  100  ,  150  ,   5   ]
ACTIONS     SIGNAL = [ FALSE, FALSE, TRUE  ]
            ALERT  = [       , RED  , CLEAR ]
            GOTO     [ ABLE , ABLE , ABLE  ]
ELSE        GOTO                             ERROR
            END
```

The columns correspond to the rules. Rule 1 of the table is tried first. Rule 1 is read: If A equals B and Q is less than R and ZR is greater than 100, then set SIGNAL equal to FALSE, do nothing with ALERT, and go to ABLE. If the conditions do not satisy rule 1, then rule 2 is tried, etc. If none of the rules apply, then the ELSE rule is invoked, and control transfers to ERROR.

Table 1 shows a sample J6 program [NIME70]. It is a procedure called INTERGUIDE, which shows how nine typical missile guidance equations are written in SPL. The program is identified by lines 1-4. Lines 5-18 show data declarations. All data are declared as fixed-point with a 12 binary-place precision; initial values are set within the standard declarations. The "overlay" declaration (line 10) equates a list of vector elements to a vector. The mode declaration (line 9) causes all subsequent undeclared variables to be declared fixed-point 12 binary precision. Thus, JP, SP, QP, J, S, Q, etc. are all fixed point 12 binary precision.

Lines 20-25 declare a CLOSE subroutine which is called from lines 27 and 36. Line 26 begins a loop that continues until TIME exceeds MAXT. At that time, the statement following line 40 is executed. The built-in functions LOG (line 21), ATAN (line 30), COS (line 38), and SIN (line 39) are preceded by a dot. Lines 28-29 show non-scalar (i.e. matrix) assignments and lines 38-39 show multiple assignments. Line 42 represents the exit from the program, and line 44 denotes the end of the program.

```
1       START  ''BEGINNING OF PROGRAM''
2       .INTERGUIDE PROC(=SINX,COSX,SINY,COSY)
3            ''SAMPLE SET OF GUIDANCE EQUATIONS CODED IN SPL.  THE
4            PROCEDURE NAME IS INTERGUIDE.''
5       DECLARE   FIXED 12, TIME = 0, MAXT = 10000, TAU, T, L,
6                 JSQP(3),
7          VEX, DG(3), XYZ(3), MAX = 1.3659, CETA,
8          V, KGN, DPSIT(3,3) = ( 8(0), 1 )
9       MODE FIXED 12
10      OVERLAY JSQP = JP,SP,QP
11      DISPLAY.  DECLARE FILE DISPLAY2
12         DEVICE = CRT2
13         ERROR = ERR2
14      DISPLAY2.  DECLARE TEXT,K1 4 = 'PHI=', PHI 12 FIXED 4
15      SIGNAL.  DECLARE FILE SIGNAL1
16         DEVICE = THRUST'CONTROL
17         ERROR = ERR1
18      DECLARE BOOLEAN,SIGNAL1
19      ''CLOSE PROCEDURE CALC DEFINED NEXT''
20      CLOSE CALC
21           L = .LOG(TAU/(TAU - T))                    ''EQUATION 1''
22           JP = TAU * L - T                           ''EQUATION 2''
23           SP = JP - T * L                            ''EQUATION 3''
24           QP = (T ** 2)/2 + TAU * SP                 ''EQUATION 4''
25      EXIT
26      WHILE TIME LS MAXT   ''BEGIN EXECUTABLE STATEMENTS IN LOOP''
27           GOTO CALC
28           J,S,Q = VEX * (JP,SP,QP)                   ''EQUATION 5''
29           X4P,Y4P,Z4P = DG * XYZ                     ''EQUATION 6''
30           PSI = .ATAN(X4P/Y4P)                       ''EQUATION 7''
31           IF PSI GR MAX
32              THEN WRITE DISPLAY
33              ELSE SIGNAL1 = ON
34                   WRITE SIGNAL1
35           END      ''END IF''
36      GOTO CALC
37      PHIT = PSI + CETA * V * T - S + KGN * T ** 2  ''EQUATION 8''
38      DPSIT(0,0) = DPSIT(1,1) = .COS(PSIT)          ''EQUATION 9''
39      DPSIT(0,1) = -DPSIT(1,0) = .SIN(PSIT)         ''EQUATION 9''
40      END     ''END OF WHILE LOOP''
41           ''. . . REST OF PROGRAM NOT SHOWN''
42      EXIT
43      TERM ()  ''END OF PROGRAM''
44      .STOP
```

Table 1. Sample J6/SPL Program called INTERGUIDE.

9.3 J6/SPL Compilers.

All the J6/SPL compilers were built by SDC using SPLIT -- SPL Implementation Tool. SPLIT was an SDC compiler-building technique that used a specialized language and automated meta-compiler tool to reduce the cost of building SPL compilers. The first SPLIT compiler was a cross-compiler built by SDC in 1969 for the UNIVAC 1824 computer. This cross-compiler as well as the SPLIT system was hosted on the CDC 7600 computer. SDC claimed that they built the UNIVAC 1824 SPL compiler using SPLIT in less than six months and at less than half the cost of an equivalent compiler built using conventional techniques [NIME70].

A SPLIT-built SPL compiler had three logical passes: 1) a syntax analyzer, which parsed SPL statements into statement trees and declared data into dictionary trees, 2) a semantics analyzer, which performed scaling, mixed mode conversions, and reduction in complexity (tree pruning) on these two trees, and 3) a code generator, which produced assembly language output [NIME70]. The first two compiler passes constituted the typical compiler front end and were target-machine independent. As with other JOVIAL compilers, retarteting only required writing a new code generator.

The first two passes were written using the SPLIT language. The SPLIT language was ideally suited to describing the syntax of formal languages. Use of the SPLIT language and meta-compiler freed the compiler writer from dealing with low level compiler-building details such as writing the lexical analyzer and building the symbol table [NIME70]. Using SPLIT, the syntax of SPL

could be described in only 528 lines. By comparison, the syntax analyzer of the IBM 360 J3 compiler, required 8192 lines of JOVIAL [NIME70].

The code generator pass was written in a descriptive graphical language that allowed the compiler writer to "see" the relationships between the operators and operands of each statement tree. This made the production of localized optimum code relatively easy. This was borne out by a comparison of the code produced by the UNIVAC 1824 SPLIT compiler for the Titan IIIC missile guidance program with equivalent hand-written machine language code produced using every known trick to minimize code. This comparison showed that the compiler produced only 10% more machine code than the hand-coded version.

SDC built two other SPLIT cross-compilers for SAMSO: one for the Honeywell 516 computer used for the Space Precision Attitude Reference System (SPARS), a star sensor for orbital guidance, and one for the RCA SCP 234 computer used for the Defense Meteorological Satellite Program (DMSP), a DOD weather satellite [BROW80]. Both compilers were built over a period of six months by a group of 4 people at SDC. Later in 1971, Logicon, Inc. designed a direct execution HOL computer, which was never built, to execute SPL/MK II code directly without a compiler or interpreter [NIEL72].

Chapter 10

JOVIAL COMPILERS AT IBM

JOVIAL was stongly identified as an SDC language: SDC had developed JOVIAL and adopted it as a corporate language. Even so, SDC's hold over JOVIAL began to diminish as JOVIAL began to be used outside of SDC. At first, other companies used the J2 and J3 compilers that SDC freely distributed. Eventually, companies (mainly IBM, CSC, and Abacus) began developing their own JOVIAL compilers. This introduced additional variation into the language, which SDC, of course, could not control. The next three chapters cover the development and use of JOVIAL compilers by these companies.

10.1 Use of 7090 J2 Compiler.

As mentioned earlier, the J2 compiler for the IBM 7090 was not integrated with the J3 common front end (chapter 5). This coupled with the popularity of the IBM 7090 computer resulted in the use of J2 by other organizations and, eventually, the production of at least one other J2 compiler. This compiler was developed for the Federal Aviation Agency's (FAA) National Airspace System (NAS), which is the air traffic control system used by controllers today.

The use of JOVIAL for the NAS system began in 1962 when the FAA established the National Aviation Facilities Experimental Center (NAFEC) near Atlantic City, NJ, to study the feasibility of automating the existing air traffic control system. NAFEC used an

IBM 7090 computer connected to various peripheral devices to provide a test bed for the proposed automated system.

Wishing to avoid programming the 7090 in assembly language, NAFEC began looking for a suitable HOL when they heard of JOVIAL, a then new language developed by SDC [KETC81]. In April 1962, NAFEC received a complete set of 7090 J2 documentation, which confirmed that JOVIAL, alone among the HOLs then available, had the needed features for describing at the bit level the many I/O message formats needed to interface with the peripherial devices NAFEC was using. Also SDC had a working compiler. A month later NAFEC received a deck of cards containing an the 7090 J2 compiler in object format from Jules Schwartz at SDC [KETC81].

From May 1962 until March of 1963, the FAA and IBM/Federal Systems Division (IBM/FSD), which was the NAFEC support contractor, used the J2 compiler to develop experimental air traffic control programs for the proposed system [KETC81]. This effort was very successful. The use of JOVIAL allowed code to be written both by FAA air traffic controllers with very little programming experience and by IBM people with little or no 7090 experience.

In November 1963 (figure 4), with the feasibility of an automated air traffic control system assured, the FAA issued an RFP for the NAS. IBM was one of the potential contractors for NAS. In preparing their proposal, the question of the language to be used came up. PL/I (called NPL at that time) and JOVIAL were the languages being considered. In December 1963, Dave Ketchum, an IBM language expert, strongly recommended the use of J2 because

of previous FAA experience with JOVIAL at NAFEC and because PL/I did not have a firm definition at the time [KETC81]. IBM proposed the use of JOVIAL. In June 1964, the NAS contract was awarded to IBM.

10.2 The IBM 9020 Compiler.

IBM's design for the NAS system was based on the new IBM 9020 computer. The 9020 was a multiprocessor version of IBM's then new System 360. The first 9020, the 9020A, was not delivered until the end of 1965 and consisted of three input/output control elements and four central processing units (CPUs), each of which was essentially a 360/50 computer with I/O and memory removed. Core memory was in separate elements equally accessible to all active CPUs and I/O control elements. The 9020A was later replaced by the 9020D based on the 360/65 CPU [KETC81].

Because of the long lead time to build the compiler, IBM contracted Computer Usage Company, Inc. (CUC) to begin building a J2 compiler for the IBM 9020 in January 1964. This compiler was to have the same capabilities as the SDC J2 7090 compiler. Until the 9020 became available, all compiler development was done on the IBM 7090 using an IBM built assembler and simulator for the IBM 360, which had the same machine instruction set as the 9020 [KETC81]. As the compiler grew in size, the simulation time on the 7090 grew too slow, and CUC began using the IBM 7030 super computer (nick-named STRETCH) for simulation. Even the STRETCH was slow: the J2 compiler required about an hour to compile a 100-statement program under simulation [KETC81]!

The compiler was delivered to IBM in September 1965 (figure 5). It translated J2 into 360 Basic Assembler Language (BAL), which was then assembled into machine code. The compiler was not constructed using SDC's approach with a separate front end and code generator. Instead, the compiler was divided into four phases, each of which translated its part of the JOVIAL source program directly into BAL. This monolithic design coupled with the fact that the compiler itself was written in 360 assembly language made the compiler virtually impossible to retarget or rehost [KETC81].

After delivery of the compiler, IBM made several enhancements. The compiler was optimized by using the special BAL byte-manipulation instructions for character manipulation, which CUC had not used. Later, major features such as pointers were added. After a few years, the compiler recognized a version of JOVIAL that was very different from the original version of J2 and that was referred to as IBM 9020 JOVIAL [IBM68]. Thus, another variation of J2 was introduced.

10.3 The IBM 360 Compiler for AWACS.

Later in 1969, IBM/FSD in Owego, N.Y., developed a J3 compiler hosted on the IBM 360 and targeted to the 360 and to the CC-1, which was a military avionics computer that IBM had developed for the Boeing Airborne Warning and Control System (AWACS) radar plane (a modified Boeing 707). The compiler was written in PL/I and complete in July 1973. Unfortunately, Boeing used another compiler for AWACS [KETC81].

Chapter 11

JOVIAL/J3 COMPILERS AT CSC

The use of JOVIAL outside of SDC was greatly accelerated when Computer Sciences Corporation (CSC), El Segundo, CA, began building JOVIAL compilers. The CSC JOVIAL compilers were the first commercially built JOVIAL compilers. The fact that JOVIAL compilers were being commercially built was a measure of JOVIAL's success, but it introduced yet another source of diversity into the language.

11.1 CSC JX.2 Compiler.

The first CSC JOVIAL compiler was built in 1964 (figure 4) for the UNIVAC 1107 computer. It implemented a subset of J3 similar in scope to the SDC JX.2 compiler. CSC developed this extremely fast, one-pass compiler as a systems programming tool. The compiler itself was implemented in the same J3 subset that it compiled and later formed the basis of CSC's SYMPL language and compiler (see below).

11.2 GENESIS/SYMPL Compilers.

CSC did not build their next JOVIAL compiler until 1968 (figure 6). At that time CSC had contracts to develop ALGOL, JOVIAL and FORTRAN compilers and was looking for techniques to reduce the amount of work required to develop new compilers. This led to the

development of the GENESIS/SYMPL multi-language, machine-independent compilation techniques and design philosophies.

The GENESIS/SYMPL compilers were written in the SYMPL language and developed using the GENESIS tool. GENESIS was a proprietary parser generator (a compiler compiler) system that produced syntax tables of the desired language from a formal grammar of the language expressed in Backus-Naur Form (BNF). These tables were then used by a canonical syntax-parsing program to provide a compiler front end for that language.

SYMPL was largely conceived by Terry Dunbar at CSC to provide a systems programming language for writing JOVIAL, ALGOL, and FORTRAN compilers [DUNB81]. SYMPL was based on ALGOL, but used concepts from both JOVIAL and FORTRAN. The first SYMPL compiler was derived from the UNIVAC 1107 J3 subset compiler by changing the front end to accept SYMPL, rewriting the compiler in SYMPL, and then self-compiling it [DUNB81].

Most of the development work for the GENESIS/SYMPL compilers took place on the UNIVAC 1107 and, later, the UNIVAC 1108, using the SYMPL compiler and the GENESIS syntax table generator. By writing all its compilers in SYMPL, CSC could re-use much of the code from one compiler to build the next.

CSC's first J3 compiler was an IBM 360 compiler developed for IBM/FSD in Gaithersburg, MD, to use on the Manned Orbiting Laboratory (MOL) program. Preliminary meetings between CSC and IBM began in December 1966, construction began in April 1967, and delivery of the first version was in April 1968 (figure 6).

Development of the compiler began on the UNIVAC 1107. When the code generator for the 360 was complete, the compiler was then self-compiled to the IBM 360, where final integration was completed. Work on an optimizer for this compiler was stopped when the MOL program was cancelled in January 1970. Only a preliminary version of the optimizer was delivered [DUNB81].

Later, the front end from this compiler was combined with a new code generator to produce a J3 compiler for the CDC 6000. In developing this compiler, CSC was able to complete work on the sophistocated machine-independent global optimizer that was begun for the 360 compiler. The CDC 6000 compiler was the first globally optimizing JOVIAL compiler ever built [CSCB79].

11.3 Other J3 Compilers.

About 1970, CSC developed two other GENESIS/SYMPL J3 compilers. One was for the UNIVAC CP-642. It had added capabilities for calling CMS-2 procedures and performing double-precision real arithmetic calculations. It was built under a Navy contract but was never finished [CSCB79].

The other compiler was built for the Hollandse Signqalapparaten (HSA) computer. The HSA was the main shipboard computer for a naval tactical defense system called DAISY that Computer Sciences International (now CSC International Division), a combined CSC and Phillips company, built for the Royal Dutch Navy. CSC developed the compiler by adding a code generator for the HSA to the GENESIS/SYMPL J3 IBM 360 compiler [ZEIT81].

Also about 1970, CSC designed and implemented a JOVIAL/J3 compiler for the Hughes Aircraft Company H4400 multiprocessor [CSCB79]. It used an original design not related to the previous GENESIS/SYMPL compilers. It was implemented in J3 rather than SYMPL and used an ad hoc parser instead of GENESIS. This compiler later served as the basis for CSC's J73/I compilers (chapter 16) and SEA's J73 compilers (chapter 20).

11.4 J3 JOCIT Compiler.

In 1973, CSC built its last GENESIS/SYMPL J3 compiler for the Honeywell Information Systems (HIS) 635 (orginally the General Electric 635). This compiler was developed for the Air Force Rome Air Development Center (RADC) as part of the JOVIAL Compiler Implementation Tool (JOCIT) contract.

JOCIT was originally intended as a compiler-building system for producing JOVIAL compilers for a modernized version of J3 being produced by an RADC committee (see chapter 14) [CSCB78]. In December 1971, when CSC submitted their bid for the JOCIT contract, the RADC committee had been working on J3 for over two years with little change to the language except to add a comprehensive I/O capability.

CSC had intended to produce the JOCIT system by modifying their GENESIS/SYMPL system to produce compilers for the modernized version of J3. But when the JOCIT contract was awarded in July 1972, the revised language no longer resembled J3 but was a

completely different language called J73 (chapter 14).

CSC spent the first six months of the JOCIT contract on two tasks. First, they began work on a compiler for a subset of J73, called J73/S [DUNB81]. The J73/S compiler was written in SYMPL and was later used by CSC to build the first J73/I compiler (chapter 16). Second, they assessed the effects that J73 would have on the original JOCIT proposal.

By January of 1973, it became clear that the effort to build a JOCIT system for the J73 language would require more resources than expected. CSC informed RADC what the impact was going to be and requested contract direction. At this time, the World Wide Military Command and Control System (WWMCCS) needed a J3 compiler for the HIS 635. In order to salvage the effort, RADC contract monitors Dick Nelson and Sam DiNitto proposed to Air Force commanders that the JOCIT contract be modified to produce a J3 JOCIT compiler for WWMCCS [DUNB81].

JOCIT was built using the GENESIS/SYMPL system [DUNB75]. The CDC 6000 J3 front end was combined with a new code generator for the HIS 600/6000 computer family. The effort was completed by moving the SYMPL and JOCIT compilers to the HIS 635 using a SYMPL cross-compiler for the HIS 635, which CSC had just completed for another project.

The JOCIT compiler contributed to the body of compiler building knowledge embodied in the GENESIS/SYMPL system in the areas of optimization and code generation [DUNB75]. The J3 JOCIT compiler

was significantly less host and target dependent than any of the preceeding GENESIS/SYMPL J3 compilers. It produced high performance object programs as a result of the extensive global optimization techniques used. It later served as a prototype compiler that reduced the cost of creating a J3 compiler for the Rolm 1666 computer. This compiler was developed for the Airborne Command Post by SEA, which still maintains both compilers [DUNB81].

Extensive testing and use proved the JOCIT compiler to be the most accurate, the most efficient (with respect to global optimization) and overall the easiest to retarget of any JOVIAL J3 compiler ever built [DINI81]. Because of its success, demands were made for the Air Force to secure full rights to the JOCIT system. This effort resulted in a 99-year lease for the rights to the GENESIS meta-compiler giving the Air Force complete control over JOCIT.

Although JOCIT was originally intended for producing several other compilers, the HIS compiler was the last JOCIT compiler produced by CSC. The JOCIT compiler was later used for a research project to build a table driven code generator. The compiler produced predictably inefficient code but was easily retargetable. Then in July 1976, CSC was awarded a contract to incorporate state-of-the-art optimization techniques into JOCIT and add double precision floating point and ASCII data handling to the HIS 635 JOCIT compiler [DEVI78].

Chapter 12

JOVIAL COMPILERS AT ABACUS

Abacus Programming Corporation, Los Angeles, CA, which was founded by Cal Jackson, who developed the Q-7 code generators at SDC, added to the diversity of JOVIAL dialects not only by building their own line of J3 compilers but also by developing a new version of JOVIAL called J70.

12.1 J3 Compilers for Hughes.

In 1966, Abacus began development of a J3 compiler for the Hughes H3118 computer. This compiler was used by Hughes Aircraft Company, Fullerton, CA, to program the command and control subsystem for the NATO Air Defense Ground Environment (NADGE). The use of J3 by Hughes for NADGE and other air defense ground systems was probably a result of the early use of JOVIAL for SAGE and SACCS by SDC.

Abacus developed the H3118 compiler by first coding it in a subset of J3 and then hand-compiling it to the H3118. The final version was then self-compiled. Besides the J3 compiler, Abacus also delivered a separate COMPOOL compiler as well as an assembler, utility routines, a loader, debugging tools, and a Quality Assurance System to Hughes [ABAC80].

Later Hughes rehosted the H3118 compiler to the Hughes H4118 and H5118 computers, which had an 18-bit architecture similar to the H3118. These compilers were used by Hughes to build similar air defense systems for other countries. Hughes also began a project to retarget the H5118 compiler to produce a cross-compiler for the 16-bit HMP-1116 multiprocessor, but Abacus was contracted to complete this effort. Later Abacus added significant new language features to the HMP-1116 compiler [ABAC80].

These Hughes compilers required about 11-12 passes per compilation. Major extensions and modifications were made to these compilers so that the language they implemented was no longer compatible with the original J3.

12.2 JOVIAL/J70.

About 1970 (figure 6), Abacus designed a new version of JOVIAL called JOVIAL/J70. J70 was an improved version of J3 with hefty extensions. It had features from FORTRAN, ALGOL, and PL/I [DINI81]. Abacus built a J70 compiler for the Digital Equipment Corporation DEC-10 computer. This compiler was developed by first specifying a J3 subset that was compatible with J70. This J3/J70 dialect was then used to implement the J70 compiler. Abacus also developed a JOVIAL/70 symbolic debugger [ABAC80].

The J70 DEC-10 compiler was built for the UCLA Center for Computer Based Studies, which was doing government-funded behaviorial research. J70 was chosen because some of the

researchers at the Center were ex-SDC employees and had existing code in J3 they planned to use [UPSH81].

The J70 compiler was later used by TRW for programming the Software Design and Verification System (SDVS) until the CSC J73/I compiler became available (chapter 16). Abacus specified a subset of J70 that was compatible with J73/I and wrote manuals for TRW describing how to write J70 programs for eventual conversion to J73/I. This subset allowed TRW to write the inital components of SDVS using the Abacus J70 compiler and later compile the same code with the J73/I compiler without extensive modification [ABAC80].

The JOVIAL DEFINE capability was used to bridge syntactical differences between the two JOVIALs. For example, the DEFINE declarations DEFINE FLOAT "F"; DEFINE INTEGER "S 35"; DEFINE FULLWORD "U 36" [BURL77] made it possible to declare data using the J/70 predefined types FLOAT, INTEGER, and FULLWORD and compile the code on the J73/I compiler.

PART II

THE LATER PERIOD

Chapter 13

STANDARDIZATION OF J3

In 1967, the Air Force issued a J3 standard and initiated efforts to gain national standardization for JOVIAL. The effort to gain national standardization eventually led to the creation of J73, an entirely new version of JOVIAL. This chapter describes these standardization efforts. The creation of J73 is the subject of the next chapter.

13.1 Air Force JOVIAL/J3 Standardization.

SDC's attempt in August 1965 (section 7.4) to require all JOVIAL compilers to implement at least the Basic JOVIAL subset was the first major effort to standardize JOVIAL compilers. But this effort applied only to SDC and only to SDC's compilers; nothing was done to address standardizing the language itself.

By this time, the Air Force recognized that considerable cost-savings could be achieved if they standardized JOVIAL. These savings were expected as a result of being able to transfer programmers from one application to another rather than programs between computers. JOVIAL applications tended to be large command-and-control systems; hence, transfer between computers was meaningless in most instances. Transferring programmers between applications, however, was deemed extremely important to reduce Air Force training costs for system maintainers [FLEI72].

As early as Fall 1961, the subject of a standard military programming language had been raised [IDA61]. A year later, SDC published a report [SDC62D] that recommended that a standard programming language be adopted for military systems [SHAW62B]. The Air Force and the Navy were considering JOVIAL as a standard language at that time, but not until June 1967 (figure 5) did the Air Force issue AFM 100-24, the JOVIAL (J3) Language Specification [AFM67]. AFM 100-24 was identical to SDC's J3 manual [PERS66B] and established J3 as a standard programming language for Air Force command and control applications. In 1977, a military standard for J3 was issued, MIL-STD-1588 [DODM77].

The push for standardization by the Air Force was the result of a DOD Standardization Program in 1965. This program assigned to the Air Force the responsiblity for Information Processing Standards for Computers and issued DOD Instruction 4120.9 which required the services "to make optimum use of the facilities of industry groups in the development of standardization documents having a present or potential DOD use" [SLAV79].

13.2 USASI Standarization of JOVIAL.

In addition to issuing its own JOVIAL standard, the Air Force also asked USASI (United States of America Standards Institute) to evaluate JOVIAL for possible standardization in 1967. Until then, USASI, which later became the American National Standards Institute (ANSI), had not paid significant attention to

JOVIAL. Only standards for FORTRAN, COBOL, ALGOL, and APT had been developed.

The USASI committee agreed that the area of command and control applications needed a standard language and that JOVIAL was the best candidate. But the committee felt that JOVIAL had not kept pace with software and hardware developments and recommended that substantial changes be made to the language [DUNB81]. Its limited user base and exclusive use on DOD programs were also cited as reasons for not establishing a USASI JOVIAL standard at the time [SLAV79].

In 1968, the Air Force funded the JOVIAL Evaluation Project to look at improving JOVIAL for future USASI standardization. Information obtained from this Project was used to develop an approach for updating J3. A report [OBRI68] was issued giving criteria for evaluating which features of the J3 standard (AFM 100-24) should be retained, deleted, or modified. This report led to establishment of the committee to modernize JOVIAL and the creation of the J73 language.

Chapter 14

JOVIAL/J73

J73 came about as a result of efforts by the Air Force to produce an acceptable version of JOVIAL for USASI language standardization. J73 served as the basis for all later versions of JOVIAL including J73B, the current standard version of JOVIAL.

14.1 The Committee to Modernize JOVIAL.

In early 1969 (figure 6), a group was formed by the Air Force Air Staff under Leo Berger to produce an updated version of JOVIAL [SCHW78]. This group directed RADC to convene a committee of industry and Air Force representatives to develop a new specification for J3 that incorporated the language change recommendations made by the JOVIAL Evaluation Project. Maj. Leon Kurtz was the first chairman; Richard Nelson took over in 1970 [DINI81].

The RADC committee worked for over two years with little progress; the only major language change was the addition of a comprehensive I/O capability [DUNB81]. Finally, in February 1972, a technical subcommittee consisting of Lynn Shirley, who was a member of Schwartz' CUSS project, Terry Dunbar, who developed JOVIAL compilers at CSC, Millard Perstein, who wrote JOVIAL manuals at SDC, and John Bengston of Abacus was funded by RADC to come up with a completed J3 language revision [DUNB81]. Cal Jackson of Abacus and the full RADC committee also contributed to the revised language.

The subcommittee was strongly influenced by Abacus' work with J70 and by CSC's experience with SYMPL [DUNB81]. One of the first changes they made was to delete all the I/O capability that the full committee had designed. Then they made substantial changes to the punctuation, data definition, and execution control structures [DUNB81]. The result was a language that no longer resembled J3 with improvements but became, instead, a new language. The technical subcommittee produced their report about May 1972. Millard Perstein wrote the final language specification [JOVI73], which was ratified by the full committee in November 1972 and published in January 1973 --hence, the name JOVIAL J73 (figure 7).

14.2 J73 Language Design.

J73 was developed to provide a common, powerful HOL suitable for a wide range of Air Force applications. It was designed to be relatively machine-independent, with features for logical operations and symbol manipulation, as well as numerical computation. Most numeric features of J73 were the same as J3. The COMPOOL and DEFINE capabilities were the same.

Important differences from J3 were removal of all I/O statements from the language and removal of the MODE specifier, requiring the user to declare all data items before he uses them. The reason for eliminating I/O was that many aspects of I/O are quite system-dependent. J73 was designed so that with the use of system routines, JOVIAL programs could be made to interface with many I/O schemes [SHIR73B].

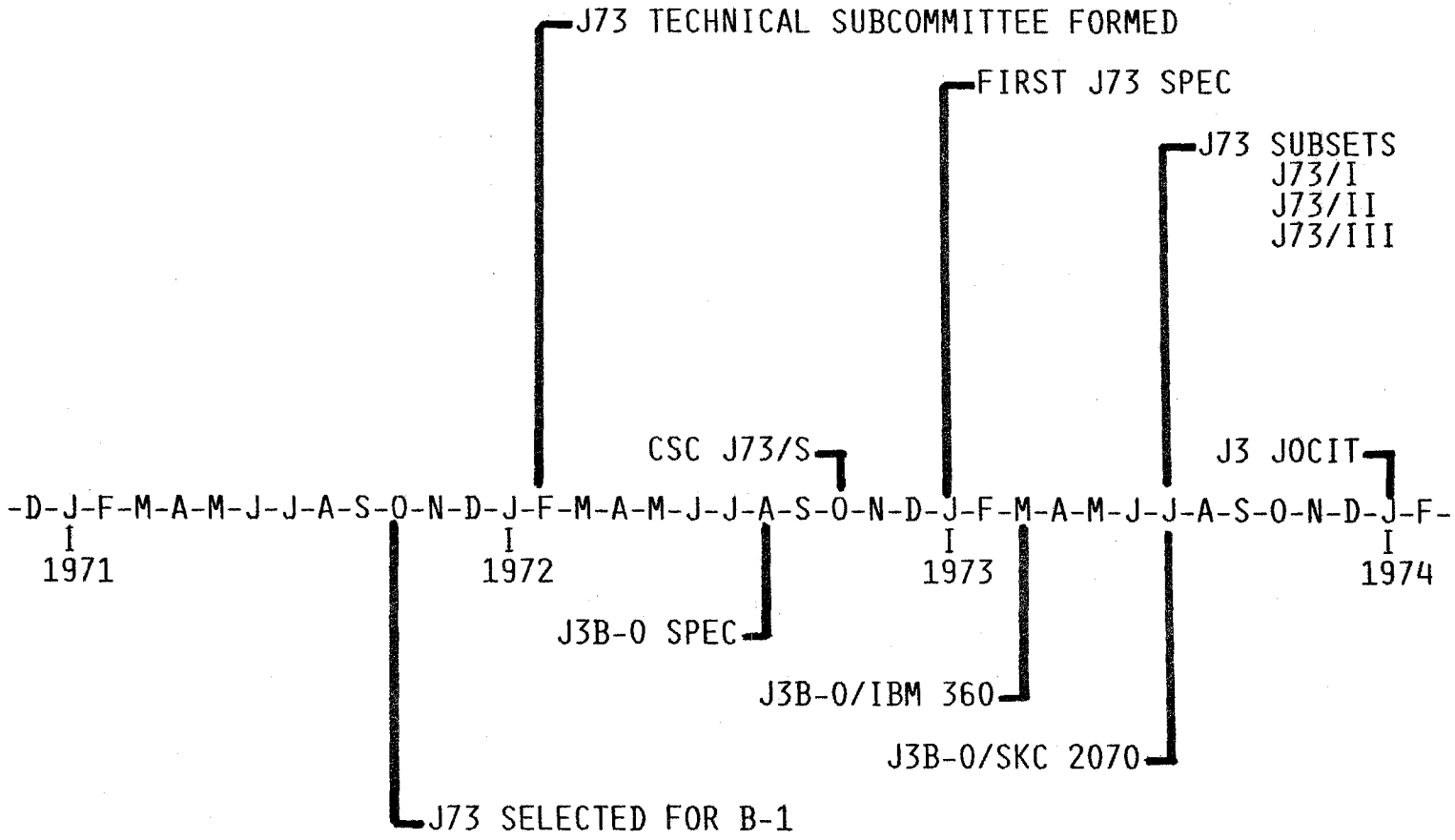


Figure 7. JOVIAL J73 and J3B (1971 -- 1974).

Extensions to J3 included expanded table declaration facilities, new compiler directives for conditional compilation and user directed optimization, recursive procedures, inclusion of elements for dynamically controlled storage allocation during execution, and format conversion aides [FLEI74].

Although J73 was designed for programming avionics systems, compilers, executives, data management systems, scientific programming, command and control, tactical systems, and real-time control [TRAI77A], the area of avionics influenced J73 the most. This was due to the B-1 bomber program.

The Air Force Aeronautical Systems Division (ASD) required the avionics flight software for the B-1 to be done in an HOL. Since ASD personnel were active members of the RADC committee and were thus aware of J73's potential benefits, they selected J73 as the language for the B-1 in October 1971 (figure 7). Being potentially the first users of J73, ASD made the largest contributions to the language. In the spring of 1972, the B-1 program office at ASD performed a survey of the avionics industry asking for desirable and necessary language features for programming avionics. Virtually all of the survey suggestions were incorporated into J73.

Unfortunately, the suggestions had to be coordinated with the definition of the proposed language, which the RADC technical subcommittee had begun only in February 1972. By June, it looked as though the final J73 specification would not be completed in time for use by B-1. So, an effort was begun to design a substitute

language that would satisfy the minimum needs of the B-1 program until J73 became available. That language was called J3B and is discussed in the next chapter.

The schedule pressures of the B-1 program had another important effect on J73. Initially, the technical subcommittee had included all the constructs that might be necessary for the intended applications in the J73 specification later intending to simplify and condense the language. But, due to the rush to produce a language specification in time for use on B-1, the subcommittee did not have time to refine the language as intended. The result was that J73 was rather large.

Because J73 was so large, three subsets were specified to insure that at least one version of J73 could be implemented on smaller computers. Initially, the Air Force only funded development of a compiler for the smallest subset, J73/I, rather than the whole J73 language (see chapter 16). In the end, a compiler for the full J73 language was never built.

Chapter 15

JOVIAL/J3B

J3B was created as a temporary language for the B-1 program until J73 was ready. But once created, it had a life of its own, the original version being revised two times. The existence of J3B also had unexpected and far-reaching effects later on when the Air Force wanted to standardize on J73/I (see chapter 18).

15.1 J3B-0.

When the Air Force began development of the Rockwell B-1 airplane, both the B-1 system program office (SPO) at ASD and Boeing Aerospace Company, the avionics subcontractor for the B-1, were aware of the problems of writing operational flight programs in assembly language and elected to use a HOL instead. The B-1 SPO was a member of the RADC committee to revise J3 and, in October 1971 [TRAI77A] selected the language that soon became J73 for the development of the B-1 avionics systems.

In June 1972, when it appeared that J73 would not be ready in time, Boeing, RADC, Abacus, and the B-1 SPO, all members of the J73 committee, began an effort to design a bare-bones version of JOVIAL, called J3B, to satisfy the needs of the B-1 until J73 was ready [TRAI77A]. To achieve a low risk, this language was designed as close as possible to J5.3 for which the government-owned IBM 360 compiler already existed (chapter 8). Wherever possible, the

language was made to look like what was known of J73 at the time.

In August 1972, Boeing issued an RFP for the development of two JOVIAL/J3B compilers and in September 1972, awarded the contract to SofTech, Inc. [HOLD77]. In refining the design for the J3B language the need for staying close to J5 was eliminated because SofTech did not plan to use the J5 compiler as a baseline. Also, the J73 technical subcommittee report was ready when the J3B specification was completed by Boeing and SofTech in October 1972. Thus, J3B was more of a derivative of J73 than J3 [TRAI77A]. This original version of J3B was termed J3B extension 0, or J3B-0.

J3B-0 was a strongly-typed language that supported a simple type definition facility, separate compilations, and a COMPOOL for defining all data shared between compilation units. The design philosophy for J3B-0 was to take the absolute minimum set of features known to be needed for avionics programming and to add new features only when needed. Thus, J3B-0 lacked features such as nested scopes and exception handling facilities. The principle features of the J3B-0 language were as follows (compare with appendices C and D):

1. Separate compilation of procedures, functions, and COMPOOLS
2. Data Types. Signed and unsigned integer, single and double precision floating point, bit strings, and character strings.
3. Data Structures. Simple ITEMs, 1 and 2 dimensional ARRAYs, TABLEs, and BLOCKs.

4. Control Structures. Assignment statement, GOTO, SWITCH, IF-THEN-ELSE, FOR and WHILE loops, PROCEDURE and FUNCTION calls.
5. Compile time evaluation of constant formulas.
6. Conditional compilation.
7. DEFINE capability.

SofTech delivered two compilers to Boeing, one for the host computer, the IBM 360, and one for the B-1 Offensive Flight Software (OFS) avionics computer, the Singer SKC 2070. Both were hosted on the IBM 360. The compiler for the IBM 360 compiler was delivered in March 1973 (figure 7) to allow some coding to begin before the cross-compiler for the SKC 2070 was ready. A month later, SofTech delivered an initial cross-compiler, which generated correct but not fully optimized code for the SKC 2070 [HOLD77].

The final SKC 2070 J3B compiler was delivered in July 1973 [SOFT74]. This compiler was capable of generating extremely good code as a result of considerable effort in the design of register usage conventions and machine dependent optimizations. This was shown during acceptance testing, which Boeing completed in December 1973 [HOLD77]. The compiler efficiency was measured by comparing typical compiler output with functionally equivalent, hand-coded assembly language programs for a variety of OFS functions. The average memory expansion of object code produced by the compiler was within 15% of the equivalent assembly language code. In most cases, this efficiency was sufficient to satisfy overall system

constraints. Critical program areas were replaced with assembly language programs [HOLD77].

There is an interesting parallel to the development of the JOCIT compiler. As mentioned before, J3B was developed because it looked like a J73 compiler would not be ready in time. As it turned out, CSC had their J73/S subset compiler, which compiled the equivalent of the J3B language as defined at that time, up and running six months before the SofTech J3B compiler [DUNB81]. So, J3B was, in many ways, unnecessary.

15.2 J3B-1.

In August 1974 (figure 8), an effort was begun to specify enhancements to J3B-0 to support development of the B-1 Avionics Defensive Subsystem, which included both the Defensive Management Subsystem and the Radio Frequency/Electronics Countermeasures Subsystem (RFS/ECMS). The requirements of this subsystem together with selection of the Litton LC-4516D (initial compiler development began with the LC-3516 and the 4516B [SOFT80]) for the avionics computer, provided motivation for the enhancements.

The resulting language was designated as J3B-1. J3B-1 was an upward compatible extension (superset) to J3B-0 with the following additional features [HOLD77]:

1. Nested blocks.
2. Preprocessed COMPOOL facility.
3. Fixed point and pointer data types added.
4. Typed tables (pointer-based).

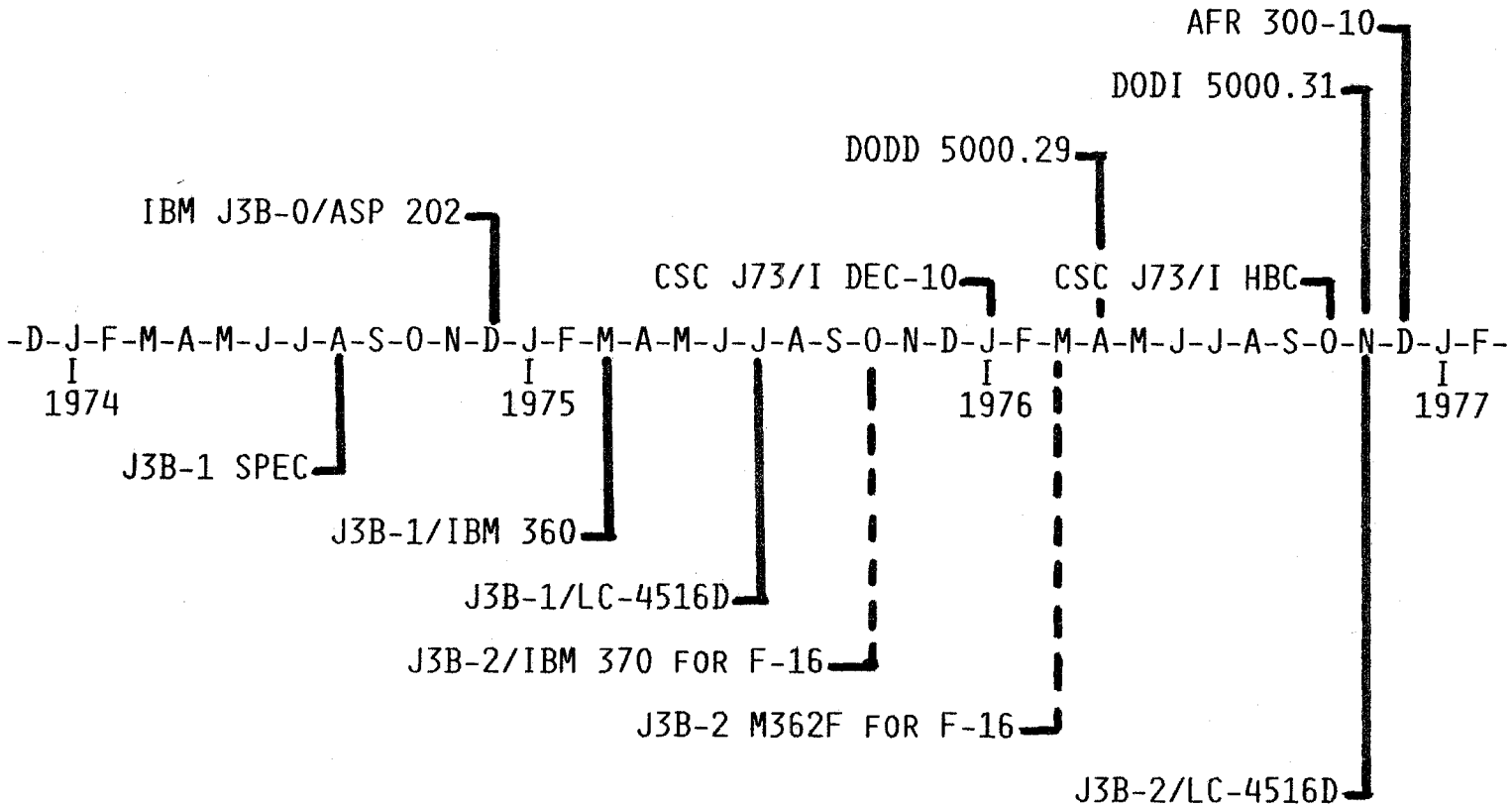


Figure 8. J3B and J73/I (1974 -- 1977).

5. Parameterized DEFINE capability.
6. IN-LINE procedure attribute.
7. Built-in procedures and functions to access machine operations.

In November 1974, Boeing awarded SofTech a contract for development of two J3B-1 compilers: one for the IBM 360 host and one for the LC-4516D. The effort required about 10 people working about 1 year and involved enhancing the language, building a new code generator for the LC-4516D, and retrofitting the existing J3B-0 compiler [BROW80]. In March 1975, SofTech delivered the IBM 360 compiler, in April, an unoptimized LC-4516D cross-compiler, and in July the final optimized LC-4516D compiler.

Acceptance testing by Boeing was completed in September 1975. The compiler was then installed at the AIL division of Cutler-Hammer (developer of the RFS/ECM software) where it was used for operational program development and an efficiency performance evaluation [HOLD77].

15.3 J3B-2.

Meanwhile, in the spring of 1975, General Dynamics chose J3B for implementating the Fire Control subsystem on the F-16 fighter aircraft. J3B-1 was deemed insufficient and SofTech was contracted to upgrade the language to J3B-2 and to deliver two J3B-2 compilers. J3B-2 was an upward compatible superset of J3B-1 with the following additional features [HOLD77]:

1. Reentrant procedures allowed.
2. Arrays with 3 dimensions and compiler allocated tables.
3. BIT and BYTE functional modifiers.
4. Global optimization capability.

SofTech delivered two J3B-2 compilers to General Dynamics, both hosted on the IBM 370: one targeted to the IBM 370 and the other to the 16-bit Delco Magic M362F-2, which was the F-16 avionics computer. At the same time, a significant new global optimization capability was added.

While this was happening, Boeing, AIL, and SofTech conducted an efficiency study of the LC-4516D compiler. Benchmark programs derived from RFS/ECMS flow charts were written in both J3B-1 and assembly language. The compiled programs were found to have a memory expansion of 58% and execution expansion of 72-106% over the hand-coded assembly language versions. This performance was unacceptable for the RFS/ECMS application [HOLD77].

Additional analysis predicted that a memory and time expansion on the order of 20% could be achieved with further compiler optimization [HOLD77]. So, in July 1976, Boeing awarded a subcontract to SofTech for enhancement of the LC-4516D and IBM 360 compilers to include the J3B-2 language extension, the global optimization capability, and re-design of the execution environment (run-time routines) [HOLD77].

In November 1976, SofTech delivered to Boeing the further optimized LC-4516D J3B-2 cross-compiler. The modifications

consisted of additional local optimization, incorporation of global optimization, redesign of the linkage and run-time environment, and addition of special functions to access machine-level operations.

After acceptance testing by Boeing, this compiler was subjected to performance testing at AIL. This analysis was completed in February 1977 and showed that, by using assembly language for time-critical sections, the LC-4516D compiler could produce code with a memory expansion of only 35% and an execution time expansion of 22%. Less than 23.4% of the total program was assembly language [HOLD77].

In October 1976, Boeing authorized a similar enhancement for a J3B-2 version of the SKC-2070 compiler, which SofTech delivered in April 1977 [HOLD77].

15.4 J3B Compiler Design.

By 1977 SofTech had built J3B-2 compilers for the IBM 370, SKC-2070, M362F, and LC-4516D compilers. The design of these compilers was based on SofTech's previously used compiler design methodology and automated tools, which were based on a compiler-writing language called ALGOL Extended for Design (AED).

The AED compiler-writing system consisted of an AED compiler, several automated compiler-building tools, and a library of machine independent procedures for I/O, free storage manipulation, and string processing. The isolation and parameterization of host-computer dependencies, and the use of the automated tools to generate portions of the syntax and semantics

analyzers, minimized the effort to rehost or retarget the AED compilers [SOFT74]. The compilers were implemented in the AED language and compiled using SofTech's AED compiler, which had multiple code generators for the desired target computers [SOFT74].

The J3B compilers each consisted of a conventional front end, intermediate language, and code generator. The front end was produced by combining a syntax analyzer, which was built by processing a syntatic description of J3B through an LALR (Lookahead Left-Right) parser generating tool, with a semantics analyzer, which was produced by processing a semantic description of J3B with SofTech's FSM automatic tool. Local optimizations in the code generator were also implemented using the FSM tool, which transformed optimizable machine code patterns into instructions that would replace these patterns in the generated target machine code with optimized code [SOFT74].

Use of the AED compiler-building tools allowed the J3B compilers to be written and debugged in minimum time -- the 48,000 lines J3B-0 compiler was produced with less than three man years of technical effort -- and yet generate efficient object code [SOFT74]. The J3B compilers required about 320K--384K bytes of main memory and compiled at 600 to 800 lines/min on an IBM 370/155 [PALM77].

15.5 IBM J3B-0 compiler and the SPS Program.

Although J3B was used primarily for avionics systems, it was also used by IBM/FSD in Westlake, CA, for the Simplified

Processing Station (SPS), which was a mobile ground receiver for strategic military satellite information. The SPS main computer was the Advanced Signal Processor (ASP-202), which consisted of several general-purpose processors connected to a central, high-speed array processor.

The J3B compiler for the ASP-202 was hosted on the IBM 360 and translated a version of J3B-0 [IBM76] into ASP-202 assembly language [SPSM76B]. It was written in PL/I using a compiler development methodology based on the Common Intermediate Language (CIL) [SPSM76A].

CIL was developed by IBM/FSD Owego, NY. CIL was typical of intermediate languages on other compilers and was designed to be compatible with the languages of interest to IBM, namely JOVIAL, PL/I, and FORTRAN, and with the target computers of interest to IBM, the IBM 360/370, the ASP-202, and the ML-1 (IBM 16-bit military computer).

The CIL-based J3B compiler was similiar in design to previous JOVIAL compilers and consisted of a conventional front end, which performed, syntactic and semantic analysis of J3B to produce the CIL Text and Dictionary files and a code generator, which translated the Text and Dictionary files into assembly language [SPSM76B]. The CIL Text consisted of quadruples (op code and three operands) representing executable source language statements. The Dictionary file (symbol table) contained attribute and storage allocation information required to establish addressability of data items [SPSM76A].

Like CSC's GENESIS, the J3B front end consisted of a general syntax parsing program that used a syntax table prepared from a BNF description of J3B by a separate LR(k) Grammar Analyzer developed at the University of Toronto.

An important feature of the compiler was the optional machine-independent optimization phase that optimized the CIL Text by removing common sub-expressions and moving loop invariant computations out of loops. The COMPOOL facility simply processed files containing source language; there was no provision for preprocessing data declarations into a formal COMPOOL [SPSM76B].

The J3B/ASP-202 cross-compiler was built by combining a front end for the J3B-0 language with a code generator for the ASP-202 that was derived from a code generator for the FORTRAN/ ML-1 compiler, which was the first compiler developed with the CIL system [KETC81]. It took about three months to build the front end. The code generator was developed in October 1974 (figure 8). The initial version of the compiler was tested in December 1974 [KETC81]. Final development of the compiler was severely complicated by the parallel development of the ASP-202 computer. Many compiler changes were required because the ASP didn't meet hardware specification and had to be changed.

The J3B application code for SPS was written using the Program Development and Maintenance System (PDMS), which provided a program support facility for program design, coding, and maintenance. PDMS consisted of a Program Support Library, which controlled the creation and accessing of programming libraries on

disk, preprocessors for JOVIAL, FORTRAN, ASL, and SPL, and a data management facility, which provided an alternative means of defining and gaining access to data without affecting the conventional facilities of the languages[SPSM77A].

Under PDMS each program consisted of a top-most unit, the TASK UNIT. The TASK UNIT, in turn, was made up of subunits, which consisted of either text, INCLUDED UNITs, or functions, FUNCTION UNITs. Each unit was limited to 50 lines or less. The J3B preprocessor replaced POMS keywords with the appropriate JOVIAL statements: START, TERM, PROC, RETURN, BEGIN and END. CALL statements cause a REF PROC declaration to be generated. A conventional IF-THEN-ELSE statement generated the appropriate JOVIAL IF statement. To encourage structured programming, the PDMS preprocessor did not support the JOVIAL GOTO statement or statement labels [SPSM77A].

15.6 IBM J3B-2 Compiler for B52 OAS.

In 1977, the CIL compiler system was used to produce a J3B-2 compiler for the IBM AP101C computer. Boeing was using the AP101C for an upgrade to the B52 Offensive Avionics System (OAS). The AP101C was very much like the ML-1, so the ML-1 back end was modified to produce code for the AP101C. By this time, the current J3B dialect was J3B-2, so IBM/FSD Owego reworked the ASP-202 J3B-0 front end to conform to the J3B-2 version of the langauge. Boeing did not use this compiler [KETC81].

Chapter 16

JOVIAL J73/I

The development of J73/I grew out of a perceived shortcoming of the original J73 definition. It formed an important stepping stone on the road to JOVIAL language standardization when it was formalized as military standard 1589 (MIL-STD-1589). Finally it served as the basis for the current approved version of JOVIAL, J73B MIL-STD-1589B (see Chapter 20).

16.1 The Subsetting of J73.

The original definition of J73 was perceived by some to be so large that many computer systems could not efficiently support the full language. To avoid a proliferation of subsets which could hurt the transportability of programs, a committee was formed in early 1973 to recommend a standard J73 subset. Abacus, CSC, and others served on that subcommittee [ABAC80].

The committee specified three subsets of J73, which were upward compatible with each other and with full J73. These subsets were arrived at by comparing the relative benefit of several hundred language features against the costs of compiler complexity, host computer memory and speed requirements, and difficulty of learning [TRAI77A]. Infrequently used features were eliminated; those that permitted the programmer to be concise were retained.

Each subset was designed to be suitable for a certain application area. Features were adjusted for consistency and to ensure that what could be accomplished with the full language could be realized in all subsets, although requiring more statements. Lynn Shirley wrote the initial subset descriptions [SHIR73B].

The base subset, J73 level I (J73/I), was designed for programming in a very explicit manner where greater control over the resultant object program was desired. J73/I was designed for a sufficiently wide range of applications to minimize a proliferation of versions [TRAI77A]. J73/I was suitable for coding avionics systems, operating systems, and other real-time systems which would be compiled on computers with small main memory (less than 32K) and executed on embedded computers with limited power and capacity. The design of J73/I was heavily influenced by J3B-0 [DINI81].

The next most restrictive subset, J73/II, provided for situations where efficiency could yield in favor of increased productivity by less skilled programmers. J73/II was suitable for data management and information systems where compilation would be done on medium-sized computers (32K - 128K memory) and execution on medium to large scale computers.

J73/III permitted the use of fixed type data and supported coding of command and control systems and large scale tactical systems where compilation was to be done on large computers (32K - 128K) and execution on systems that required the storage efficiency of fixed type data. Tables 2 and 3 summarize the features which were excluded from the three subsets.

1. Bit form.
2. Chain comparison.
3. Concatenation in bit formulas.
4. Character form.
5. All built-in functions except SHIFT, i.e. EXRAD, fraction part, integer part, signed, type, number of words per entry (NWDSEN), and alternate entrance built-in functions..
6. Both sides of assignment statement having multiple variables and/or formulas.
7. Multiple loop controls in a loop statement.
8. REMQUO built-in procedure.
9. EXIT statement.
10. ZAP statement.
11. DIRECT statement.
12. List-directed formatting.
13. Packed simple items.
14. Tight tables.
15. Table allocation increment.
16. Subordinate overlays.
17. Dynamic allocation of procedure instructions.
18. Tables with variable bounds.
19. Procedure use before declaration.
20. DEF and REF within a single program.
21. Index range in conjunction with number of entries built-in function.
22. Actual input parameters specifing labels for STOP, RETURN, TEST, and EXIT.
23. Use of other type values in a character type context.
24. Character formula used as a format list.
25. Rearrangement by index value of statements contained within a switch statement.
26. Multiple names declared by a specified table item declaration.
27. Constant formulas containing the operators &, @, and @@.
28. Constant formulas containing floating and fixed constant operands.

Table 2. J73 Features Excluded from All Three Subsets.

FEATURES OF J73/III NOT IN J73/II and J73/I.

1. Fixed type data.
2. Evaluation control.
3. Attribute association

FEATURES OF J73/III AND J73/II NOT IN J73/I.

1. DEFINE capability.
2. ROUND attribute in an item declaration.
3. Format-directed formatting
4. Alternate entrance declaration
5. Status declaration
6. Exchange statement
7. Indexed variable range
8. Multiple names declared by a single item declaration
9. Concatenation in character formulas

Table 3. J73 Features Excluded from Specific Subsets.

16.2 The DAIS Program.

Due to its limited size, J73/I was chosen for the first compiler to limit implementation risk. In 1974, the Avionics Laboratory of the Air Force Wright Aeronautical Laboratories (AFWAL) at Wright-Patterson AFB in Dayton, OH, awarded CSC a contract to build two J73/I compilers for the Digital Avionics Information System (DAIS) program (Figure 8).

DAIS was a research program concerned with applying current technology to reduce the number of separate computer systems and displays in the cockpit in order to stem the rising costs of developing avionics weapons systems [STAN80]. Applications modeled after the A-7D and the A-10 attack aircraft [STAN80] were demonstrated to show the benefits of applying modern techniques such as digital flight computers and HOLs [CHAL77] to the development of avionics. Significant among the technologies incorporated in DAIS were the use of structured programming practices and an HOL, namely J73/I [STAN80].

Physically DAIS consisted of a cockpit simulator called the Hot Bench, which consisted of several Westinghouse AN/AYK-15 avionics computers organized as a system and connected to a set of five multi-purpose programmable CRT cockpit displays instead of the usual flight instruments [STAN80]. The AN/AYK-15 was also called the Hot Bench Computer (HBC).

CSC built two J73/I compilers for DAIS. Both were hosted on the DEC-10 computer. One was targeted to the DEC-10 and the other to the Westinghouse HBC. These compilers were developed using

the J73/S compiler CSC developed for JOCIT (chapter 11). J73/S was a subset of J73/I. Initially, the J73/S compiler was hosted on CSC's UNIVAC 1108, in El Segundo, CA and was written in SYMPL. First, it was then rewritten in J73/S and self-compiled. Next, a DEC-10 code generator was added to form a UNIVAC 1108-to-DEC-10 cross-compiler, which was then bootstrapped to the DEC-10. This resulted in a J73/S compiler on the DEC-10.

The J73/I compiler itself was written in J73/S. It was largely developed and checked out on the 1108 using the UNIVAC-hosted version of the J73/S compiler. The compiler was then transferred to the DEC-10 at AFWAL and compiled with the DEC-10-hosted J73/S compiler, which was discarded as soon as the J73/I compiler could support itself. The DEC-10 compiler was delivered in early 1976. It required about 50K words of memory and compiled at 1500 lines/min on the DEC-10 model KI-10 [PALM77].

The AN/AYK-15 compiler was derived from the DEC-10 compiler by adding a new code generator. Initial development work on this compiler required a software simulator for the AN/AYK-15 as the computer had not yet been delivered. This compiler was delivered in mid 1976 [DUNB81].

The design of these compilers was completely different from that used for the GENESIS/SYMPL compilers and was derived from the CSC Hughes H4400 compiler. Rather than a syntax table-based front end, these compilers used an ad hoc front end designed specifically to recognize J73/I. This approach produced a faster, more efficient compiler. This same design was used for the J73 compilers built by

SEA (chapters 19 and 20) [DUNB81].

The CSC J73/I compilers were used by Intermetric, TRW, and the Charles Stark Draper Laboratory to develop software for DAIS. Intermetrics developed the Avionics Executive and Mission Software systems. The PALEFAC support software was developed by Draper Laboratory [CHAL77]. TRW developed the Software Design and Verification System (SDVS), which was a set of software development and management tools used by the other contractors to develop DAIS software. SDVS was initially coded in J70 until the CSC J73/I compiler was available (chapter 12).

16.3 MIL-STD-1589.

As happened with the J2 compilers, the J73/I language was modified even before the first compilers were developed. The original specifications of J73 [JOVI73] and its subsets [SHIR73B] were essentially rough drafts that contained many inconsistencies and ambiguities. Many of these were discovered by TRW while attempting to write a formal semantic definition of J73 using their Semantic Oriented Language (SEMANOL)[BERN75]. Also, J73/I did not adhere to the philosophies of modern language design as espoused Hoare [HOAR73B], Dijkstra [DIJK69] and others [DINI82].

In 1973, RADC contracted Prof. Tony Hoare (then of Queens University at Belfast) to review and, subsequently, revise the J73/I specification [HOAR73]. One of Hoare's graduate students, Jim Haubiph, did most of the actual work in consultation with Terry Dunbar (CSC), D. Lynn Shirley, John Mergiton (Abacus) and

Sam DiNitto. The revised specification was delivered in mid 1975 and, after further revision, was issued as MIL-STD-1589 in February 1977 [DODM77] (figure 9). Robert Storer, Sam DiNitto, and John McLean of RADC managed the effort [DINI82].

It is interesting to note that in revising J73/I many of the features of J73 originally excluded from the J73/I subset were reinstated. The result was a language closer to the full J73 language than to the J73/I subset. Hence, the original job of paring down the full J73 language, which was not done due to the B-1 schedule pressures, was finally accomplished, although by a more circuitous chain of events.

16.4 Other CSC J73/I Compilers.

Following publication of MIL-STD-1589, the changes to make the DEC-10 compiler compatible with MIL-STD-1589 were not made until CSC received a separate contract to build a MIL-STD-1589 DEC-10 hosted cross-compiler for the Delco M362F computer [UPSH81]. The changes required by MIL-STD-1589 were then retrofitted to the DEC-10 and HBC code generators.

The M362F compiler was developed to provide a compiler for use on the F-16. However, it was never used. Later, it was modified by Proprietary Software Systems (PSS) under contract to DELCO to provide a compiler for the M362S, a spaceborne version of the M362F. The M362S was used by Boeing Aerospace Company as the main flight computer for the Inertial Upper Stage (IUS) rocket,

which is deployed from the Space Shuttle to boost satellites into the higher geosynchronous orbits required for communications.

In 1978, CSC developed a DEC-10 hosted cross-compiler for the Raytheon Fault Tolerant Spaceborne Computer (FTSC). The FTSC was built by Raytheon for SAMSO to provide a common spaceborne computer for Air Force satellites. SAMSO had contracted for development of the FTSC cross-compiler; RADC acted as the contract monitor.

At the time the compiler was built, Raytheon had only developed a brass-board version of the FTSC. Development of the FTSC was cancelled in 1980 before a production version could be built, and the FTSC compiler was never used for actual program development.

16.5 SEA J73/I Compilers.

After CSC had developed the DEC-10 compilers for DAIS, AFWAL let a follow-on contract for the maintenance of these compilers in 1976. CSC and TRW, both of whom were involved with the inital DAIS effort, bid for the contract. TRW won and subcontracted the compiler maintenance work to Abacus. Thus from the end of 1976 until 1978, Abacus maintained and enhanced the DEC-10 J73/I compilers [ABAC80].

In the spring of 1978, TRW switched the maintenance contract to a small firm called Software Engineering Associates (SEA). SEA was formed in June 1977 by Terry Dunbar and others who had worked on the JOVIAL compilers at CSC. One of the first tasks

SEA did under the TRW/AFWAL contract was to combine the CSC DEC-10 hosted compilers into a single compiler with four code generators. This compiler served as the basis for all subsequent SEA J73/I and J73 compilers (chapter 20).

In July 1978, SofTech subcontracted SEA to develop an interim JOVIAL J73/I compiler hosted on the IBM 370 to use in developing the J73 JOCIT compiler (see chapter 19). This interim compiler had the same front end and syntax analyzer as the AFWAL integrated compiler, except for a few host-dependent parts. A code generator for the IBM 360 was first added to the DEC-10 compiler, and then the entire compiler was cross-compiled to the 360. The other targets for the DEC-10 compiler were not initially rehosted. Work started in September 1978 and delivery was in February 1979.

As part the AFWAL support contract with TRW, SEA added code generators to this compiler for the AN/AYK-15 and its successor, the AN/AYK-15A. In 1980, the machine instruction set of the AN/AYK-15A was standardized as MIL-STD-1750. The idea behind MIL-STD-1750 was to standardize only the machine instructions and registers available to the programmer, i.e. the instruction set architecture (ISA), instead of the design of the entire computer.

Computers having the same ISA could then be built by different contractors using their own design and yet still be programmed with the same compiler or assembler. Also the underlying technology of the computer could be improved without necessitating

changes to previously written software. MIL-STD-1750 later evolved to MIL-STD-1750A.

The ISA concept is best illustrated by IBM's 360, 370, 4331, 4341, and 303x computer family. All computers execute essentially the same instruction set as do the so-called plug-compatible AMDAHL and ITEL computers. Thus, although a compiler is said to be hosted or targeted to the IBM 370, it can, in fact, run on any of these machines. Actually, SEA developed the J73/I compiler for the ITEL computer at Wright-Patterson AFB, not the IBM 360.

SEA later rehosted the J73/I DEC-10 compiler to the UNIVAC 1100 for Rockwell/Collins and added a code generator for the CAPS-7 (Collins Adaptive Processing System) computer. Collins used the compiler to write programs for the ground receiver units developed for SAMSO's NAVSTAR Global Positioning System (GPS).

In summary, SEA built three J73/I compilers:

1. The DEC-10 hosted compiler for AFWAL with DEC-10, AN/AYK-15, MIL-STD-1750, DELCO M362F, and FTSC code generators.
2. The IBM 370 hosted compiler for SofTech, built by added a 370 code generator to the DEC-10 compiler and rehosting to the IBM 370.
3. The UNIVAC 1108 hosted compiler for Rockwell/Collins with UNIVAC 1108 and CAPS-7 code generators.

Chapter 17

STANDARDIZATION OF J73

In 1976, the DOD began a concerted effort to standardize on a few high order programming languages in an effort to cut the costs of software development for. The effect of this effort on JOVIAL was the creation by the Air Force of a formal structure to control the configuration and use of JOVIAL. Another effect was the development of a single standard version of JOVIAL. This chapter describes the establishment of the JOVIAL control structure that led to the development of the new JOVIAL standard (MIL-STD-1589A), which is discussed in the next chapter.

17.1 DOD Standardization Efforts.

By 1974, the large number of JOVIAL dialects (J2, J3, J4, J5.2, J5.3, J6, J70, J3B, and J73/I) made it clear that earlier standardization efforts had failed. Furthermore, the problems with JOVIAL were symptomatic of a proliferation of languages and language versions within the DOD in general. Two extremes existed within the DOD at the time: either HOLs were not being used at all or too many HOLs were being used. As late as 1976, a study of sixty-four Air Force weapons systems [LORI76] showed that approximately fifty percent of all programming was done in assembly language.

On the other hand, the proliferation of HOLs, often dialects of the same language, was widespread. One report [FISH78] stated that at least 450 different languages and dialects were being used on DOD embedded computer applications; none was widely used [SLAV79]. This situation existed despite previous standardization efforts. It was obvious that language standardization without effective enforcement was useless [SLAV79].

This lack of language standardization had resulted in skyrocketing costs for software. Each language required not only its own compilers but also its own compiler maintenance, training, documentation, and support software tools [SLAV79]. With the prospect of future software cost estimated to be in the billions of dollars annually [CCIP85], the DOD took action. In January 1975, the DOD initiated a study [HATH77A] to define a set of standard HOLs for all defense systems. The result was two landmark DOD policy statements.

In April 1976 (figure 8), DOD Directive 5000.29 [DODD76] established the policy for managing the acquisition of computer resources for defense systems. In the area of programming languages, 5000.29 required that only approved HOLs be used. Each approved HOL was assigned a Designated Control Agent (DCA) responsible for compiler validation and language configuration control.

The second policy was DOD Instruction 5000.31 [DODI76] which specified the approved list of HOLs referenced in 5000.29: TACPOL, CMS-2, SPL-1 (Signal Processing Language), FORTRAN, COBOL,

and JOVIAL (J3 and J73/I). J3B was not included as a standard language due to its similarity to the broader based J73/I standard. The Air Force was made the DCA for JOVIAL.

17.2 Air Force Standardization Efforts.

In December 1976, responding to 5000.29 and 5000.31, the Air Force issued Air Force Regulation AFR 300-10 [AFR76], which assigned control responsibility for JOVIAL to Air Force Systems Command (AFSC) and limited Air Force programs to using COBOL, FORTRAN, ATLAS, and JOVIAL. Requests for use of assembly language or non-standard HOLs required an approved waiver from AFSC prior to system acquisition.

The AFSC had already begun a standardization effort in October 1974, when the High Order Language Standardization Program was established to develop a plan to control proliferation of HOLs in new Air Force weapons systems. After two years of data collection, a draft of the AFSC HOL Standardization Policy and Implementation Plan was issued in late 1976 by RADC [HATH77B]. Drafting of the Plan was managed by RADC project officers Sam DiNitto and Dick Slavinsky.

In the middle of reviewing this draft, the 5000.29, 5000.31 and AFR 300-10 were issued. In May 1977, AFSC held a meeting to revise the Plan to be more compatible with DOD and Air Force policy [HATH77A]. In December 1978 (figure 9), the final version of this Plan was formalized as AFSC Supplement 1 to AFR 300-10 [AFSC78].

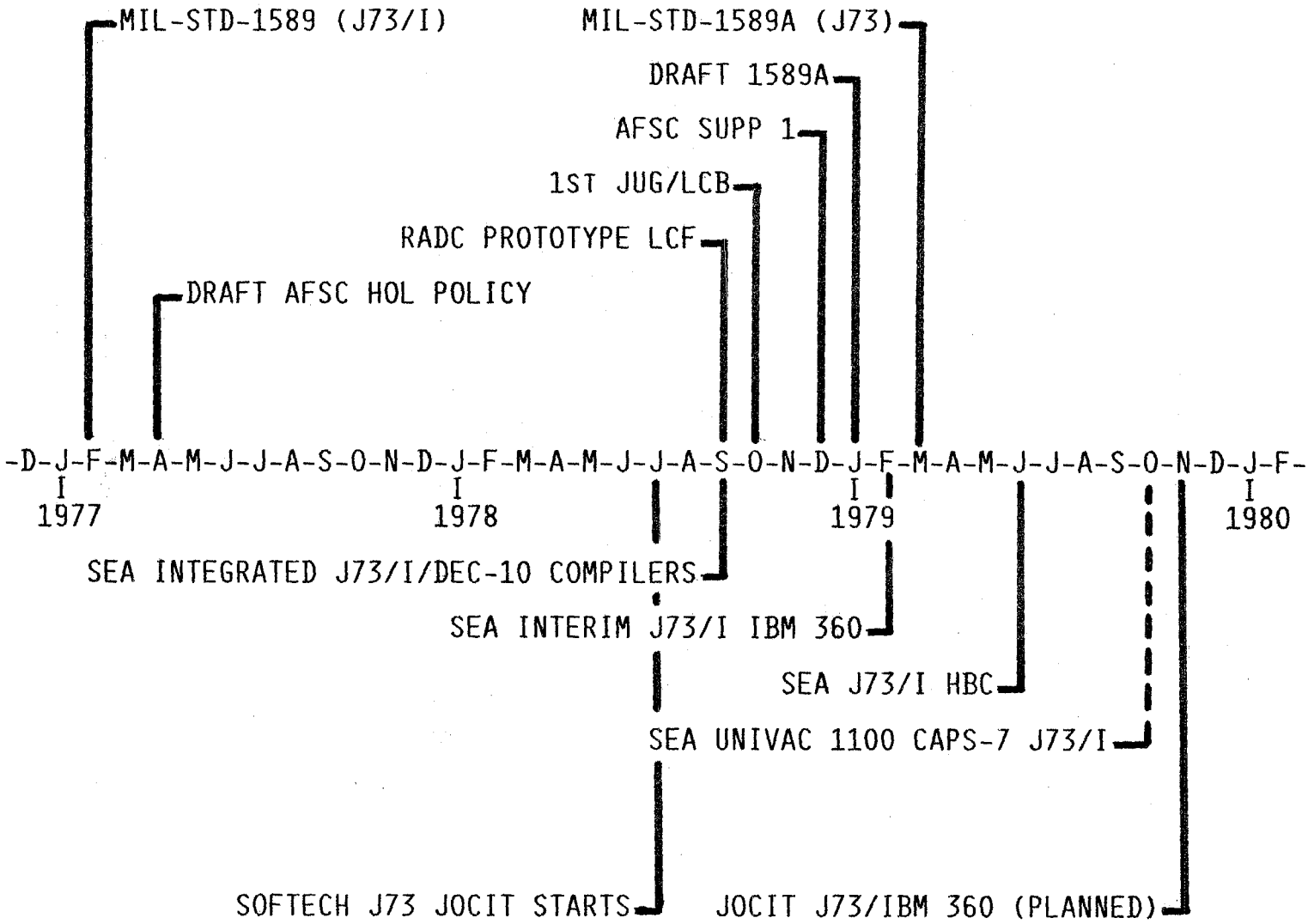


Figure 9. JOVIAL J73/I AND J73A (1977 -- 1980).

Supplement 1 applied to all AFSC weapons systems programs and established a formal JOVIAL language control structure to carry out AFSC's responsibilities as the JOVIAL Designated Control Agent. Actually, Dick Nelson and Sam DiNitto at RADC had conceived of the idea for a JOVIAL control facility as early as 1972 [DINI82]. Major points of Supplement 1 were:

1. Designation of RADC as interim JOVIAL Language Control Agent (LCA) responsible for base line control of the J3 and J73/I.
2. Establishment of the JOVIAL Language Control Facility (LCF), to assist the LCA in performing the JOVIAL control functions.
3. Requiring that each JOVIAL compiler be validated by the JOVIAL LCA prior to acceptance or use.

A three-phase program was also adopted for JOVIAL language control over the next ten years. In the near-term phase through 1979, AFSC planned to limit use of JOVIAL to two standard languages, J3 and J73/I, and to begin development of an LCF to ensure the quality and control of these languages. In the medium-term phase through 1983, efforts were to be directed toward development and use of a single JOVIAL language for all Air Force applications. In the long-term phase through 1986, a JOVIAL programming environment would be developed to further reduce the costs of software development, testing, and maintenance [SLAV79].

17.3 JOVIAL Control Structure.

The control structure established by Supplement 1 was comprised of four elements: the Designated Control Agent (DCA), the JOVIAL Language Control Agent (LCA), a formal JOVIAL Users Group (JUG), and the JOVIAL Language Focal Points [SLAV79]. The same control structure is still in place today.

The DCA is the authority element in the control structure [SLAV79] with final authority over approval of requests for all JOVIAL waivers, extensions and subsets. The DCA interfaces directly with the DOD and Air Force headquarters when exceptions to their respective HOL policies are requested.

The LCA, together with the Language Control Facility (LCF) and the Language Control Board (LCB), constitute the technical element. The LCA handles the technical aspects of JOVIAL control and is responsibile for assuring the stability of the language. The LCF is the technical arm of the LCA and performs JOVIAL language configuration control and compiler testing. The LCB is the decision making arm of the LCA and is comprised of the LCA, the DCA and the JOVIAL Language Focal Points. The LCB typically meets in conjunction with the JOVIAL User's Group meetings and makes all official decisions concerning language control.

The JOVIAL Language Focal Points constitute the DOD community interface element [SLAV79]. Each Focal Point is appointed by an AFSC product division (e.g. ASD, ESD, SAMSO) or other participating DOD organization to be responsible for assuring

communication between his respective organization and the JOVIAL control structure.

The JUG (later renamed the JOVIAL-Ada User's Group) provides the user community interface for injecting user ideas and opinions into the language control process independently of the Air Force. JUG membership is open to anyone interested in JOVIAL.

Lt.Col. John Marciniak, director of AFSC Computer Resources Policy and his deputy, Capt. Al Kopp, were the first JOVIAL DCA. RADC project officer Richart T. Slavinski was the LCA and director of the LCF. The prototype LCF was established at RADC in September 1978 as a continuation of its role in developing the draft AFSC HOL Policy and Implementation Plan. SofTech Inc., Waltham, MA. was selected as the LCF support contractor on 8 January 1979 [SOFT81]; Michael Moore was the SofTech LCF director. The initial tasks of the LCF were to [SLAV79]:

1. Control and maintain the JOVIAL language standards (J3, MIL-STD-1588 and J73/I, MIL-STD-1589).
2. Maintain an information directory of JOVIAL users, compilers, and support tools.
3. Assist in the acquisition, development, and maintenance of all JOVIAL compilers.
4. Validate all JOVIAL compilers for conformance to the appropriate JOVIAL language standards.
5. Provide direct technical guidance in evaluating all JOVIAL language change, subset, and extension requests received by the DCA and LCA.

6. Support the JUG through maintenance of user mailing lists and distribution of bulletins and meeting minutes.

Formal procedures were established for how the various elements in the language control structure would deal with reviewing language change requests and updating the language standards. These procedures emphasized JUG participation. Each proposed language change was reviewed by the LCA who in turn submitted the proposal to the JUG for initial review and comment. The JUG-approved change requests were then presented to the LCB, which recommended approval or disapproval. The LCB recommendation together with the JUG comments are used by the DCA to make the final approval determination [SLAV79].

These procedures were put into practice immediately to develop a single JOVIAL standard as part of the AFSC three-phased plan. This standard was developed by the newly formed LCB and JUG in less than six months and is the subject of the next chapter.

Chapter 18

J73A LANGUAGE DEVELOPMENT

In 1979, J73/I (MIL-STD-1589) was revised to become J73A (MIL-STD-1589A) in order to create an acceptable JOVIAL standard for all Air Force programs. The creation of J73A was largely due to political forces within the Air Force and provides an exceptional example of how non-technical influences cause language change.

18.1 The J73/I Upgrade.

When the AFSC JOVIAL plan calling for a single JOVIAL standard by 1983 was formulated, RADC had planned that J73/I would be the accepted standard. About this same time (1978), the B1 program had just been cancelled, and the electronics systems on the B52 and the F-111 needed to be upgraded for these aircraft to fill the gap left by the cancelled B1. These upgrades would necessarily involve the use of avionics computers, which would have to be programmed in J73/I if this were the standard.

But the B52 and F-111 SPO's were most familiar with J3B-2 used on the B1 and were reluctant to commit a major defense program to a language like J73/I that had been developed by a research organization (RADC) and only tried out in a laboratory environment (DAIS at AFWAL). The bottom line was that the B52 and F-111 programs were going to go with J3B-2.

Such an action on the part of two major program offices at a time when the Air Force was embarking on a major effort to standardize on a single language would have had a disastrous effect on getting any other program to go along with the standard. The only course of action acceptable to the program offices at ASD was to merge the two languages.

In September 1978 (Figure 9), RADC was formally directed to develop an upgraded version of J73/I, temporarily called J73*. SofTech was given the actual task of writing the specification for J73* as part of their J73 JOCIT contract with RADC (next chapter). Dr. John Goodenough led the J73/I upgrade effort for SofTech.

The purpose of J73* was to incorporate into J73/I some of the proven features of J3B-2 in order to establish a standard JOVIAL language suitable for all command-and-control, avionics, space, and armament defense application [JUG79A]. That J73/I and J3B were quite similar made the merger feasible. Many of the differences between the two languages were trivial syntactic ones. J3B was designed about the same time as J73/I, and the J3B designers had access to the J73/I preliminary design documents.

Apart from syntatic and semantic differences, the major difference between the languages was that J3B contained more features that enforced safe, structured programming practices resulting in increased program reliability and maintainability and thus decreased software costs. For example, J3B was a strongly-typed language, which provided type declarations and "typed" pointers instead of arbitrary access to machine addresses;

data objects could only be used according to their delcared data type. J3B also restricted unstructured GOTOs outside of local scope.

The major design task in the upgrade was to provide these desirable J3B features but at the same time ensure that no useful J73/I capability was deleted [SOFT79B]. SofTech was directed to resolve differences by adopting J73/I syntax except where J3B was clearly superior.

Although AFSC Supplement 1 was not official until December 1978, RADC planned for the upgrade to be accomplished within the framework of the new JOVIAL language control structure. Specifically, RADC as the LCA would supervise the proposed changes, the JUG would provide input from actual users, and the LCB would approve the changes.

18.2 The Influence of the JUG.

The sequence of JUG meeting and language drafts to draft the upgraded language that transpired during the next several months were in marked contrast to the design of J2 by Schwartz. The revision of J73/I was a committee effort involving the 30 - 40 members of the JUG as well as the 5 - 10 members of the LCB.

RADC convened the first meeting of the JUG and the LCB, on 4-5 October 1978, at Wright Patterson AFB, Dayton, OH, to consider the proposed language changes [JUG78A]. At that meeting, Lt.Col. Marciniak presented the groundrules that SofTech and the JUG were to follow in considering the proposed language changes: 1) use MIL-STD-1589 as the baseline, 2) do not recommend new features other

than those specified for consideration in the JOCIT contract, 3) resolve syntax and semantic conflicts in favor of modern language principles, 4) make it possible to translate existing J3 and J73/I programs to the upgraded language, and 5) ensure the upgraded J73 would be useful for all Air Force embedded computer systems [SOFT78].

Also at that meeting, SofTech requested ideas for subroutine parameter passing, typed pointers and TABLES, status type and TYPE matching rules, and loop statements and surveyed the JUG members to find out what features JUG members felt should be deleted or included in J73/I. The results of that survey along with the requirements from the JOCIT contract were used to draft the first description of J73* [SOFT78]. This description was distributed to the Air Force and the JUG for comment on 1 November 1978 and was discussed at the second JUG/LCB meeting held 13-14 November 1978 at Wright-Patterson AFB.

Some of the key changes that SofTech proposed in this document were:

1. Allow multiple subroutines in all compilation units.
2. Eliminate J73/I implicit conversions, i.e., add strong typing.
3. Incorporate J3B CONTSTANT declaration capability and permit TABLEs as well as ITEMs to be declared as constants.
4. Require data declarations to precede and subroutine declarations to follow all executable statements as in J3B.

5. Add recursive (REC) and reentrant (RENT) and INLINE capability to subroutines.
6. Permit labels to be passed as subroutine parameters as in J3B.
7. Require empty parentheses for parameterless subroutines.
8. Allow functions to have out parameters as in J3B.
9. Allow size-unresolved TABLES as formal parameters to permit passing TABLEs with varying sizes.
10. Make STATUS a true data type.
11. Forbid nested subroutines from external reference via a DEF specification.
12. Use the J3B keyword LABEL for a label declaration instead of the J73/I keyword NAME and require that the LABEL of a GOTO be declared in the same scope as the GOTO.

Votes were taken at that meeting to determine the acceptance of these proposed changes. JUG members also raised other language issues [JUG78B]:

1. Change J73/I subroutine invocation by adding the keyword CALL.
2. Prohibit default size specification for unsigned, signed, and character data type declarations.
3. Delete the MONITOR capability and SET data types
4. Allow packed data variables to be used as procedure output parameters.

5. Require explicit typing for pointers.

6. Retain square brackets in the JOVIAL character set.

7. Omit empty parenthesis for parameterless subroutines.

At their meeting following the JUG meeting, the LCB voted to accept all the SofTech proposals that were endorsed by a majority of the JUG members as well as some alternatives discussed by the JUG. The LCB action on these language issues was as follows [JUG78B]:

1. Retain the J73/I method for calling subroutines.

2. Permit default size specification for unsigned, signed, and character data type declarations.

3. Delete the MONITOR capability and SET data types

4. Do not include a standard collating sequence for using relational operators with alphanumerics.

5. Prohibit implicit pointer dereferencing.

6. Retain the short circuit evaluation for boolean expressions.

The results of the voting at the JUG meeting together with the LCB directives were used by SofTech to produce a preliminary draft of MIL-STD-1589A [SOFT79A], which was distributed to JUG members in early January 1979. At the third JUG meeting, 6-7 February 1979, in Fort Worth, TX, clarifications of the syntax and semantics of the draft were considered on a paragraph by paragraph basis. In addition, the JUG voted on several amendments to the SofTech draft as well as their previously proposed language changes. Some of the votes were reversals of previous JUG votes.

Those issues that carried by a two-thirds majority were forwarded to the LCB for final approval at their meeting on 21-22 February 1979 at Eglin AFB, FL. This was the last LCB meeting before publication of MIL-STD-1589A [JUG79A]. The LCB approved 7 of the 10 issued submitted by the JUG. Several language changes were deferred until the next language revision, which came just a year later (chapter 20) [JUG79B]:

The final version of MIL-STD-1589A [DODM79] was complete on 15 March 1979 (figure 9). The standard was approved by Lt.Col. Marciniak on 29 March and by Air Force headquarters on 27 April. May 1979 was the official release date. The designation of the upgraded language was simplified from J73* to just J73. The designation J73A has been used in this paper to avoid confusion with the original J73 developed in 1973.

With the revision of J73/I, references to MIL-STD-1589 in the various DOD and Air Force policy documents needed to be changed. AFR 300-10 was changed in August 1979, AFSC Supplement 1 was revised in September 1980, and a draft revision to DODI 5000.31 was drafted in January 1980. The final approval of 5000.31 was held up until 1983 due to reasons unrelated to JOVIAL.

Chapter 19

J73A COMPILER DEVELOPMENT

The upgrade of J73/I was intimately involved with RADC's second attempt to build a J73 JOVIAL Compiler Implementation Tool (JOCIT). CSC's JOCIT contract had failed to produce a J73 compiler because the language changes to J3 were more radical than expected. Unbelievably, this second JOCIT compiler failed to produce the desired J73 compiler for nearly the same reason.

19.1 The JOCIT Compiler.

Following publication of MIL-STD-1589 in February 1977, RADC issued an RFP for a J73/I JOCIT compiler in the fall of 1977. JOCIT called for a highly optimizing, state-of-the-art J73/I compiler that was designed to be easily retargetable and rehostable. The inital compiler was to be developed for the IBM 370. Later, compilers for the DEC 10 and CDC 6600 were to be developed as well as a code generator for the MIL-STD-1750 instruction set.

The contract was finally awarded to SofTech, Inc. in team with SEA in July 1978 after some delay due to the contract award being contested by Proprietary Software Systems (PSS), who were the other bidders for JOCIT. By this time, however, RADC had been tasked to revise J73/I as discussed in the previous chapter. SofTech was to make these language changes; SEA was subcontracted to

develop the compilers, which included an interim J73/I compiler for the IBM 370 that would be used to build the JOCIT compiler.

The JOCIT contract originally specified only a few changes to J73/I such as adding fixed point arithmetic. These language changes were to be defined by November 1978 and incorporated into the JOCIT compiler. But, as described in the previous chapter, the Air Force greatly enlarged the scope of the language changes resulting in MIL-STD-1589A.

The extensive changes contained in MIL-STD-1589A adversly affected the JOCIT delivery schedule (figure 10). The first schedule slip occurred when SEA delivered the interim J73/I compiler a month late in February 1979. At this point SofTech took over development of the JOCIT compiler.

Delivery of the initial, non-optimizing version of the JOCIT compiler, originally set for June 1979, was changed to August 1979. Delivery of the fully optimizing version was changed from July to November 1979. These dates were never met. Instead, the final dates were revised several times with SofTech delivering a series of intermediate versions beginning with the Baseline 1 (B1) version in February 1980 as shown in figure 10. Each baseline implemented progressively more of the J73A language until the full language was implemented in the B9.5 version, released July 1981.

Although the final JOCIT compiler was to implemented in its own language, the preliminary versions were implemented in a J73/I compatible subset of J73A so that the interim J73/I compiler could be used during development. The use of the JOVIAL DEFINE feature

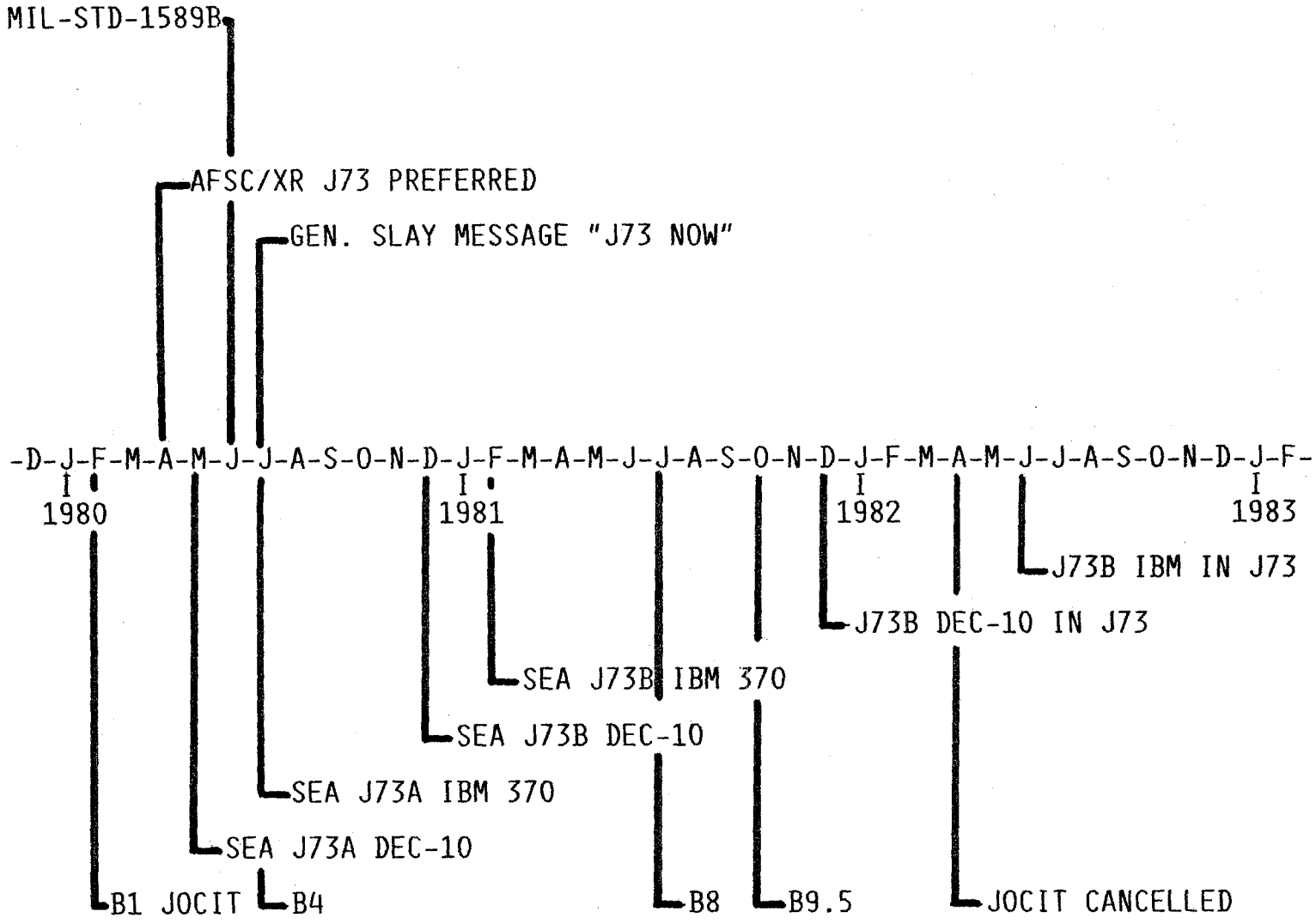


Figure 10. JOVIAL J73B (1980 -- 1983)

allowed much of the code to be written using J73 syntax, as was done by TRW with the J70 compiler (chapter 12).

With the unexpected delays in delivering JOCIT, the funds for the CDC 6600 and DEC-10 compilers were used to continue the over-budget JOCIT contract. The contract officially ended in December 1981, when all the money for JOCIT ran out before the compiler could be translated into J73A. The final version delivered by SofTech was the B9.5 baseline that was still implemented in J73/I. A 1750A code generator was started but was never finished.

19.2 Other SofTech J73A Compilers.

SofTech used the JOCIT compiler as the basis for several other J73A compilers hosted on the IBM 370. These compilers were implemented in J73/I and were never validated but were still quite usable. SofTech delivered a JOCIT-based compiler for the Bendix 930 flight computer, that Martin-Marietta used for the Army Pershing II missile [SOFT80].

For the Air Force Aramament Laboratory's Digital Integrating Subsystem (DIS) program, SofTech delivered a JOCIT compiler with code generators for the PDP 11/34 and Zilog Z-8000 to General Dynamics Corp. in San Diego, CA [SOFT80]. The Z-8000 compiler was used by General Dynamics to program the flight software for the Midcourse Guidance Correction (MGC) demonstration of an air-to-air missile.

Due to the many errors and problems with the JOCIT base compiler, SofTech's San Diego office independently worked out the errors and provided General Dynamics with a compiler sufficiently reliable to develop code for this program. Like the other JOCIT compilers this compiler was written in J73/I. General Dynamics also used the Z8000 compiler for programming the flight computer of the Medium Range Air to Surface Missile (MRASM).

As part of the MX missile program, SofTech developed a J73 compiler and support tools for both the ground computers and the MECA missile flight computer. The MX compiler was a slightly different version of the JOCIT compiler with a MECA code generator and a few extra features. Its development was, thus, closely tied to the JOCIT compiler. When the delays in the MX compiler exceeded a year, the MX SPO were concerned that SofTech would not be able to deliver the MX compiler by September 1981 as scheduled. Because MX had a high strategic priority, SofTech was directed to concentrate its manpower on MX rather than JOCIT or Pershing. This, of course, further delayed delivery of JOCIT. As it turned out, SofTech did not meet the September 1981 deadline either, and Rockwell/Autonetics, the MX flight software contractor, completed development of the MX flight software using assembly language.

One reason for the delays of the MX compiler may have been the requirement that the MECA code generator achieve an object code space efficiency of 5% over hand-written assembly code [SOFT80]. The problems that SofTech encountered in trying to meet this requirement diverted SofTech from completing either the MX or the

JOCIT compiler. In retrospect, the 5% code expansion requirement was somewhat unrealistic.

Until 1981, the SofTech compilers were the most widely used J73 compilers on real military acquisition programs (as opposed to laboratory programs) [DINI82]. However, when J73A was revised to MIL-STD-1589B in 1980 (next chapter), the government did not fund conversion of the JOCIT compiler to the new standard. Eventually, with the availability of the SEA J73B compilers, use of SofTech's JOCIT compiler diminished.

19.3 SEA J73A Compilers.

When MIL-STD-1589A became the approved Air Force version of JOVIAL in March 1979, AFWAL wanted to upgrade their J73/I compilers to the new standard. So, in April 1979, SEA was directed to make the following modifications to the J73/I compilers:

1. Modify the front end of the DEC-10 hosted compiler to accept the MIL-STD-1589A language.
2. Revise the DEC-10, AN/AYK-15, AN/AYK-15A (MIL-STD-1750), and M362F code generators to support the front end changes.
3. Add a new MIL-STD-1750A code generator (1750 had been upgraded by this time).
4. Integrate the IBM 370 code generator, which SEA had built for the JOCIT interim J73/I compiler, with the DEC-10 front end.

5. Rehost the compiler with the IBM 370, 1750, and 1750A code generators to the IBM 370 to provide a J73A compiler for the IBM 370. RADC initiated and funded this last item to provide a backup for the JOCIT compiler, which was falling behind schedule [DINI82].

These compilers were developed by SEA as part of their subcontract with TRW, which was the maintenance contractor for the AFWAL J73/I compilers. TRW performed the integration, problem analysis, and configuration control of the compilers as well as the compilation and linking of the IBM 370 hosted compiler.

SEA completed the J73A DEC-10 compiler with the DEC-10 and 1750 (AN/AYK-15A) targets in May 1980 and the AYK-15 and M362F code generators in June 1980 (figure 10). Final delivery of the IBM compiler was in July 1980, although an initial delivery was ready in April 1980.

These compilers were the first fully implemented J73A compilers available, since the JOCIT compiler was delayed. The design of the SEA J73A compilers were identical to the J73/I compilers and is explained in detail in the next chapter. These compilers were validated by the JOVIAL Language Control Facility using the JOVIAL Compiler Validation System (JCVS) developed by SofTech under their LCF contract.

Chapter 20

JOVIAL/J73B

The current standard version of JOVIAL, J73 MIL-STD-1589B (J73B) was published in June 1980, just a little over a year after J73A. The costs of revising the language even before the first J73A compilers were completed demonstrated that the benefits of language standardization and control required more than just having a single language standard. It also required that this standard remain in effect for enough time to reap those benefits.

20.1 MIL-STD-1589B.

Just as the preliminary versions of the SEA J73A compilers were being delivered and well before any reasonably complete version of the SofTech JOCIT compiler was available, J73A was revised. When MIL-STD-1589A was issued in March 1979, no revisions could be made to J73 for at least a year according to DODI 5000.31. During that year, the Language Issues Committee of the JOVIAL Users Group continued the process of changing the language that was begun when J73/I was revised. At each of the quarterly JUG/LCB meetings, various language change requests were considered and approved first by the JUG and then by the LCB. Finally, after the April 1980 meeting, all of the changes approved since MIL-STD-1589A was issued were collected and published as MIL-STD-1589B in June 1980. Over 86 language changes to MIL-STD-1589A were incorporated into 1589B.

These changes were mostly limited to clarifications and extensions to the specification; all except three were compatible with programs written in J73A [SOFT81].

The publication of MIL-STD-1589B caused considerable problems for Air Force program offices and contractors. Not only did the revision of J73A cause the J73A compilers to be obsolete even before they were delivered, but this lack of compilers were compounded by pressure from the Air Force to use J73 for all new defense programs.

Although DOD and AF language standard policy had been in effect since 1978, few Air Force programs were using J73. This situation prompted AFSC to send a message to all AFSC product divisions in May 1980, which said that J73 was the preferred language for all software for embedded computer systems. All AFSC weapons systems acquisition programs that had not passed the Critical Design Review (CDR) stage were to use J73 unless compelling reasons existed to the contrary, in which case a waiver request was required.

Not satisfied with the slow progress toward acceptance of J73 on the part of AFSC program offices, General Slay, AFSC commander, reiterated this policy in his strongly worded 1 July 1980 message: "I am dead serious about language standardization on JOVIAL J73 now and Ada when it is ready."

These two actions caused considerable stir in many AFSC program offices because of the limited availability of J73 compilers. The only working compilers were the SEA J73A compilers.

There were no J73B compilers.

As a result of Slay's message, the Embedded Computer Standardization Program Office (ECSPO) was established in October 1980, at ASD to coordinate the acquision and configuration management of JOVIAL J73 compilers and tools. The ESCPO was an new addition to the JOVIAL language control structure discussed in Chapter 17. The establishment of the ESCPO marked a change in emphasis from JOVIAL language configuration control and change to JOVIAL compiler acquisition and configuration control. In mid-1982, the ECSPO assumed maintenance and configuration control of the SEA J73B compilers from the AFWAL Avionics Laboratory.

With the added impetus of General Slay's order that all AFSC programs must use JOVIAL, the attention of the Air Force shifted away from language changes to building more compilers. Future changes to the JOVIAL standard were required to be upward compatible and were limited to not less than every two years. This would have made yet another version, MIL-STD-1589C, possible in mid 1982. But no change was made at that time because the LCA and LCB were reluctant to make further changes to the language before an adequate number of mature compilers was available. The next revision to MIL-STD-1589 was Change Notice 1, which was approved in 1983. Change 1 was issued to correct errors and resolve ambiguities and included only two small extensions.

20.2 J73B Compiler Development.

When 1589B was issued, SEA was again tasked by AFWAL to upgrade the newly-completed J73A compilers to J73B. SEA delivered the J73B DEC-10 compiler in November 1980. The February 1981 version (version 020581) of this compiler with DEC-10, 1750, 1750A, AN/AYK-15, and M362F targets was validated by the LCF in March 1981 [SOFT81].

The IBM 370 hosted J73B compiler (version 021281) with IBM 370, 1750, and 1750A targets was delivered in February 1981. It was validated in March 1981. Subsequent versions of both compilers were released in July 1981, December 1981, April 1982, and June 1982 to correct errors discovered by users.

Both the SEA J73A and J73B compilers were derived from the SEA J73/I compiler by changing the front end to accept J73, but were still implemented in J73/I. The Air Force JOVIAL policy was that all JOVIAL compilers must be written in JOVIAL so they were self-compiling. When it became clear that the SEA compilers would become the only compilers to implement the current version of JOVIAL, AFWAL funded SEA to translate their compilers from J73/I to J73B. The translation effort began about April 1980. The DEC-10 compiler was finished in September 1981 and the IBM 370 compiler in April 1982. The translation was facilitated by the use of a J73/I-to-J73 translator that SEA built by combining the J73/I compiler front end with a back end that produced J73 code.

The AFWAL/SEA compilers have served as the basis for several other J73B compilers. SEA has added code generators for the Z-8002, TI-990, TI-9900, and VAX 11/780 computers to the compiler as well as rehosted the compiler to the VAX. PSS developed code generator for the INTEL 8086/8087 microprocessor chip for INTEL Corporation. PSS also rehosted the compiler to the VAX 11/780 for the ECSPO. Because of the wide distribution and use of the AFWAL compiler, it is expected that other developments will evolve from this baseline [LCFB82].

20.3 SEA Compiler Design.

The SEA J73B compilers, as well as the previous J73A and J73/I compilers they were derived from, are divided into several phases. The compiler executive program (EXEC) is resident in main memory during the entire compilation and provides utility routines such as symbol table control and printing error messages to the other five phases, which loaded one at a time by EXEC.

The first compiler phase (JOV1) initializes a table containing a description of the desired target computer according to the target computer selected by the user. This table provides succeeding phases with target-specific information while allowing them to be target-independent.

There are two syntax analyzer phases (JOV2A and JOV2B) in the J73 compilers; the J73/I compilers only required one [DUNB81]. JOV2A performs lexical analysis and processes declarative

(non-executable) statements; JOV2B translates the executable statements into a reverse Polish intermediate language.

The code generator phase (JOV3) consists of a target-independent optimizer and code generator that expands the intermediate language into logical machine instructions, which the editor phase (JOV4) uses to produce relocatable object code. The code generator and editor phases are organized into pairs, one for each target computer. The last phase generates an external COMPOOL file to be used during subsequent compilations and is invoked only if the compiler is processing a COMPOOL.

Unlike many previous compilers, the source code for both the DEC-10 and IBM 370 hosted compilers is kept on the DEC-10. SEA is able to accomplish this by isolating most of the host-dependent parts of the compiler in the executive. In other host sensitive places, the JOVIAL parameterized DEFINE capability is used. Even so, there are a few areas, such as tightly packed TABLES which depend on the word size of the host computer, that cannot be specified with DEFINES. These are are handled by using !BEGIN-!END directives to cause conditional compilation (appendix D, 8.2).

20.4 Use of J73B Compilers.

The DEC-10 hosted J73B compiler was used by the System Integration Branch at AFWAL to convert the DAIS Avionics Executive and Mission Software from J73/I to J73. TRW, Dayton, OH used this compiler to develop the ALINKS 1750A Linker and the JOVIAL Interactive Debugger (JID), both for AFWAL.

Other Air Force agencies have used the compiler to develop such software systems as the JOVIAL Programming Support Library, JOVIAL Code Auditor, and the JOVIAL Automated Verfication System. The IBM-hosted J73B compiler is being used by IBM/FSD Gaithersburg for the Data Systems Modernization program at the Satellite Control Facility (see chapter 8).

PART III

CONCLUSIONS

Chapter 21

TECHNICAL INFLUENCES

The next four chapters summarize the lessons that can be learned from JOVIAL. Hopefully, understanding the events that influenced JOVIAL will provide some insight into the factors affecting high order languages in general. The technical factors that affected JOVIAL are summarized in this chapter. Chapter 22 provides a similar overview of the non-technical factors. Chapter 23 deals with the issues of why JOVIAL never achieved greater acceptance. And Chapter 24 summarizes what influence JOVIAL has had on other languages and on the field of language design.

As with other languages [WASS80], the technical influences on the development of JOVIAL can be divided into four main categories: the technical basis for the language, the application problem domain to be supported, hardware constraints of the computers involved, and any programming methodologies to be supported.

21.1 Technical Basis.

The technical basis of a language is the body of technical data and features taken from the literature and from other programming languages that were used to as the starting point for

its design. As mentioned before, the technical basis for J2 was ALGOL-58 and, to some degree, CLIP which was also derived from ALGOL. Schwartz chose ALGOL as the basis for JOVIAL because "at the time it appeared as if it would become the commonly accepted standard for languages, and it seemed like a better technical basis for a new language than FORTRAN, which was the only other possibility" [SCHW78].

The fundamental structure and notation of JOVIAL as well as many of its features were borrowed directly from ALGOL. Such features as the compound statement, the FOR loop statement, the relational operators, the GOTO and SWITCH statements, and the provisions for procedures and functions were taken directly from ALGOL and give JOVIAL the overall character of other block structured, von Neuman-type languages such as Pascal, PL/I, and Ada.

The major characteristic of such languages is a flow of program execution that is sequentially oriented, as opposed to languages like SNOBOL, LISP, and APL which are pattern and functionally oriented. Sequentially oriented languages have been favored by software engineering advocates and, indeed, seem to be well suited to development of large, complex systems. Also, we seem to understand these languages better.

This basic structure has been retained in all versions of the language. JX.2, JS, J4, J5.3, J6, and Basic JOVIAL were either subsets of extensions to J3 and retained most of its constructs.

The addition of the decision table control structure to J6 was a significant departure from the J3 control structures. But it was an addition to J3 and did not replace the other control statements.

The creation of J73 was not so much an enhancement of J3 as it was the creation of new language. While retaining J3's basic form, J73 incorporated many new ideas that were not known when J3 was designed. By 1973, ALGOL-68 had been defined, IBM had introduced PL/I, and Pascal had been designed [WIRT71], although it was not as popular as it is now.

The most important feature added to J73 was the TYPE declaration capability. This feature first appeared in Pascal and has since been incorporated into other languages. The ability to specify user types is an important facility for achieving data abstraction, simplifying program correctness, and enhancing understandability. Another important addition to J73 was the pointer constuct, which allowed the creation of dynamic data structures.

Later versions of JOVIAL retained many of the new feature of J73. The MIL-STD-1589 version of J73/I had nearly all the features of the original J73 language. Although J3B was initially based on J5, it was later enhanced to include many of the ideas included in J73, whose definition took place at the same time as the definition of J3B-0.

J73A (MIL-STD-1589A) was, of course, directly derived from J73/I and J3B-2. Although no new ideas were supposed to be included in J73A, the influence of the JOVIAL Users Group on the final language design did result in some constructs being implemented in much different ways than they were in J73/I. J7B was simply J73A with added refinements. J73B was a result of the continuation of the process begun when J73A was specified, the intent being to make the clarifications and changes that were missed in J73A.

21.2 Problem Domain.

The expected problem of a programming language probably has the most important impact on the technical content of the language. Most languages were created to provide features for a particular application. In fact, at the time JOVIAL was first designed, HOLs were called problem oriented languages, i.e. languages designed for a specific type of processing problem.

In the case of JOVIAL, the problem domain changed little during the first 10 years of usage. The early versions of JOVIAL (J2, J3 and J4) were used by SDC to program command-and-control systems and to develop operating systems, compilers, and other system software.

Whereas languages designed before JOVIAL, such as FORTRAN and FLOWMATIC, were oriented toward either scientific or business processing, JOVIAL was the first language (besides CLIP of course)

that was designed with features for information processing. JOVIAL contained features for things such as text and bit manipulation, data re-formating, database accessing and decision making as well as for the ususal arithmetic processing done by previous languages. Eventually, other languages like PL/I, Pascal, SIMULA, and Ada were designed that provided features for character manipulation and complex decision logic as well as for arithmetic calculations, so today we take for this capability in an language for granted. But at the time, this was a pretty significant accomplishment.

J2 was designed for programming the SACCS system, which everyone assumed would be like SAGE. Two of the most unique JOVIAL features, the TABLE and the COMPOOL, were added from experience with SAGE. TABLES are like arrays of records in Pascal or Ada. It would be nice to think that the JOVIAL TABLE influenced the creation of the record construct, but there is no evidence to support this. Both probably arose out the need to provide for a similar data structuring feature. The TABLE has survived in every version of JOVIAL since J2. In J73, TABLES were expanded to be multi-dimensional and the ARRAY construct, originally taken from ALGOL, was dropped, since an ARRAY could be implemented simply as a table with one item per table entry.

J6/SPL was probably the best example of the influence of the expected problem domain. J6 was developed to program spaceborne computers. Thus, matrix operators for adding, multiplying,

transposing, inverting, and taking the scalar product, sum, and difference were added to J3. Features for interrupt handling were added to the already powerful DIRECT and functional modifiers to provide enhanced machine accessibility. Minor cosmetic changes were also made such as dropping the $ statement terminator.

One of the major uses of JOVIAL has been for avionics programming. This use of JOVIAL came about due to development of J3B for programming the B-1 avionics systems. Later, the first application of J73/I was to program the DAIS avionics testbed. Even so, the overall capability of the language changed little with the change in application area (compare J3 and J73B, appendices C and D respectively). Had JOVIAL not been suitable for programming avionics computers to begin with it is doubtful that it would have been chosen for the B1 program in the first place.

The early omission from J2 of any input/output (I/O) capability was a clear demonstration of the large impact of the intended application. In SAGE, all I/O was done by the central control program, which acted essentially like an operating system. The I/O in SAGE was very hardware dependent and required the power of assembly language to program. It was assumed that I/O in SACCS would likewise have to be done in assembly language. Thus, it was unnecessary for I/O features to be included in the first JOVIAL.

The I/O features in other versions of JOVIAL have also been influenced by the expected application. J4 included added I/O

features needed to access files in the disk operating environment of the CDC 3800 computer. The I/O commands included in the TINT interpreter were dictated by the need for interactive communication between the programmer and his running program.

Although I/O facilities were added to J3, I/O statements were again removed when J3B and J73 were designed and have not been included in any later versions. Knowing that the first application of J73 was going to be for programming the avionics computers of the B-1, the J73 technical subcommittee felt the usual I/O facilities would be of little use. The type of I/O done by embedded computers, such as the avionics computers on B-1, is very specific to the application and involves operations like testing sensors and activating motors. Embedded avionics computers don't need facilities for ejecting pages, printing lines of data, creating files, and rewinding tapes.

The use of J2 and J3 to program the first compilers also had a considerable influence on the early versions of JOVIAL and resulted in features to access low-level machine details without resort to assembly language. This capability was provided principally by the functional modifiers, which have been retained in later versions of the language, although the same capabilities and functions have been somewhat changed, e.g. the SIGN function was dropped from J73 .

Another feature that allowed access to machine details was the DIRECT feature. This is one of the most controversial features of JOVIAL. It allowed programmers the ability to insert assembly language code directly in-line with JOVIAL code (see appendix C). The designers of J2 and J3 included this feature, because they did not want to get caught short handed if they encountered a problem that required assembly language.

Those in favor of the DIRECT feature cited the need for assembly language to do even minimal I/O. After I/O facilities were added to J3, the feature was retained. It must be remembered that in the early 60's, HOLs were not trusted to provide the right capabilities in all situations.

The argument against the DIRECT feature was that programmers could simply wrap an assembly language program in JOVIAL START/TERM statements and say their system was implemented in an HOL. Thus, the Air Force felt that the DIRECT feature could be used by a system developer to get around a contract requirement to program the system in JOVIAL and caused the feature to be dropped from J73 and later versions.

This was not to say that assembly language would never be needed. Most embedded computers, in fact, must used assembly language to interact directly with hardware devices. The current approach is to confine all assembly language to separate subroutines that are linked together with the JOVIAL programs that call them.

Thus, all machine-dependent assembly language is isolated in easily identifiable blocks of code for easier maintenance instead of sprinkled throughout every section of a program.

In considering the expected problem domain, language designer also take into account the anticipated ability of the programmer. This can be seen in the abbreviated style of the C programming language as compared to COBOL, which was designed to make the computer accessible to the unsophisticated business users, or Ada, which was designed for readability by maintenance programmers years after a system was developed. Since it was expected that J2 would be used by sophisticated programmers, a rich concise notation for the syntax was preferred to a loose English-like style such as that of COBOL [SHAW78]. This resulted in the rather cryptic syntax of the data declarations in J3 (see appendix C). Unfortunately, this style was carried over into J73 and its successors (see appendix D).

21.3 Hardware Constraints.

The influence of the expected hardware limitation, such as memory size and processor speed, interact somewhat with the constraints imposed by the application problem. The influences of the hardware are quite evident in other programming languages. For example, the LISP functions CAR and CDR came about because each instruction in the IBM 704 computer contained an address part and a

decrement part. The function CAR stood for "Contents of the Address part of Register number ..." and similarly for CDR [WEXE81].

Hardware influences are less important today than when JOVIAL was designed. The designers of J2 thought quite narrowly in terms of the SACCS program and the IBM 709 and Q-31 computers. SACCS required that the storage of large amounts of data in the limited memories of the computers used then. Several important features were added to JOVIAL to accomplish this such as the TABLE, the TABLE packing attributes and the functional modifiers.

Significantly, this power has been retained in all versions of JOVIAL (compare the REP function in J73 with the BIT and BYTE functions of J3, appendices D and C, section 6.3). JOVIAL is still one of the few HOLs that gives the programmer this much control over the internal representation of his data. It is one of the strongest features of the language and has contributed to JOVIAL's popularity for embedded computer applications.

Another example of the effect of the hardware environment is seen in the JOVIAL STOP statement and origin specification on the TERM statement (appendix C, sections 4.12 and 7.1). Few of the computer systems in the early 60's had operating systems. Each program ran on the computer by itself. The programmer frequently operated the computer himelf. On the STOP statement, the programmer could specify a statement label where execution was to resume if the operator restarted the program. This feature was useful for handling exceptional situations and for debugging.

Because the program ran by itself, the programmer could also specify the absolute machine location where his program should begin in the START statement in the first line of his program; otherwise, the compiler would chose the starting location. These two function were the first to go when JOVIAL compilers were designed to run under operating systems (see J5, chapter 8).

21.4 Support for Programming Methodology.

The idea that a programming language should provide features that support a particular style of programming and omit those that lead to error-prone and unstructured programming is relatively new. In the early 1960's, no one ever imagined that certain programming styles could lessen the cost of later changes (the maintenance process) or lessen the number of mistakes made during initial coding. Certainly the original designers of J2 were just interested in providing a useful programming tool.

The idea of structured programming was first introduced by Dijkstra in his seminal paper "Notes on Structured Programming [DIJK69] and was refined and generalized over the next five years by Wirth and Hoare [WASS80]. By the time J73 and later versions were designed, the need to provide for structured programming was recognized. Thus, J73 has the symmetrical IF-THEN-ELSE statement, a CASE statement (which was designed by Hoare [HOAR73] and first implemented in Pascal), and a loop statement with a WHILE clause.

The user-defined TYPE declaration facility for data abstraction and pointers for dynamic control of data allocation were also added to J73 to support data structures such as dynamic linked lists and binary trees.

Old habits die hard, and the battle over the GOTO resulted in few converts by 1973. Thus, J73B still retained the GOTO statement. Even more recently, Ada was designed with a restricted GOTO statement.

21.5 Other Technical Influences.

In addition to the above influences, the objectives of language flexibility, power, and generality had also influenced the design of many versions of JOVIAL and were reasonably well satisfied. In later versions, the objective of compactness and language elegance might be added.

In the early J2 version, these objectives led to reliability, speed, and code generation difficulties. But a decision was made to forego compilation speed for language flexibility and power. Schwartz later concluded that the flexibility and generality of the language were worth the initial difficulties with compiler speed [SCHW78].

Chapter 22

NON-TECHNICAL INFLUENCES

Non-technical influences can have a great impact on the final technical content of a programming language, many times determining the difference between its success or failure. These non-technical influences are more varied and not as easily categorized as the technical influences. Many times, rather than influencing a language directly, these factors have their effect by driving the technical influences.

The non-technical influences on language design have not been well studied. An important goal of this paper has been to point out how non-technical issues have affected the history of JOVIAL. In fact, the most important lesson learned from this study is that the non-technical influences on language development and change are much more important than previously realized. This chapter summarizes these issues to provide some insight on how these factor affect programming languages in general.

Based on the history of JOVIAL, some of the significant non-technical influences on language development seem to be standardization efforts, the use of formal specifications, language experience of the designers, schedule pressures, and design by an individual vs a committee.

22.1 Standardization.

One of the most common non-technical influences on JOVIAL has been language standardization efforts. Almost every effort to standardize JOVIAL has resulted in a proliferation of languages versions rather than a limitation of language variation. The same phenomenon has occured with other languages that were standardized after they were in use. However, this has not been true with Ada, possibly because the language was not used before efforts were begun to standardize it.

The creation of J3 was a result of SDC's decision to use JOVIAL as a corporate language. SDC, therefore, needed to improve J2 to make it more machine-independent. Again in 1965, SDC's policy to require that any new version that did not implement J3 exactly could not be called J3 resulted in the creation of the J4 and J5 dialects.

The effort to improve J3 in 1969 to gain ANSI approval resulted in the RADC committee, which created J73, the first non-upward compatible version of JOVIAL. The J73 subsets (J73/I, J73/II, and J73/III) were created to prevent a proliferation of non-standard versions. When J73/I was formalized as MIL-STD-1589, the language had been extensively revised. MIL-STD-1589 was more or less a different version than the original J73/I subset, causing the initial CSC J73/I compilers to differ from MIL-STD-1589 for a time.

The best example of how standardization attempts can lead to new language versions was the creation of J73A. RADC wanted to standardize on J73/I (MIL-STD-1589), but ASD felt that only J3B-2

was adequate for upgrading the B-52 Offensive Avionics System. To prevent a major rift just when a uniform language policy was desired, J73/I and J3B were merged to create J73A. J73A was, thus, created to effect a political compromise.

In retrospect, there seem to be two factors growing from standardization that affect language change. One is that standardization leads to efforts to clean up and perfect a language so that it will be acceptable as a standard. These efforts inevitably lead to changes. The other factor is that standardization forces separate user communities to unify their differences, which can result in compromise versions of a language.

22.2 Language Specifications.

J2 was developed without any preliminary detailed language descriptions. Moreover, the language was designed in parallel with the development of the compiler. This was not the case with other early languages. For example, a preliminary specification for FORTRAN I [IBM54] was completed and subjected to outside review two months before development of the compiler began in January 1955 [BACK78].

Other versions of JOVIAL were developed in a more conventional manner. The J3 specification was based on the description of J2 and was substantially completed before coding of the compiler began. The formal specifications for J4, J5 and J6 were worked out in great detail before any compilers were

constructed. Thus, the lack of a firm language specification prior to compiler construction was only a factor in the design of J2.

A different problem, but one related to the lack of a formal baseline prior to compiler construction, is the problem of having more than one language specification during compiler development. This problem has not been confined to JOVIAL and has affected almost every language ever developed.

The first time this happened with JOVIAL was when SDC began designing J3 even before the first J2 compilers were completed (chapter 5). It appeared again when CSC began development of the JOCIT compiler in 1973 but discovered that the revised J3 language was too different and implemented JOCIT for J3 (chapter 11).

Again when J73/I was specified, CSC began development of the J73/I compilers about 1974. But RADC changed the J73/I specification during compiler construction culminating with the publication of MIL-STD-1589 in February 1977. CSC attempted to implement changes as they became known, but there was still a period of catching up. The DEC-10 and HBC compilers did not conform to MIL-STD-1589 until some years after the first delivery (chapter 16).

The same situation occurred again when J73A was developed. During construction of the JOCIT compiler, J73/I was being revised. The Air Force had originally intended to make a few small changes which would have allowed completion of the compiler on schedule. However, the danger in such an approach always seems to be that the changes are much more extensive than originally thought.

The extensive changes in MIL-STD-1589A led to an initial slip in the JOCIT delivery schedule. So, one of the causes contributing to the failure of the JOCIT compiler was the lack of a firm language definition before beginning compiler development. That the original schedule was probably unrealistic to begin with and the attempt to meet the difficult optimization requirements of the MX MECA code generator also contributed to its failure.

Another contributing factor to the failure of JOCIT was the publication of yet another version of J73, MIL-STD-1589B, in June 1980, only a year after J73A had been defined. With the version of JOVIAL recognized by the JOCIT compiler already obsolete before the compiler's completion and with the availabiliy of the J73B compilers from SEA, there was little economic justification for RADC spending more money to complete JOCIT.

One thing that the failure of JOCIT did accomplish was to slow the development of new JOVIAL language standards. A standard seems to need to remain in existence at least 5-7 years to achieve acceptance. This can be seen with the ANSI FORTRAN and COBOL standards. ANSI requires that its language standards be re-examined every five years. With FORTRAN 66 this process began in 1971, but a new standard was not agreed upon until 1977. That the new Ada standard, ANSI/MIL-STD-1815A (22 January 1983), cannot change for at least five years will help to ensure Ada's success.

Although having a firm formal language specification does simplify design and construction of the compiler, the trade off is that a certain amount of naturalness and speed of development is

lost. Schwartz felt that co-development of the J2 language and compilers resulted in a language that was very natural for people to use [SCHW78]. On the other hand, he felt that designing with little time or forethought did result in some awkward syntax in the language. The language lacked elegance and was difficult to memorize, but once learned was rather easy to use.

The entire J2 language and 709 compiler were designed and implemented in a little over two years. In contrast, the development of Ada demonstrated the opposite extreme in perfecting the language before beginning compiler development. It took over seven years to develop the ANSI Ada standard, during which the now-famous series of requirements (STRAWMAN, WOODENMAN, TINMAN, IRONMAN, and STEELMAN) was written. Needless to say, a good deal of speed was sacrificed for perfection.

22.3 Experience of Designer.

It seems that a lack of previous language training or experience has not had a poor effect on the design of programming languages. Few of those who originally designed J2 had a foundation in language design, due to the shortage of people trained in language design. The same situation was true in the case of FORTRAN [BACK78]. Although the lack of formal training of the J2 design team did result in the formal definition of the language being ignored prior to and during development of the compiler, this problem has since been solved.

Those who worked on the J73 technical subcommittee and those who designed J73A at SofTech (chapter 18) did have a high degree of language experience but it is not clear that their work was so much better than earlier versions of JOVIAL. On the other hand, several members of the JOVIAL Users Group, which was responsible for the production of MIL-STD-1589B, had little previous language design training and yet were quite productive.

Therefore, if the language design team leaders have had some language training, it seems unnecessary that language designers need extensive experience and training in language design . In fact, the opposite may be true. The lack of training of Schwartz' design team may have been a source of new ideas. By not knowing what could be not be done, the team tried the impossible.

22.4 Schedule Pressures.

Another important part of the J2 development and possibly the development of other versions, was the pressure of the schedule. In the case of J2 the pressure was probably more extreme as the development of the rather expensive SACCS systems was dependent on completion of the compilers. Although the development of FORTRAN was also late (the original schedule of six months eventually took about two years) [BACK78], the only pressure was to get a product out to IBMs customers rather than the possibility of a multi-million dollar project failing.

In retrospect, the most important effect of the schedule pressure on J2 was that the J2 709 compiler was not integrated with the other J3 compilers. Distribution of the J2 7090 compiler kept multiple versions of the language around when SDC wanted to standardize on J3. Because the J2 7090 compiler was distributed to the FAA, IBM contracted CUC to develop a J2 compiler for the IBM 7090 for the National Airspace System contract. Thus, the FAA still uses a version of J2 to maintain the NAS programs (chapter 10). This wasn't the only thing that prevented standardization on J3 but it didn't help the situation.

The effect of schedule pressures was again significant in the creation of J3B by Boeing and SofTech for the B1 avionics systems. J3B was created because in 1972 it looked like J73 would not be ready in time for B-1. J3B was a bare-bones version of J3 and was created to be used until J73 compilers could be built.

As it turns out, CSC had a subset version of J73, J73/S, which they planned to use to build the first J73 JOCIT compiler, ready by December 1972 (chapter 11). This subset implemented more of the language than SofTech's first version of J3B-0 which was delivered in March 1973. Thus, B-1 could have used the CSC compiler and J3B need never have existed. But J3B lived on and was enhanced two more times (J3B-1 and J3B-2). Boeing has revived J3B for the initial delivery of the avionics for the B1-B program.

Since many software projects run into schedule problems, not much can be said about letting scheduling difficulties force decision to be made that will have long term effects. Schedules will always be a part of any language design effort. Of course, the trade off to the pressure of schedules is that development of the language takes a long time (e.g. eight years for development of Ada).

22.5 Lack of Committees.

Many of the most successful languages have been designed by a single individual or small group with the final say being the responsibility of a single individual. FORTRAN was designed mostly by Backus [BACK78]. Pascal was the sole effort of Niklaus Wirth [WITR71]. Every feature of Ada was given final approval by Jean Ichbiah, even though Ada was reviewed by more people than any other language in history. All of these languages are both characterized by a high degree of consistency and uniformity of style.

The original JOVIAL work was also an individual effort. There were relatively few people who actually designed the language. The fact that it wasn't a committee effort in many ways created the positive characteristics of the language as well as some of its negative ones [SCHW78]. The lack of committees during the development of J2 not only provided a certain consistency of style but also shortened development time. Schwartz felt that for initial efforts on language design, an individual effort was probably a good idea [SCHW78]. This lack of committee is one of the important lessons learned to be from the history of JOVIAL.

On the other hand, the committee approach has resulted in many language failures. PL/I was designed by the combined SHARE-IBM Advanced Language Design Committee and resulted in its large size and grab-bag array of features.

The lack of effectiveness of the committee approach with JOVIAL was demonstrated by the effort to modernize J3. The large RADC committee met for nearly three years without producing a language document, yet the small technical subcommittee was able to come up with what was considered quite a large language in a little over nine months (chapter 14).

Later in 1978, when J73A was specified, most of the work was accomplished by the JOVIAL Users' Group (JUG). During development of J73A, every paragraph of the language was considered and voted on by the eitire JUG, which number about 25-50 people at the time. Members of the JUG were primarily implementors; experience in language design was not a criterion for JUG membership. The result, according to one implementor, was that J73/I had been turned into an inconsistent and difficult to implement language. Thus, the result of this large committee input to the design of J73A was an increase in the complexity of the language with little increase in power.

Chapter 23

SUCCESS OF THE LANGUAGE

No version of JOVIAL has achieved great success. Despite a potential application area wider than any other language (until PL/I) and compilers for a large number of machines, J3 was not used much except for writing its own compilers and for military command-and-control applications [SAMM69]. Other early versions (J4, J5, J6 and J3B) were used even less. Likewise, the use of J73/I and its successors, J73A and J73B, has been confined to Air Force military systems, mainly for programming avionics and large ground-systems computers. By comparison FORTRAN IV, developed in 1964, is still extremely popular and only now is slowly being replaced by FORTRAN-77.

This chapter examines those factors that might have prevented JOVIAL from achieving wider success to see what lessons may be learned. Even though JOVIAL was not particularly successful compared with FORTRAN and COBOL, it has been more extensively used than was ever anticipated by the early developers, who thought of it in very short-range terms [SCHW78]. Considering that JOVIAL originally was only planned for SACCS 709 and Q-31 computers and had little support from SDC, JOVIAL succeeded quite well!

23.1 Technical Influences on Success.

23.1.1 Compiler Performance.

An important influence on the success of JOVIAL seems to have been the performance of its compilers. Backus attributed much of the success of FORTRAN to the early emphasis on object program efficiency rather than on language details [BACK78]. The FORTRAN I compiler was a single-pass compiler implemented in IBM 704 assembly code and was quite fast.

One of the reasons that PL/I failed to gain early widespread acceptance was due to the slowness of the first compilers. Because PL/I combined the capability of FORTRAN and COBOL, users were more inclined to use the faster FORTRAN or COBOL compilers, depending on the application, rather than wait for the early PL/I compilers.

There are two facets to the issue of compiler performance. One is the speed of the compiler; the other is the speed of the compiled object program. If the compiler is implemented in its own language, as were many JOVIAL compilers, the first parameter will necessarily be affected by the second.

Most of the early complaints about JOVIAL concerned the slowness of the compilers. Compiler speed and size was given little attention during development of J2. The first 709 compiler took eight hours to compile itself but was only 50,000 instructions long. The low reliability of the initial compilers compounded the lack of speed. The result was often long periods of waiting for

wrong or nonexistent results. These factors dampened early enthusiasm for the language [SCHW78].

Many things contributed to the slowness of the early compilers. The language was designed for generality; capabilities were added to the language without considering their potential effects on compilation speed. There was the rather low level of experience of the staff. Also, the compiler was programmed in its own language, and the computers used were extremely slow by today's standards.

Some improvement was achieved in the SDC J3 compilers; Schwartz stated that once the early compiler problems were overcome, the language was considered satisfactory [SCHW78]. However, in 1961, the Rand Corporation tested five programs in the area of command and control on both JOVIAL and FORTRAN. FORTRAN won on all of the array problems; JOVIAL compared favorably on the non-matrix problems. But JOVIAL was much worse in compilation time [HAVE63]. Also, the JX.2 and JS were later devised by Bratman to provide faster JOVIAL compilers (chapter 7).

By the time the CSC JOVIAL compilers were produced, the problems with slow compilers were alleviated by the increased speed of the computers available. But, the problem surfaced again with the J73A and J73B compilers. The baseline versions of the SofTech JOCIT compiler were very slow. The JOCIT compiler, however, was written using an interim, non-optimizing J73/I compiler, and SofTech claimed that when the compiler was self-compiled its speed would have improved greatly.

The performance of the SEA J73B compilers has been much better and SEA is making improvements to them all the time. Still, all of the modern JOVIAL compilers are large and slow compared with FORTRAN and Pascal compilers. This is mainly due to the complexity and large size of the language.

23.1.2 Lack Of I/O.

The lack of I/O in J2 may have prevented much initial use of JOVIAL. But by 1961, J3 had a reasonably adequate I/O capability and the language enjoyed a fairly widespread usage. However, I/O was again removed from J73 and has not been included in any later version. The lack of I/O is one of the first things that novice users of J73B notice.

This lack of I/O is probably the single most important factor that has prevented J73 and its successors from being used outside of the military embedded computer environment. Few scientific and commercial users are willing to write their own I/O routines. Many users have felt that the general lack of success of ALGOL-60 was the lack of I/O in the language. The lack of all but primitive I/O for Pascal bothers many users. In many respects, the I/O features of FORTRAN and COBOL have been the reason for the success of these languages. In fact, many users of the J73B compilers use the FORTRAN I/O run-time routines to provide an I/O capability.

23.1.3 Compiler Availability.

One aspect of insuring the success of any software product, including a programming language, is providing an implementation of the product for the most popular computers. This fact is well known by most software vendors. One reason for the great interest in Pascal and the success of BASIC has been that these languages are economically implementable on small desk-top computers. The initial popularity of FORTRAN was due as much to its being available on the very popular IBM 704 computer as to its being supported by IBM.

At least one version of JOVIAL, several versions in some cases, has existed on just about every major manufacturer's computers over the years. A minimum of ten companies have produced JOVIAL compilers. Yet the success of JOVIAL was not a foregone conclusion.

In the case of JOVIAL, J3 was prevented from achieving wider success because there was no compiler for the very popular IBM 709, 7090, 7094 computers. J73/I was created in order to allow compilers for more computers to be implemented. One of the arguments in favor of an Ada subset is that only large computers users will have access to it.

When J73/I was first implemented it was available only for the DEC-10 computer. Later the compiler was re-hosted to the IBM 360. If J73B achieves any degree of success, one reason will be that compilers are available for not only the DEC-10 and popular IBM 370 family, but within the last year compilers for the highly popular VAX 11/780 computers have become available.

23.2 Non-Technical Influences.

23.2.1 Vendor Support.

One reason often offered for JOVIAL's lack of major acceptance is the lack of direct support by a major vendor [SAMM69]. The existence or lack of vendor support, however, has not been determinant of the success of other languages.

JOVIAL was initially regarded as an SDC proprietary language. SDC used the language mostly on defense systems for the Air Force. By the time the Air Force published the J3 standard, AFM 100-24, J3 had come to be regarded as an Air Force language. Yet there was no substantial funding from either SDC or the Air Force for J3. All the JOVIAL compilers that were produced until CSC began building compilers were done under a government contract for a specific system.

FORTRAN was initially regarded as an IBM language but had the financial support of IBM to along with it. However, even with IBM's support PL/I never achieved great popularity.

It might seem that because the language was regarded as belonging to SDC and the Air Force (both non-profit organizations), other software vendors were reluctant to step in and support the language. However, Ada belongs to the DOD and at least 10 vendors are building Ada compilers without government support.

Computer manufacturers were not a major factor in producing or preserving the language initially, although versions of JOVIAL

have existed for almost all major and not-so-major computers since the start of JOVIAL. When most manufacturers were supporting COBOL and FORTRAN, JOVIAL maintained its progress. However, it was probably the long range lack of support that caused interest in JOVIAL to be dropped. Recently, the Air Force financed the construction of several J73B compilers. Whether such an effort will spell success for J73B remains to be seen. At least compilers for the VAX and other popular avionics computers will be available.

23.2.2 Standardization.

In 1980, the Air Force Systems Command required that all Air Force embedded computers be programmed in the current version of JOVIAL J73, which is now MIL-STD-1589B (chapter 20). Three years after this order was made, the effect has been a large increase in the usage of J73B. The SCF's Data Systems Modernization program (chapter 7) is using J73B. This contract is very large involving hundreds of programmers and costing several hundred million dollars. The ground systems for the MX missile system are being programmed in J73B, and several avionics systems have been programmed in JOVIAL.

Thus, one way to achieve success is to mandate it. The DOD plans to take the same approach with Ada. The only question is how soon to require the use of Ada. Many users have urged the DOD to wait until good compilers are available. AFSC's experience with J73 showed that if no compilers are available such an order can have costly effects on defense programs.

23.2.3 JOVIAL Culture.

One of the factors that seems to be necessary for the success of a language is that it achieve a certain culture about it. For example, APL and LISP are both regarded as rather successful languages, although neither language is as successful as, say, FORTRAN and COBOL. This is because of the culture surrounding their use. The best example of this effect is seen with the UNIX operating system originally implemented in 1971 on the DEC PDP-11 computer. For many years, UNIX received almost no recognition or support from Bell Labs. Yet, its popularity spread by word of mouth mainly in academic and research circles. This created a ground swell of interest and demand for the product, which forced Bell to provide an organization to market and distribute it. Establishing such a culture around a language or software product therefore seems to be an important way to achieve success.

This could be one of the reasons for the lack of JOVIAL's success. Schwartz almost personally designed J2. There was no time for review of the language. J3 was designed by a small group people working only within SDC. The same was true of J4, and J5. J6 was done for an Air Force contract. Widespread review of these versions was never even considered.

Later, when J73 was designed it received more extensive review by the members of the RADC committee. However, it was not until J73A was created that JOVIAL received anything that could qualify as widespread user review. This review was provided by JOVIAL users in the JUG. Appropriately, the members of the JUG were the first to support J73A and later J73B. After all, they themselves had voted on what features should be in the language. It will be interesting to see just how successful the Air Force has been in creating the proper culture for J73.

The DOD has carefully established a similar phenomenon for Ada. This was done by involving as many people as possible in reviewing the design of the language, compilers, and Ada Programming Support Environment. Thus, users are familiar with the language and are committed to its support even before the final version is produced.

Right now it seems that this approach has succeeded. People within the military community are predicting the success of the Ada due to its acceptance by industry and academia. On the other hand, industry and academia feel that Ada will be a success due to the support and heavy financial backing of the DOD. In the end, the Ada's success will likely be a result of the self-fulfilling prophesy.

Chapter 24

INFLUENCES OF JOVIAL ON LANGUAGE DESIGN

Having looked at the factors that have influenced JOVIAL, it seems fitting to conclude by examining what influence JOVIAL has had on other programming languages as well as the field of programming language and compiler design in general.

24.1 Contributions to Other Languages.

JOVIAL has served as a direct contributor to at least several languages. One language which is in use today in Europe is CORAL [WOOD70]. Many of the initial concepts for CORAL were taken from the J2 and J3 [SCHW78].

CSC's SYMPL language, which was used as a systems programming language to build several J3 as well as FORTRAN and ALGOL compilers (chapter 11), consisted of elements of JOVIAL and FORTRAN and other features developed at CSC [DUNB81].

Those involved in the production of J73A (MIL-STD-1589A) influenced the Ada effort in several ways. First, many of the JUG members who influenced the design of J73A provided extensive inputs to the several revision of Ada between the publication of the preliminary version in May 1979 and the acceptance of Ada as an ANSI standard in February 1983. At least two members of the JUG were

members of the elite Ada Distinguished Reviewers group and several others participated in the review of the July 1982 version. So, although features were not taken directly from J73, experience with J73A and J73B became a part of Ada.

The second way that JOVIAL influenced Ada was more direct. The success of the JOVIAL language control structure (chapter 17) provided a model for the establishment of the Ada Joint Program Office and subsequent management of configuration control over Ada. Also, the use of the JUG to review and provide user input to the MIL-STD-1589A effort served as a model for the way that the DOD conducted the world-wide review of Ada.

PL/I was directly influenced by JOVIAL [WEXE81, p594]. Fixed binary numbers, fixed binary arithmetic and the specified binary point came directly from JOVIAL, and the COMPOOL concept inspired the PL/I INCLUDE facility. Other languages borrowed features from JOVIAL but in a sufficiently different format that a direct connection with JOVIAL cannot be deduced. For example, CMS/2, which was developed by CSC for the Navy, was no doubt influence by CSC's experience with building JOVIAL compilers.

24.2 Contributions to Language Design.

Although referred to as a language for command and control, JOVIAL was one of the first (if not the first) system programming language and helped show that HOLs were capable of serving that purpose [SCHW78]. Right from the start, JOVIAL had constructs that permitted programming more than just mathematical computations. As

previously discussed, JOVIAL provided extensive features for access to subparts of items and machine words and for handling of text and bit data which suited it to systems programming applications.

J3 was less problem oriented than most languages of the time and helped to pave the way for more general HOLs like PL/I, Pascal, C, and Ada which can be used for both scientific and commercial applications as well as systems programming. Other areas that J3 was used for include writing a program to simplify JOVIAL source programs [CLAR67], creating a teaching program [MARS64], and automatic essay paraphrasing [KLEI65].

JOVIAL was later used for programming small embedded avionics computers and showed that an HOL was equally well suited to embedded applications (embedded computers are those that form part of a larger system, such as a missile or aircraft flight control computer). In fact, most usage of J73/I, J73A, and J73B has been for Air Force avionics applications.

JOVIAL's ability to provide low-level machine access at the HOL level and to pack large amounts of data efficiently into small memories led to J73 being chosen for the B-1 avionics systems. When J73 was late, J3B was developed as a substitute. This was the first use of JOVIAL for avionics and opened a new area for JOVIAL applications. The F-16 flight control computer was programmed in J3B-2. Later AFWAL developed the first J73/I compilers to program DAIS. Although there is no direct evidence, the use of JOVIAL for embedded computers probably contributed to the notion that an HOL,

namely Ada, could and should be developed for programming all DOD embedded applications.

Two important JOVIAL language constructs, the COMPOOL and the TABLE, appear to have influenced later language designs, such as SIMULA and Ada, although the connections with JOVIAL can only be surmised as little has been written on how these languages were designed, and technical literature does not usually give the source of language ideas. Both constructs originated with the SAGE program, but were most successfully implemented in JOVIAL.

Almost every version of JOVIAL has implemented some form of COMPOOL. The COMPOOLs in the J2 and J3 compilers were implemented by first copying the symbol TABLE following compilation of the COMPOOL out onto tape or disk and later copying it back in during compilation of the modules that accessed the COMPOOL data. This is also the form of the COMPOOL in the J73/I, J73A and J73B compilers. The J3B-0 compiler only provided copying of source code of data declarations into the source code followed by normal compilation.

The modern concepts of a data dictionary, the SIMULA CLASS feature, and the Ada PACKAGE feature are all extensions of the COMPOOL concept, the difference being that CLASSes and PACKAGEs may contain executable code and subroutines whereas COMPOOLs may only contain data declarations and (in J73) subroutine heading declarations. Like the COMPOOL, data declared in PACKAGEs and CLASSes is external to the modules that access them.

The JOVIAL TABLE was the first instance of the record type of data structure as implemented in Pascal and Ada. The implementation of similar structures in JOVIAL as Pascal could have been due to Wirth having knowledge of JOVIAL or could simply have been a similar solution to a common programming problem. A TABLE of entries is essentially an array of records. Each JOVIAL entry and each Pascal record may contain data of varying types. J2 and J3 had features for arrays as well as TABLEs. Later in J73, the array and TABLE concepts were combined by allowing TABLEs of only one item.

24.3 Contributions to Compiler Design.

The design and implementation of the JOVIAL compilers is now regarded as the norm for compiler design. The JOVIAL compilers were some of the earliest compilers to be written entirely in their own language. Since then, most JOVIAL compilers have been written in their own language. The advantage of this approach is that re-hosting the compilers is simplified and maintenance of the compiler can be accomplished by users without requiring a second compiler (see appendix B).

Another design feature of the JOVIAL compilers was the division of the compiler into a separate front end and code generator communicating via a machine independent intermediate language. This technique was first put forth by the UNCOL group (appendix B) and picked up by the CLIP group.

The JOVIAL compilers provided a more concrete example of the success of the UNCOL type of design than the CLIP compiler did because there was only one CLIP compiler. The construction of the first three J3 compilers was accomplished with a common front end and separate code generators. This design has been used over and over again for both JOVIAL and other languages. Whether its use for other languages was due to the use on JOVIAL is difficult to determine. Certainly, in the early 1960's, SDC was preeminent in computer science research with some of the world's top computer experts, who gave papers at many conferences. So, it seems entirely possible that SDC's success with this compiler design would be widely known.

24.4 Final Note.

All in all, although JOVIAL has not achieved the same stature as the more popular languages, its influence has been significant. Even more significant are the lessons learned from studying its history. The most important of these lessons is that the most significant effects on the design of a programming language and on its widespread acceptance are non-technical. Non-technical elements shaped JOVIAL's history and will continue to be an important influence on the programming languages in the future.

BIBLIOGRAPHY

This bibliography contains references cited in the text as well many other JOVIAL documents of historical interest. Most of the early JOVIAL documents were developed at System Development Corporation (SDC) and were cited in an earlier bibliography by Shaw [SHAW62C]. SDC's document numbering scheme had the following format: series-location-number/volume/revision. Thus, FN-LX-393/212/02 reads: Field Note, Lexington, number 393, volume 212, revision 2. TM stands for technical manual series. Location codes are LX: Lexington, MA; LO: Paramus, NJ; WD: Falls Church, VA; and no code: Santa Monica, CA.

[ABAC80] ABACUS Brochure, ABACUS Language Processor Development Experience, ABACUS Programming Corporation, Los Angeles, CA (1980).
Summarizes ABACUS' language and compiler experience.

[AFM67] Air Force Manual, Standard Computer Language for Air Force Command and Control Systems, (CED 2400), AFM 100-24, Washington, D.C. (15 June 1967).
Official description of the JOVIAL J3 language.

[AFR76] Air Force Regulation, Data Automation, Computer Programming Languages, AFR 300-10 (15 December 1976), Change 1 (14 February 1979), Change 2 (13 August 1979), Change 3 (21 May 1980).
Policy for procurement of programming languages and compilers. Implements DOD Directives 5100.40, 5000.29, 7900.1 , and DOD Manual 4120.3-M (1972).

[AFSC78] AFSC Supplement, Data Automation, Computer Programming Languages, AFSC Supp 1 to AFR 300-10 (14 December 1978), superceded 2 September 1980.
Establishes JOVIAL and MIL-STD-1750 control policy for AFSC.

[ANSI67] "JOVIAL Usage and Compilers (as of 8/67)", Appendix 1, Minutes of USASI X3.4 meeting (29 September 1967)
Contains list of JOVIAL compilers as of 1967.

[BACK78] Backus, J., "The History of FORTRAN I, II, and III", ACM SIGPLAN History of Programming Languages Conference, SIGPLAN Notices, 13(8) (August 1978) 165-180; also in [WEXE81], pp. 25-74, along with transcript of conference.

[BANH59] Banhagel, E.W.; R.W. Beeler; E. Book; H. Bratman; H.M. Isbitz; H.Manelowitz; and E. Myer, A Compiler and Language for Information Processing - Report #2 (CLIP), SD-3159, SDC, Santa Monica, CA (6 May 1959).
A technical description of the CLIP-1 language in terms used to describe ALGOL.

[BAUM81] Baum, C., The System Builders, SDC, Santa Monica, CA (1981).
A history of SDC from 1956 to 1980; written and published by SDC.

[BERN75A] Berning, P.T., A SEMANOL(73) Implementation Standard for JOVIAL (J73), RADC-TR-75-211-Vol-1 (of four), TRW Systems Group, Redondo Beach, CA (30 June 1975).
SEMANOL (Semantics Oriented Language) was improved and used to produce a syntactic and sematic specification of of J73 and J73/I. A SEMINOL interpreter for the RADC HIS-6180 under Multics was produced.

[BERN75B] Berning, P.T., A SEMANOL (73) Implementation Standard for JOVIAL (J73), SEMANOL (73) Reference Manual, RADC-TR-75-211-Vol-2, TRW Systems Group, Redondo Beach, CA (June 1975).

[BLEI64] Bleier, M. and H. Bratman, J-S Compiler Maintenance Manual, TM-1682/001/01, SDC, Santa Monica, CA (October 1964).

[BLOC62] Block, P.R., Description of the Format of the JOVIAL Compool for the Philco 2000, FN-6231/000/01, SDC, Santa Monica, CA (April 1962).
Maintenance manual for the JOVIAL COMPOOL for the 2000 compiler.

[BOCK59] Bockhorst, J. and J. Reynolds, Introduction to JOVIAL Coding, FN-LO-139, SDC, Lodi, NJ (September 1959).

[BOCK61] Bockhorst, J., JOVIAL I/O (7090), FN-LO-34-3, S1, SDC, Paramus, NJ (1961).

[BOEI73] Boeing, Higher Order Language Application Study, JOVIAL-73, (July 1973).

[BOOK60] Book, E. and H. Bratman, Using Compilers to Build Compilers, SP-176, presented at ACM meeting in Milwaukee, WI, 24 August 1960, SDC, Santa Monica, CA (31 August 1960).
Describes use of CLIP to implement the CLIP and JOVIAL compilers.

[BOOK62] Book, E. and V.L. Cohen, JOVIAL Generator Description: Computer Dependencies With Respect to the 1604 Compiler, TM-555/301/00, SDC, Santa Monica, CA (July 1962).
Maintenance manual for the 1604 compiler front end.

[BOOK63A] Book, E. and H. Bratman, Operating Instructions and Outputs of the J-X.2 One-pass Compiler and Loader, TM-970/003/01, SDC, Santa Monica, CA (25 April 1963).
Maintenance manual for the JX.2 compiler.

[BOOK63B] Book, E., H. Bratman, and J. Schwartz, Design Features and Structure of the J-X.2 One Pass JOVIAL Compiler, TM-970/004/00, SDC, Santa Monica, CA (9 October 1963).
A technical description of the J-X.2 JOVIAL compiler.

[BRAT59] Bratman, H., Project CLIP, SP-106, SDC, Santa Monica, CA (11 September 1959).
Describes modification of ALGOL to arrive at CLIP and the CLIP compiler.

[BRAT61A] Bratman, H. and H. Isbitz, Project CLIP - A Report and Summary, FN-3593-2, SDC, Santa Monica, CA (4 January 1961).

[BRAT61B] Bratman, H., "An Alternate Form of the 'UNCOL Diagram'", Comm. ACM, 4(3) (March 1961) 142.

[BRAT62] Bratman, H. and V. Cohen, Notes for JOVIAL Programmers (CDC 1604), The 1604 JOVIAL Library, FN-5140/010/00, SDC, Santa Monica, CA (June 1962).
Reference describing the 1604 compiler subroutine library.

[BRAT62] Bratman, H., Notes for JOVIAL Programmers (CDC 1604), Examples of ALGOL Transliterations, FN-5140/011/00, SDC, Santa Monica, CA (June 1962).
Example of two ALGOL programs translated into JOVIAL with compilation output listings in CODAP.

[BRAT62] Bratman, H. and L. Jacoby, Notes for JOVIAL Programmers (CDC 1604), The Current 1604 JOVIAL Compiler, FN-5140/012/00, SDC, Santa Monica, CA (June 1962).
Reference 1604 version of JOVIAL in terms of its difference from that described in [PERS66B].

[BRAT62] Bratman, H. and W.C. Brandstatter, The 1604 JOVIAL Operating System, FN-6752 & supplements, SDC, Santa Monica, CA (July 1962).
Reference manual for operating the 1604 compiler.

[BRAT62] Bratman, H., Notes for JOVIAL Programmers (CDC 1604), 1604 Compiler Error Messages, FN-5140/014/00, SDC, Santa Monica, CA (September 1962).
1604 compiler error messages.

[BRAT68] Bratman, H.; H.G. Martin; E.C. Perstein, "Program Composition and Editing with an On-line Display", AFIPS Fall Joint Computer Conference, Vol 33 (1968) 1349-1360.
Describes implementation of a display terminal with the SDC Interactive Programming Support System.

[BRAT81] Bratman, H., personal interview, SDC, Santa Monica, CA (23 November 1981).
Interview conducted by the author with Harvey Bratman.

[BREH62] Brehm, W.H., JOVIAL Programming Techniques, I/O Operations, FN-WD-255/008/01, SDC, Santa Monica, CA (July 1962).
Reference manual for 1604 compiler I/O operations. Contains examples with flow charts, JOVIAL LISTINGS, AND CODAP listings.

[BROS77] Brosgol, B.M.; R.E. Hartman; J.R. Nestor; M.S. Roth; and L.M. Weissman, Candidate Languages Evaluation Report, USACSC-AT-76-11, Intermetrics, Inc., Cambridge, MA (January 1977).
Evaluation of TACPOL, CS-4, J73/I, FORTRAN, COBOL, and PL/I with respect to the DoD "Tinman" Requirements for High Order Computer Programming Language

[BROW80] Brown, G.E., Past JOVIAL Compiler Development Experiences, internal document, Aerospace Corporation, El Segundo, CA (18 June 1980).
Survey of JOVIAL compilers used for Air Force space program.

[BUDE69] Budelman, F.C. and D.A. Barley, Program Production System Software Support and JOVIAL Compiler Quality Assurance, RADC-TR-69-265, SDC, Rome, NY (August 1969).

[BURL77] Burlakoff, M.; M.E. Hollowich; M.G. McClimens; and W.L. Trainor, "The Development of an Avionics Software Support System Using Modern Programming Techniques", Proceedings of the IEEE 1977 National Aerospace and Electronics Conference, NAECON 1977 (May 1977) 1011-1016.

[BURR65] Burroughes D-825, JOVIAL Compiler Characteristics, Burroughs Corporation (January 1965).

[CALL69] Callender, E.D. and N.W. Rhodus, J-3, PL/I and a Data Base, SAMSO TR-69-25, TR-0200(S9990)-4, Aerospace Corporation, San Bernardino, CA (1969).
JOVIAL J3 and PL/I were evaluated for space applications. PL/I was better because of its interrupt handling, asynchronous tasking, and dynamic memory allocation.

[CART62] Cartmell, D.J., The Intermediate Language (IL) Table, TM-555/050/00, SDC, Santa Monica, CA (2 January 1962). Reference manual for intermediate language for the Q-7, 2000, and 1604 JOVIAL compilers.

[CART62] Cartmell, D. and V. Cohen, JOVIAL Generator Description, Section V, Comp Dependencies with Respect to the S-2000 Compiler for SSRL, FN-6096/020/00, SDC, Santa Monica, CA (February 1962).
Reference manual describing machine dependencies of the 2000 compiler front end.

[CCIP78] AFSC Development Planning Study, "Information Processing/Data Automation Implications of Air Force Command and Control Requirements in the 1980s, Executive Summary", CCIP-85 (February 1972).

[CHAL77] Chalstrom, H.B. Jr., PALEFAC, AFAL-TR-77-167, Charles Stark Draper Lab, Inc, Cambridge, MA (September 1977).

[CHRI66] Christie, L.S. and D. Boreta, Interfacility Transfer of JOVIAL Code, N-23161/000/00, SDC, Santa Monica, CA (14 April 1966).

[CHU78] Chu, E.; E. Halb; H. McCoy; and R. Morton, Automated Code Generators for Compilers, RADC-TR-78-157, Computer Science Corporation, El Segundo, CA August (1978).

[CLAP74] Clapp, J.A. and J.E. Sullivan, SIMON: Finding the Answers to Software Development Problems, MTP-152, The MITRE Corporation, Bedford, Massachusetts (May 1974).

[CLAR62A] Clark, E., Error Detection in the Philco 2000 JOVIAL Translator, FN-64, SDC, Santa Monica, CA (April 1962). Reference manual for 2000 compler error messages.

[CLAR62B] Clark, E., Philco 2000 JOVIAL Subroutine Library, TM-555/202/00, SDC, Santa Monica, CA (April 1962). Description of Philco 2000 compiler subroutine library.

[CLAR62C] Clark, E., Phase I of the Philco 2000 JOVIAL Translator, TM-555/211/00, SDC, Santa Monica, CA (3 August 1962). Reference manual for Phase 1 of the 2000 compiler code generator.

[CLAR62D] Clark, E., Phase II of the Philco 2000 JOVIAL Translator, TM-555/212/00, SDC, Santa Monica, CA (August 1962).

[CLAR65A] Clark, E.R. and N. Isbitz, SURE User's Guide, TM-1354/445/01, SDC, Santa Monica, CA (9 March 1965). SURE examines a JOVIAL program and produces a new JOVIAL program which has been rearranged. Comments are added. Alphabetical listings of where each identifier is defined, set, used, initialized, or tested.

[CLAR65B] Clark, E.R., SURE, Set/Use Reformatter, TM-2295/041/00, SDC, Santa Monica, CA (23 June 1965). Describes the SURE processor which optimizes JOVIAL source code by deletion of redundant BEGIN/END brackets, redundant transfers of control, unused variables, unreferenced statement labels, unreachable statements, consecutive statement labels as well as simplification of path mode and Boolean expressions.

[CLAR67] Clark, E.R., "On the Automatic Simplification of Source-Language Programs", Comm. ACM, 10(3) (March 1967) 160-165. same as [CLAR65B]

[COCK70] Cocke, J. and J.T.Schwartz, Programming Languages and Their Compilers, Preliminary Notes, Second Revised Version, Courant Institute of Mathematical Sciences, New York University (April 1970). A text on the inner workings of a variety of programming languages, from a compiler point of view.

[COFF61] Coffman, E.G., Jr., A Brief Description and Comparison of ALGOL and JOVIAL, FN-5618, SDC, Santa Monica, CA (9 June 1961). An informal comparison of JOVIAL/J3 and ALGOL 60.

[COHE62A] Cohen, V.L. and M.H. Perstein, JOVIAL Generator Description, TM-555/020/00, TM-555/020/00A and TM-555/020/00B, SDC, Santa Monica, CA (July, August and September 1962) Reference manual for the Q-7, 2000, and 1604 compiler front end.

[COHE62B] Cohen, V.L., JOVIAL Generator Description: Computer Dependencies with Respect to the 1604 Compiler, TM-555/301/01, SDC, Santa Monica, CA (3 December 1962).

[COHE65] Cohen, V.L. and J. S. Hopkins, Phase I of the JOVIAL Generator (GEN1), TM-555/021/01, TM-555/021/01A, SDC, Santa Monica, CA (17 July 1965, 13 January 1966).

[COMP68] Computer Command and Control Co., Philadelphia, PA, Jovial Program Support System. Vol I. System Manual, RADC-TR-68-166-Vol 1 (May 1968).

[COND77] Condict, M.N.; C.A. Landauer; and J.M. Morris, High-Order Language Extensions for Concurrent Processing, RADC-TR-77-394, Pattern Analysis and Recognition Corp., Rome, NY (December 1977).
Describes extensions to JOVIAL for concurrent programming.

[COOP64] Cooperband, A. S., Print-formatting Routines for the Philco 2000 JOVIAL Library, SDC, Santa Monica, CA (28 Sept 1964).

[COZI62A] Cozier, W. and T. Dunbar, Q-7 JOVIAL Utility System, Part 7: Compiler Control Maintenance Manual, FN-6212/007/00, SDC, Santa Monica, CA (March 1962).
Reference manual for the Q-7 compiler control program.

[COZI62B] Cozier, W.A., Q-7 JOVIAL Utility System, Part 20: Compass Library Preprocessor User's Manual, FN-6212/020/00, SDC, Santa Monica, CA (August 1962)
Reference manual for the processor the Compass (Q-7 Assembly Program) subroutine library for use by the Q-7 compiler.

[CSCB78] Computer Sciences Corporation, Language Capability Brochure (1978).
Description of CSC past accomplishments in languages and compilers.

[CSCM77] Computer Sciences Corporation, JOVIAL J73/I Computer Programming Manual, Computer Sciences Corporation, El Segundo, CA (October 1977).

[DAVI62A] Davis. R.O., Q-7 JOVIAL Utility System, Part 6: JOVIAL Library User's Manual, FN-6212/006/00, FN-6212/006/00B, and FN-6212/006/00A SDC, Santa Monica, CA (March, May and July 1962).
Reference manual for the Q-7 subroutine library.

[DAVI62B] Davis, R.O., Q-7 JOVIAL Utility System, Part II: JOVIAL Prestore Update User's Manual, FN-6212/011/00, SDC, Santa Monica, CA (August 1962).
Reference manual on how to update prestored JOVIAL programs.

[DAVI62C] Davis, R.O., Q-7 JOVIAL Utility System, Part 1: Q-7 JOVIAL PROGRAMMING Manual, FN-6212/001/01, SDC, Santa Monica, CA (September 1962).
Q-7 JOVIAL language reference manual.

[DAVI62D] Davis, R.O., Q-7 JOVIAL Utility System, Part 21: Q-7 JOVIAL Utility System User's Manual, FN-6212/021/00, SDC, Santa Monica, CA (October 1962).
Q-7 operating manual.

[DAVI62E] Davis, R.O. & W. A. Cozier, Q-7 JOVIAL Utility System, Part 31: Assembler Maintenance Manual, FN-6212/031/00, SDC, Santa Monica, CA.
Reference manual for the assembler in the Q-7 compiler.

[DEAS62] Deason, J.A., The Use of the Arithmetic Section of the 1604 JOVIAL Compiler, FN-WD-255/006/00, SDC, Santa Monica, CA (March 1962).
Programming manual for effective use of arithmetic formulas in the 1604 JOVIAL compiler.

[DEVI78] DeVine, T.; L. Hyde; H. McCoy; and R. Rushall, Improvements to JOCIT, RADC-TR-78-144, Computer Sciences Corporation, El Segundo, CA (October 1978).

[DIJK69] Dijkstra, E.W., Notes on Structured Programming, in Dahl, Dijkstra, and Hoare, Structured Programming, Academic, NY (1972).

[DINI78] Dinitto, S.A., Jr., "High Order Language Standardization", Proceedings of the IEEE 1978 National Aerospace and Electronics Conference, NAECON 1978 (May 1978) 1139-46.

[DOBK68] Dobkin, Richard M., et. al., An Investigation of Advanced Programming Techniques, RADC-TR-68-367, SDC, Rome, NY (October 1968).

[DOBR62A] Dobrusky, W.B., Central JOVIAL Procedure Library, FN-6184/000/00, SDC, Santa Monica, CA (January 1962).
Reference manual for the central, subroutine reference library.

[DOBR62B] Dobrusky, W. B., Design for JOVIAL Compiler for the Small Computer, TM-739/000/00, SDC, Santa Monica, CA (7 August 1962).
Describes a J3 subset for implementation on small computers in terms of differences with [PERS68].

[DODD76] Department of Defense, Directive 5000.29, (29 April 1976).
Establishes policy for the management and control of computer resources during the development, acquisition, deployment and support of major Defense systems.

[DODI76] Department of Defense, Instruction 5000.31, (24 November 1976).
Specifies the High Order Programming Languages which are approved for use in conjunction with DOD Directive 5000.29.

[DODM77] Department of Defense, Military Standard, JOVIAL (J73/I), MIL-STD-1589 (28 February 1977).
Official description of the J73/I language.

[DODM79] Department of Defense, Military Standard, JOVIAL (J73), MIL-STD-1589A, supercedes MIL-STD-1589 (15 March 1979).
Official description of the J73A language.

[DODM80] Department of Defense, Military Standard, JOVIAL (J73), MIL-STD-1589B, supercedes MIL-STD-1589A (6 June 1980).
Official description of the J73B language.

[DREW69] Drews, C. M., Raytheon-UNIVAC 1108 JOVIAL Compiler, Program System Description, TM-WD-1506/001/00, SDC, Falls Church, VA (22 October 1969).

[DUNB75] Dunbar, Terry L., JOCIT JOVIAL Compiler Implementation Tool, RADC-TR-74-322, Computer Sciences Corporation, Los Angeles, CA (January 1975).
Final technical report on the JOCIT system.

[DUNB81] Dunbar, T.L., personal interview, Software Engineering Associates, Torrance, CA (30 October 1981).
Notes from several interviews conducted by the author.

[EANE77] Eanes, R.S.; J.B. Goodenough; J.R. Kelly; D.K. Klotzbach; and A.J. Nickelson, Communications High-Order Language Investigation, RADC-TR-77-341, Softech Inc., Waltham, MA (October 1977).
IRONMAN, J73C (new JOVIAL language/criterion), J73/I, PASCAL, PL/I, CS-4, and J3 were evaluated as a basis for a communications HOL. J73/I was selected.

[ELSP76] Elspas, B.; R.S. Boyer; R.E. Shostak; and J.M. Spitzen, A Verification System for JOVIAL/J73 Programs (Rugged Programming Environment - RPE/1), RADC-TR-76-58, Stanford Research Inst., Menlo Park, CA (March 1976).

[ELSP77] Elspas, B., et. al., A Verification System for JOCIT/J3 Programs (Rugged Programming Environment - RPE/2), RADC-TR-229 (June 1977).

[ELSP79] Elspas, B.; M.W. Green; M.S. Moriconi; and R.E. Shostak, A JOVIAL Verifier, RADC-TR-79-195 (July 1979).
Describes a program verification system supporting design, development, and formal verification of J73/I programs. Programs designed using the SRI Hierarchical Development Methodology (HDM) were verified. JOVIAL constructs that could lead to aliasing were excluded. DEFINES were removed.

[ENGE71] Engel, Frank Jr., The Air Force JOVIAL Compiler Validation System (JCVS), ESD-TR-71-236, Mitre Corporation, Bedford, MA (August 1971).

[ENGL61] Englund, D. and E. Clark, "The CLIP Translator", Comm. ACM, 4(1) (January 1961), 19-22; also referenced as FN-4455, SDC, Santa Monica, CA (31 October 1960). Describes the code generator for the CLIP IBM 709 compiler.

[FELT82] Felty, J., personal correspondence, SofTech, Inc., Waltham, MA (February 1982).

[FISH78] Fisher, D.A., "DOD's Common Programming Language Effort," Computer (March 1978) 24-33.

[FLEI72] Fleiss, J.E.; G.W. Phillips; A. Edwards; and L. Rieder, Programming for Transferability, RADC-TR-72-234, International Computer Systems Inc., Los Angeles, CA (September 1972).
Gives techniques for easing the transfer of software (FORTRAN, JOVIAL, and COBOL programs in particular) from one computing environment to another. Specific suggestions for improvement of program design are included.

[FLEI73] Fleiss, J.E. and G.W. Phillips, JOVIAL J73 Programmers' Reference Manual, Proprietary Software Systems, Beverly Hills, CA (January 1973).

[FLEI74] Fleiss, J.E. and G.W. Phillips, A Statistics Gathering Package for the JOVIAL Language, RADC-TR-73-381, Proprietary Software Systems, Inc., Beverly Hills, CA (January 1974).
A study of statistics and data collection techniques resulting in requirements for a JOVIAL 73 statistics gathering package.

[FLEI77] Fleischer, R. and R. Spitler, "A Project Management System for Software Development," 15th IEEE Computer Society International Conference, Fall COMPCON (September 1977) 110-114.
The Software Implementation Monitor (SIMON) collects information on project status and produces reports for use by managers and programmers. The first implementation is tailored for JOVIAL.

[GANN77A] Gannon, C.; N. B. Brooks; and R. J. Brooks, JAVS Technical Report: User's Guide, RADC-TR-77-126-Vol-1, General Research Corporation, Santa Barbara, CA (April 1977).

[GANN77B] Gannon, C. and N.B. Brooks, JAVS Technical Report: Reference Manual, RADC-TR-77-126-Vol-2, General Research Corporation, Santa Barbara, CA (April 1977).

[GANN77C] Gannon, C. and N.B. Brooks, JAVS Technical Report: Methodology Report, RADC-TR-77-126-Vol-3, General Research Corporation, Santa Barbara, CA (April 1977).
Describes the JOVIAL/J3 Automated Verification System (JAVS)

[GANN77D] Gannon, C.; N.B. Brooks; W.R. Wisehart, JAVS Final Report, RADC-TR-77-201 (June 1977).
Describes the results of the test and evaluation of JAVS at RADC on a large and complex operational program, the current configuration of JAVS, and proposed capabilities of future tools.

[GANN77E] Gannon, C., "Verification Case Study", AIAA/NASA/IEEE/ACM Computers in Aerospace Conference, A Collection of Technical Papers, Los Angeles, CA (October 1977) 349-353.
Gives an overview of JAVS. JAVS assists in the testing and documentation of large JOVIAL programs by performing static and dynamic program analysis via instrumented source code.

[GANN80] Gannon, C., JOVIAL J73 Automated Verification System - Study Phase, RADC-TR-80-261, General Research Corporation, Santa Barbara, CA (August 1980).

[GENE70] General Electric, The GE-600 Line JOVIAL, CPB-1650, General Electric Co., Information Systems, Phoenix, AZ (1970)
Technical data on the GE-625/635 JOVIAL/J3 compiler developed by DigiTec and PRC for GE.

[GILB66A] Gilbert, P.; D.M. Gunn; and C.L. Schager, Automatic Programming Techniques, RADC-TR-66-54, Teledyne Systems Co., Hawthorne, CA (July 1966).

[GILB66B] Gilbert, P. and W.G. McLellan, Compiler Generation Using Formal Specificaion of Procedure-Oriented and Machine Languages, RADC-TR-67-454, Measurement Analysis Corporation (July 1966).
Describes a compiler generator generates both the front end and code generator. The system can produce ALGOL, FORTRAN, and JOVIAL compilers.

[GILB67] Gilbert, P.; D.M. Gunn; L. Craig and W. Testerman, Automatic Programming Techniques Volume I, RADC-TR-66-665-Vol-1, Teledyne Systems Co., Hawthorne, CA (January 1967).

[GILI62] Gilinsky, R.J., Q-7 JOVIAL Utility System, Part 22: Q-7 JOVIAL Tape File Maintenance User's Manual, FN-6212/022/00, SDC, Santa Monica, CA (October 1962).
Reference manual for the JOVIAL file maintenance program for binary tape.

[GODD78] Goddard, G.; M. Whitworth; and E. Strovink, JOVIAL Structured Design Diagrammer (JSDD). Volume I, Report Summary, RADC-TR-7809-Vol. 1, Charles Stark Draper Lab, Inc., Cambridge, MA (February 1978).

[GOOD76] Goodenough, J.B., An Exploratory Study of Reasons for HOL Object Code Inefficiency, ECOM-75-0373-F, SofTech, Inc., Waltham, MA (August 1976).
Two real-time system were rewritten in J3B to study programmer use of the language. Having to code around compiler inefficiencies adversely affected source code readability and style.

[GRAN64] Grant, V., JOVIAL Documentation and Compiler Distribution, N-19493/000/01, SDC, Santa Monica, CA (14 January 1964).
Lists all recipients who have been issued JOVIAL compilers and documentation as of December 1963.

[GRIF72] Griffiths, Thomas R., NMCSSC JOVIAL J5.2 Compiler and JOVIAL-NIPS Interface, NMCSSC-TM-70A-71, National Military Command System Support Center, Washington, DC (December 1972).

[GRIF62] Griffitts, V.A., Errors Detected by the Philco 2000 JOVIAL, TM-555/203/00, SDC, Santa Monica, CA (October 1962).
Reference manual for 2000 compiler error messages.

[HAGG64] Haggerty, D.P., Philco 2000 JOVIAL Subroutine Library, TM-555/202/01, SDC, Santa Monica, CA (13 May 1964).

[HATH77] Hathaway, A. and P.L. Loring, AFSC HOL Standardization Program: Strawman Policy and Implementation Plan, WP-21476, MITRE (30 Sept 77).

[HATH77A] Hathaway, A. and P.L. Loring: High Order Language Standardization Task: Year End Summary FY77, MTR-3485, Mitre Corp., Bedford, MA (30 September 1977).

[HAVE63] Haverty, J.P. and R.L. Patrick, Programming Languages and Standardization in Command and Control, RM-3447-PR (DDC AD-296 046), Rand Corporation, Santa Monica, CA (1963).

[HAVE64] Haverty, J.P., Programming Language Selection for Command and Control Applications, P-2967, RAND Corp., presented at Symposium on Computer Programming for Military Systems, SHAPE Technical Centre, The Hague, Netherlands (2 October 1964).
The Air Force Assistant for Data Automation compared ALGOL, CL-1, COBOL, FORTRAN II & IV, JOVIAL J2 & JX.2, MAD, NELIAC, and SCAT assembly language on the 7090. JOVIAL was best in execution time, worst in compilation speed.

[HAYE62] Hayes, E., Philco 2000 JOVIAL String Routines, TM-555/201/00, SDC, Santa Monica, CA (17 April 1962).
Reference manual on string manipulation subroutines of the 2000 compiler.

[HAYE63] Hayes, E., Recent Improvements to the Philco 2000 JOVIAL Translator, TM-555/204/00, SDC, Santa Monica, CA (9 July 1963).

[HAYE64] Hayes, E., Additions to the Philco 2000 JOVIAL Compiler, TM-555/205/00, SDC, Santa Monica, CA (10 September 1964).

[HAYS77] Hays, G.G., "Dips and Flatpacks Come to Software Design", Proceedings of the IEEE 1977 National Aerospace and Electronics Conference, NAECON 1977 (May 1977) 410-415.
Just as hardware designers use multi-device packs, Westinghouse software designers are using fully database driven software packages for its EAR operational software and JOVIAL compiler.

[HIR67A] Hirschfield, G.A., and C.J. Shaw, et.al., Spaceborne Software Systems Study, Vol III, Recommendation for a Common Space Programming Language, SSD-TR-67-11-Vol-3, SDC, Santa Monica, CA (January 1967).
Describes the benefits of using a common HOL for space. An evaluation of existing languages was made; a description of recommended language is given.

[HIR67B] Hirschfield, G.A., and C.J. Shaw, et.al., Spaceborne Software Systems Study, Vol IV, Survey of Software Techniques, SSD-TR-67-11-Vol-4, SDC, Santa Monica, CA (January 1967).
A collection of techniques applicable to development of software in general and to spaceborne software in particular.

[HIRS68] Hirschfield, G.A., and L.J. Carey, Space Programming Language (SPL/J6) Programmer's Manual, SAMSO-TR-68-383, SDC, Santa Monica, CA (November 1968).
Programmer's manual describing SPL/JOVIAL 6 and how to use it.

[HOAR73A] Hoare, C.A.R., Critique of the Standard Computer Programming Language JOVIAL (J73), (1973).

[HOAR73B] Hoare, C.A.R., Hints on Programming Language Design, Stanford University Computer Science Department Technical Report CS-73-403 (December 1973).

[HOAR75] Hoare, C.A.R., Hints on the Design of a Programming Language for Real-time Command and Control, (1975).

[HOGA81] Hogan, M.O., JOVIAL J73 Status Report, TOR-0081(6902-03)-1, Aerospace Corporation, El Segundo, CA (25 March 1981).

[HOLD77] Holden, M.T., "JOVIAL/J3B Compiler Optimization for Time Critical B-1 RFS/ECMS Application", Proceedings of the IEEE 1977 National Aerospace and Electronics Conference, NAECON 1977 (May 1977) 486, abstract only, complete paper obtained from the author.

[HOPK39] Hopkins, J.S., POOL--The COMPOOL Search Program for the Philco 2000 JOVIAL Compiler, TM-555/207/00, SDC, Santa Monica, CA (18 November 1964).

[HOWE60] Howell, H.L., JOVIAL -- Variable Definition for the 709 Translator, Rep. FN-LO-34-2-52, SDC, Santa Monica, CA (1960).

[HOWE62] Howell, H.L.; H. Isbitz; and J.I. Schwartz, Documentation of the Jovial Language and Compiler for the IBM 7090 Computer, FN-6223 and supplements, SDC, Santa Monica, CA Reference manual for the 7090 compiler and the language it accepts.

[IBM54] IBM Programming Research Group, Applied Science Division, Specifications for the IBM Mathematical FORmula TRANslating System, FORTRAN, preliminary report, IBM Corp., New York (1954).

[IBM68] IBM/FAA Programming Project Office, IBM 9020 Data Processing System JOVIAL Language Reference Manual, FAA-64-WA-5223, Atlantic City, NJ (August 1968).

[IBM71] IBM Corp., The System/360 JOVIAL Compiler, Program Information Department, Contributed Program Library Type III, 360-03.2.010, IBM Corp., Hawthorne, NY.
Reference manual for the JOVIAL compiler hosted on the IBM 360/50 at MITRE used to test the JCVS.

[IBM76] IBM Corp., JOVIAL/J3B Language Specification and Reference Manual, IBM No. 6122801 (1976).

[IDA61] Institute for Defense Analyses, Computers in Command Control, Tech. Rept. 61-12 (November 1961).

[ISBI59] Isbitz, H., CLIP: A Compiler Language for Information Processing, SP-117, SDC, Santa Monica, CA (October 1962).

[ISBI62] Isbitz, H., Detailed Description of Various Jovial 7090 Library Routines, FN-6343/000/00A, SDC, Santa Monica, CA (25 May 1962).
Reference manual for the 7090 subroutine library.

[JACK62] Jackson, C.W., and W. Haueisen, Q-7 JOVIAL Utility System, Part 8: Translator Maintenance Manual, FN-6212/008/00 & FN-6212/008/00A, SDC, Santa Monica, CA (May and June 1962).
Reference manual, with flow charts, for Q-7 code generator.

[JACK74] Jackson, C.W., and L. Danberg, The Improved JCVS, RADC-TR-74-232, Abacus Programming Corp., Santa Monica, CA (September 1974).

[JACO62A] Jacoby, L., Calling Sequences for 1604 Library Subroutines, FN-WD-255/007/00, SDC, Santa Monica, CA (January 1962).
Reference manual for the 1604 subroutine library with listings.

[JACO62B] Jacoby, L., Introduction to Maintenance Documents for the 1604 JOVIAL Translator, FN-WD-5545/528/00, SDC, Santa Monica, CA (February 1962).
Reference manual for 1604 code generator.

[JACO62C] Jacoby, L., JOVIAL Programming Techniques (1604), Programming Techniques for Compiling JOVIAL Programs, FN-WD-255/003/01, SDC, Santa Monica, CA (July 1962).
Operation manual for the 1604 compiler.

[JACO62D] Jacoby, L., 1604 JOVIAL Compiler, JOVIAL Programming Guide, TM-WD-555/301/00, SDC, Santa Monica, CA (October 1962).
JOVIAL manual for the 1604 compiler.

[JAME72] James, T.A.; B.C. Hall; and P.M. Newbold, Advanced Software Techniques for Data Management Systems, Final Report, Vol 3: Programming Language Characteristics and Comparison Reference, NASA-CR-115515, Intermetrics, Inc., Cambridge, MA (February 1972).
9 languages of aerospace programming interest were evaluated: PL/I, HAL, J3, SPL/J6, CLASP, ALGOL 60, FORTRAN, and MAC 360.

[JOVI73] Standard Computer Programming Language JOVIAL (J73), draft specification (1 January 1973).

[JUG78A] JOVIAL User's Group, Minutes of the First (Fall 1978) Meeting, Wright-Patterson AFB, OH (4-5 October 1978).

[JUG78B] JOVIAL User's Group, Minutes of the Second (Fall 1978) Meeting, Wright-Patterson AFB, OH (13-14 November 1978).

[JUG79A] JOVIAL User's Group, Minutes of the Third (Winter 1979) Meeting, Fort Worth, TX (6-7 February 1979).

[JUG79B] JOVIAL User's Group, Minutes of the Fourth (Spring 1979) Meeting, El Segundo, CA (1-2 May 1979).

[JUG79C] JOVIAL User's Group, Minutes of the Fifth (Summer 1979) Meeting, Seattle, WA (24-25 July 1979).

[JUG79D] JOVIAL User's Group, Minutes of the Sixth (Fall 1979) Meeting, Waltham, MA (30-31 October 1979).

[JUG80A] JOVIAL User's Group, Minutes of the Seventh (Winter 1980) Meeting, Goleta, CA (8-9 January 1980).

[JUG80B] JOVIAL User's Group, Minutes of the Eighth (Spring 1980) Meeting, Dayton, OH (15-16 April 1980).

[JUG80C] JOVIAL User's Group, Minutes of the Ninth (Summer 1980) Meeting, El Segundo, CA (15-16 July 1980).

[JUG80D] JOVIAL User's Group, Minutes of the Tenth (Fall 1980) Meeting, Minneapolis, MN (7-8 October 1980).

[JUG81A] JOVIAL-Ada User's Group, Minutes of the Eleventh (Winter 1981) Meeting, Tampa, FL (20-21 January 1981).

[JUG81B] JOVIAL-Ada User's Group, Minutes of the Twelevth (Spring 1981) Meeting, Sacramento, CA (14-15 April 1981).

[JUG81C] JOVIAL-Ada User's Group, Minutes of the Thirteenth (Summer 1981) Meeting, Johnson City, NY (14-15 July 1981).

[JUGM81D] JOVIAL-Ada User's Group, Minutes of the Fourteenth (Fall 1981) Meeting, Baltimore, MD (13-14 October 1981).

[KAPP59] Kappler, M.O., Computer Programming for Command Control Systems, SP-123, SDC, Santa Monica, CA (30 October 1959).

[KEEL70] Keeler, F.S.; A.P. Grebert; and D.A. Nelson, Computer Architecture Study, SAMSO-TR-70-420, Information and Communication Applications, Inc., Silver Springs, MD (October 1970).

[KENN62] Kennedy, P.R., A Simplified Approach to JOVIAL, TM-780/000/00, SDC, Santa Monica, CA (1962).
An primer and reference manual for the beginning JOVIAL programmer.

[KENN65] Kennedy, P. R., The TINT Users' Guide, TM-1933/000/02, SDC, Santa Monica, CA (8 March 1965).
Describes how to the On-line Teletype JOVIAL INTerpreter (TINT) on SDC's Time-Sharing System (TSS).

[KETC81] Ketchum, D., personal interview, IBM/Federal Systems Division, Owego, NY (14 October 1981).
Interview and personal papers obtained by the author.

[KLEI64] Klein, P. E., Results of a Study of the Bidirectional Transfer of Computer Programs between the IBM-7090 and the CDC-1604A, TM-WD-423/000/00, SDC, Falls Church, VA (12 August 1964).

[KLEI65] Klein, S., "Automatic Paraphrasing in Essay Format", Mechanical Translation, 8 (3 & 4) (June and October 1965) 68-83.
Use of JOVIAL for paraphrasing essays.

[KNEE62] Kneemeyer, J.A., The Translator Pass of the 1604 JOVIAL Compiler, FN-WD-5545/530/00, SDC, Santa Monica, CA (March 1962).
Reference manual describing the second pass of 1604 code generator.

[KOCH62] Koch, M.L., and L. Owens, JOVIAL Training Manual, FN-LO-762 and supplements. SDC, Santa Monica, CA (September 1962).
Manual for JOVIAL/J2 as implemented by the Q-31v compiler.

[LCFB80A] LCF Bi-Monthly Newsletter, J73 Compiler Developements, 2(4), JOVIAL Language Control Facility, Rome Air Development Center (RADC/ISISL), Griffiss AFB, NY (August 1980).
Contains listing of currently available J73 compilers.

[LCFB80B] LCF Bi-Monthly newsletter, J73 Compiler Developements, 2(5), JOVIAL Language Control Facility, Rome Air Development Center (RADC/ISISL), Griffiss AFB, NY (August 1980).
Contains listing of currently available J73 compilers.

[LCFB80C] LCF Bi-Monthly newsletter, J73 Compiler Developements, 2(6), JOVIAL Language Control Facility, Rome Air Development Center (RADC/ISISL), Griffiss AFB, NY (August 1980).
Contains listing of currently available J73 compilers.

[LALO71] LaLonde, W.R., "An Efficient LALR Parser Generator," University of Toronto, Technical Report CSRG-2 (February 1971).

[LICK60] Licklider, J.C.R., "Man-Computer Symbiosis," IRE Transactions on Human Factors in Electornics, V.HFE-1 (March 1960) 4-10.

[LORI76] Loring, P.L.; E.A. Lamagna; and L.J. LaPadula, Programming Languages, Standards, Use, and Selection: Air Force and Other Government Agencies, Vol. I Analysis and Vol. II System Summaries, MTR-3346, MITRE Corp., Bedford, MA (30 September 1976).

[MARS64] Marsh, D.G., "JOVIAL in Class", Annual Review in Automatic Programming, Vol. 4 (R. Goodman, ed.), Macmillan, New York (1964) 176-181.
Describes use of JOVIAL for a teaching program.

[MANE59] Manelowitz, H., ANCHOR, An Algorithm for Analysis of Algebraic and Logical Expressions, SP-127, SDC, Santa Monica, CA (9 November 1959).
ANCHOR is an efficient method for processing expressions involving complex logical decision. ANCHOR reduces Algebraic and Logical expressions to an ordered sequence of operand-operator-operand triplets.

[MATY77] Matysek, T.E., "HOL in Operational Software - From a User's Point of View", Proceedings of the IEEE 1977 National Aerospace and Electronics Conference, NAECON 1977 (May 1977) 494-501.

[MATY78] Matysek, T.E., "The Versatility of JOVIAL J73 in Avionics Systems", Proceedings of the IEEE 1978 National Aerospace and Electronics Conference, NAECON 1978 (May 1978) 928-33.

[MCIS62] McIsaac, P.V., 2000 JOVIAL Programming Tips, FN-LX-394/201/00, SDC, Santa Monica, CA (September 1962).
Informal JOVIAL reference manual for the 2000 compiler.

[MCLE67] McLellan, W.G., and P. Gilbert, Compiler Generation Using Formal Specification of Procedure-Oriented and Machine Languages, RADC-TR-67-454, Rome Air Development Center, Griffiss, AFB, NY (August 1967).

[MELA56] Melahn, W. S., et.al., "PACT I", (A series of 7 papers), J. ACM 3(4), (1956) 266-313.

[MEYE62] Meyer, W.E., Distribution Information for the Philco 2000 JOVIAL, FN-6867/000/00, SDC, Santa Monica, CA (October 1962).
Reference describing the 2000 compiler master tape.

[METR80] Metropolis, N.; J.Howlett; and G-C. Rota, A History of Computing in the Twentieth Century, Academic Press, New York, (1980).
Papers presented at the International Research Conference on the History of Computing, held at the Los Alamos Scientific Laboratory, 10-15 June 1976.

[MITR62] FAST -- Fortran Automatic Symbol Translator (reference manual), SR-24, MITRE Corporation, Bedford, MA (January 1962).
Describes use of the COMPOOL before JOVIAL.

[MOLT68] Moltzau, LtJG, JOVIAL Procedure Library, FOCCPAC-TN-3, Navy Fleet Operations Control Center (April 1968).
Lists procedures and functions in the JOVIAL Procedure Library at FOCCPAC.

[MOOR62] Moore, B.B., JOVIAL Training Problems, FN-WD-430/001/00, SDC, Santa Monica, CA (August 1962).
Training manual with programming problems for the 1604 compiler.

[MUHL73] Muhlhauser, R.R., DM-1 Implementation, RADC-TR-73-68, Auerbach Corp., Phildelphia, PA (March 1973).
Data Manager-1 (DM-1) is a data base management system implemented in JOVIAL on a Honeywell G-635 under the GCOS-III operating system.

[NAUR60] Naur, P., Ed., "Report on the Algorithmic language ALGOL 60", Comm. ACM, 3(5) (May 1960).
Known as the ALGOL report, introduced the Backus-Naur Form (BNF) for describing the syntax of ALGOL.
dont have

[NAUR63] Naur, P., Ed., "Revised report on the Algorithmic language ALGOL 60", Comm. ACM, 5(1) (January 1963).
Corrected many of the ambiguities in the original report.

[NEEB62A] Neeb, D., The DPC-JOVIAL-To-DPC Compiler Error Message List, FN-LO-581/000, SDC, Paramus, NJ (July 1962).
Reference manual of the error messages for the Q-31v (DPC) compiler.

[NEEB62B] Neeb, D., IOC DPC Utility and Test Subsystem Programmer's Handbook - DPC JOVIAL Compiler, FN-LO-741/040/00, SDC, Santa Monica, CA (September 1962).
JOVIAL reference manual for the Q-31v (DPC) compiler.

[NIEL72] Nielsen, W.C., Aerospace HOL Computer, Volume I. Summary, AFAL-TR-72-292-Vol-1, Logicon, Inc., San Pedro, CA (October 1972).
Vol 1 of a four volume report summarizes a study to design a computer architecture to execute SPL/Mk II directly.

[NIME70] Nimensky, R.E., "Space Programming Language: Flight Software Comes of Age," Proceedings of the IEEE 1970 National Aerospace and Electronics Conference, NAECON 1970 (May 1970) 39-45.
Overview of SPL/J6 and the UNIVAC 1824 compiler developed by SDC.

[OBRI68] O'Brien, W.M., Approach for Change. Jovial Evaluation Project, ESD-TR-68-455, Data Dynamics, Inc., Los Angeles, CA (December 1968).
Establishes criteria for improving features of J3 (AFM 100-24). The information was based on the JOVIAL Evaluation Project.

[OLIN62] Olin, P., Q-7 JOVIAL Utility System, Part 5: Reformatter User's Manual, FN-6212/005/00, SDC, Santa Monica, CA (October 1962).
Reference manual for using SURE.

[OLSO60] Olson, W.J.; K.E. Petersen; and J.I. Schwartz, JOVIAL and its Interpreter, a Higher Level Programming Language and Interpretive Technique for Checkout, SP-165, SDC, Paramus, New Jersey (1960).

[PALM77] Palmer, R.; J.W. Crenshaw; K.D. Dannenberg; and D. R. Griffin, HOL Evaluation for Missile Software Applications, CSC/TR-77/5495, Computer Sciences Corporation, Huntsville, AL (8 November 1977).
Evaluated J3B (SofTech Report 7072.1), J73/I, CMS-2, TACPOL, and SPL-1 for use on Pershing II and Digital Autopilot (DAP). J73/I was recommended.
Returned AL#5064194

[PARS79] Parsley, B.J.; H.G. Lehtman; and S. Kahn, On-Line Programmer's Management System, Final Technical Report, RADC-TR-79-205, Augmentation Resources Center, Tymshare, Inc., Cupertino, CA (August 1979).
The JOVIAL Do-All interactive Debugger (JDAD) was constructed by modifying the NLS/NSW Do-All Debugger for use with JOVIAL on the DEC PDP-10 under TENEX.

[PATT62] Patterson, W.M., 1604 JOVIAL Compiler Error Log, FN-WD-277/000/03, SDC, Santa Monica, CA (October 1962).
Describes outstanding errors in the 1604 compiler.

[PERL58] Perlis, A.J. and K. Samelson, "Preliminary Report -- International Algebraic Language", Comm. ACM, 1(12) (December 1958), 8-22.
Original ALGOL-58 report.

[PERS61] Perstein, M.H.; E. Clark; and E. Hayes, Implementation of JOVIAL in the Systems Simulation Research Laboratory, TM-555/200/00 & TM-555/200/00A, SDC, Santa Monica, CA (March & December 1961).
JOVIAL reference manual for the 2000 compiler in terms of its differences from TM-555/002/01 [PERS66B].

[PERS62A] Perstein, M.H., JOVIAL for the Dilettante, Part 1, TM-555/061/00, SDC, Santa Monica, CA (8 October 1962).
Explains a limited portion of JOVIAL sufficient to construct complete programs.

[PERS62B] Perstein, M.H., JOVIAL for the Dilettante, Part 2, TM-555/062/00, SDC, Santa Monica, CA (5 November 1962).
Continues TM-555/061/00 to provide an introductory JOVIAL course.

[PERS63] Perstein, M.H., JOVIAL for the Dilettante and Beyond: Compiler Error Detection Lists, TM-555/063/00, SDC, Santa Monica, CA (2 January 1963).
Presents the latest compiler error lists for J3 on Q-7, CDC 1604, and Philco 2000 computers.

[PERS63B] Perstein, M.H., JOVIAL News, Vol I No. 1, 2, and 3, TM-555/071/01, TM-555/072/00, and TM-555/073/00, SDC, Santa Monica, CA (11 March 1963, 20 November 1963, 14 April 1964).

[PERS64] Perstein, M.H., JOVIAL (J3) Grammar and Lexicon, TM-1682/003/00, SDC, Santa Monica, CA (15 September 1964).
Compare with [PERS66B]

[PERS66A] Perstein, M.H., Numbered-Line Syntactic Description of JOVIAL (J3), SP-2311/000/00, SDC, Santa Monica, CA (1 January 1966).
Syntactic description of J3 in a complex and unreadable notation inspired Iversion's specification of ALGOL 60 in Comm. ACM Oct 1964, pp 588-589.

[PERS66B] Perstein, M.H., The JOVIAL Manual. Part 2. The JOVIAL (J3) Grammar and Lexicon, TM-555/002/04B, SDC, Santa Monica, CA (May 1966) previous versions are M.H. Perstein, TM-555/002/04 (20 October 1965) and C.J. Shaw, TM-555/002/02 (1964), and TM-555/02/01 (June 1961).
Official SDC reference manual for full JOVIAL J3 language.

[PERS67] Perstein, M.H. and J.K. Igawa, JOVIAL Dialect Difference Documents, TM-555/007/01, SDC, Santa Monica, CA (13 November 1967).
Uses the proposed J5.2 dialect as a model for preparing documents showing how JOVIAL dialects compare with Basic JOVIAL [PERS68].

[PERS68] Perstein, M.H., Grammar and Lexicon for Basic JOVIAL, TM-555/005/01, SDC, Santa Monica, CA (4 July 1968), previous version TM-555/005/00 (10 May 1966).
Describes the minimum subset of J3 that every JOVIAL compiler must implement.

[RIZZ62] Rizzo, M., Critique of the JOVIAL User's Manual, FN-LO-34-3, Internal Rep. N-LO-2109/000/00, SDC, Paramus, New Jersey (1962).

[ROBI73] Robinson, R.A. and D.R. Williams, JOVIAL Compiler Validation System User's Guide, RADC-TR-73-315, Vol I & II, RADC, Griffiss Air Force Base, NY. (November 1973).
The JCVS measures the compliance J3 compilers against AFM 100-24. Vol II is a complete listing of all JCVS tests for the Honeywell 6000/600 Computer System.

[RUBE68] Rubey, R.J.; R.C. Wick; W.J. Stoner; and L. Bentley, Comparative Evaluation of PL/I, ESD-TR-68-150, Logicon, Inc., San Pedro, CA (April 1968).
Data management programs were coded in PL/I and J3. The GE 635 version 36 J3 compiler was compared with the PL/I level F compiler for the IBM 360/40. PL/I was found superior to JOVIAL.

[SAMM69] Sammet, J., Programming Languages: History and Fundamentals, Prentice-Hall, Englewood Cliffs, NJ (1969).
The first major work on the history of programming languages.

[SAND65] Sandin, N.A. and E.B. Foote, JTS User's Manual, TM-1577/000/01, SDC, Santa Monica, CA (April 1965). Describes the JOVIAL Time-Sharing (JTS) compiler.

[SAND67] Sandin, N.A., An Introduction to JS as Implemented on the IBM S/360, SP-2824, SDC, Santa Monica, CA (April 1967).

[SAND69] Sandin, N.A., TS/DMS User's Guide to JOVIAL (prJ5.3), TM-4454/000/00, SDC, Santa Monica, CA (5 November 1969). Describes the use of the JOVIAL/J5.3 Compiler, which operates under the TS/DMS timesharing system.

[SCHE80] Scheer, L.S. and M.G. McClimens, "DoD's Ada compared to Present Military Standard HOLs, a Look at new Capabilities, Proceedings of the IEEE 1980 National Aerospace and Electronics Conference, NAECON 1980 (May 1980) 539-544.
Compares J73, CMS-2 and FORTRAN to Ada. Except for strong typing, real-time processing, exception handling and separate translation all languages provide capabilities sufficient for most military problems.

[SCHU61] Schultz, L. and B.C. Sweet, JOVIAL: A New Computer Language System, BR-14, SDC, Santa Monica, CA (April 1961).
An SDC brochure on JOVIAL, its compilers, and some of the advantages of their use.

[SCHW59A] Schwartz, J.I., Preliminary Report on JOVIAL, FN-LO-34, SDC, Lodi, NJ (April 1959).

[SCHW59B] Schwartz, J.I., JOVIAL--Report #2, FN-LO-34-1, SDC, Lodi, NJ (May 1959).

[SCHW59C] Schwartz, J.I., JOVIAL--Primer #1, FN-LO-154, SDC, Lodi, NJ (September 1959).

[SCHW60A] Schwartz, J.I., JOVIAL--A Description of the Language, FN-LO-34-2, SDC, Paramus, NJ (1960).

[SCHW60B] Schwartz, J.I., JOVIAL--Clarifications and Corrections for, FN-LO-34-2, FN-LO-34-2-51, SDC, Paramus, NJ (1960).

[SCHW60C] Schwartz, J.I.; K.E. Petersen; and W.J. Olson, JOVIAL and its Interpreter, A Higher Level Programming Language and an Interpretive Technique for Checkout, SP-165, SDC, Santa Monica, CA (1 April 1960).
Describes the JOVIAL/J0 interpreter implemented before the first compiler.

[SCHW61] Schwartz, J.I. and H.L. Howell, The JOVIAL (J2) Language for the 7090 Computer, FN-6223/100/00, SDC, Santa Monica, CA (1961).

[SCHW62] Schwartz, J.I., "JOVIAL: A General Algorithmic Language", Procedings of the Symposium on Symbolic Languages in Data Processing, Gordon & Breach, NY (1962) 481-493.
First formal presentation on JOVIAL briefly describing the language.

[SCHW64] Schwartz, J.I.; E.G. Coffman; and C. Weissman, "A General-Purpose Time-Sharing System," SDC, Santa Monica, CA, Proceedings of the Spring Joint Computer Conference (1964) 397-411.
Describes the SDC Time-Sharing System (TSS) and the TINT JOVIAL interpreter which operates on TSS.

[SCHW65] Schwartz, J.I., "Programming Languages For On-Line Computing", Proceedings of the IFIP Congress, Vol. 2, Spartan Books, Washington, D. C. (1965) 546-547.

[SCHW69] Schwartz, J.I, Advanced Development Prototype, SDC-TM-3628/003/00, SDC, Santa Monica, CA (January 1969).

[SCHW78] Schwartz, J.I., "The Development of JOVIAL", ACM SIGPLAN History of Programming Languages Conference, SIGPLAN Notices, 13(8) (August 1978) 201-214; also in [WEXE81], pp. 369-401, along with transcript of conference.

[SDC60] SDC, The S-2000 JOVIAL Library, 2000 JOVIAL Library Index & JOVIAL Procedure Writeups, FN-LX-394/300/00, SDC, Santa Monica, CA.
Describes the 2000 compiler subroutine library in Lexington.

[SDC62A] SDC, 1604 JOVIAL Compiler, Program Description of the Translator Pass 1, TM-555/302/00, SDC, Santa Monica, CA (June 1962).
Compiler maintenance manual for the first pass of the 1604 code generator.

[SDC62B] SDC, 1604 JOVIAL Compiler, Description of System-Dependent Procedures, WD-555/304/00, SDC, Santa Monica, CA (September 1962).
Compiler maintenance manual for host-dependent portions of 1604 compiler

[SDC62C] SDC, Preliminary JOVIAL Training Manual, FN-WD-430/002/00, SDC, Santa Monica, CA (October 1962).
1604A JOVIAL programming manual.

[SDC62D] SDC, Computer Programming Standards in Command and Control, TM-688/000/01, SDC, Santa Monica, CA (1962).
Recommends adoption of standard military programming language.

[SEPA62] Sepan, A.V., Input-Output Routines for Philco 2000 JOVIAL Programs & 2000 JOVIAL I/O Operations, FN-5530/001/00 & FN-5530/002/00, SDC, Santa Monica, CA (November 1962 & September 1962).
Reference manual for the input-output features of the 2000 compiler.

[SHAW60A] Shaw, C.J., The Compleat JOVIAL Grammar, FN-4178, SDC, Santa Monica, CA (1960).

[SHAW60B] Shaw, C.J., The JOVIAL Lexicon: A Brief Semantic Description, FN-4178, 51, SDC, Santa Monica, CA (1960).

[SHAW60C] Shaw, C.J., Programming Languages and JOVIAL, SDC Brochure, 3(11), SDC, 2500 Colorado Avenue, Santa Monica, CA (November 1960).
A brochure introducing JOVIAL, its compilers, and the people involved in its early development.

[SHAW60D] Shaw, C.J., The JOVIAL Manual, Part 1, Computers, Programming Languages and JOVIAL, TM-555/001/00, SDC, Santa Monica, CA (20 December 1960).
Introduction to JOVIAL language and compilers.

[SHAW61A] Shaw, C.J., "SDC's Procedure-Oriented JOVIAL", Datamation, 7(6) (June 1961) 28-32.
Informal introduction to JOVIAL language and its compilers.

[SHAW61B] Shaw, C.J., "A Programmer's Look at JOVIAL, in an ALGOL Perspective," Datamation, 7(10) (October 1961) 46-50.
Informal report on the language and its relationship to ALGOL

[SHAW61C] Shaw, C.J., The JOVIAL Manual, Part 3, The JOVIAL Primer, TM-555/003/00, SDC, Santa Monica, CA (26 December 1961).
J3 programming manual with syntactic definitions, examples, and exercises.

[SHAW62A] Shaw, C.J., A Comparative Evaluation of JOVIAL and NELIAC, FN-6609, SDC, Santa Monica, CA (5 June 1962).

[SHAW62B] Shaw, C.J., JOVIAL, SDC Brochure, 5(6), SDC, 2500 Colorado Avenue, Santa Monica, CA (June 1962).

[SHAW62C] Shaw, C.J., A Programmer's Introduction to Basic JOVIAL, TM-629, SDC, Santa Monica, CA (August 1961).
Programming reference manual for Basic JOVIAL subset.

[SHAW62D] Shaw, C.J., JOVIAL and Its Documentation, SP-1013, SDC, Santa Monica, CA (30 October 1962).
Extensive list of JOVIAL references.

[SHAW63B] Shaw, C.J., "JOVIAL and Its Documentation", Comm. ACM, 6(3) (March 1963) 89-91.
Same as [SHAW62D].

[SHAW63A] Shaw, C.J., "JOVIAL-- A Programming Language for Real-time Command Systems", in Annual Review in Automatic Programming, Vol. 3 (R. Goodman, ed.), Pergamon Press, New York (1963) 53-119.
An informal report on JOVIAL with syntactic definitions and examples.

[SHAW63C] Shaw, C.J., "A Specification of JOVIAL", Comm. ACM, 6(12) (December 1963) 721-736.
Formal specification of JOVIAL syntax using Backus Naur Form is included.

[SHAW64] Shaw, C.J., A Comparative Evaluation of JOVIAL and FORTRAN IV, N-21169, SDC, Santa Monica, CA (January 1964) also in Automatic Programming Information, College of Technology, Brighton, England (August 1964).

[SHAW81] Shaw, C.J., personal interview, Xerox Computer Systems, Los Angeles, CA (July 1981).
Personal interview conducted by the author.

[SHAW78] Shaw, M.; G.T. Almes; J.M. Newcomer; B.K. Reid; and W.A. Wulf, A Comparison of Programming Languages for Software Engineering, RADC-TR-78-58, Carnegie-Mellon Univ., Pittsburgh, PA (April 1978).
A core of essential features for FORTRAN, COBOL, JOVIAL AND Ada are compared in the light of modern software engineering.

[SHIR73A] Shirley, D.L., "Compiler Specification Guidance, JOVIAL J73", Draft submitted to RADC, Griffiss AFB, NY (31 July 1973).
Provides guidelines for preparing compiler procurement specifications. Addresses the issues of user interface, hardware, compiler construction, JOVIAL J73 language aspects, documentation, and quality assurance.

[SHIR73B] Shirley, D.L., JOVIAL J73 Subsetting, RADC report (31 July 1973).
Initial specification describing the three JOVIAL J73 subsets.

[SLAV79] Slavinsky, R.T., "JOVIAL Language Control", Proceedings of the IEEE 1979 National Aerospace and Electronics Conference, NAECON 1979 (May 1979) 1283-1288.
Describes development of the JOVIAL a language control structure.

[SOFT74] SofTech, An Introduction to the JOVIAL/J3B Language and Compilers, 7015, SofTech, Inc., Waltham, MA (November 1974).

[SOFT75] SofTech, JOVIAL/J3B Extension 2 Language Specification, 2044-4, SofTech, Inc., Waltham, MA (1 June 1975).

[SOFT78] SofTech, The Proposed Upgrage of JOVIAL J73/I, 1049-3, SofTech, Inc., Waltham, MA (1 November 1978).
Describes proposed features of J3B to be included in J73/I.

[SOFT79A] SofTech, MILITARY STANDARD JOVIAL J73, 1049-8 and revision 1049-8.1, SofTech, Inc., Totten Pond Road, Waltham, MA (8 January 1979 and revision 20 February 1979).
Draft revision to MIL-STD-1589A.

[SOFT79B] SofTech, Inc., SofTech Non-Technical Overview on the J73 Upgrade, distributed to JOVIAL Users Group (October 1979).
Summary of the J73/I language upgrade.

[SOFT80] SofTech note, "SofTech's Experience in Developing Compilers."
This note summarizes SofTech's experience in developing compilers intended for use in a production environment.

[SOFT81] SofTech, Annual Digest, Final Submission, Vol. II, Part 1, 1056-14.3, SofTech, Inc., Waltham, MA (13 March 1981, resubmission 1 May 1981).
Summarizes SofTech's second year (1980) of supporting the JOVIAL Language Control Facility at RADC.

[SOFT82] SofTech, Computer Programming Manual for the JOVIAL (J73) Language, 2133-11, SofTech, Inc., Waltham, MA (July 1982).
Learning manual for J73 (1589B) produced by SofTech as part of their JOVIAL Language Control Facility support contract.

[SPER74] Sperry UNIVAC, Drone Control and Data Retrieval System (DCDRS). Preliminary Design Study Final Report. Volume III. Trade Studies and Analyses. Part VIII. Basic Program Language Trade Study Analysis, ASD-TR-74-5-Vol-3-Pt-8, Sperry UNIVAC Defense Systems Division, St. Paul, MN (April 1974).
Compares J73 to other languages for use on the DCDRS.

[SPIE61] Spierer, M., The 7090-JOVIAL-TO-7090 Checker Specifications, FN-LO-504/000/01, SDC, Santa Monica, CA (31 July 1961).
Reference manual for the JOVIAL Checker, a source language debugging tool for use with the 7090 compiler.

[SPSM76A] Simplified Processing Station Manual, Common Intermediate Language Compiler, Contract F04701-75-C-0074 CDRL A037, Doc # 6122804 (29 December 1976).
Describes J3B compiler built by IBM for the SPS program.

[SPSM77A] Simplified Processing Station Manual, Computer Programming Manual, Standards and Conventions, Vol 3, Program Development and Maintenance System, Contract F04701-75-C-0074 CDRL B006, Doc # 6188007 (16 May 1977).
Describes use of JOVIAL with PDMS.

[SPSM77B] Simplified Processing Station Manual, Computer Programming Manual, Standards and Conventions, Vol 5, JOVIAL/J3B Language, Contract F04701-75-C-0074 CDRL B006, Doc # 6188007 (16 May 1977).
Programming manual for using JOVIAL directly without PDMS.

[STAN80] Stanten, S.F.; P.Y. Williams; D.A. Flanders; S.Z. Stein; and S.E. Adams, Digital Avionics Information System (DAIS): Mission Software, AFWAL-TR-80-1003, Intermetrics Inc., Dayton, OH (February 1980).

[STEE60] Steel, T.B., "UNCOL, Universal Computer Oriented Language Revisited", Datamation, (January/February 1960) 18-20.

[STEE66] Steel, T.B., Some Observations on the Relationship Between JOVIAL and PL/I, TM-2930/000/01, SDC, Santa Monica, CA (May 1960).

[STEE73] Steel, S.A. and L.J. Galbiati, "Higher Order Language Evaluation for Real-Time Computer Systems", RCA, Morrestown, NJ, 7th Asilomar Conference on Circuits, Systems and Computers (27 November 1973) 417-427.
Comparison of FORTRAN, CMS-2, JOVIAL, PL/1, and assembly languages for a real-time radar control. FORTRAN was chosen due to training, available skills and compiler maturity.

[STOV77] Stover, R.E. Jr., A Statistics Collection Package for the Jovial J3 Programming Language, RADC-TR-77-293, Rome Air Development Center, Griffiss AFB, NY (September 1977).

[SYST62B] System Support Group, 1604 JOVIAL Compiler Vol. III: Program Description of the Translator Pass 1, TM-555/302/00, SDC, Santa Monica, CA (1 June 1962).

[SYST62B] System Support Group, MC JOVIAL Library Procedure Write-ups, FN-LO-661/002/00 and supplements, SDC, Santa Monica, CA (October 1962).
Reference documents describing the Q-31v (MC) compiler subroutine library.

[SYST67] System Support Group, How Pr J5.2 Differs From Basic JOVIAL, TM-WD-999/002/00, SDC, Falls Church, VA (22 May 1967).
Compares the proposed J5.2 JOVIAL language with Basic JOVIAL A better version of this comparison is given in [PERS67].

[TIER67] Tiernan, J.C., Programming Languages for Digital Weapon Systems: Evaluation, NELC-1527, Naval Electronics Lab Center for Command Control Communications, San Diego, CA (December 1967).
A qualitative evaluation is made of various procedure-oriented languages from the viewpoint of Navy digital weapon systems requirements.

[TJOM60] Tjomsland, I.A., The 709 JOVIAL Compool, FN-3836, SDC, Paramus, NJ (1960).

[TRAI77A] Trainor, W.L., Report on High-Order Language Standardization for Avionics, AFAL-TR-76-254, Vol. 2, Air Force Avionics Laboratory (AFAL/AA), Wright-Patterson AFB, OH (January 1977).
An in depth comparison of the two most prominate HOLs in avionics: J73 and J3B.

[TRAI77B] Trainor, W.L. and H.M. Grove, "Higher Order Language Standardization for Avionics", Proceedings of the IEEE 1977 National Aerospace and Electronics Conference, NAECON 1977 (May 1977) 487-493.
Conference version of [TRAI77A].

[TRAI77C] Trainor, W.L. and M. Burlakoff, "JOVIAL-73 Versus Assembly Language -- An Efficiency Comparison", Proceedings of the IEEE 1977 National Aerospace and Electronics Conference, NAECON 1977 (May 1977) 502-507.
The DAIS J73/I cross compiler to AN/AYK-15 resulted in a 9.8% run-time expansion and approximately 11% memory expansion.

[TRAN62] Translator Group, P-2000, Phase 2 of the Philco 2000 JOVIAL Translator, TM-555/212/00, SDC, Santa Monica, CA (22 August 1962).
Reference manual for the instruction generation portion of the Philco 2000 code generator.

[TRAN64] Translator Group, Phase 1 (and Phase 2) of the Philco 212 JOVIAL Translator, TM-555/213/00 (and TM-555/214/00), SDC, Santa Monica, CA (1 September 1964).

[UPSH81] Upshaw, S., personal interview, Abacus Programming Corporation, Los Angeles, CA (17 September 1981).
Personal interview conducted by the author.

[VANA62] Van Auken, D., The Military Computer Checker, FN-LO-646,SDC, Santa Monica, CA (8 February 1962).
Reference manual for the JOVIAL Checker used with the AN/FSQ-31v compiler. See [SPIE61] for the 7090 compiler.

[VAND69] Van Dyke, J.G.O., M555 Coordination Control System, Final Report, RADC-TR-69-71, Informatics Inc., Bethesda, MD (April 1968).
Describes development of a JOVIAL compilation system to produce programs executable on the UNIVAC M555.

[VERH70] Ver Hoef, E.W.; J.L. Berg; and D.L. Shirley, Block File and Multics Systems Interface Investigation and Programming. Volume I, RADC-TR-69-400-Vol-1, Informatics, Inc. Bethesda, MD (April 1970).
Describes development of a test compiler which produces reentrant GE 635 GMAP assembly code and discusses implementation of two segmentation algorthms.

[WALL62] Wallace, R.W., The DPC-JOVIAL-to-DPC Compiler System, FN-LO-608/040/00, FN-LO-608/041/00 and FN-LO-608/042/00, SDC, Paramus, NJ (April 1962, June 1962 and November 1962).
Reference manual for the Q-31v (DPC) compiler. See [NEEB62A] for error messages.

[WASS80] Wasserman, A.I., Tuitorial Programming Language Design, IEEE Computer Society (October 1980).
Introductory articles and reprints of research on programming language design including extensive sections on Pascal and Ada.

[WEXE81] Wexelblat, R.L., Ed., History of Programming Languages, Academic Press, New York (1981).
Contains papers and transcripts of ACM SIGPLAN History of Programming Languages (HOPL) Conference, Los Angeles CA, 1-3 June 1978.

[WILE68] Wiley, G., Analyzation of the Jovial Compilers for the GE-635 and UNIVAC M1218 Computers, RADC-TR-68-1, SDC, Rome, NY (April 1968).

[WILK61] Wilkerson, M., JOVIAL User's Manual, subtitled JOVIAL Language Specifications for the 7090 and MC Compilers, FN-LO-34-3, SDC, Paramus, New Jersey (1961).

[WILK61] Wilkerson, M., 7090 Jovial Checker, An Automatic Checkout System For Higher Level Language Programs, The Western Joint Computer Conference, Los Angeles, CA (May 1961). Conference version of [SPIE61].

[WIRT71] Wirth, N., "The Programming Language Pascal", Acta Informatica, 1(1) (1971) 35-63.
Designed to combine the machine independence of Algol W with the efficiency of PL/360. New features are the type definition capability.

[WOLP58] Wolpe, H., "Algorithm for Analyzing Logical Statements to Produce Truth Function Tables", Comm. ACM, 1(3) (1958), 4-13.
Article which inspired the CLIP compiler.

[YOTT68] Yott, J.H., An Investigation of Advanced Programming Techniques, RADC-TR-68-367, SDC, Rome, NY (October 1968).

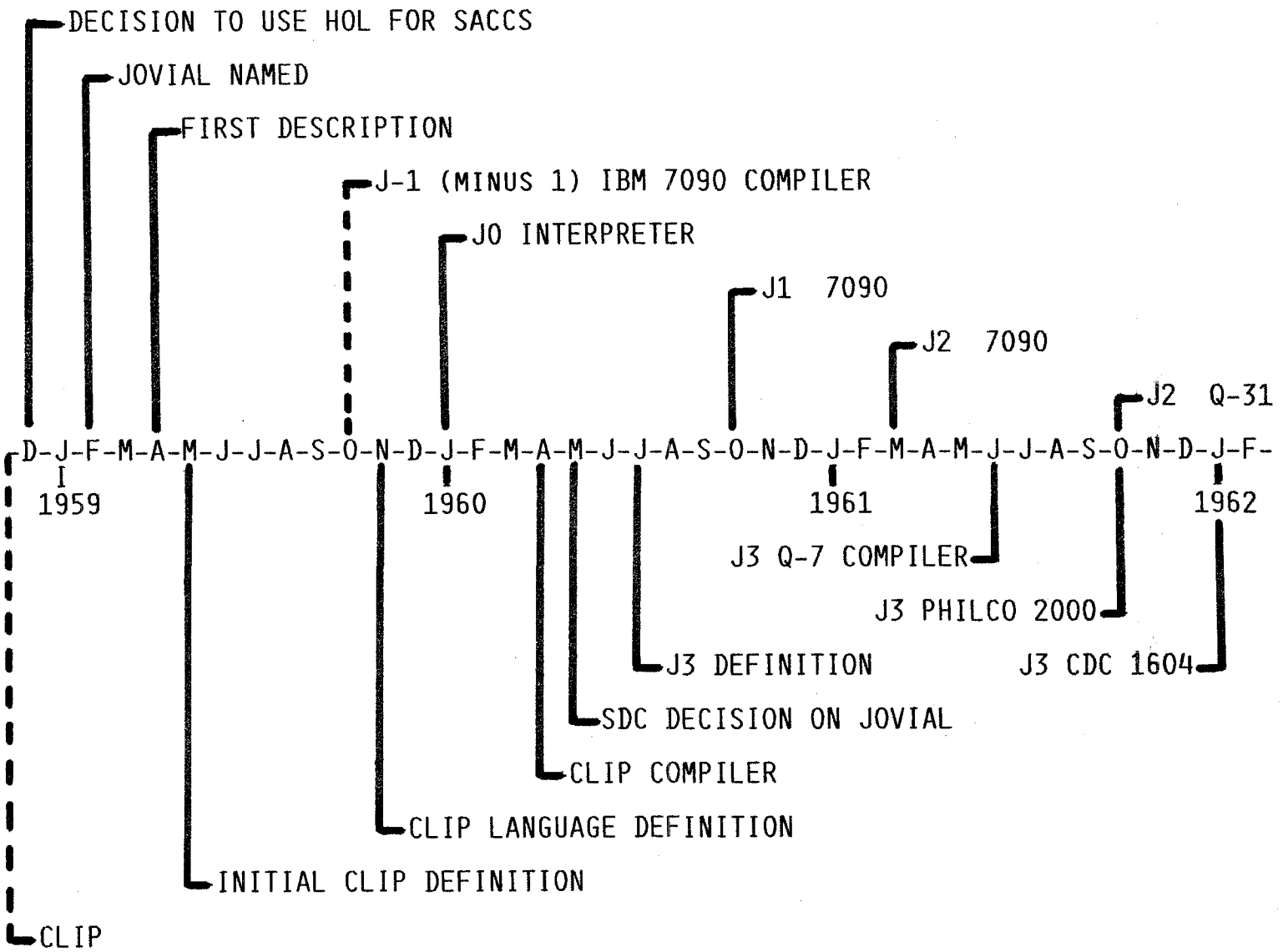
[ZEIT81] Zeitlin, L., personal interview, Abacus Programming Corporation, Los Angeles, CA (September 1981).
Personal interview conducted by the author.

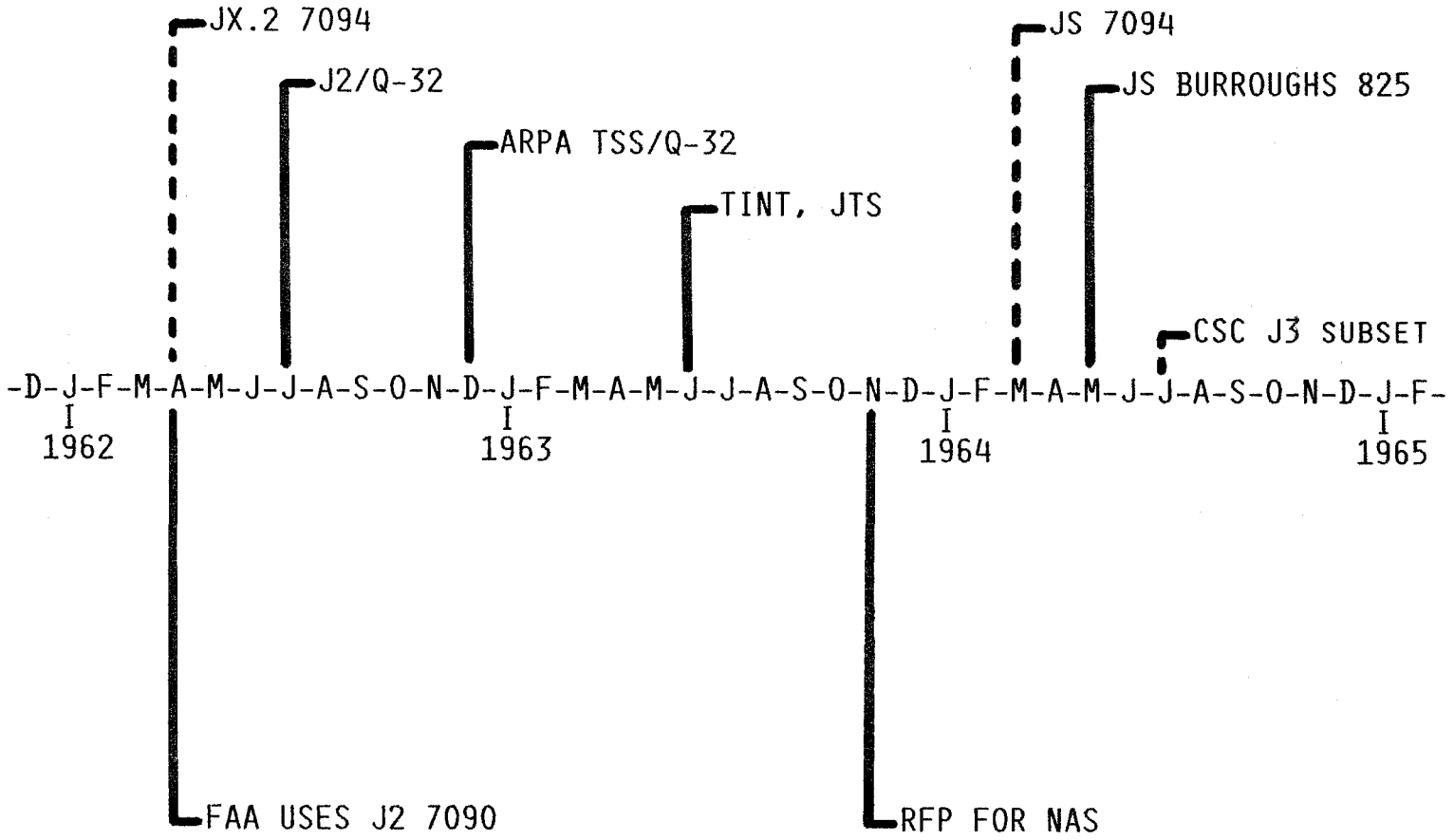
Appendix A

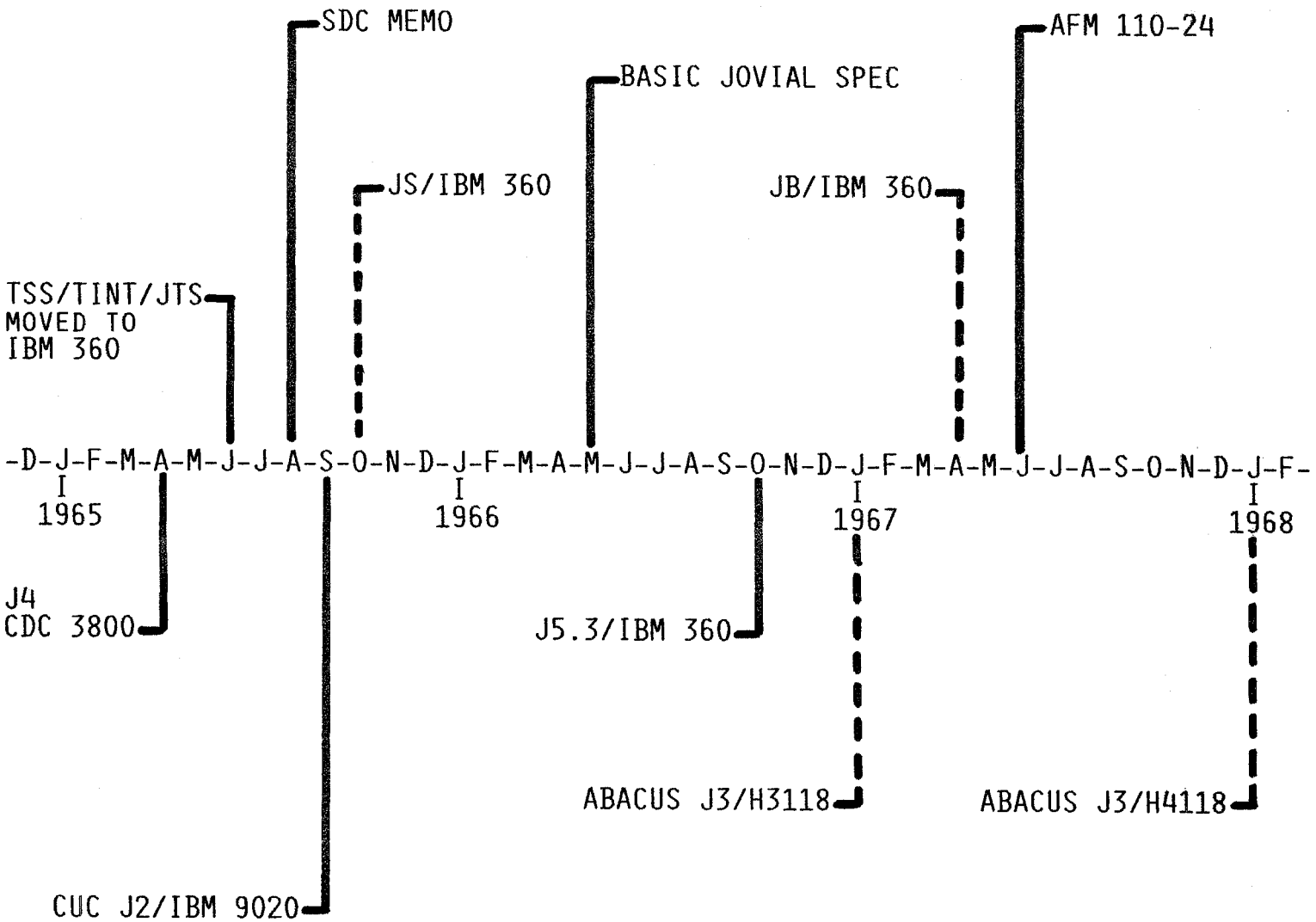
JOVIAL TIME LINE

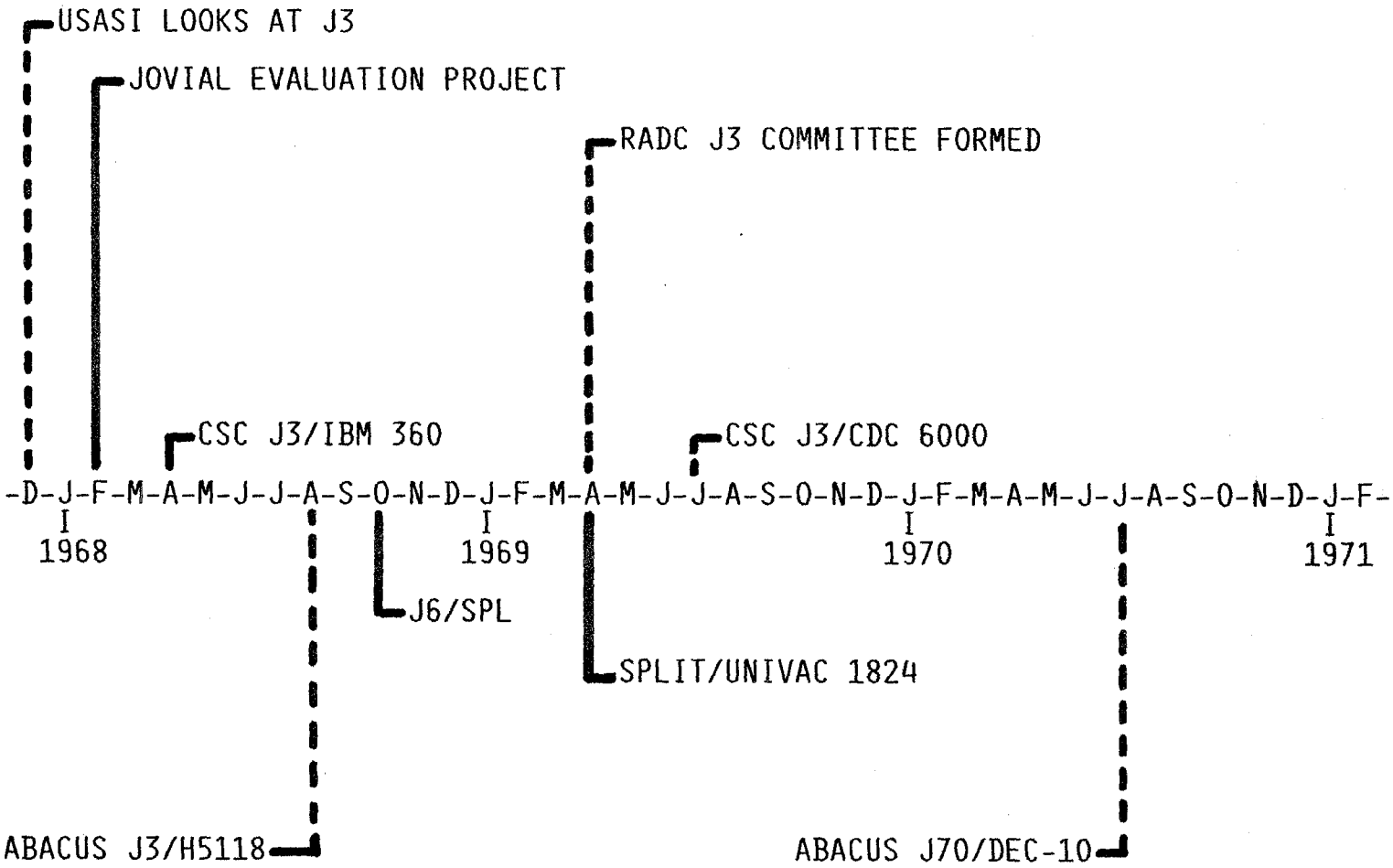
These figures present a time line of the history of JOVIAL to aid the reader in sorting out the various dates. Many events which happened concurrently are presented in different chapters in the paper. This time line allows the reader to see the relationship of these events.

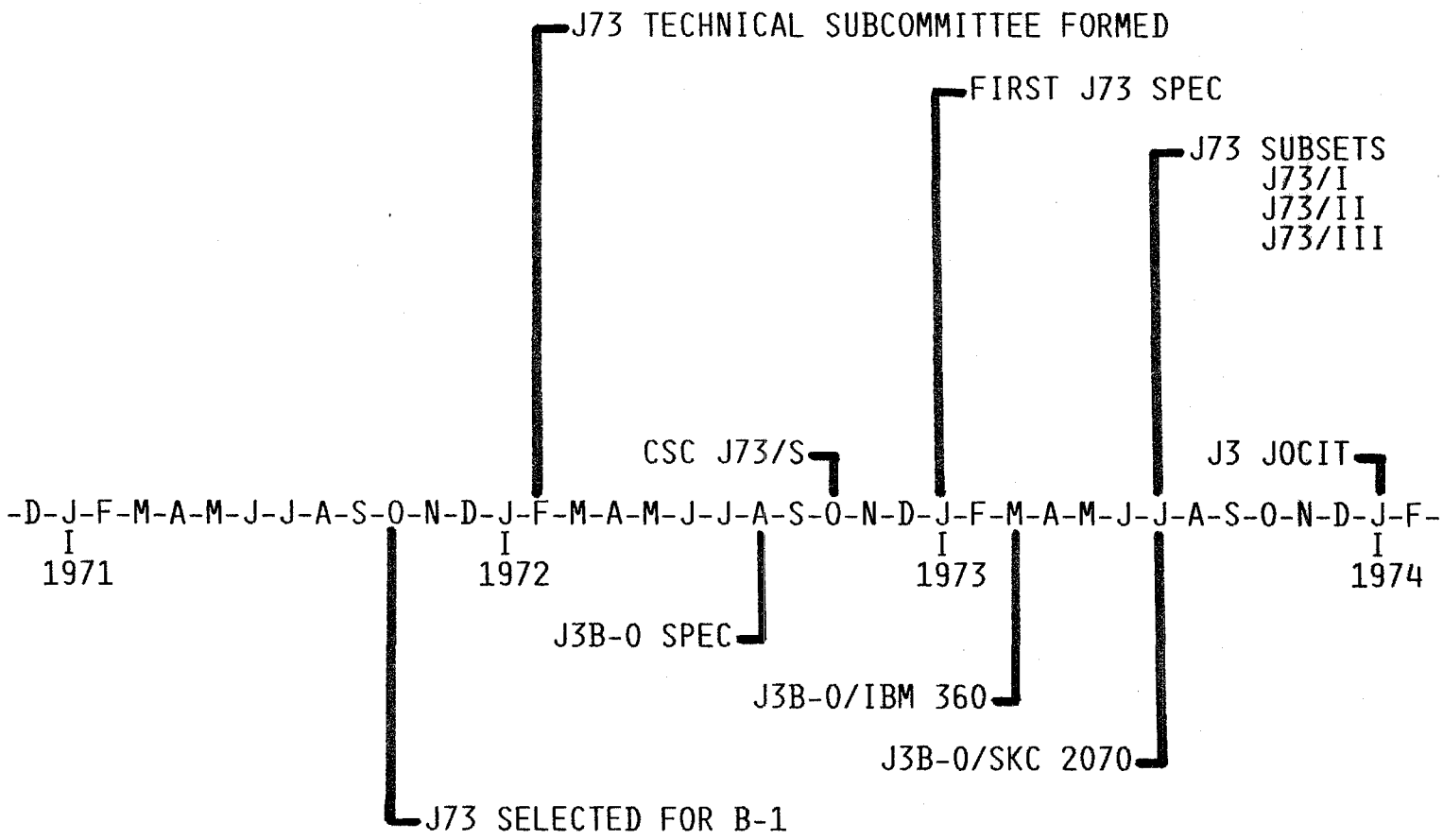
The events marked with a dotted line represent only approximate dates; those with a solid represent actual dates.

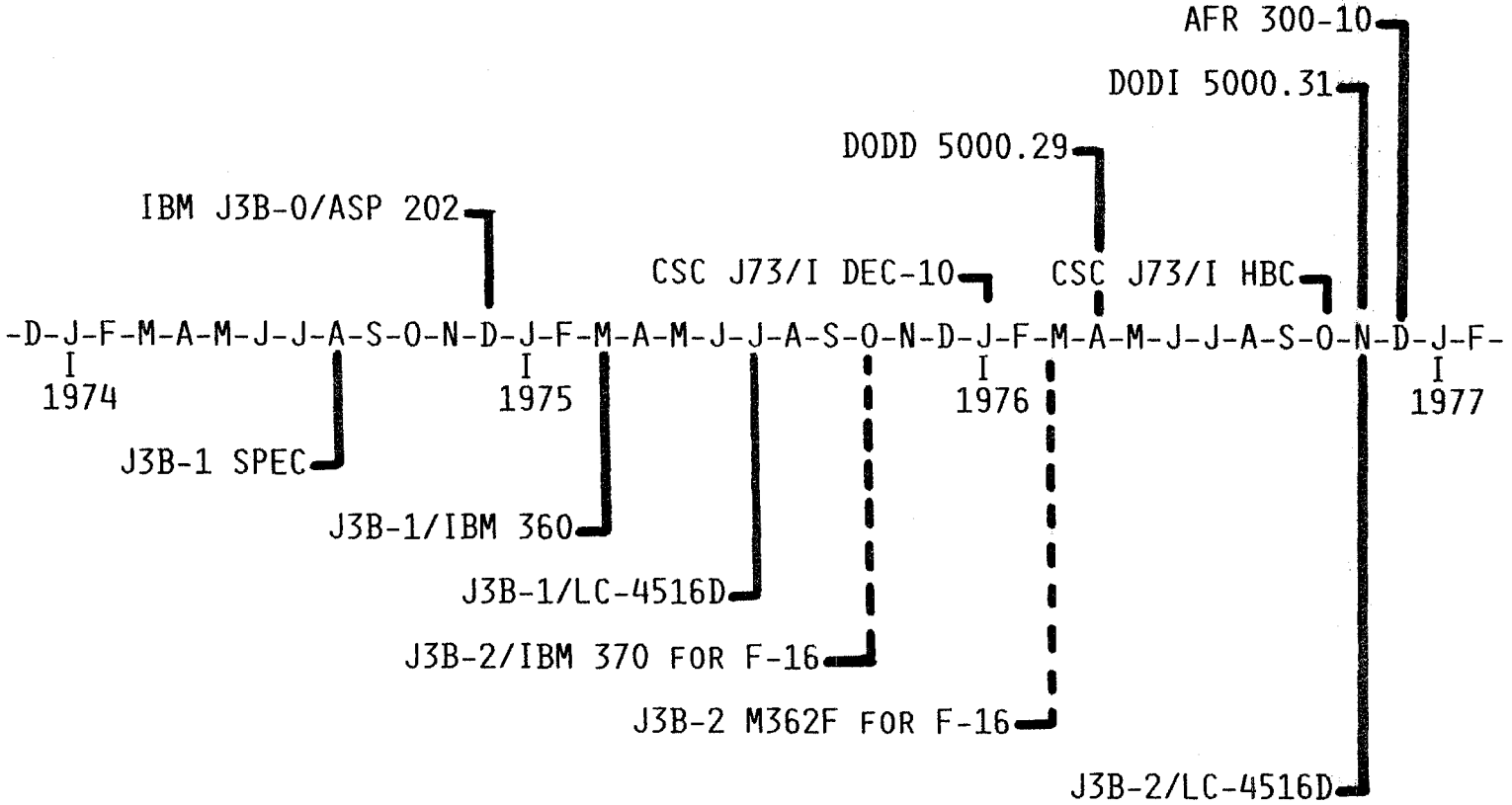

DECISION TO USE HOL FOR SACCS
JOVIAL NAMED
FIRST DESCRIPTION
J-1 (MINUS 1) IBM 7090 COMPILER
J0 INTERPRETER
J1 7090
J2 7090
J2 Q-31
D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-
1959
1960
1961
1962
J3 Q-7 COMPILER
J3 PHILCO 2000
J3 CDC 1604
J3 DEFINITION
SDC DECISION ON JOVIAL
CLIP COMPILER
CLIP LANGUAGE DEFINITION
INITIAL CLIP DEFINITION
CLIP

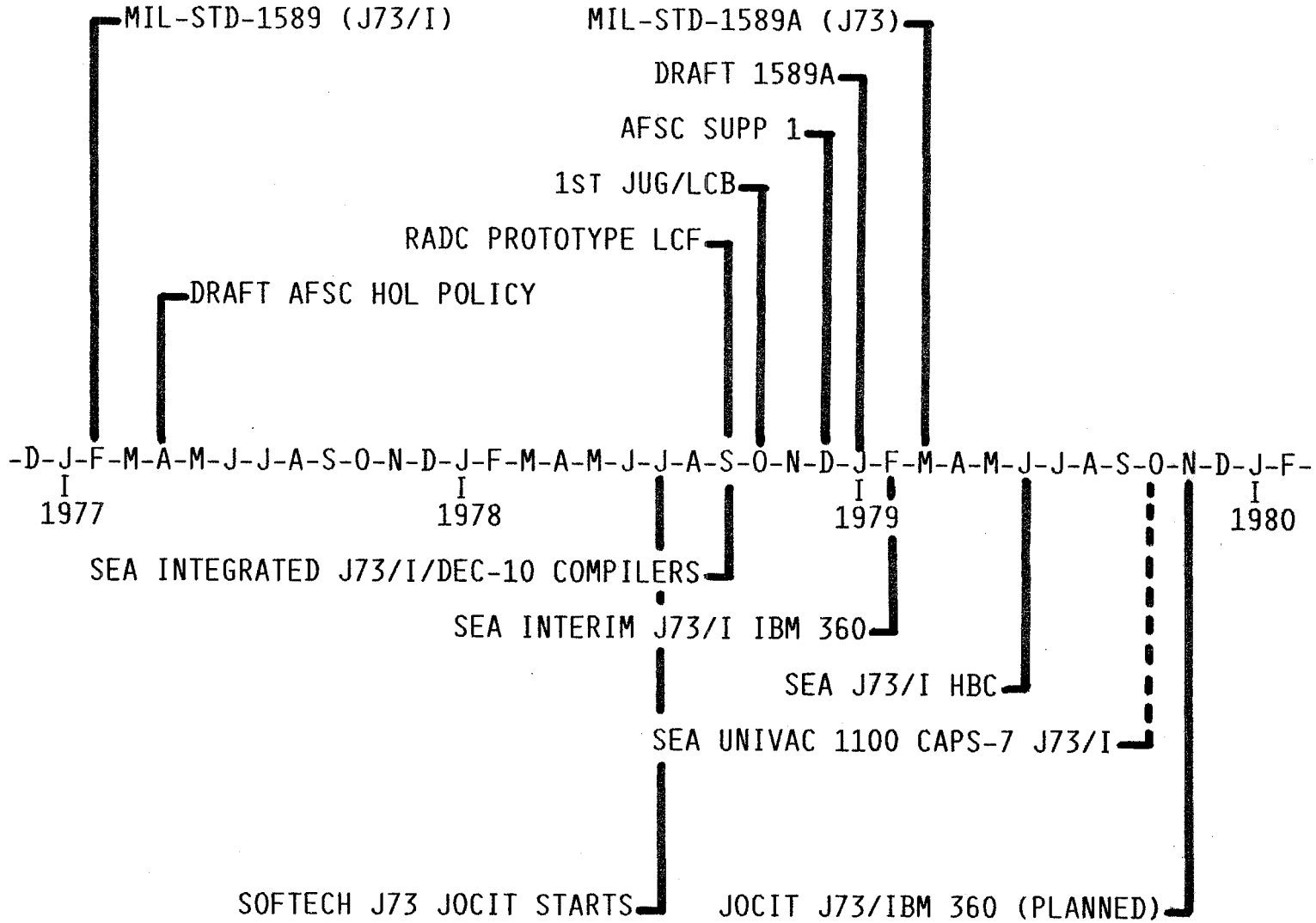

JX.2 7094
J2/Q-32
ARPA TSS/Q-32
TINT, JTS
JS 7094
JS BURROUGHS 825
CSC J3 SUBSET
-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-
1962
1963
1964
1965
FAA USES J2 7090
RFP FOR NAS

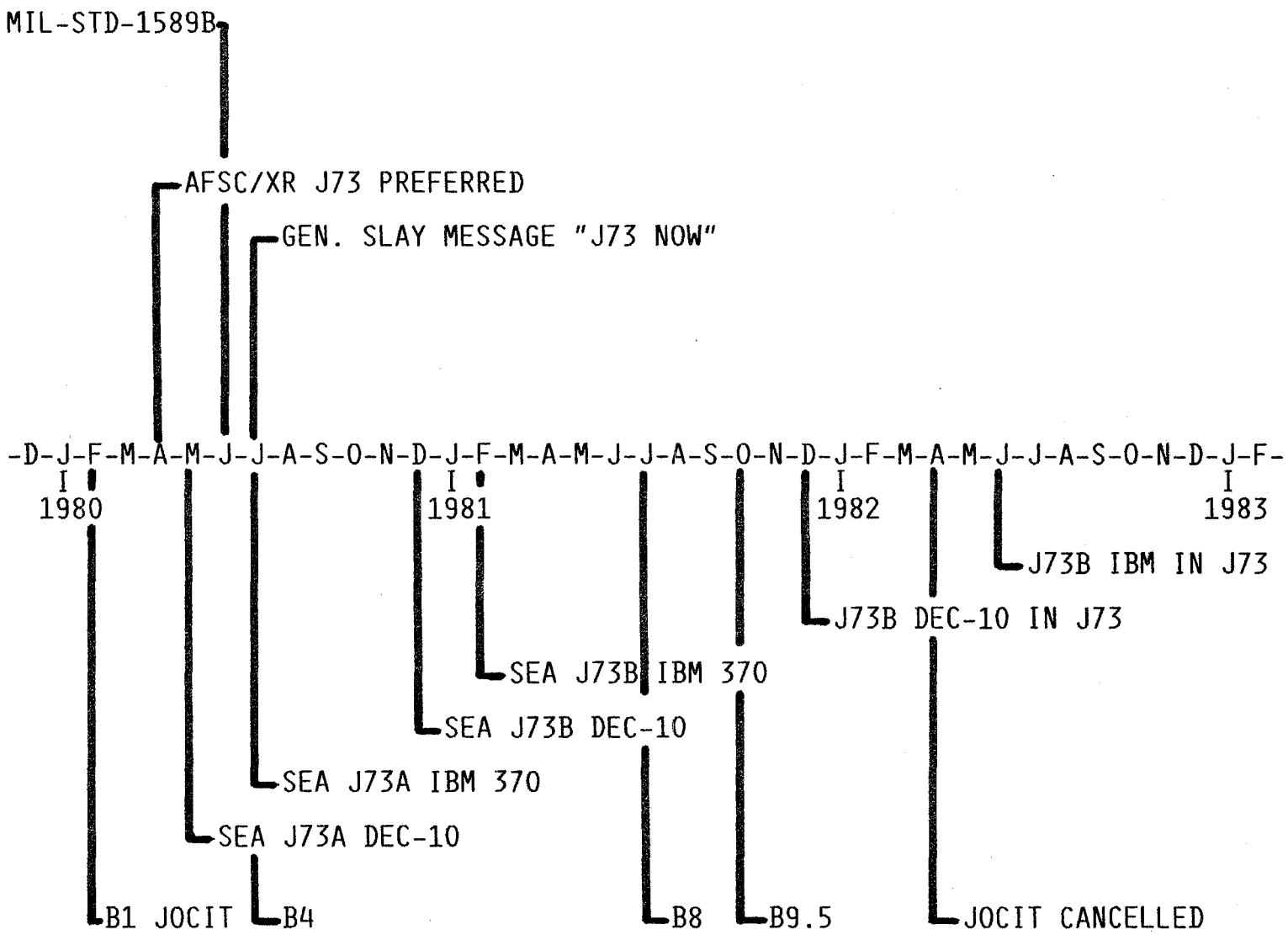

SDC MEMO
BASIC JOVIAL SPEC
AFM 110-24
JS/IBM 360
JB/IBM 360
TSS/TINT/JTS
MOVED TO
IBM 360
-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-
1965
1966
1967
1968
J4
CDC 3800
J5.3/IBM 360
ABACUS J3/H3118
ABACUS J3/H4118
CUC J2/IBM 9020


USASI LOOKS AT J3
JOVIAL EVALUATION PROJECT
RADC J3 COMMITTEE FORMED
CSC J3/IBM 360
CSC J3/CDC 6000
-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-
1968
1969
1970
1971
J6/SPL
SPLIT/UNIVAC 1824
ABACUS J3/H5118
ABACUS J70/DEC-10


J73 TECHNICAL SUBCOMMITTEE FORMED
FIRST J73 SPEC
J73 SUBSETS
J73/I
J73/II
J73/III
CSC J73/S
J3 JOCIT
-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-
I I I I
1971 1972 1973 1974
J3B-O SPEC
J3B-O/IBM 360
J3B-O/SKC 2070
J73 SELECTED FOR B-1


AFR 300-10
DODI 5000.31
DODD 5000.29
IBM J3B-0/ASP 202
CSC J73/I DEC-10
CSC J73/I HBC
-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-
1974
1975
1976
1977
J3B-1 SPEC
J3B-1/IBM 360
J3B-1/LC-4516D
J3B-2/IBM 370 FOR F-16
J3B-2 M362F FOR F-16
J3B-2/LC-4516D


MIL-STD-1589 (J73/I)
MIL-STD-1589A (J73)
DRAFT 1589A
AFSC SUPP 1
1ST JUG/LCB
RADC PROTOTYPE LCF
DRAFT AFSC HOL POLICY
-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-
1977
1978
1979
1980
SEA INTEGRATED J73/I/DEC-10 COMPILERS
SEA INTERIM J73/I IBM 360
SEA J73/I HBC
SEA UNIVAC 1100 CAPS-7 J73/I
SOFTECH J73 JOCIT STARTS
JOCIT J73/IBM 360 (PLANNED)


MIL-STD-1589B
AFSC/XR J73 PREFERRED
GEN. SLAY MESSAGE "J73 NOW"
-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-M-A-M-J-J-A-S-O-N-D-J-F-
1980
1981
1982
1983
J73B IBM IN J73
J73B DEC-10 IN J73
SEA J73B IBM 370
SEA J73B DEC-10
SEA J73A IBM 370
SEA J73A DEC-10
B1 JOCIT
B4
B8
B9.5
JOCIT CANCELLED

Appendix B

COMPILER TERMINOLOGY

This appendix explains the terms used in this paper to talk about programming language compilers, and JOVIAL compilers in particular. A compiler is a computer program that translates a computer program written in a high order language into a program written in either assembly language or machine language. If the program is translated into assembly language, it must be further processed by an assembler into machine langauge before the program can be executed.

The compiler executes on a computer called the host computer and produces code that executes on the target computer. The host and target computers may be the same or they may be different. The term cross-compiler is applied to a compiler that runs on one computer but produces code for a different computer.

The input high order language is called the source language. The source language for a JOVIAL compiler is JOVIAL. The source language program is translated into the target language of the target computer. The terms object language, object code, target language, and target code are used interchangeably.

The compiler itself is written in a language, called the implementation language. Today most compilers are implemented in an HOL which must be compiled into the machine language of the host computer before the compiler can run. The implementation language

of most JOVIAL compilers has been JOVIAL, which allows the compiler to be able to compile a source language version of itself.

Most compilers consist of two major parts: the front end and the back end. Sometimes there is a "middle end", which is sandwiched between the front and back ends and which consists of a global optimizer.

The front end consists of a lexical analyzer and a parser. The lexical analyzer divides the stream of input characters into tokens such as reserved words, data names, and special symbols like ($, $), and +. The parser checks that these tokens are combined using the syntax of the source language. The parser then produces a symbol table containing information on all declarations, and it translates the imperative statements into a machine-independent intermediate language (IL). In addition to the program being compiled, the front end of JOVIAL compilers must accept COMPOOLs. A COMPOOL (COMmunications POOL) is essentially an an external file containing the symbol table of global data declaration that have been previously compiled.

The back end consists of a code generator and, in some compilers, a local optimizer. The terms back end and code generator are used interchangeably. The code generator uses the symbol table and the IL produced by the front end to produce the target machine code. The local optimizer then makes changes to the target code to improve the speed of the resulting target code. The local optimizer

is target-dependent and makes use of special knowledge about the target computer. Local optimizations apply only to single statements. Well known local optimizations are re-ordering of formula evaluation, register assignments optimized to reduce the number of register loads and saves in a program, and instruction scheduling optimization in computers that overlap instruction decoding with memory access or perform look ahead in access and decoding of instructions.

The middle end consists of the global optimizer, which works on the intermediate language and is language and target computer independent. Global optimizations are applied across many statements and may extend over an entire program. Some well known global optimizations are flow analysis to eliminate instruction paths that can never be reached, code straightening to eliminate unnecessary transfers between instruction sequences, common subexpression recognition and elimination, code redistribution by moving code segments to a path that eliminates redundant execution of the code, strength reduction by replacing an operation by one or more operations that are less time consuming, and dead variable analysis to eliminate assigning a value to a variable that is never used.

Optimization is costly both in resources utilized by the compiler in performing optimization and in the actual cost of building a compiler with optimization capability. This cost will be an effective tradeoff only if programs produced by the compiler are used frequently enough to recover the cost of optimizing.

In the early literature, the front end was called the "generator" and the code generator was called the "translator". These terms as well as the original idea of building a compiler with separate front and back ends came from a SHARE sponsored project called the UNiversal Computer Oriented Language (UNCOL).

UNCOL was aimed at reducing the work to implement compilers for new computers. With the increasing number of new computers being developed and the growing number of high order languages, it looked as if portability of programs written in m programming languages across n computers would require m x n compilers. Furthermore, the time to develop a compiler for a new computer (2-3 years) was about the same as the life time of the computer itself.

The UNCOL approach was to design the compiler with an independent front end and back end communicating via a machine-independent IL -- a UNiversal Computer Oriented Language. Thomas B. Steel, Jr., who was director of Information Processing Research at SDC, was one of five national participants in this industry-renowned project [STEE58] and the UNCOL concept was first used on the CLIP (Chapter 2) and JOVIAL compilers (Chapter 3) built at SDC.

The primary reason for dividing the compiler into separate front and back ends communicating via a machine-independent IL is to facilitate rewriting the compiler to produce code for a different target computer (a process called retargeting). Both the front end

and the IL are entirely machine-independent allowing as much analysis as possible to be done on the source language before producing language that is too machine dependent. To retarget a compiler designed in this way only requires writing a new code generator for the new target computer and combining it with the same front end. The work of developing a new compiler is cut in half. In this way, a single front end may be joined to many code generators to produce a compiler that generates code for several different computers. Furthermore, several front ends, one for each language, can be combined with different code generators, one for each computer, to produce compilers for many combinations of languages and computers.

If the compiler is written in the same language that it compiles, i.e., the source and implementation languages are the same, the work of rewriting the compiler to run on a different host computer computer (a process called rehosting) can also be considerably reduced. Rehosting is accomplished by first writing a new code generator for the new host computer. Then the compiler is self-compiled using the new code generator to produce a version of the compiler in the machine language of the new host computer. Finally, the compiler is installed on the new host computer by modifiying the operating system interface routines. The term bootstrap compiler is sometimes used to refer to the existing compiler with the new code generator because this compiler is used to bootstrap itself to the new host by self-compilation.

The possibility of having a compiler with multiple front ends or back ends introduces a problem in naming the compiler: should a compiler with a single front end and several back ends (code generators) be referred to as one compiler or many? What if the compiler has multiple front ends too?

The convention used in this paper is that compilers are regarded as different if they have different front ends. Front ends are different for different languages or different host computers. If a front end has been separately combined with different code generators and is maintained separately, eventually, through repeated error corrections and enhancements, the compilers will diverge in the way they handle the same source program. Thus a compiler with multiple front ends is regarded as multiple compilers. In the more frequent case where a single front end is combined with more than one code generator, the compiler will be regarded as a single product.

Appendix C

TABLE OF JOVIAL J2, J3, and J4 FEATURES

1. PROGRAM ELEMENTS

1.1 Characters

Letters:	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
Digits:	1 2 3 4 5 6 7 8 9 0
Marks:	+ - * / . , = ( ) ' $ blank

1.2 Identifiers

A sequence of 2 to 120 (J4 allows 2-136) letters, digits, or primes beginning with a letter or prime. No marks other than prime. No two consecutive primes. Last sign must not be a prime. Single letters are index characters not identifiers (see section 4.5).

1.3 Reserved Words

ABS	CLOSE	GOTO	LQ	OPEN	SHUT
ALL	DEFINE	GQ	LS	OR	START
AND	DIRECT	GR	MANT	ORIF	STOP
ARRAY	END	IF	MODE	OUTPUT	STRING
ASSIGN	ENT	IFEITH	NENT	OVERLAY	SWITCH
BEGIN	ENTRY	INPUT	NOT	POS	TABLE
BIT	EQ	ITEM	NQ	PROC	TERM
BYTE	FILE	JOVIAL	NWDSEN	PROGRAM	TEST
CHAR	FOR	LOC	ODD	RETURN	

1.4 Operators

Assignment:	=				
Arithmetic:	+	-	*	/	**
Boolean:	AND	OR	NOT		
Relational:	EQ	(Equal)		NQ	(Not equal)
	LS	(Less than)		LQ	Less then/equal to)
	GR	(Greater than)		GQ	(Greater then/equal to)

1.5 Separators

($ $)	encloses subscript expressions
(* *)	alternate form for indicating exponents
(/ /)	absolute value of enclosed expression
' '	encloses comments

START TERM	encloses the program
BEGIN END	encloses compound statements
DIRECT JOVIAL	encloses direct machine code
CLOSE END	encloses closed subroutines
PROC END	encloses subroutines (functions & procedures)
.	statement labels are followed by period
,	separates lists of items or parameters
==	exchange operator
...	expected range of numeric items
$	statement separator.

1.6 Program format and Comments

JOVIAL programs are free format. Any number of blanks or new line may be used between symbols. Blanks must not be embedded in names or reserved words. A comment is a string of JOVIAL characters (except the symbols $ and '') delimited by '' and '' (double primes).

2. DATA TYPES

2.1 Item Declarations

Data items (the term item was taken from SAGE) are scalar variables. Items can be either full or part words. In J2, no item could be more than one word long, but this was changed in later versions of the language [SCHW78]. Only one variable may be declared per statement. Examples are:

```
Integer:        ITEM TALLY I 15 U $
                ITEM SUM   I 10 S R 0...100 $
Fixed:          ITEM AREA  A 25 S 7 0.A7...1.E5A7 $
Floating:       ITEM PP    F P 1.23E-4 $
Dual:           ITEM PLACE D 16 S 3 $
Boolean:        ITEM SPARE B $
Character:      ITEM FIRST H 30 $
                ITEM LAST  T 40 $
Status:         ITEM OP'TYPE S 6 V(ARI) V(REL) V(LOG) $
Implicit:       MODE F $
```

TALLY is an an unsigned (U) integer item that is stored in at least a 15-bit internal machine representation.
SUM is a 10-bit signed (S) integer that is rounded (R) rather than truncated upon assignment and ranges from 0 to 100 in value. This optional range information may be used by the compiler for optimization.
AREA is a fixed-point (A) item that has 25 bits, of which 1 is a sign bit, 7 are fractional bits after the binary point, and the remaining 17 bit are to the left of the binary point.
PP is a rounded, floating-point (F) item which is preset (P) with an initial value of 1.25E-4.

PLACE is a dual (D), fixed-point, signed item with 16 bits per each component and 3 bits to the right of the binary point.
SPARE is a Boolean (B) item.
FIRST is Hollerith (H) character item with a maximum length of 30 characters.
LAST is a character item stored in standard Transmission (T) code (STC) format.
OP'TYPE is a status (S) item. Status items have programmer-defined symbolic values, which can be referenced without the programmer having to know the underlying representation. The 6 indicates that the values of OP'TYPE are stored internally as 6-bit integers. Correspondence between two identical status values and their associated items is done by context.
The MODE directive allows implicit declaration of undeclared variables. The above statement causes any undeclared variable to be floating-point until a new MODE statement is encountered. The default MODE is INTEGER.

2.2 Literals

The following illustrate typical literal values for the items just presented.

Integer:	27, -0039, +331E9 (exponents are to base 10)
Octal:	O(27), O(39)
Fixed:	-123.A4, 5.5A5, +.678E9A-20
Floating:	27., -0.039, +3.31E-6
Dual:	D(2,2), D(-5.9E-5A5,89.1E-4A5)
Boolean:	1 denotes true, 0 denotes false
Character:	9H(THIS IS IT), 11T(THIS WAS IT)
Status:	V(HEARTS), V(CLUBS), V(SPADES), V(DIAMONDS)
Location:	Positive integer designating machine address

Character literals are represented interally as either six-bit Hollerith or six-bit standard Transmission code. Octal literals denote unsigned integers and also the machine representation of characters. For example, O(371625) and 3T(ZIP) are represented internally by the same 18-bit symbol.

2.3 ARRAY Declarations

Arrays may have any number of dimensions, but the number must be fixed at compile time. Subscripts range from 0 to the maximum defined for each dimension minus 1. Array elements may be of any type. Subscript types may be any expression that evaluates to an integer, including other subscripted variables. Some examples are:

```
ARRAY ALPHA 2 4 3 I 7 U 0...99 $
ARRAY TIC'TAC'TOE 3 3 Status V(EMPTY) V(ZERO) V(CROSS) $
ARRAY ALPHA 4 F $ BEGIN 1.14 .67 .203 3.14 END
```

ALPHA is a 2 by 4 by 3 array of positive integers less than 100. TIC'TAC'TOE is a 3 by 3 matrix of status values. ALPHA is an array of 4 floating-point values with a preset values of 1.14, .67, etc.

2.4 TABLE Declarations

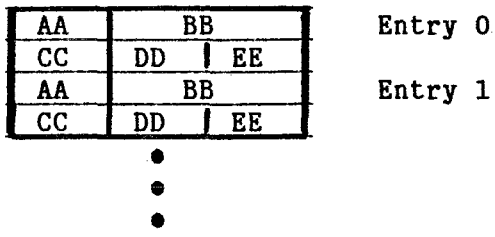
TABLEs are data structures that consists of one or more entries. Each entry, in turn, may consist of one or more items that may differ in size and type. All entries of a TABLE have the same composition and structure.

Table Dimension Lists. Tables may have a rigid (R) or fixed number of entries, or they may have a variable (V) number of entries up to some maximum. For example:

```
TABLE PAYROLL V 1000 S M $
BEGIN
ITEM EMP'NAME  H 18 $
ITEM MAN'NMBR  I 12 U $
ITEM ORG'CODE  S V(SALES) V(PROD) V(ENG) $
END
```

PAYROLL is a variable table which may have up to 1000 entries, each of which has three items, EMP'NAME, MAN'NMBR, and ORG'CODE.

Table Structure. Tables may also have serial (S) or parallel (P) structure. In a serial table, all words containing an entry are stored contiguously in memory. In a parallel table similar items are stored together. Referencing individual items is independent of the entry size or parallel/serial format, and invisible to the programmer. For example, a serial table containing the items AA, BB, CC, DD, and EE, requiring three words per entry would be stored as follows:



A parallel table containing the same items would be organized as follows:

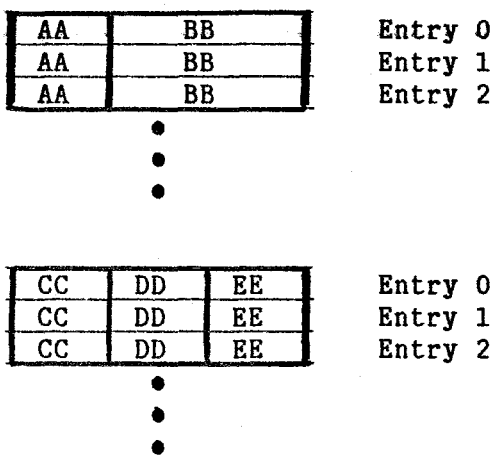


Table Packing. Storage allocation (packing) by the compiler may be optionally specified according to three schemes. No packing (N) means that each item in an entry will occupy one or more full computer words. Medium packing (M) means that each item will occupy one or more machine-oriented fields of a computer word. In early computers, these would most likely have been the address field or the operation field. Today most machine-oriented fields probably would be bytes. Dense packing (D) means that storage is allocated in bit units, so that each item will occupy one or more consecutive bit positions.

Specified Table Entries. Table can be either ordinary or specified. The declaration of ordinary tables was just described. Specified tables allow the programmer to specify the position of each item of an entry. For example, the following specified table declaration is given for a computer with 36-bit words and 6-bit bytes:

```
TABLE XX R 500 P 3''words per entry''$
BEGIN           ''Word  Bit  Packing''
ITEM AA F R        0    00     N   $
ITEM BB A 20 S 5   1    00     D   $
ITEM CC T 8        1    25     M   $
END
```

Table XX is a has a fixed (R) length of 500 entries and parallel structure with 3 words per entry. Item AA begins at bit 00 of word 0 of the entry and has no packing. Item BB begins at bit 00 of word 1 of the entry and has dense packing. Item CC begins at bit 25 of word 1 and has medium packing. Item CC can begin in the same word as item BB because BB is only 20 bits long and has dense packing.

STRING Declarations. A STRING is an item that occurs many times per entry. Each such occurrence is called a bead. STRINGs are a rather complex structure that were very difficult to implement. The STRING capability went way beyond the original concept of items and character string items [SCHW78]. String items are declared similarily to specified items. Instead of ITEM, the word STRING is used, and two additional factors are added to the declaration: 1) an interval factor giving the frequency of occurence of the string item in the entry, and 2) a packing factor giving the number of beads in each word of the entry. For example, table XX is a rigid table with 10 entries, 3 words per entry, with string item ZEUS:

```
TABLE XX R 10 S 3 $
BEGIN
STRING ZEUS I 11 U 0 1 2 3 $
END $
```

This declares that the beads of ZEUS are 11-bit Unsigned Integers. The first bead starts in word 0, bit 1 of the entry. There are beads in every 2nd word of the entry, and there are 3 beads in each word of the entry that contains beads. The starting bit position remains the same in every word that contains beads. A two-component index is used to reference string items. ZEUS ($0,4$) indicates bead 0, in entry 4 of the table. The following illustrates entry 4 of TABLE XX:

word 0	ZEUS ($0,4$)	ZEUS ($1,4$)	ZEUS ($2,4)
word 1			
word 2	ZEUS ($3,4$)	ZEUS ($4,4$)	ZEUS ($5,4)

LIKE Table Declarations. Sometimes it is necessary to have two or more tables with the same entry structure. To avoid redeclaration of each table, the LIKE descriptor (L) is used. The new table is declared using the name of previously declared table with a distinguishing letter or numerial suffix and the LIKE descriptor:

```
TABLE PAYROLLA R 10 P D L $
```

PAYROLLA is a new table with a Rigid number of 10 entries, in Parallel with Dense packing. The table items are Like those in PAYROLL and are automatically declared to be EMP'NAMEA, MAN'NMBRA, and ORG'CODEA.

2.5 Overlay Declarations

The OVERLAY declaration specifies the names of variables that share the same memory locations. For example:

```
OVERLAY AA=BB=CC,DD,EE $
```

This causes the items AA, BB and CC to be allocated storage beginning at the same word or byte boundary. In addition, item CC, DD, and EE are allocated space in consecutive words. OVERYLAY is used with previously declared item, arrays, and tables.

2.6 Data References

Simple items are reference by giving the name of the item. Arrays and items which are part of tables (indexed variables) are referenced by giving the array name or item name followed by an index in parentheses. Each subscript value in the index specifies a dimension of an array. Thus AA($1,2,0$) indicates row 1, column 2, and plane 0 of array AA.

There is no way to qualify the table items with the table name, e.g., PAYROLL.EMP'NAME for the table declared in section 2.4. Items are referenced as though they were separate items and not part of a table. Therefore, names of table items must be unique for each table; no two tables can have the items of the same name. Thus, EMP'NAME(3) refers to the item EMP'NAME in entry 3 of the table PAYROLL declared above.

3. EXPRESSIONS

3.1 Expression Structure

Expressions describe the computation of a value. The value of an expression has a type associated with it, which is determined by the types of its operands. Like FORTRAN J3 handles most type conversions implicitly. although there are elaborate rules for scaling dissimilair types. Evaluation of expressions proceedes according to the following operator precedence (from highest to lowest):

-	unary minus
** (* *)	exponentiation
* /	multiplication and division
+ -	addition and subtraction
EQ NQ LS LQ GR GQ	relational operators
NOT	logical negation
AND OR	logical conjunction and disjunction

Evaluation of operators of equal precedence is in strict left to right sequence.

3.2 Numeric Expressions

```
Integer:    TALLY = SUM + 1 $
Floating:   PP =  AA/3.0 + 4.**(BB/3.) + CC/3 $
Fixed:      AREA = 3.1415900A7 * 2A7 * RADIUS $
Dual:       PLACE = D(4.5A3, 9.A3) ** D(2, 2) $
          ''PLACE is now equal to D(20.250A3, 81.000A3)''
```

3.3 Bit Expressions

Relational operators (EQ NQ LS LQ GR GQ) apply to all scalar types: numeric, Boolean, status, and character. Boolean expressions involve the three logical operators, AND OR and NOT, and apply only to Boolean variables. There are no logical operators that work on bit strings, but the BIT function treats its operands as if they were strings of bits even though the operands are declared as numeric, logical, Boolean, or character.

3.4 Character Expressions

Character values are represented by character types and octal literals. Character expressions are limited to character literals, character items, and character functions. Character types may be assigned and compared and operated on by the BIT and BYTE functions.

4. STATEMENTS

4.1 Statement Structure and Format

Statement may begin in any column from 1-72 and may extend over as many lines as necessary with more than one statement per line. Simple Statements are separated by the $ mark. Compound Statements are enclosed in BEGIN END brackets and can be placed anywhere a simple statement can.

4.2 Statement Labels

Statement labels consist of an identifier followed by a period and are declared by their use. There are no label variables. There may be multiple labels per statement:

```
LAST. CEASE. DESIST.  STOP $
```

4.3 Assignment Statement

An assignment statement assigns an expression to a variable. All scalar variables (not tables or arrays) may be assigned to: numeric, Boolean, status or character. Numeric types are integer, fixed-point, floating-point, dual fixed-point and are converted upon assignment. For other types both sides must agree in type. Multiple assignments are not available. The format for the assignment statement is:

```
variable = expression $
```

4.4 Exchange Statement

An exchange statement exchanges the values of the two variables. Exchange of all scalar types is allowed The exchange statement has the following format:

```
AA == BB $
```

4.5 IF Statement

The basic IF statement has the following form:

```
IF AA - BB LS 2 $    ''IF AA IS LESS THAN BB,''
   AA = 25   $       ''THEN SET AA TO 25     ''
```

If the Boolean expression is true, the statement following it is executed; otherwise it is skipped. IF statements may be nested to implement complex logic:

```
IF COND EQ V(RED) $                ''IF COND EQ RED THEN,''
   BEGIN IF SPEED EQ V(MAX)$       ''IF SPEED EQ MAX THEN,''
            ACTION = 1 $           ''SET ACTION TO 1''
            GOTO NEXT $            ''SKIP OVER ELSE CLAUSE''
   END
                                   '' IF COND NQ RED THEN,''
ACTION = 2 $                       '' SET ACTION TO 2''
NEXT: . . .                    '' NEXT STATEMENT IN PROGRAM''
```

4.6 ALTERNATIVE Statement

The function of a CASE statement is accomplished with the alternative statement, thus:

```
IFEITH  B1 $  S1$    ''If B1 is true, then do S1''
ORIF    B2 $  S2$    ''else if B2 is true, then do S2''
...     ...   ...       ...
ORIF    1  $  Sn$    ''otherwise, do Sn''
END
```

4.7 Loop Statements

J3 has only one loop statement, the FOR statement. The loop control index is a signed integer variable designated by a single alphabetic letter. Loop controls are activated and assigned initial values by the execution of FOR statements. Transfers out of loops via GOTO's are allowed at any point; however, the loop control is undefined outside of the loop. Basic forms of the FOR statement are as follows:

```
FOR L=I $                    Where:  L = index subscript
FOR L=I,D $                          I = initial value
FOR L=I,D,F $                        D = decrement or increment
FOR L=ALL(table'name ) $             F = final value
```

The I, D, and F may be any positive or negative numeric value that is truncatable to integer. Furthermore, D and F may change during execution, as they are recomputed after each iteration. The first form of the FOR statement merely activates the subscript, L, and assigns it the value specified by evaluating I and converting to integer by truncation. The second form causes indefinite looping with L incremented by D after each iteration. The third form causes L to be incremented by D after each iteration with looping terminated when L exceeds F. The fourth form causes implicit iteration thru all data group elements.

The first statement after the FOR statement is the iterated statement. Multiple statements must be grouped as a compound statement. The evaluation of the inner product of two vectors, AA and BB illustrates use of the FOR statement:

```
ARRAY AA 50 F R $
ARRAY BB 50 F R $
ITEM  DOT   F R P 0.0 $
BEGIN                    ''TO LOOP IN REVERSE DIRECTION USE:''
FOR I = 0,1,49 $          ''FOR I = 49,-1,0 ''
      BEGIN
      DOT = DOT + AA($I$)*BB($I$) $
      AA($I$) = 0 $   BB($I$) = 0 $
      END
END
```

4.8 TEST Statement

The TEST statement, TEST $ or TEST L $, causes control to skip to the loop and the loop to recycle. If a loop subscript (L in this case) is given, control passes to the end of the indicated loop; otherwise control passes to the innermost loop.

4.9 GOTO Statement

There are four forms of GOTO statement:

```
GOTO  statement'label $
GOTO  close'name $
GOTO  program'name $
GOTO  switch'name $
```

The first causes an uncondition jump to the statement with the label given in the statement. The second causes invocation of a CLOSE statement. The third calls an external procedure. The fourth form is explained under the switch statement.

4.10 SWITCH Statement

Multiway branches are implemented with SWITCH statements. There are two kinds of switch statements: index (numeric) switches and item (value) switches. The index switch was taken from the ALGOL and is used as follows:

```
SWITCH TOGGLE =  (BL97,  , LOOP, EMIT ) $
GOTO TOGGLE ($ K $) $
```

When the GOTO statement is executed, if K=0, control will transfer to the statement labelled BL97. If K=1, no action will be taken. IF K=2, control will transfer to LOOP, etc.

JOVIAL is the only language with an item switch. The index switch of ALGOL only provided for branching based on integer items. This was not sufficient for the status items and character item allowed by JOVIAL. So the item switch was created. An item switch is declared and used thusly:

```
SWITCH WHICH (BETA) =  (3H(ARY)=ST34, 3H(0L9)=S01,
                        3(''')=EXIT) $
GOTO WHICH ($ BETA $)
```

When GOTO WHICH is encountered, if the value of BETA is equal to ARY, control is transferred to statement ST34, etc.

4.11 RETURN Statement

A RETURN statement, RETURN $, effects a normal return from a subroutine (procedure, function, or close) from anywhere in the subroutine body. An implicit RETURN occurs at the END statement.

4.12 STOP Statement

A STOP statement causes program execution to stop until restarted by the operator. If STOP STEP1$ is written and the program is restarted, execution will resume at statement STEP1, otherwise execution would resume at the statement following stop.

4.13 I/O Statements

Input/Output statments were not a part of JOVIAL/J2. A file declaration is used to name and describe a FILE on some hardware device used for I/O. For example:

```
FILE   SNAP  H  200  V  120
       V(READY)  V(BUSY)  V(ERROR)  V(EOR)
       TAPE5 $
```

SNAP is declared to be a file of Hollerith (H) characters with a maximum of 200 records. Each record has a variable (V) length of up to 120 bytes. Files may also be binary (B), where I/O is done without conversion from internal machine format and records may be rigid (R). The size of a binary record is given in bits per record. The status constants give information on device status. TAPE5 is an installation defined device name.

Files may be opened, read or written, and shut with the following statements:

```
OPEN INPUT filename   $      OPEN OUTPUT filename  $
INPUT filename block  $      OUTPUT filename block $
SHUT INPUT filename   $      SHUT OUTPUT filename  $
```

where filename is a previously declared file and block is a single variable, an entire array or table, or a consecutive set of table entries.

4.14 FORMAT Statements

Formatting facilities are only provided in J4. The I/O format description is written as part of the I/O statements. Free format is not available. Device control is tailored to specific systems. Examples of format statements are:

```
FORMIN($U$)(format list)$
FORMOUT($U$)(format list)$
DECODE($buffer$)(format list)$
ENCODE($buffer$)(format list)$
format list includes variables and formats.
```

5. SUBROUTINES

A subroutine may be either a procedure or a function. Recursive subroutine calls are prohibited. Subroutines may not contain other subroutine declarations, although subroutines may call other subroutines.

5.1 Procedures

Procedure declarations are similar to other languages. The procedure REMQUO illustrates a procedure declaration:

```
PROC REMQUO (NUM, DEN = QUO, REM) $
      ITEM NUM I 36 S $        ITEM DEN I 36 S $
      ITEM QUO I 36 S $        ITEM REM I 36 S $
BEGIN
      QUO = NUM/DEN$     REM = NUM-QUO*DEN$
END
```

5.2 Functions

A function differs from a procedure by the presence of an item declaration with the same name as the function name. The type of the function is defined by the type of this item. This item is the sole output parameter since the function declaration does not provide for a formal output parameter. Unlike a procedure, a function is used in an expression and may not contain an ABORT phrase. An example of a function is as follows:

```
PROC NEAR (AA,BB) $
  ITEM NEAR B $ ''INDICATES NEAR IS A BOOLEAN FUNCTION''
  ITEM AA F $
  ITEM BB F $    ''BEGIN END NOT REQUIRED FOR SINGLE STMT''
  NEAR = CHAR(AA) EQ CHAR(BB)$
```

5.3 Parameters

Subroutines may or may not have parameters. Input parameters appear to the left of the = separator, and output parameters to the right. If the parameter list contains only output parameters, the = must be present to indicate that they are output parameters. Input parameters may be ITEMs, ARRAYs, TABLEs, or close statement names. A close name must be followed by a period. Output parameters may be ITEMs, ARRAYs, TABLEs, or statement labels. Statement labels must be followed by a period. Formal parameters are declared like other data objects.

J3 has two types of binding: value or reference (name). In value binding, formal parameters are allocated memory space and

the value of the actual parameter is copied into that memory. In call by reference, no new object is created; the actual and formal parameters denote the same physical object. Any change in the formal parameter results in an immediate change to the value of the actual parameter.

The way in which an actual parameter is bound to its formal parameter is determined by whether or not a declaration for the parameter is given in the subroutine. If a declaration is given, a new data object is created and binding is by value; otherwise, the parameter is passed by reference.

5.4 CLOSE Subroutines

A CLOSE subroutine is a group of statements that act as a program dependent subroutine. A CLOSE subroutine is closed in the sense of being removed from the normal statement execution sequence. The compiler isolates this group of statements by providing a transfer around it if the programmer fails to do so by means of a GOTO statement. The CLOSE subroutine has no parameters; it is restricted to using the global variables in its containing procedure. The CLOSE statement was added to JOVIAL because it can be called within the domain of an index subscript and hence access the index subscript [SCHW78]. The format of a CLOSE subroutine is as follows:

```
CLOSE close'name $
BEGIN
 ...     ''SEQUENCE OF STATEMENTS''
END
```

5.5 Subroutine Call Statements

Procedures are invoked by giving the procedure name and list of actual parameters: REMQUO(13, 3 = QQ, RR) $. Functions are called by use of the function name in an expression:

```
IF NEAR (3.14, 3.14E10) $
XX = RANDOM () $
```

The second statement illustrates a parameterless function call. CLOSE statements are invoked with the GOTO statement and may be invoked only within the procedure or main program in which it is declared. Control is returned to the statement following the GOTO after execution of the CLOSE.

6. BUILT-IN FUNCTIONS

There are two kinds of built-in functions: functions and functional modifiers. BIT, BYTE, CHAR, ENT, MANT, NENT,

NWDSEN, POS, and SIGN are functional modifiers. Functional modifiers were added during the early development of JOVIAL to allow machine access yet retain freedom from machine specifics. They differ from functions in that they can be used on either side of assignment statements.

6.1 ABS

ABS returns the absolute value of the item passed to it. Two forms are allowed: ABS (-25) = (/ 25 /) = 25.

6.2 SIGN

The sign of an item may be accessed with the SIGN modifier. The following sets the sign of ITEM1 to 1:

```
SIGN(ITEM1) = 1$
```

6.3 BIT and BYTE

A significant feature of JOVIAL is the provision for access to bits and bytes of items provided by the BIT and BYTE modifiers (The term BYTE was was originally coined by Werner Buchholz as part of the definition of the IBM 7030, STRETCH, computer and later used in the Q-31 description). Although these functions were powerful, they were not efficient to implement in the first J2 and J3 compilers [SCHW78].

The internal value of any item may be considered a string of bits or, in the case of character items, of bytes. In either case, bits or bytes are indexed left to right, beginning with zero. The BIT function allows access to any substring of an item:

```
ITEM MASK I 4 U P 1 $          BIT ($0,3$)(MASK) = 6 $
```

The string of 3 bits starting at bit 0 of item MASK (which is a four bit item with initial value of 1 = 0001) is assigned the number 6, which is the bit string "110". The resulting value of MASK is 1101 = 13. The BYTE function is similar:

```
BYTE($I, 3$)(SYM) = BYTE($J, 3)(BOL) $
```

sets the three bytes starting with the Ith byte of SYM to the three bytes of item BOL starting with the Jth byte.

6.4 LOC

LOC provides the capability of accessing relocatable information. LOC(ITEM) returns an integer equal to the absolution machine location of an item. For tables, named statements, or programs, the address of the first word is returned.

6.5 NWDSEN

The Number of WorDS per ENtry (NWDSEN) returns the number of words of storage allocated to each table entry in the table given as an argument. NWDSEN is useful in programs that perform dynamic storage allocation. For example, the total storage for table PAYROLL is given as:

```
PAYROLL'SIZE'IN'WORDS = NWDSEN(PAYROLL)*NENT(PAYROLL) $
```

6.6 NENT

A vital parameter in table processing is the Number of ENTries. For rigid tables, NENT acts as a preset parameter that can be used to parameterize statements that refer to the number of entries in a table. For variable length tables, NENT acts as a counter of the current number of table entries. The programmer would code the following for each new entry to table PAYROLL:

```
NENT(PAYROLL) = NENT(PAYROLL)+1 $
```

6.7 ENT

A table entry is conglomeration of related items. The ENTry function allows an entry to be manipulated as a single value, which is not denotable, except when all the bits in the entry have the value 0, i.e. aggregate values are not denotable as in Ada. Entries can be assigned, exchanged, or compared for equality or inequality. For example:

```
ENT(PAYROLL($47$)) = ENT(PAYROLL($52$)) $
ENT( PAYROLL($47$) ) = 0 $
```

6.8 MANT and CHAR

A floating point item consists of a mantissa and a characteristic. Either component may be accessed as a fixed-point value with the MANT and CHAR functions. Thus, the fixed-point value of the floating-point item ALPHA can be expressed as:

```
MANT(ALPHA)*2**CHAR(ALPHA)
```

6.9 ODD

ODD returns a Boolean value of true if the least significant bit of a subscript or numeric item is a one (i.e. for integers if the number is odd) and a value of false if the least significant bit is a zero, i.e. even.

6.10 REM and REMQUO

REM is function that yields the remainder in a division of two integers. REMQUO is a procedure that yields both remainder and quotient of a division of two integers.

6.11 ALL

The ALL function causes a FOR loop to cycle through an entire table without concern for table length or structure. The following forms are equivalent:

```
FOR T=0,1,NENT(PAYROLL)-1 $     ''STEP IN ASCENDING ORDER''
FOR T=NENT(PAYROLL)-1,-1,0 $    ''STEP IN REVERSE ORDER  ''
FOR T=ALL(PAYROLL) $            ''DIRECTION NOT SPECIFIED''
```

6.12 POS

POS designates the record position of a file. The value 0 corresponds to a position before the first record of the file. For a file of k records, the value k corresponds to the position after the last record, i.e., end of file. POS changes whenever the file is used for input or output. POS can be changed by assignment to reposition a file. For example, the first statement backspaces the file inventory one record; the second rewinds the file:

```
POS(INVENTORY) = POS(INVENTORY) - 1 $
POS(INVENTORY) = 0 $
```

7. PROGRAM STURCTURE

A JOVIAL program consists of one or more main programs together with zero or more compools. Each main program may contain subroutines and can call any other main program. Because of this mutual calling ability, JOVIAL main programs can have parameters and may be classed into the same types as subroutines: procedures, functions, and closes.

7.1 Main Programs

A main program is a string of declarations, statements, and directives enclosed between START TERM delimiters. A program can be compiled as an independent entity, or contained in another program as a close statement. The format of a program is as follows:

```
CLOSE name $
START origin $
      statements ...
TERM $
```

The first line is optional. If the program begins with CLOSE, it is a closed subroutine and the name before START is the name of this subroutine. Otherwise it is an independent program. If the origin is specified, it gives the machine location at which the compiled program begins; if not, the compiler chooses the starting location.

7.2 Compools

The compool is perhaps the most notable feature of JOVIAL, although it originated in the SAGE program. Compools contain the data declaration for all the item and tables to be shared by the independent programs of the system. The compool is contains the same information as the data declaration section within a program.

A compool may be used like a data dictionary to allow the programmer to use an item without haveing to specify where to place the value or its internal form. The compool can be used to implement a debugging program or serve as a tool for data reduction [TJOM60] after tests are run.

7.3 Scope of Names

Names fall into three categories: 1) device names, 2) statement and program names, and 3) data names. Duplicate names between categories are allowed. The meaning of duplicate names within categories depends on program scope. Names defined in main programs are global to all subprograms. Names defined within a subprogram are local to it and hide names defined outside. Duplicate names within the same scope are not allowed. Compool names are global to all programs.

8. COMPILER AND MACHINE INTERFACE

8.1 DEFINE Declaration

The define capability provides a compiler macro capability. A define declaration associates a name with a string of JOVIAL code. The compiler then substitues the associated code everywhere the define identifier appears in the source text. Defines can be used to parameterize programs and to make code more readable. For example, if the following defines are used:

```
AA = BB + CC * DD ;
DEFINE PLUS       ''+'' $
DEFINE MULTIPLIED ''*'' $
DEFINE REPLACED   ''='' $
DEFINE IS         '' '' $
DEFINE BY         '' '' $
```

A programmer can write the following, otherwise invalid JOVIAL statement in his program:

```
AA IS REPLACED BY BB PLUS CC MULTIPLIED BY DD $
```

and the compiler will substitute the defined strings for the define names to arrive at the following valid JOVIAL statement which it then compiles:

```
AA = BB + CC * DD $
```

8.2 DIRECT/JOVIAL Directive

Although the purpose of JOVIAL was to provide for machine oriented programming without the use of machine language, JOVIAL allows the flexibility of including machine language in JOVIAL source programs. The keyword DIRECT signals the beginning of machine language code and the keyword JOVIAL the end.

Within the machine language section, access to JOVIAL items, tables, and arrays achieved via the ASSIGN operator, which causes JOVIAL items to be moved into or out of the accumulator. Scaling and shifting is handled by the compiler. For example: ASSIGN A(10) = ALPHA($I,J**2$) $ assigns the subscripted value of ALPHA to the accumulator as a fixed-point value with 10 fractional bits.

Appendix D

TABLE OF JOVIAL J73 (MIL-STD-1589B) FEATURES

1. PROGRAM ELEMENTS

1.1 Characters

```
Letters:   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
Digits:    1 2 3 4 5 6 7 8 9 0
Marks:     + - * / . , = ( ) ' $ blank
           < > @ : ; " % !
```

Lower case letters are allowed but are not distinguished from upper case except in character literals.

1.2 Identifiers

A sequence of 2 to 31 letters, digits, or primes beginning with a letter or a dollar sign. A name may not begin with a prime. Single letters are index characters, not identifiers.

1.3 Reserved Words

ABORT	CONDITION*	FREE*	MOD	REC	TERM
ABS	CONSTANT	GOTO	NENT*	REF	THEN
AND	DEF	HANDLER*	NEW*	REGISTER*	TO*
BEGIN	DEFAULT	IF	NEXT	RENT	TRUE
BIT	DEFINE	IN*	NOT	REP	TYPE
BITSIZE	ELSE	INLINE	NULL	RETURN	UBOUND
BLOCK	ENCAPSULATION*		NWDSEN	SGN	UPDATE*
BY	END	INSTANCE	OR	SHIFTL	WHILE
BYREF	EQV	INTERRUPT*	OVERLAY	SHIFTR	WITH*
BYRES	EXIT	ITEM	PARALLEL	SIGNAL*	WORDSIZE
BYTE	EXPORTS*	LABEL	POS	START	WRITEONLY*
BYTESIZE	FALLTHRU	LAST	PROC	STATIC	XOR
BYVAL	FALSE	LBOUND	PROGRAM	STATUS	ZONE*
CASE	FIRST	LIKE	PROTECTED*	STOP	
COMPOOL	FOR	LOC	READONLY*	TABLE	

Starred words are not used in J73B. They were included in the standard for future language features.

1.4 Operators

```
Assignment: =
Arithmetic: +  -  *  /  **  MOD
```

```
Boolean:     AND  OR  NOT  XOR  EQV
Relational:  =    (Equal)              <>   (Not equal)
             <    (Less than)          <=   (Less then/equal to)
             >    (Greater than)       >=   (Greater then/equal to)
Dereference:@
```

1.5 Separators

(* *)	type conversions
" " or % %	encloses comments
START TERM	encloses the program
BEGIN END	encloses compound statements
PROC END	encloses procedures and functions
:	statement label, case label, loop control, dimension, subrange, and parameter separator
,	separates lists of items or parameters
;	statement separator
!	indicates compiler directives and define-string formal parameters

1.6 Program format and Comments

JOVIAL programs are free format. Any number of blanks or new lines may be used between symbols. Blanks must not be embedded in names, reserved words, or symbols except in a character literal, a comment, or a define-string. A comment is a sequence of characters delimited " and " (double quotes) or % and % (percents).

2. DATA TYPES

A data-declaration delares a variable, a constant, or a type as template for later declarations. Variables can change values during program execution; constants cannot. Storage allocation for variables is static unless a data object is declared in a subroutine in which case the allocation is automatic unless the declaration has the keyword STATIC. Allocation for constants is always considered to be static, eventhough they may not actually be allocated.

2.1 Item Declarations

An item is a scalar variable. In an integer, floating or fixed type description, the minimum number of bits occupied by the item may be specified. Only one item may be declared per statement; lists are not allowed. Examples are:

```
Integer:        ITEM TALLY U 15;
                ITEM SUM STATIC S 10;
                ITEM TIME U;
Fixed:          ITEM AREA  A 17,7;
```

```
Floating:       ITEM PP  F,R 30 = 1.23E-4;
                ITEM VELOCITY F;
Bit:            ITEM MASK B 3 = 1B'101';
                ITEM SPARE B;
Character:      ITEM FIRST C 30 = 'JOHN';
Status:         ITEM OP'TYPE  STATUS
                (V(ARI),V(REL),V(LOG),V(ASN)) = V(REL);
Pointer:        ITEM PTR P = NULL;
                ITEM LINK  P PARTS = LOC(DATA);
```

TALLY is an an unsigned (U) integer item that is stored in at least a 15-bit internal machine representation.
SUM is a 10-bit signed (S) integer (plus an extra bit for sign) with STATIC allocation permanence even if it is declared in a subroutine.
TIME is an unsigned integer with default size. If BITSINWORD is 16 (section 6.11), TIME is allocated a minimum of 15 bits and can assume the values of 0 through MAXINT(15).
AREA is a fixed-point (A) item with at least 25 bits, of which 1 is a sign bit, 7 are after the binary point (fraction of 7), and the remaining 17 bits are to the left of the binary point (scale of 17).
PP is a floating (F) item with a precision of 30, i.e. its mantissa occupies a minimum of 30 bits. It has a preset (initial) value of 1.25E-4 that is rounded (R) before assignment to PP. If T or Z were used, the value would have been truncated toward minus infinity or toward zero, respectively, before assignment.
VELOCITY is a floating variable with default precision. If FLOATPRECISION is 12 (section 6.11), its mantissa occupies 12 bits.
MASK is a bit variable 3 bits long with an initial value of 101.
SPARE is a bit variable with default size of 1 bit. Bit variables one bit long are used for Boolean values.
FIRST is a character variable 30 characters long with an inital value of 'JOHN'.
OP'TYPE is a status (S) variable. The position of a status-constant in in the declaration determines order for relational expressions, e.g. V(ARITH) < V(RELAT).
Pointers contain address of items, tables, and blocks. PTR is an untyped pointer. LINK is a typed pointer and can only point to objects of type PARTS and has an initial value of LOC(DATA) the location of the object DATA. DATA must be of type PARTS.
All data must be declared. There are no implicit data declarations in J73.

2.2 Literals

The following illustrate typical literal values for the items just presented.

```
Integer:    27, -0039
Real:       27., -0.039, +3.31E-6
Bit:        4B'10AC6' = 1B'00010000101011000110'
            1B'1' denotes true, 1B'0' denotes false
Character:  'THIS IS IT', 'THIS WAS IT', '2+2=4'
Status:     V(HEARTS), V(CLUBS), V(SPADES), V(DIAMONDS)
Pointer:    NULL
```

A real literal can be interpreted as a floating or fixed type depending on its context.
A bit literal represents a bit string value. A bit literal is composed of a string of beads. The number of bits in each bead is given at the beginning of the bit literal as bead-size. The form of a bit literal is bead-size B 'bead ...' . Blanks are not permitted anywhere in a bit literal. The bead-size of a bit literal can be 1 through 5. Beads can be any digit or any letter from A through V. The digits 0 - 9 represent their actual values; the letters A - V represent the values 10 through 31, respectively. 4B'10AC6' has a bead size of 4 and is represents a bit literal in hexadecimal notation. It is equivalent to the second bit literal above. Boolean values are represented by one-bit literals. The value 1B'1' represents the Boolean literal TRUE, and 1B'0' the Boolean literal FALSE.

A character literal is a string of characters enclosed in single quotes.
Status values are declared by their appearence in a status type declaration. The form of a status literal is the letter V followed by a parenthesized name, which may be a name, a letter, or a reserved word.
There is only one pointer literal, namely, NULL.

2.3 Table Declarations

Table Dimension Lists. A table is a collection of data objects. A table can be dimensioned or undimensioned. An undimensioned table has only one entry. All entries of a TABLE have the same composition and structure. Tables are either ordinary tables or specified and can have parallel, serial, or tight serial structure. For example:

```
TABLE PAYROLL (1:5, 1000);
BEGIN
ITEM EMP'NAME  C 18;
ITEM MAN'NMBR  U 12;
ITEM ORG'CODE  STATUS (V(SALES), V(PROD), V(ENG));
END
```

PAYROLL is a serial table with two dimensions. The first dimension has a lower-bound of 1 and upper-bound of 5. The second dimension has an default lower bound of 0 and an upper bound of 1000. Thus, unlike J3, there are 1001 entries for the

second dimension and 5*1001 = 5005 entries in PAYROLL. Each entry contains three items, EMP'NAME, MAN'NUMBR, and ORG'CODE. The item EMP'NAME in the first entry is referenced as EMP'NAME(1,0). The bounds of a table may be status types as well as integer types:

```
ITEM INDEX STATUS (V(USA), V(CAN), V(ENG), V(FRA));
TABLE VOYAGE (V(USA): V(FRA));
      ITEM TIME U;
```

Arrays in J73 can be represented as either tables with a single item per entry or as tables with an unnamed entry. For example:

```
TABLE ALPHA (2, 4, 3);
  ITEM PART U 7;

TABLE TTT (3, 3) STATUS (V(EMPTY), V(ZERO), V(CROSS));
```

The first entry of ALPHA is referred to as PART(0,0,0). The first entry of TTT is referenced as TTT (0,0).

Table Structure. Tables may have serial or PARALLEL structure. In a serial table, all words containing an entry are stored contiguously in memory. In a parallel table similar items are stored together. The PARALLEL structures may be specified only for a table in which none of the items of an entry occupy more than one word. Referencing individual items is independent of the entry size or parallel/serial format. Given the following declaration:

```
TABLE RACE1 (100);
      BEGIN
      ITEM A1 U;
      ITEM B1 S;
      END
```

```
TABLE RACE2 (100) PARALLEL;
      BEGIN
      ITEM A2 U;
      ITEM B2 U;
      END
```

RACE1 is specified by default as a serial table and RACE is specified as having parallel structure. The compiler would store these tables as follows:

For serial tables the tight (T) structure may be specified, which causes the compiler to pack as many entries per word as possible. Along with the T specifier, bits-per-entry allows entries to be allocated with a specified number of bits. For example:

```
TABLE ATTENDANCE (1:1000) T 8;
      ITEM COUNT U 5;
```

causes each five bit item, COUNT, to be allocated in 8 bits. Tight structure defines the allocation of storage between entries in a dimensioned table, whereas packing defines the allocation of storage within an entry of an ordinary table. The default packing spec for tight tables is dense (D).

Ordinary Table Entries: Packing Specifier. Storage allocation (packing) of items within ordinary table entries may be optionally specified according to three schemes. No packing (N) means that each item in an entry is allocated in a new word. Medium packing (M) is implementation defined. Dense packing (D) means that storage is usually allocated in bit units (sometimes in bytes), so that several items of an entry may be packed into a word, but each entry begins in a new word. The packing specifier may be given as part of the table declaration, indicating that it applies to the entire table, or only for individual items in the table. All items are packed according to the table packing spec except those items that have a packing spec in their declarations.

Specified Table Entries. Tables can be either ordinary or specified. The declaration of ordinary tables was just described. Specified tables allow the programmer to specify the position of each item of an entry. Instead of a packing spec, specified tables have a table-kind specifier that indicates whether the table has fixed-length entries of a certain size (W) or variable-length entries (V). The position of each item in a specified table entry is given by a pos-clause: POS (startbit, startword), where the first bit of a word and the first word of an entry are numbered 0. Item positioning must take into account the number of bits in a word or entry.

For example, the following specified table declaration is given for a computer with 36-bit words and 6-bit bytes:

```
TABLE XX (1:500) PARALLEL W 3 "words per entry";
BEGIN                  "Bit  Word"
ITEM AA F,R          POS (00 , 0);
ITEM BB A 14,5       POS (00 , 1);
ITEM CC C 6          POS (00 , 2);
END
```

Table XX is a has parallel structure and 500 fixed-length entries with 3 words per entry. Item AA begins at bit 00 of word 0 of the entry. Item BB begins at bit 00 of word 1 of the entry. Item CC must begin in word 2 because J73 does not allow multi-word items in parallel tables as J3 does.

The TIGHT structure specifier may also be used with fixed-length specified tables. The compiler uses the minimum number of bits necessary for an entry, unless the bit-per-entry is specified. The entry-size (word-per-entry) must not be given following the W.

Each logical entry in a table with variable-length entries is composed of a different number of a different number of physical entries (words). Variable-length tables may not contain a preset or PARALLEL or TIGHT structure spec. A physical entry is one word long. It is up to the programmer to keep track of where each logical entry begins. For example:

```
TABLE PARTS (100) V;
      BEGIN
      ITEM ENTRY'SIZE  U   POS (0,0);
      ITEM NUMBER      C 5 POS (0,1);
      ITEM ON'HAND     U   POS (0,2);
      ITEM DEFECTIVE   U   POS (0,3);
      END
```

The total number of physical entries is 101. Logical entries of two, three, and four words are possible. The programmer keeps track of the number of physical words per entry in ENTRY'SIZE. Thus, ENTRY'SIZE(I) for an entry with all four items would be 4.

Like Table Declarations. Sometimes it is necessary to have two or more tables with the same entry structure. To avoid redeclaration of each table, the LIKE option is used. The like option is used only with table type declarations (see below). The new table type is declared using the name of previously declared table type with the keyword LIKE. Unlike J3, the type name of the new table need not conform to the type name in the LIKE option:

```
TYPE PAY'TYPE TABLE;
      BEGIN
      ITEM EMP'NAME  C 18;
      ITEM MAN'NMBR  U 12;
      ITEM ORG'CODE  STATUS (V(SALES) V(PROD) V(ENG));
      END

TYPE NEW'PAY'TYPE TABLE (1:10) PARALLEL LIKE PAY'TYPE D;
      ITEM PAY'CODE  U 5;
```

NEW'PAY'TYPE is a new table type with 10 entries, parallel organization, and dense packing. Each entry consists of the items like the type PAY'TYPE together with the additional item PAY'CODE: EMP'NAME, MAN'NMBR, ORG'CODE, and PAY'CODE.

2.4 Block Declarations

A block groups items, tables, and other blocks into contiguous storage and gives the collection of data objects a single name so the data may be manipulated as a whole. For example, blocks can be passed a parameters or declared external. To cause a block declared within a subroutine to have static allocation, the STATIC attribute is used. Initial values may be given for static blocks by giving presets for the data objects in the block. For example:

```
BLOCK MAINGROUP;
   BEGIN
   ITEM MASTER U = 22;
   ITEM MASTERCODE U ;
   TABLE INVENTORY (1000) = 1001(0,"XXXXX",0);
      ITEM ORDER'NUM U;
      ITEM VISUAL C 5;
      ITEM ONHAND U ;
      END
   BLOCK SUBGROUP;
      BEGIN
      ITEM MINOR U = 6;
      TABLE MINORTAB(100,100);
         ITEM SCORE U;
      END
   END
```

2.5 Type Declarations

A type declaration declares a name that may be used in a data declaration to describe the type attribute of that data object. This allows programmers to define their own data types and to use thosee types for program consistency. A type declaration declares a new type by associating a name with a data description. Item, table, and block data descriptions can be given type names. For items, the following type declarations together with the item declarations are equivalent to those given in section 2.1:

```
TYPE COUNTER ITEM U 15;          ITEM TALLY COUNTER;
TYPE INTEGER ITEM S 10;          ITEM SUM STATIC INTEGER;
TYPE FLOAT   ITEM F,R 30;        ITEM PP FLOAT = 1.23E-4;
TYPE BOOLEAN ITEM B;             ITEM SPARE BOOLEAN;
```

The type declaration can be used to declare a number of tables with similar entry-descriptions. For example:

```
TYPE PARTS TABLE;          TABLE BOLTS PARTS;
   BEGIN                   TABLE NUTS (100) PARTS;
   ITEM ID C 10;           TABLE WRENCHES (10, 20) PARTS;
   ITEM COUNT U;
   END
```

The above three tables have items with the same names, so those items must be qualified to indicate the table from which they come when the items are referenced. Qualification is achieved by using pointers (see below). Block type declarations must also be qualified. An example of a block type declaration and data declaration is as follows:

```
TYPE INVENTORY BLOCK          BLOCK XSTORE INVENTORY =
      BEGIN                         50, (50 (' ', 0));
      ITEM MAXSIZE U;
      TABLE PARTS (1:50);     BLOCK PARTSLIST INVENTORY;
            BEGIN
            ITEM ID C 10;
            ITEM COUNT U;
            END
      END
```

2.6 Constant Declarations

A constant item declaration begins with the reserved word CONSTANT and concludes with an item-preset. The value of a constant may not be changed during execution:

```
Integer:    CONSTANT ITEM TALLY U 15 = 47;
Fixed:      CONSTANT ITEM NET   A 17,7 = 4.7;
Bit:        CONSTANT ITEM TRUE  B = 1B'1';
            CONSTANT ITEM FALSE B = 1B'0';
Table:      CONSTANT TABLE THRESHOLDS (1:10);
              ITEM LEVEL U = 2, 4, 6, 8, 10, 12, 14, 16, 18, 20;
```

The elements of a contant table cannot be changed during program execution.

2.7 Initialization of Data Objects (Presets)

The form of item presets is illustrated in section 2.1. Table may also have preset values. Some or all of the items in a table can be preset. Presets can be given for individual items within the table or in the table attribute list. For example the following declarations are equivalent:

```
TABLE TT(1:3);                TABLE TT(1:3)=1,7,2,4,3,8;
  BEGIN                          BEGIN
  ITEM SPEED U=1,2,3;            ITEM SPEED U;
  ITEM DISTANCE U=7,4,8;         ITEM DISTANCE U;
  END                            END
```

Both declarations preset the items as follows:

```
SPEED(1)=1        SPEED(2)=2        SPEED(3)=3
DISTANCE(1)=7     DISTANCE(2)=4     DISTANCE(3)=8
```

Positioners aid in specifying the starting position for presetting one or more values. Repetition counts aid in presetting multiple values. For example, the following declaration:

```
TABLE SCHOOLSYSTEM(1:100);
      ITEM CLASS'SIZE U = 10(5), POS(50):10(0), POS(70):
1,1 ;
```

sets the first ten entries (1 - 10) to 5, the 10 entries beginning at position 50 (50 - 59) to 0, and entries 70 and 71 to 1.

2.8 Overlay Declarations

The OVERLAY declaration is used for allocating several data objects beginning at the same place in storage, for assigning data to a specified machine address, or for specifying the allocation order of a set of data objects. The names given in an overlay declaration must be previously declared. They may be names of items, tables, or blocks. They may not be names of items within tables or blocks. Data objects in an overlay declaration must all be in the same scope and all have either static or automatic storage allocation. For example, consider the following declarations:

```
TABLE AA (1:10);              ITEM BB U;        ITEM DD U;
      ITEM PARTS U;           ITEM DD U;        ITEM EE U;

OVERLAY AA : BB : CC,DD,EE ;
OVERLAY AA : W 9, BB: CC, W 8, DD, EE;
OVERLAY AA : BB : CC, (DD : EE);
OVERLAY POS (4701) : AA : BB;
OVERLAY AA,BB,CC;
```

This causes the items AA, BB and CC to be allocated storage beginning at the same word or byte boundary. In addition, item CC, DD, and EE are allocated space in consecutive words. Thus, PARTS(1), BB and CC all share the memord word; PARTS(2) and DD share the same word; and PARTS(3) and EE share the same word. OVERLAYs may also indicate the number of words to skip when allocating data in the overlay. The second declaration causes BB and PARTS(10) to share the same word. The third declaration causes items DD and EE to share the same word as PARTS(2). The fourth declaration causes AA to be allocated beginning at word 4701, Thus, PARTS(1) is allocated to word 4701, PARTS(2) to 4702, etc. Also, BB is allocated to word 4701. The overlay

declaration is used to specify the allocation order of items. Unlike the !ORDER directive, which spcifies allocation order within a table or block, the overlay directive specifies order between items, tables, or blocks. The fifth declaration assures the order of allocation for items AA, BB, and CC.

2.9 Data References

Simple references are made by giving the name of an item, a table, a block, or an item in an unsubscripted table:

```
TABLE STATISTICS;
      BEGIN
      ITEM COUNT U;
      ITEM WEIGHT F;
      END
```

```
BLOCK PARTSLIST;
      BEGIN
      ITEM MAXSIZE U;
      TABLE PARTS (1:50);
            BEGIN
            ITEM ID C 10;
            ITEM COUNT U;
            END
      END
```

A simple reference may be made to the table STATISTICS and the items within it (COUNT and WEIGHT) or the block PARTSLIST and the item and table within it (DATE and PARTS). Items declared within a dimensioned table are referenced by giving the item name followed by an index subscript. Each subscript in the index specifies a dimension of the table. Thus, AA(1,2,0) indicates row 1, column 2, and plane 0 of array AA.

For tables or block not declared with a type name, there is no way to qualify the table items with the table name. Items in these tables must be referrenced as though they were separate items and must be unique for each table in a scope; no two tables in a scope can have items with the same names.

Pointers in J73, unlike Pascal and Ada, are not used for creating and accessing dynamic data objects. Pointers may reference any data object. Pointers must be used to access the components of tables or blocks declared using a type name. A pointer is assigned the address of a particular data object, and the components of that data object are referenced with respect to the pointer. A pointer qualified reference contains a dereference, i.e. a memory location. A dereference treats the data object found at the address given by the value of the pointer as an object of the type associated with the pointer. For example, consider following declarations:

```
TYPE PARTS TABLE;
   BEGIN
   ITEM ID C 10;
   ITEM COUNT U;
   END
```

```
ITEM PTR P PARTS;
TABLE BOLTS2 PARTS;
TABLE NUTS (100) PARTS;
TABLE BOLTS1 PARTS;
TABLE WRENCHES (10,20) PARTS;
```

The pointer PTR is a type pointer declared to reference objects of type PARTS. The LOC built-in function, described in section 6, is used to obtain a machine address which is used to set the pointer PTR. For example:

```
PTR = LOC (BOLTS1);
```

causes the address of table BOLTS1 to be assigned to pointer PTR. Once PTR is set to point to BOLTS1, a reference may be made to the item COUNT in table BOLTS1 using a dereference:

```
COUNT @ PTR;
```

With dimensioned tables, items of individual entries may be referenced using two equivalent methods. The LOC function may be used to set a pointer to any given entry in the table. The items in that entry are then referenced as before. For example, to reference item ID in the 47 entry of table NUTS, we could first set PTR to the 47th item of NUTS which is also of type PARTS and then just dereference the item. Alternatively, we could set PTR to point to the address of the entire table NUTS and then reference the 47 entry of that table. These two mechods are illustrated below:

```
PTR = LOC( NUTS (47) );        ID @ PTR
```

or

```
PTR = LOC( NUTS );             ID (47) @ PTR
```

3.0 FORMULAS

3.1 Formula Structure

Formulas describe the computation of a value. The value of an formula has a type associated with it, which is determined by the types of its operands. With certain exceptions, the types of the operands must be the same, i.e. J73 is a strongly typed language (but not as strontly types as Ada or Pascal). Evaluation of formulas proceeds according to the following operator precedence (from highest to lowest):

```
-                    unary minus
**                   exponentiation
*  / MOD             multiplication, division, modulus
+  -                 addition and subtraction
= <  >  <=  >=  <>   relational operators
NOT AND OR XOR EQV   logical operators
```

Evaluation of operators of equal precedence is not specified unless the !LEFTRIGHT directive is in effect, in which case evaluation is from left to right.

3.2 Numeric Formulas

```
Integer:    TALLY = (SUM MOD 3)**3 + 1;
Floating:   PP = AA*3.0 + 4.** (BB/3.) + CC/3;
Fixed:      RADIUS =  (* A 17,7 *) DIAMETER/2.00;
            AREA = 3.14159 * 2.0 * RADIUS;
```

MOD is not defined for floating or fixed types. Exponentiation is not defined for fixed. The result of fixed point division must be explicitly converted to the desired scale and fraction via the type conversion operator (* desired-type *).

3.3 Bit Formulas

Bit formulas apply only to bit types and consist of bit literals (strings of bits), bit items, bit functions, the logical operators (i.e. AND, OR, NOT, XOR, and EQV), bit conversions (e.g. (*B3*)(TALLY)) and REP conversions (see 6.3). Any data type except a block may be explicitly convereted to a bit type.

The relational operators (EQ NQ LS LQ GR GQ) apply to all scalar types: numeric, Boolean, status, and character and return Boolean values, ie. bit values one bit long. An example of a bit formula is:

```
ITEM FF B 5 = 1B'00111';      ITEM GG B 5 = 1B'11110';
              FF AND GG = 1B'00110'
```

3.4 Character Formulas

Character values are represented by character types. Character formulas are limited to character literals, character items, and character functions (e.g. BYTE built-in function). Character conversions, e.g. (* C 10 *)(TALLY), are the only character operations; there are no character operators in J73.

3.5 Status, Pointer, and Table Formulas

Status, pointer, and table formulas involve literals, variables and functions which are status, pointer, and table types. There are no status, pointer, or table operators in J73.

4. STATEMENTS

J73 has no exchange statement. There is a null statement which consists of a single semicolon.

4.1 Statement Structure and Format

Statements may begin in any column and may extend over as many

lines as necessary with more than one statement per line. Simple Statements are separated by the ; mark. Compound statements are enclosed in BEGIN END brackets and can be placed anywhere a simple statement can.

4.2 Statement Labels

Statement labels consist of an identifier followed by a colon and are declared by their use. There are no label variables. There may be multiple labels per statement:

```
LAST: CEASE: DESIST:  STOP;
```

Statement labels within the controlled statement of a loop (see 4.5 below) may not be accessed from outside the loop. The controlled statement may only be executed by executing the loop itself. A goto statement in a for loop may transfer control only to a labelled statement within the loop.

4.3 Assignment Statement

An assignment statement assigns the value of a formula to a variable. All scalar variables --numeric, Boolean, status or character types-- as well as tables and blocks may be assigned to provided the type of the formula, which may be a single variable, matches the type of the target variable. Thus, two tables or table entries can be assigned without requiring the special ENT function as in J3. The type of the formula is determined from the types of its operands. Numeric types are integer, fixed-point, floating-point, dual fixed-point and are converted upon assignment. For other types both sides must agree in type or be made to match via a type conversion. Multiple assignments are also allowed. For example:

```
HEIGHT, LENGTH = SIZE;
```

4.4 NULL Statement

A null statement fulfills the requirement for a statement but performs no action. A null statement can be either a solitary semicolon or an empty BEGIN END pair. An example of the use of the null statement is to resolve the "dangling ELSE" in an IF statement.

4.5 IF Statement

The basic IF statement has the following form:

```
IF AA < BB ;          "IF AA IS LESS THAN BB,"
   AA = 25 ;          "THEN SET AA TO 25     "
```

If the Boolean formula is true, the following statement is

executed; otherwise it is skipped. The Boolean formula must be a bit formula of size 1 (see 5.2.2); bit formulas of size greater than 1 are not implicitly converted to Boolean. A second form of the IF statement has an ELSE clause. If the value of the Boolean formula is TRUE, the true-alternative statement is executed; otherwise, the false-alternative is executed. A compound statement is used to include more than one statement in an alternative:

```
IF COUNT = THRESHOLD;          "IF COUNT = THRESHOLD,"
   BEGIN
     COUNT = 0;                   "SET COUNT TO ZERO"
     ERROR (11);                  "CALL ERROR PROCEDURE"
   END
ELSE                           "IF COUNT <>THRESHOLD,"
   COMPUTE (: RESULT);             "CALL PROCEDURE COMPUTE"
```

IF statements may be nested to implement complex logic:

```
IF COND = V(RED);              "IF COND = RED THEN,"
   BEGIN IF SPEED = V(MAX);       "IF SPEED = MAX THEN,"
            ACTION = 1;              "SET ACTION TO 1"
   END
ELSE                           "IF COND <> RED THEN,"
   ACTION = 2;                     "SET ACTION TO 2"
```

In a nested IF-statement, an else clause that could be interpreted as belonging to more than one IF statement is associated with the closer IF statement. If the above statement were written as follows, the resulting logic would be different:

```
IF COND = V(RED);              "IF COND = RED THEN,"
   IF SPEED = V(MAX);             "IF SPEED = MAX THEN,"
      ACTION = 1;                    "SET ACTION TO 1"
   ELSE                           "IF SPEED <> MAX THEN,"
      ACTION = 2;                    "SET ACTION TO 2"
                               "IF COND <> RED THEN,"
                                  "NO ACTION"
```

To preserve the intended logic a null statement could have been used in place of the BEGIN..END pair shown above:

```
IF COND = V(RED);              "IF COND = RED THEN,"
   IF SPEED = V(MAX);             "IF SPEED = MAX THEN,"
      ACTION = 1;                    "SET ACTION TO 1"
   ELSE;                          "IF SPEED <> MAX,"
                                     "NO ACTION"
ELSE                           "IF COND <> RED THEN,"
   ACTION = 2;                    "SET ACTION TO 2"
```

4.6 CASE Statement

The CASE statement provides alternative execution of one or more statements based on the value of the case selector. The case selector may be an integer, bit, character, or status formula. The case indices designate the statement to be executed for a particular value of the case selector. If there is no case index corresponding to the value of the case selector, the DEFAULT option is executed; if no DEFAULT option is given, the effect of the case statement is undefined. There can be only one DEFAULT option, which may appear anywhere in the case body. After executing the statement, control passes to the statement following the case statement, unless a FALLTHRU option is given. For example:

```
CASE CODE;  BEGIN
            (0:100)           : ACTION1; FALLTHRU
            (100:500, 600)    : ACTION2;
            (700)             : ACTION3;
            (DEFAULT)         : ACTION4;
            END
```

If the value of CODE is between 0 and 100, ACTION1 is called, and since the FALLTHRU option is given, ACTION 2 is also called. If the value of CODE is between 100 and 500 or equal to 600, ACTION 2 is called. IF CODE is equal to 700, ACTION3 is called. For any other value of CODE, ACTION4 is called.

4.7 Loop Statements

J73 has two loop statements, the WHILE loop and the FOR loop. The WHILE loop repeatedly executes a controlled statement while a specified Boolean formula is TRUE. The FOR loop includes the WHILE clause and adds additional information for controlling the values of a loop control item. Both forms of the loop statement check for loop termination at the beginning of each iteration. The loop control item may be a declared item or a single alphabetic letter, which is implicitly declared by its use in the loop statement. Transfers out of loops via GOTO's are allowed at any point; however, single letter loop controls are undefined outside of the loop. Basic forms of the loop statement are as follows:

```
FOR L:I;                      Where: L = item subscript
FOR L:I BY D;                        I = initial value
FOR L:I BY D WHILE L<=F;             D = decrement or increment
FOR L:I THEN R;                      F = final value
FOR L:I THEN R WHILE B;              R = repetion value
WHILE B;                             B = any Boolean formula
```

The I, F, and R may be any numeric or status value. D must have a type and value that can be added to the loop control

item. For single letter loop controls D may not change during execution. The WHILE clause may have any Boolean formula as its argument; the formula L<=F was used to illustrate the loop described for J3. The WHILE clause may also be used with the THEN clause. The formula in the THEN clause is evaluated and reassigned to the loop control after each execution of the loop. This allows stepping through a linked list. The first form of the FOR statement merely activates the subscript, L, and assigns it the value specified by evaluating I and converting to integer by truncation. The second form causes indefinite looping with L incremented by D after each iteration. The third form causes L to be incremented by D after each iteration with looping terminated when L exceeds F. The fourth form causes implicit iteration thru all data group elements.

The first statement after the FOR statement is the iterated statement. Multiple statements must be grouped as a compound statement. The evaluation of the inner product of two vectors, AA and BB, illustrates use of the FOR statement:

```
TABLE AA (50) F;
TABLE BB (50) F;
ITEM  DOT F = 0.0;
BEGIN                              "TO COMPUTE IN REVERSE USE"
FOR I:0 BY 1 WHILE I < 50;         "FOR I:49 BY -1 WHILE I>=0;"
   BEGIN
   DOT = DOT + AA(I)*BB(I);
   AA(I) = 0;  BB(I) = 0;
   END
END
```

4.8 EXIT Statement

The EXIT statement, EXIT, causes control to exit the immediately enclosing loop. Unlike the J3 TEST statement, the loop is not repeated; control passes to the first statement following the innermost loop statement.

4.9 GOTO Statement

There is only one form of GOTO statement:

```
GOTO  statement'label;
```

The GOTO causes an unconditional jump to the statement with the label given in the statement. The statement label must not be in a loop statement or in a subroutine or in another module.

4.10 RETURN Statement

A RETURN statement, RETURN;, effects a normal return from a subroutine (procedure or function) from anywhere in the subroutine body. An implicit RETURN occurs at the END statement.

4.11 ABORT Statement

An ABORT statement, ABORT;, effects an abnormal return from a procedure. When the ABORT statement is executed, control passes to the statement name given in the abort phrase of the procedure call statement that activated the procedure call. If no abort phase is given, the effect of an ABORT statement is the same as that of a STOP statement.

4.12 STOP Statement

A STOP statement causes program execution to stop until restarted by the operator. If STOP stop-code; is written, the stop-code, an integer formula, is supplied to the environment in which the JOVIAL program is running. The meaning of the stop-code is implementation dependent. If a STOP statement is executed within a subroutine, the output parameters are not set.

5. SUBROUTINES

A subroutine may be either a procedure or a function. A subroutine may be re-entrant or recursive by using the RENT or REC attributes in the subroutine declaration. A recursive subroutine may call itself, either directly or indirectly. A re-entrant subroutine is one that may be executing in several concurrent processes. Subroutines may contain other subroutine declarations and may call other subroutines. The INLINE declaration directs the compiler to insert the object code for the body of the subroutines named at the points of their invocation instead of generating object code to call them.

5.1 Procedures

Procedure declarations are similar to other languages. The subroutine body must be enclosed in BEGIN END brackets if it has parameters or multiple statements. The procedure REMQUO illustrates a procedure declaration:

```
PROC REMQUO (NUM, DEN : QUO, REM);
BEGIN                        "QUO & REM ARE OUTPUT PARAMETERS"
  ITEM NUM S 31;  ITEM DEN S 31;
  ITEM QUO S 31;  ITEM REM S 36;
  QUO = NUM/DEN;  REM = NUM-(QUO*DEN);
END
```

5.2 Functions

A function differs from a procedure by the presence of a type description for the result. A function returns a value of that type. The type of the function is defined by the type of the result. Unlike a procedure, a function is used in an formula and may not contain an ABORT phrase. Unlike other languages, a J73 function may return values via formal output parameters as well as via the function call. An example of a function is as follows:

```
PROC NEAR (AA,BB) B;        "DECLARES NEAR AS A BOOLEAN FCN"
BEGIN                       "NO COLON INDICATES AA & BB ARE"
  ITEM AA F; ITEM BB F;     "INPUT PARAMETERS."
  NEAR = (AA = BB);
END
```

5.3 Parameters

Subroutines may or may not have parameters. The parameters given in a subroutine declaration are called formal parameters; the parameters passed in a subroutine call are actual parameters. Input parameters are given first. Output parameters follow input parameters and must be preceeded by the : separator. The number of input and output actual parameters must agree with the number of formal parameters, respectively. Actual input parameters may be formulas, data objects (i.e. items, tables, and blocks), statement labels, or subroutine names. Formal input parameters may be data objects, labels, or subroutines. Actual and formal output parameters may only be data objects.

Formal parameters must be declared in the subroutine. Formal parameters are declared like other data objects, except that a formal table parameters may be declared with asterisk dimensions, which indicate its bounds are determined from the actual parameter on each subroutine call, to allow a single subroutine to handle tables with different sizes. Formal parameters may not be constants or types and may not contain allocation specifiers or presets.

If a formal parameter is a statement label, it is declared by a LABEL declaration. A GOTO to a label formal parameter results in an abnormal subroutine exit; control is passed to the actual parameter label. Subroutine formal parameters must be re-declared giving the information necessary to call the subroutine including parameter declaration; the subroutine body is not repeated.

Actual parameters are bound to formal parameter in four ways: value, value-result, result, and reference. The type of binding is determined either explicitly from a binding

specifier or by default from its its type and input/output status. The binding specifiers, BYREF, BYVAL, or BYRES, cause explicit binding by reference, by value or value-result, or by result respectively. If no binding specifier is given, items and tight table entries are bound by value on input and value-result for output parameters. Default binding of blocks, tables, and other table entries is by reference.

Value binding applies only to input parameters. In value binding, the compiler assures the value of the actual parameter is not changed. This may be done by allocating memory for the formal parameter and copying the value actual parameter into that location.

In value-result binding, a new data object of the type of the formal parameter is created and the actual parameter assigned to that object, as with value binding. But on normal subroutine termination, the value of the formal parameter is copied out to the actual parameter. Value-result binding applies to parameters that are both input and output.

In result binding, the value of the formal parameter is undefined on entry to the subroutine but is otherwise like value-result binding. Result binding apply only to output parameters.

In reference binding, no new objects are created; the actual and formal parameters denote the same physical object. Any change in the formal parameter results in an immediate change to the value of the actual parameter.

5.5 Subroutine Call Statements

Procedures are invoked by giving the procedure name and list of actual parameters followed by an optional abort phrase that provides a label to which control is sent if an ABORT statement is encountered during the execution of the procedure. Functions are called by use of the function name in an formula:

```
REMQUO(13, 3 : QQ, RR) ABORT ERROR2;
IF NEAR (3.14, 3.14E10) ;
```

Parameterless subroutines may optionally contain an empty parameter list:

```
XX = RANDOM;        or optionally      XX = RANDOM ();
```

6. BUILT-IN FUNCTIONS

The JOVIAL built-in functions provide a way of getting information that would otherwise be inaccessible to the

programmer or time-consuming for the programmer to calculate. Three of the built-in functions, BIT, BYTE and REP, may be used as either functions or as pseudo-variables on the lefthand side of an assignment-statement.

6.1 ABS

ABS returns the absolute value of the item passed to it:

```
ABS ( -25 ) returns = 25.
```

6.2 SGN

SGN returns +1, 0, or -1 if the value of the input parameter is greater than, equal to, or less than zero: SGN(-25) returns -1. The sign function may not be used to set the sign of an item.

6.3 BIT, BYTE, and REP

A significant feature of JOVIAL is the provision for access to bits and bytes of items provided by the BIT, BYTE, and REP functions. (The term BYTE was was originally coined by Werner Buchholz in describing the IBM 7030, STRETCH, computer and was picked up from the decripiton of the IBM FSQ-31 computer). The BIT function in J73 extracts and assigns substrings from bit type items and formulas only, unlike J3 which allow BIT to act on items of any type. Bits are indexed left to right, beginning with 0. For example:

```
ITEM MASK B 4 = 1B'0001';      BIT (MASK,0,3) = 6 ;
```

changes the value of MASK to 13 = 1101 by causing the string of 3 bits starting at bit 0 of item MASK (which is a four bit item with initial value of 1 = 0001) to be assigned the number 6, which is the bit string "110".

Similiarly, the BYTE function operates on character type expresssion and variables. Characters are numbered from left to right, beginning with zero. For example:

```
ITEM SYM C 6 = 'BULLET';       ITEM BOL C 5 = 'RIGHT';
BYTE(SYM,1,4) = BYTE(BOL,0,4;
```

sets the four characters starting with the byte 1 OF SYM to the four bytes of item BOL starting with byte 0. The value of SYM becomes 'BRIGHT'.

The REP function converts any data object to a bit string whose size is the actual number of bits occupied by the object. The exact machine representation of the object is obtained. For example, given a declaration of an item COUNT:

```
ITEM COUNT U 3 = 7;
```

The value of REP(COUNT) is 1B'0000000000000111', if the default number of bits per word is 16. REP many also be assigned to as a pseudo-variable.

6.4 LOC

LOC provides the machine address of the word in which its argument is contained. The argument of LOC can be a data object, a label, a subroutine or a block. LOC returns a pointer value that is used in a pointer-qualified reference (see section 2.9).

6.5 NWDSEN

The Number of WorDS per ENtry (NWDSEN) returns the number of words of storage allocated to each table entry in the table or table type given as an argument. NWDSEN is useful in programs that perform dynamic storage allocation. For the table PAYROLL in section 2.3 NWDSEN (PAYROLL) returns the value 7: 5 words for EMP'NAME, 1 word for MAN'NMBR, and 1 word for ORG'CODE.

6.6 BITSIZE, BYTESIZE, and WORDSIZE

BITSIZE returns the size of its argument in bits, BYTESIZE returns the size in bytes, and WORDSIZE in words. Arguments may be items, tables, blocks, or type names of these types. The three functions are related to one another by the implementation parameters BITSINBYTE, BITSINWORD and BYTESINWORD. The BITSIZE of an item is related to the size given in its declaration. The BITSIZE of a table is the total number of bits from the first bit of the first entry to the last bit of the last entry. For a table with no structure or packing specifier, BITSIZE is the product of the number of words per entry, the number of entries, and BITSINWORD. The BITSIZE of a block is the number of words in the block times BITSINWORD.

6.7 LBOUND and UBOUND

LBOUND and UBOUND return, respectively, the lower and upper bounds of a specified dimension of a given table or table type. The type of the value returned is either integer or status, depending on the declaration of the table. For tables PAYROLL and VOYAGE (section 2.3), the following are true:

```
LBOUND(PAYROLL, 0) = 1          UBOUND(PAYROLL, 0) = 5
LBOUND(PAYROLL, 1) = 0          UBOUND(PAYROLL, 1) = 1000
LBOUND(VOYAGE, 0) = V(USA)      UBOUND(VOYAGE, 0) = V(FRA)
```

6.8 NEXT

If the argument to NEXT is a pointer formula, NEXT returns the arithmetic sum of a pointer argument and an increment. NEXT can be used to return the next entry of a table declared using a type. For example, consider the table NUTS(100) and pointer PTR declared in section 2.9. If PTR = LOC(NUTS(47)), then NEXT(PTR, 3) adds the correct increment to the address of NUTS(47) to return a pointer to the address of entry NUTS(50). NEXT can thus be used to step through performing some action on each entry.

If the argument to NEXT is a status formula, NEXT returns the successor or predessor of the value of the formula. For example, if the value of the status item OP'TYPE (section 2.1) is initially V(REL), then NEXT (OP'TYPE, 2) returns the second successor to V(REL), which is the value V(ASN). NEXT(OP'TYPE, -1) returns the first predecessor to V(REL), which is the value V(ARI).

6.9 FIRST and LAST

FIRTS and LAST return, respectively, the first and last values associated with their argument, which may be either a status formula or status type name. For item OP'TYPE, FIRST(OP'TYPE) returns V(ARI) and LAST(OP'TYPE) returns V(ASN).

6.10 SHIFTL and SHIFTR

The shift functions perform logical shifting of bit formula. Shifting occurs in the direction indicated by the function name the specified (non-negative) number of bits with vacated bit zero filled and shifted-out bits lost. For example, if the value of MASK declared above is 1B'1010', SHIFTL(MASK, 2) return the value 1B'1000' and SHIFTR(MASK,1) the value 1B'0101'.

6.11 Implementation Parameters

Although not really built-in functions, implementation parameters are an important feature of J73 which supply information about the underlying machine representation of a program. Implementation parameters vary with each computer: the nuber of bits in a byte, in a word, and in an address varies from implementation to implementation. These values affect the representation and behavior of data in the actual program. J73 implementaton parameters allow these values to be accessed symbolically by the programmer. Also implementation of data declarations are defined in terms of these parameters. Examples of implementation parameters are as follows:

BITSINBYTE	Number of bits in a byte.
BITSINWORD	Number of bits in a word.

BYTESINWORD	Number of complete bytes in a word.
FLOATPRECISION	Default number of bits to hold the mantissa of a floating point item.
MAXINTSIZE	Maximum size for integers excluding sign
MAXINT(size)	Maximum integer value representable in size+1 bits including sign bit.
MAXBYTES	Maximum value for character strings.
MAXBITS	Maximum value for bit-size; maximum value of words per entry in a table is MAXBIT/BITSINWORD; maximum BITSIZE of a table entry is MAXBITS.

7. PROGRAM STURCTURE

A JOVIAL program consists of one or more modules, which are compiled separately and then linked together for execution. JOVIAL has three kinds of modules: main program, procedure and compool. A program must have exactly one main program module. Execution of the program begins with the first statement of this module and ends with the last statement or a stop statement. A program may have zero or more procedure modules which contain data and subroutines. Unlike some previous versions of JOVIAL, J73 allows subroutines to be nested. A program may have zero or more compool modules, which contain declarations that are shared among other modules.

7.1 Main Program Modules

All three modules are enclosed between START TERM delimiters. The format of a main program module is as follows:

```
START optional:origin $
PROGRAM name;
  BEGIN
    declarations ...              "ZERO OR MORE DECLARATIONS"
    statements ...                "AT LEAST ONE STATEMENT"
    nested subroutines ...        "ZERO OR MORE DECLARATIONS"
  END
    non-nested subroutines ..."ZERO OR MORE DECLARATIONS"
TERM
```

Only non-nested subroutines can be designated as external by the DEF reserved word (section 7.2).

7.2 External Declarations

Communication between modules is accomplished by external names. An external declaration declares an external name in one module that may be referenced in other modules. J73 has two kinds of external declarations: DEF and REF.

A DEF specification delares an external name and allocates storage for that name; it is used to export a name outside the module in which the DEF appears. Data or subroutines declared in a DEF specification in a module are physically allocated in that module. Items, tables, blocks, or labels may be given in a DEF declaration in any kind of module. Subroutines may not be DEFed in a compool module. DEFed labels may not be used as the target of a goto statement in another module. A DEF specification may be referenced by a REF specification in any number of modules. Since all modules share the same scope, it is illegal to have two DEF specifications with the same name in the same program.

A REF specification provides inormation about an external name that is declared in another module by a DEF; it is used to import information about external data and subroutines declared by a DEF specification in a procedure or main program module. A REF specification must agree in name, type, and all other attributes with the name declared in the DEF specification. Presets must not be given in a REF specification for an item or table. A constant item or table may not appear in either a DEF or REF specification. For example, A REF specification is used in a compool to make a subroutine declared in a procedure module available through the compool.

7.3 Compool Modules

Compools provide for the communication of names between separately compiled modules. A compool may be used like a data dictionary to allow the programmer to use an item without having to specify where to place the value or its internal form. The compool feature in J73 is more integrated into the language than in previous versions of JOVIAL. All J73 compilers must implement the compool feature and do so in a consistent manner, unlike J3 where compool processors were not part of the compiler and were sometimes not implemented at all.

Declarations are placed in a compool so that they may be used by more than one module. A compool module may only contain declarations. The following kinds of declarations are allowed in a compool: constants, types, defines, overlays, DEFs for data or labels, and REFs for data or subroutine declarations. The names in a compool are made available to another module by using a !COMPOOL directive immediately following the START in the module using the compool. The !COMPOOL directive may optionally contain a list of only those names to be imported into the module using the compool; otherwise all names in the compool are imported. The names imported from a compool may be used without further declaration. Although communication between modules may be accomplished directly with DEF and REF specifications, no checking is performed. At compile time the compiler assumes that the type and attributes given in the REF

specification agree with the DEF specification. At link time, the references are bound together, but no check of type or attributes is made because that information is no longer available. By using the compool for communication between modules, the compiler can detect any incompatibility between the DEF and REF specifications.

7.4 Procedure Modules

A procedure module provides a way in which subroutines can be separately compiled, since a compool can only contain declarations, not executable statements. Any type of declaration may be given in a procedure module.

An example of using a compool to provide subroutine declarations contained in a procedure module to a main procedure module is now given:

```
START COMPOOL TYPES;
  TYPE KEY STATUS (V(RED), V(BLUE), V(GRN));
  TYPE TABTYPE TABLE (100);
      BEGIN
        ITEM CODE KEY;
        ITEM VALUE U;
      END
TERM

START !COMPOOL('TYPES');
  DEF PROC GETVAL (ARG);
    BEGIN
      TABLE ARG TABTYPE;
      ...
    END
  DEF PROC EXCEPTION (CODE);
    BEGIN
      ITEM CODE U;
      ...
      END
TERM


START !COMPOOL ('TYPES') COMPOOL DATABASE;
  DEF TABLE DATA TABTYPE;
  DEF ITEM CURVAL U;
  REF PROC GETVAL (ARG);
    TABLE ARG TABTYPE;
  REF PROC EXCEPTION (CODE);
    ITEM CODE U;
TERM
```

```
START
!COMPOOL ('TYPES'); !COMPOOL ('DATABASE');
PROGRAM SEARCH;
  BEGIN
    GETVAL (DATA);
    CURVAL = RETRIEVE (V(RED));
    PROC RETRIEVE (ARG1) U;
      BEGIN
      ITEM ARG1 KEY;
      FOR I: 0 BY 1 WHILE I <= 100;
         IF CODE (I) = ARG1;
            RETRIEVE = VALUE (I);
      END
  END
TERM
```

7.5 Scope of Names

Each name in a program must be declared in a data, type, subroutine, label, define, external, overlay, or inline declaration. A reserved word cannot be used in a declaration. An external name must be declared by exactly one DEF specification. Each declaration in a program supplies information about a particular name. A given declaration of a given name does not necessarily apply to all occurrences of that name. The occurrences of a name to which a declaration does apply is the scope of that declaration. The scope of a declaration is the smallest block of code that contains a declaration for that name. Thus, the meaning of declarations of duplicate names depends on the scope in which the name is used.

System defined names such as implementation parameters and machine specific subroutines belong to system scope. All names imported from referenced compools, the names of the compools themselves, and the name of the module being compiled belong to the compool scope. The system scope and a compool scope enclose, i.e. are global to, the module being compiled. These names are global to the module. Other scopes are the module body scope and the scope of nested and non-nested subroutines. The names in a module are global to all the subroutines contained in it. The names declaraed in a subroutine are local to it but are global to global to the nested subroutines it contains.

When the name is used, the compiler determines which declaration of the data object is meant by looking for a declaration of the referenced name in a scope containing the current reference to the name. If no declarations of the name lie in the current scope, scope which are global to the current are search successively for a declaration of the name. If none

is found, the program is invalid. If exactly one declaration of the name is found, that declaration applies. If several declarations of the name are found in the scopes which contain the reference to the name, the declaration with the smallest scope applies, i.e. local names hide declarations of the same name global to the. Duplicate names within the same scope are not allowed as the compiler could not resolve to which declaration the name refers.

8. COMPILER AND MACHINE INTERFACE

8.1 DEFINE Declaration

The define capability provides a compiler macro capability. A define declaration associates a name with a string of JOVIAL code. The compiler then substitutes the associated code everywhere the define identifier appears in the source text. Only the identifiers in the program are substituted; the compiler does not replace characters within comments and character literals. Defines can be used to parameterize programs and to make code more readable but cannot be used to create new symbols. For example, if the following defines are used:

```
DEFINE PLUS       "+" ;
DEFINE MULTIPLIED "*" ;
DEFINE REPLACED   "=" ;
DEFINE IS         " " ;
DEFINE BY         " " ;
```

A programmer can write the following, otherwise invalid JOVIAL statement in his program:

```
AA IS REPLACED BY BB PLUS CC MULTIPLIED BY DD;
```

and the compiler will substitute the defined strings for the define names to arrive at the following valid JOVIAL statement which it then compiles:

```
AA = BB + CC * DD ;
```

A define declaration may contain formal parameters, which are indicated by single letters within a parenthesized parameter list and by the letter preceded by an exclamation point within the define string. The formal parameter is assigned the value of the define actual parameter given in the DEFINE call. For example, the following notation for incrementation may be used:

```
DEFINE INC (N) "!N = !N + 1" ;
...
INC (COUNT);   is replaced by COUNT = COUNT + 1;
```

A define-string may itself include define calls. For example, given the following define declarations:

```
DEFINE C1 (A,B) "!A / !B ** EXP";        DEFINE EXP "2" ;
```

The following define call (left) produces the expansion shown (right) by first expanding C1 and then EXP:

```
XX = C1 (YY, 5);              XX = YY / 5 ** 2;
```

A define actual parameter may itself be a define call. Given the above define declarations the following define call produces the following expansion:

```
XX = C1 (EXP, 5);             XX =  2 / 5 ** 2;
```

8.2 Directives

Directives provide supplemental information to the compiler about the program. Directives are preceeded by an exclamation point. J73 has 22 directives, which affect output format, optimization, data and subroutine linkage, debugging information, and compools; implementations may also support additional directives. The standard directives are grouped as follows:

!COMPOOL The !COMPOOL directive identifies the compool and the set of names from that compool that are to be imported. Three forms are used:

```
!COMPOOL ('RAWDATA');
!COMPOOL 'RAWDATA' GRID, LENGTH;
!COMPOOL 'RAWDATA' (GRID), LENGTH;
```

The first form imports all the names in compool RAWDATA. The second form imports only the names GRID and LENGTH. The third form imports all the components of the table GRID.

!COPY The !COPY directive names a file of source text that is to be copied into the program at the point where the directive is given.

!SKIP
!BEGIN
!END The conditional compilation directives, !BEGIN and !END, followed by a single letter delimit a block of code that is to be skipped depending on whether a !SKIP directive followed by the same letter is encountered. For example:

```
START PROGRAM MAIN;
  BEGIN
   ...              "declarations & statements"
    !SKIP A;
    !BEGIN A;
      ...           "code for block A"
    !END;
    !BEGIN B;
      ...           "code for block B"
    !END;
   ...              "rest of program"
  END
TERM
```

The above directives cause the compiler to omit the block of code labelled A. If the letter B had been used with the !SKIP directive, the compiler would omit the block of code labelled B.

!LIST
!NOLIST
!EJECT

The source listing directives direct printing of the source code. !NOLIST suppress listing of the source code until a !LIST directive is encountered. !EJECT tells the compiler to skip to a new page, i.e. insert a page break in the listing.

!LISTINV
!LISTEXP
!LISTBOTH

The define listing directives allow programmer control over the text to be included in the source program listing for define calls. !LISTINV is the default and causes the define call to be listed as in the original source. !LISTEXP causes the expended define string to be put in the listing in place of the define call. !LISTBOTH causes both the define call and the resulting expansion to be listed.

!INITIALIZE

The !INITIALIZE directive causes all static data that is not initialized by a preset to be set to zero bits. The effect of the directive extends to the end of the current scope.

!ORDER

!ORDER instructs the compiler to allocate storage for the data objects in a block or ordinary table in the order of their declaration in the text.

!LEFTRIGHT
!REARRANGE

The evaluation order directives indicate how formulas are evaluated. !LEFTRIGHT causes evaluation of operators of equal precedence from left to right within an formula. !REARRANGE is the default and allows evaluation of operators in any order to produce more efficient code, as long as

the commutative, associative, and distributive laws are observed.

!INTERFERENCE This directive prevents the compiler from making optimizations based on the assumption that the data names given in the directive do not overlap. Although the knows about overlapping data objects given in specified tables and overlay declarations, other cases occurr such as when two objects are assigned the same absolute address in different overlay declarations.

!REDUCIBLE The !REDUCIBLE directive is given in a function heading and tells the compiler that all calls of the function with the same actual parameter values result in the same return value and output parameters and that the function only modifies automatic data declared within the function. This directive allows the compiler to optimize identical function calls by saving the values produced by the first call.

!BASE
!ISBASE
!DROP Register directives affect target computer register allocation. !BASE causes the specified register to be loaded with the address of the given data name. !ISBASE tells the compiler to assume the specified register contains the address of the data object and !DROP fress the specified register for other uses.

!LINKAGE The !LINKAGE directive is given in a subroutine heading and tells the compiler that the subroutine does not obey standard JOVIAL linkage conventions. This directive is used with externally declared subroutines. The !LINKAGE FORTRAN directive is implemented in many J73 compilers.

!TRACE The !TRACE directive causes the compiler to output a trace of the specified names. A message is output each time a specified subroutine is called or when a data name is modified. The !TRACE directive applies from the point at which it is given to the end of the scope.