AD-A033 893 SOFTECH INC WALTHAM MASS F/G 9/2
EVALUATION OF ALGOL 68, JOVIAL J3B, PASCAL, SIMULA 67, AND TACP--ETC(U)
DEC 76 J B GOODENOUGH, C L MCGOWAN, J R KELLY DAAB07-75-C-0373
UNCLASSIFIED ECOM-75-0373-F NL

1 OF 2
AD A033893

ADA033893

Research and Development Technical Report

ECOM -75-0373-F

EVALUATION OF ALGOL 68, JOVIAL J3B, PASCAL, SIMULA 67, and TACPOL vs TINMAN REQUIREMENTS FOR A COMMON HIGH ORDER PROGRAMMING LANGUAGE

John B. Goodenough
Clement L. McGowan
John R. Kelly
SOFTECH INC.
460 Totten Pond Road
Waltham, MA 02154

December 1976

Final Report for Period July 1976 - October 1976

DDC
RECEIVED
JAN 3 1977
B

Prepared for:

ECOM

US ARMY ELECTRONICS COMMAND FORT MONMOUTH, NEW JERSEY 07703

DISTRIBUTION STATEMENT A
Approved for public release;
Distribution Unlimited

HISA FM 2957-73

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position, unless so designated by other authorized documents.

The citation of trade names and names of manufacturers in this report is not to be construed as official Government indorsement or approval of commercial products or services referenced herein.

Disposition

Destroy this report when it is no longer needed. Do not return it to the originator.

ESC-PM 5700-45

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: ECOM-75-0373-F	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle): Evaluation of ALGOL 68, JOVIAL J3B, PASCAL, SIMULA 67, and TACPOL vs. TINMAN requirements for a common High Order Programming Language	5. TYPE OF REPORT & PERIOD COVERED: Final Report for Period July 76 - Oct 76
	6. PERFORMING ORG. REPORT NUMBER: 1021-14
7. AUTHOR(s): John B. Goodenough, Clement L. McGowan, John R. Kelly	8. CONTRACT OR GRANT NUMBER(s): DAAB07-75-C-0373
9. PERFORMING ORGANIZATION NAME AND ADDRESS: SofTech Inc. 460 Totten Pond Rd. Waltham, MA 02154	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 672725.12.724.02.02
11. CONTROLLING OFFICE NAME AND ADDRESS: Computer Software Tech Area (DRSEL-NL-BG-2) Center for Tactical Computer Science US Army ECOM Ft. Monmouth 07703	12. REPORT DATE: Dec 76
	13. NUMBER OF PAGES: 171 p.
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report): unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release, distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Computer Programming, high level languages, language evaluation, programming languages, SIMULA 67, TACPOL, JOVIAL J3B, ALGOL 68, PASCAL.

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report evaluates how well ALGOL 68, JOVIAL J3B, PASCAL, SIMULA 67, and TACPOL satisfy the Department of Defense requirements for high order computer programming languages.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

405 932

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

PREFACE

The evaluations presented in this report were performed under the direction of John B. Goodenough, who also was responsible for the evaluation of JOVIAL J3B. Clement L. McGowan was primarily responsible for the evaluation of PASCAL, SIMULA 67, and TACPOL. John R. Kelly evaluated ALGOL 68.

ACCESSION for
NTIS White Section ☑
DDC Buff Section ☐
UNANNOUNCED ☐
JUSTIFICATION
BY
DISTRIBUTION/AVAILABILITY CODES
Dist. AVAIL. and/or SPECIAL
A

TABLE OF CONTENTS

INTRODUCTION AND SUMMARY OF RESULTS		1
	Recommendations	3
DETAILED LANGUAGE EVALUATIONS		6
	ALGOL 68 Evaluation Summary	6
	J3B Evaluation Summary	8
	PASCAL Evaluation Summary	11
	SIMULA 67 Evaluation Summary	14
	TACPOL Evaluation Summary	17
A1.	Typed Language	20
A2.	Data Types	21
A3.	Precision	23
A4.	Fixed Point Numbers	24
A5.	Character Data	26
A6.	Arrays	27
A7.	Records	29
B1.	Assignment and Reference	30
B2.	Equivalence	32
B3.	Relationals	33
B4.	Arithmetic Operations	35
B5.	Truncation and Rounding	36
B6.	Boolean Operations	37
B7.	Scalar Operations on Composite Data Structures	39
B8.	Type Conversion	40
B9.	Changes in Numeric Representations	43
B10.	I/O Operations	44
B11.	Power Set Operations	47
C1.	Side Effects (Due to Evaluation Order)	48
C2.	Operand Structure	49
C3.	Expressions Permitted	51
C4.	Constant Expressions	52
C5.	Consistent Parameter Rules	54
C6.	Type Agreement in Parameters	56
C7.	Formal Parameter Kinds	59
C8.	Formal Parameter Specifications	60
C9.	Variable Numbers of Parameters	62
D1.	Constant Value Identifiers	63
D2.	Literals	64
D3.	Initial Values of Variables	65
D4.	Numerical Range and Step Size	67
D5.	Variable Types	68
D6.	Pointer Variables	70
E1.	User Definitions Possible	72
E2.	Consistent Use of Types	73
E3.	No Default Declarations	74
E4.	Can Extend Existing Operators	75

E5. Type Definitions 76
E6. Data Defining Mechanisms 78
E7. No Free Union or Subset Types 80
E8. Type Initialization 81

F1. Separate Allocation and Access 82
F2. Limiting Access Scope 84
F3. Compile Time Scope Determination 86
F4. Libraries Available 88
F5. Library Contents 89
F6. Libraries and Compools Indistinguishable 90
F7. Standard Library Definitions for Machine-Dependent Interfaces 92

G1. Kinds of Control Structures 94
G2. The GOTO 95
G3. Conditional Control 96
G4. Iterative Control 98
G5. Routines 99
G6. Parallel Processing 101
G7. Exception Handling 102
G8. Synchronization and Real Time 104

H1. General Syntax Characteristics 105
H2. No Syntax Extensions 107
H3. Source Character Set 108
H4. Identifiers and Literals 109
H5. Lexical Units and Lines 111
H6. Key Words 112
H7. Comment Conventions 113
H8. Unmatched Parentheses 115
H9. Uniform Referent Notation 116
H10. Consistency of Meaning 119

I1. No Defaults in Program Logic 121
I2. Object Representation Specifications Optional 122
I3. Compile Time Variables 124
I4. Conditional Compilation 125
I5. Simple Base Language 127
I6. Translator Restrictions 129
I7. Object Machine Restrictions 131

J1. Efficient Object Code 132
J2. Safe and Effective Optimization Possible 134
J3. Machine Language Insertions 136
J4. Object Representation Specifications 137
J5. Open and Closed Routine Calls 139

K1. Operating System Not Required 140
K2. Program Assembly 142
K3. Software Development Tools 143
K4. Translator Options 144
K5. Assertions and Other Optional Specifications 146

L1.	No Superset Implementations	147
L2.	No Subset Implementations	148
L3.	Low Cost Translation	150
L4.	Many Object Machines	151
L5.	Self-Hosting Possible	152
L6.	Translator Checking Required	153
L7.	Diagnostic Messages	154
L8.	Translator Internal Structure	156
L9.	Self-Implementable Language	157
M1.	Existing Language Features Only	158
M2.	Unambiguous Definition	159
M3.	Language Documentation Required	161
M4.	Control Agent Required	162
M5.	Support Agent Required	163
M6.	Library Standards and Support Required	164

SECTION 1

INTRODUCTION AND SUMMARY OF RESULTS

This report evaluates how well ALGOL 68, JOVIAL J3B, PASCAL, SIMULA 67, and TACPOL satisfy the Department of Defense requirements for high order computer programming languages, as described in "A Common Programming Language for the Department of Defense -- Background and Technical Requirements," Institute for Defense Analyses, Arlington, Va., Paper P-1191, June 1976, known informally as the "TINMAN." The body of this evaluation report consists of stating for each TINMAN requirement (e.g., "A6. Arrays") the degree to which each language satisfies the requirement (e.g., "Not Satisfied"), together with a justification for this evaluation, keyed to language reference documents. Bibliographic references to these documents are found at the beginning of Section 2, where each language's deficiencies are summarized. The remainder of Section 2 consists of the detailed language evaluations. Each evaluation begins by quoting relevant parts of the TINMAN requirement, and then each language is evaluated separately with respect to that requirement.

A summary of how well each language satisfies each requirement is presented below, where S indicates the requirement is satisfied, P indicates it is partially satisfied, N indicates it is not satisfied, and U indicates it was not possible to determine whether the requirement was satisfied or not.

		ALG	J3B	PAS	SIM	TAC
A1.	Typed Language	S	P	P	P	P
A2.	Data Types	P	P	P	P	P
A3.	Precision	N	N	N	N	N
A4.	Fixed Point	N	N	N	N	P
A5.	Character Data	N	N	N	N	N
A6.	Arrays	P	P	P	P	P
A7.	Records	P	P	P	P	N
B1.	Assignment	P	P	P	P	P
B2.	Equivalence	P	P	P	P	N
B3.	Relationals	P	P	P	P	P
B4.	Arithmetic Operations	P	P	P	S	P
B5.	Truncation/Rounding	P	P	U	P	N
B6.	Boolean Operations	P	P	N	P	P
B7.	Scalar Operations	P	P	P	N	N
B8.	Type Conversion	N	P	N	N	P
B9.	Range Exceptions	N	P	P	N	S
B10.	I/O Operations	S	N	P	S	P
B11.	Power Set Operation	N	N	P	N	N
C1.	Side Effects	N	N	N	N	N
C2.	Operand Structure	N	P	P	N	P
C3.	Expressions Permitted	S	S	S	S	U
C4.	Constant Expressions	N	P	N	N	N

C5.	Parameter Rules	P	P	P	P	N
C6.	Parameter Agreement	P	P	P	P	S
C7.	Parameter Kinds	P	P	N	N	P
C8.	Parameter Specifications	N	N	N	P	N
C9.	Numbers of Parameters	S	N	N	N	N
D1.	Constant Value Identifiers	S	S	S	N	S
D2.	Numeric Literals	P	P	P	P	P
D3.	Initialization	P	P	N	N	N
D4.	Ranges and Step Sizes	N	N	N	N	P
D5.	Variable Types	P	P	S	P	P
D6.	Pointer Variables	S	P	N	S	N
E1.	User Definitions	S	N	P	P	N
E2.	Consistent Use	S	N	N	P	N
E3.	No Default Declarations	S	S	S	P	P
E4.	Extend Existing Operators	P	N	N	N	N
E5.	Type Definitions	N	P	P	P	N
E6.	Data Definitions	P	P	S	P	N
E7.	No Free Union	S	N	N	S	N
E8.	Type Initialization	N	N	N	P	N
F1.	Allocation and Access	P	P	P	P	P
F2.	Limit Access Scope	P	P	N	P	P
F3.	Static Scope Determination	S	S	S	S	S
F4.	Libraries Available	N	N	N	P	N
F5.	Library Contents	N	P	N	N	P
F6.	Libraries and Compools	N	N	N	P	P
F7.	Machine-Dependent Interfaces	S	N	S	S	S
G1.	Control Structures	P	P	P	N	P
G2.	The GOTO	N	P	N	N	N
G3.	Conditional Control	P	N	P	N	N
G4.	Iterative Control	P	N	N	N	P
G5.	Routines	P	S	P	P	U
G6.	Parallel Processing	P	N	N	N	N
G7.	Exception Handling	N	N	N	N	P
G8.	Synchroniztion	N	N	N	N	N
H1.	Syntax Characteristics	S	P	P	S	P
H2.	No Syntax Extensions	N	S	S	S	S
H3.	Source Character Set	S	S	P	N	S
H4.	Identifiers and Literals	P	P	P	P	P
H5.	Lexical Units	N	N	N	N	N
H6.	Key Words	S	P	S	S	U
H7.	Comment Conventions	P	P	P	N	P
H8.	Unmatched Parentheses	S	S	S	S	S
H9.	Uniform Referent Notation	P	P	N	N	S
H10.	Consistency of Meaning	P	N	P	N	N
I1.	Defaults in Program Logic	S	N	N	N	N
I2.	Object Representation	P	S	P	N	P

I3.	Compile Time Variables	P	N	N	N	N
I4.	Conditional Compilation	N	S	N	N	N
I5.	Simple Base Language	P	P	S	P	N
I6.	Translator Restrictions	N	P	N	N	P
I7.	Object Machine Restrictions	P	N	P	P	S
J1.	Efficient Object Code	N	P	P	N	P
J2.	Optimizations Possible	P	P	P	P	N
J3.	Machine Language	N	N	N	N	P
J4.	Object Representations	N	P	N	N	P
J5.	Open/Closed Routines	N	S	N	N	N
K1.	Operating System	P	S	S	P	S
K2.	Program Assembly	N	S	N	P	S
K3.	Software Tools	U	U	U	U	U
K4.	Translator Options	U	U	U	U	P
K5.	Assertions	S	N	N	N	N
L1.	Supersets	N	N	S	N	N
L2.	Subsets	N	N	S	S	N
L3.	Low-Cost Translation	P	S	S	P	S
L4.	Many Object Machines	S	S	S	S	N
L5.	Self-Hosting	U	S	S	P	S
L6.	Translator Checking	S	S	U	P	S
L7.	Diagnostic Messages	N	P	N	N	N
L8.	Translator Structure	S	S	S	S	S
L9.	Self-Implementable	S	P	S	P	P
M1.	Existing Features	S	S	S	S	S
M2.	Unambiguous Definition	S	P	P	P	N
M3.	Language Documentation	S	P	S	S	N
M4.	Control Agent	N	N	N	N	N
M5.	Support Agent	N	N	N	N	N
M6.	Library Standards	N	N	N	N	N

Recommendations

One of the purposes of this evaluation effort was to determine whether any language satisfies the TINMAN requirements sufficiently that it can be used unchanged (or with relatively minor modifications) as the DoD Common Language for embedded computer applications. None of the languages we have evaluated, however, requires only minor changes to satisfy these requirements, and so none is suitable for immediate adoption by the DoD.

A second objective of the evaluation was to select languages which could usefully serve as the basis for developing a common language. Among the reasons for choosing a basis language (and modifying it, perhaps extensively) are:

1) the common language can draw on a population of programmers familiar with the use of the basis language, assuming the changes made to it are not too drastic;

2) similarly, compilers and other tools developed for the basis language can be reused or modified if the changes required are not too extensive;

3) perhaps more important, by starting with a basis language whose deficiencies and strengths are understood, modifications can be guided by the desire to eliminate the language's deficiencies while preserving its strengths; this can help to focus language design issues;

4) design effort can be concentrated on the more difficult design issues, since choosing a basis language resolves many trivial issues immediately, and provides an orientation that can guide the resolution of other issues.

Keeping these goals in mind, a suitable basis language should satisfy at least the following criteria:

- it should have a well specified language definition (so it is clear what the basis language is);

- the additions and deletions required to satisfy the TINMAN requirements should be relatively independent, i.e., one should not have to delete a feature to satisfy a requirement and then add a similar feature to satisfy the same requirement; such modifications vitiate the value of building from the selected basis language;

- experience in using the language, especially for applications similar to embedded computer applications, is important in understanding the strengths and weaknesses of a basis language relevant to DoD applications;

- the features of a basis language that satisfy TINMAN requirements should not be a subset of another language's features, i.e., if two languages are both otherwise acceptable basis languages, one should select that language having the greatest number of acceptable features, since the experience gained in using a larger number of features is of greater benefit in guiding the modification effort.

Of the languages evaluated, only PASCAL and ALGOL 68 appear suitable as a basis for creating a language satisfying all the TINMAN requirements. PASCAL is suitable essentially because relatively few features need to be deleted from the language. On the other hand, so many features need to be added that the final language may bear little resemblance to PASCAL. In particular, maintaining PASCAL's simplicity and consistency will be difficult, but attempting to do so will in itself be a useful constraint on the design effort.

ALGOL 68 is suitable as a basis for modification primarily because relatively few features need to be added to it, and the features that need to be added would need to be added to virtually any other language selected as a design base. Most of the ALGOL 68 modifications are deletions and

simplifications. Experience has been gained in England in the use of ALGOL 68 for systems implementation. An advantage of using ALGOL 68 as a design base is its consistency -- its use of a few linguistic concepts uniformly combined. Another advantage, from the point of view of modification, is its syntactic and semantic description method, which helps to enforce the consistency and uniformity that is typical of the language. Modifications to ALGOL 68 can be described in terms of this syntax, and the ramifications of deleting or modifying some feature can be more readily traced out due to the specification method. This may well prove productive in bringing modification problems to a design team's attention. Since it is the interaction among language features that makes language design difficult, having a formal basis for tracking these interactions should prove productive.

JOVIAL J3B is unsuitable as a basis for modification primarily because of its numerous arbitrary restrictions and inconsistencies (see, for example, H1 and C5). These would all have to be removed and resolved before J3B could serve as a suitable basis, and this would not be a productive use of a design team's time.

TACPOL is unsuitable as a basis for modification primarily because its acceptable features are a (rather small) subset of the acceptable features of other languages, e.g., PASCAL.

SIMULA 67 is unsuitable primarily because the class concept, which is what distinguishes SIMULA from other languages, is unsuitable for embedded computer applications, and if this construct is deleted from the language there is no longer any virtue in using SIMULA as a basis language rather than ALGOL 60. The difficulty with the class concept is primarily that it supports record data structures only indirectly. Looking a a SIMULA class definition, it is not easy to see what class "attributes" correspond to fields in an underlying record structure, even though identification of these fields is necessary when the object representation of a record structure is to be specified explicitly by a programmer (TINMAN requirements I2 and J4). The fact that SIMULA classes abstract away from the physical structuring of a record is a virtue in its intended application area, but for embedded computer applications, where the physical structuring of data tends to be very important, this abstraction will tend to make a SIMULA-based language difficult and unwieldy to to use. Even so, it is clear that much can be learned from the SIMULA experience in deciding how to adapt other languages to meet the TINMAN requirements.

SECTION 2

DETAILED LANGUAGE EVALUATIONS

ALGOL 68 Evaluation Summary

SofTech's evaluation of ALGOL 68 with respect to the Department of Defense requirements for high order computer programming languages ("TINMAN," June 1976) is summarized here. Detailed evaluations of ALGOL 68 with respect to each TINMAN requirement are contained in separate sections of this report. These sections begin with an overall assessment of the extent to which the requirement is satisfied, followed by a justification of this assessment referencing sections of the ALGOL 68 defining document:

van Wijngaarden, A. et. al. (eds.) Revised Report on the Algorithmic Language ALGOL 68. Springer-Verlag, 1976.

Summary of Changes Required for ALGOL 68

We list here the additions and deletions that must be made to ALGOL 68 to bring it into conformance with the TINMAN requirements. An addition is considered major (and marked with an asterisk) when it potentially interacts with many language features or when it significantly changes the character of the language and its use. Each addition or deletion is cross-referenced to a TINMAN requirement justifying the modification. A more detailed description of the nature of the required change will be found in the discussion of the referenced requirement.

Required Additions (ALGOL 68)

* change arithmetic types from implicit implementation dependent precision (via long) to just int and float with precision specified separately (A2, A3, A4, D4)

. problem-oriented floating point precision specification (A3)

. global default floating point precision specification (A3)

. "exact" fixed point real arithmetic, with specifiable precision (A4)

* specifiable character sets (A5)

. arrays indexable by enumeration types (A6)

* programmer control over discrimination of alternative record structures without having free union (A7)

* assignment and access definable for new types (B1)

. extend built-in equivalence definition to apply to all built-in atomic types (B2)

. unordered enumeration types (B3)

. define dyadic arithmetic operators for operands of mixed precision (B4)

. define rounding/truncation rules for real arithmetic (B5)

. short circuit evaluation of AND and OR operators (B6)

* define scalar operators over conformable arrays (B7)

* run time exception handling ability for range errors (B9)

. powerset of enumeration type, with operators (B11, E6)

- left-to-right evaluation order for operands, subscripts, and subroutine parameters (C1)
- require parentheses at same precedence level for non-associative operators (C2)
- require constant expressions to be evaluated at compile time (C4)
- change integer case clauses to use explicitly labeled alternatives (G3)
* exception handling, including overflow, range, and subscript errors (C7, B9, D4, G7)
- generic procedures (C8)
- require program and data numeric literals to have the same value (D2)
- run time uninitialized variable test (D3)
- range declarations (D4)
- enumeration types (D5, E6)
* group applicable operators with a type definition (E5)
* associate initialization, finalization, allocation, and deallocation actions with programmer defined types (E8)
* limit access to separately defined structures (F2)
* application-oriented library mechanism (F4, F5, F6)
- iterative control with termination condition occurring anywhere in loop body (G3, G4)
* asynchronous interrupt handling (G8)
* delay until time (real or simulated) or situation occurs (G8)
- real time clock access (G8)
- separate quoting of each line of long character string literals (H4)
- modify field selection syntax (H9)
- mechanisms for specifying object representation (I2, J4)
- extend set of environmental enquiries (I3)
- conditional compilation (I4)
- define translator restrictions/limits (I6)
- require that translators not build in object machine dependent language restrictions (I7)
* programmer control over movement between main and backing store (J1)
- machine language insertions (J3)
- associate identifiers with object machine addresses (J4)
- open and closed routine calls (J5)
* mechanism for integrating separately written modules (K2)
- software development tools (K3)
- translator options (K9)
- prohibit supersets and subsets (L2, L3)
- provide suggested error and warning messages (L7)
- DoD designated control agent, support agent, and library support (M4, M5, M6)

Required Deletions (ALGOL 68)

- string data type as built-in type (A2, I5)
- procedure variables (A2)
- bits type (A2)
- bytes type (A2)
- non-constant lower bounds for arrays (A6)
- flexible arrays (A6)
- widening coercion (implicit integer to real conversion) (B8, C6)

- formatted I/O (B10)
- priority (operator precedence) declarations (C2, H2)
- array substructuring (trimmers) as an operation with its own special syntax (C5)
- automatic inheritance by new types of the properties and operators of the underlying representation types (B7, E4, E5)
- dynamic storage allocation (F1)
- jumps from inner to outer lexical scope (G2)
- optional else and out clauses (G3)
- procedure definition within the bodies of recursive procedures (G5)
- arbitrary nesting of procedures and parallel clauses (G6)
- lexical units (including comments) continued across end of line (H5, H7)
- square brackets for array subscripting (H9)
- left hand side function calls (H10)
- alternate notations for conditional and case clauses (I5)
- conditional and case expressions (I5)
- valued assignment statement (I5)
- statements preceding declarations within a lexical scope (L3)

J3B Evaluation Summary

SofTech's evaluation of JOVIAL J3B with respect to the Department of Defense requirements for high order computer programming languages ("TINMAN", June 1976) is summarized here. Detailed evaluations of J3B with respect to each TINMAN reuirement are contained in separate sections of this report. These sections begin with an overall assessment of the extent to which the requirement is satisfied, followed by a justification of this assessment referencing sections of the J3B defining document:

> JOVIAL/J3B Extension 2 Language Specification. SofTech, Inc., Waltham, Mass., Document 2044-4.2, July 1975.

(A later version of this specification, dated September 1976, contains only minor changes that do not affect the evaluations reported here.)

Summary of Changes Required for J3B

We list here additions and deletions that must be made to JOVIAL J3B to bring it into conformance with the TINMAN requirements. An addition is considered major (and is marked with an asterisk) when it potentially interacts with many language features or when it significantly changes the character of the language and its use. Each addition or deletion is cross-referenced to a TINMAN requirement justifying the modification. A more detailed description of the nature of the required change will be found in the discussion of J3B with respect to the referenced TINMAN requirement.

Required Additions (J3B)

- require explicit discrimination among record variants with programmer control over how the discrimination condition is tested (A1, A7, E7, G3)

. Boolean data type (A2)
. problem-oriented floating point precision specification (A3)
. global default floating point precision specification (A3)
. exact fixed point data type (and modified scaling rules), with specifiable precision (A4, D4)
. ability to define new character sets (A5)
. enumeration data type (A5, E6), with and without ordering (B3)
. programmer-specified lower array bound (A6)
. arrays indexable by enumeration types (A6)
. permit upper subscript array (and table) bounds to be determined on entry to allocation scope (A6)
* assignment and access functions definable for new types (B1)
. return arrays and records as values of functions (B1, E1, E2)
. floating point equivalence defined for minimum (rather than maximum) precision of operands (B2)
* programmer-specified definition of equivalence operation (B2)
. unordered enumeration types (B3)
. relational comparisons defined for character strings (B3)
. fixed point exponentiation operator (B4)
. remainder (modulus) operator (B4)
. rounding required in floating point computations (B5)
. explicit ROUND operator available for fixed and floating point data (B5)
. require short circuit evaluation of Boolean operators (B6)
. generalized array and record assignment (B7)
* ability to apply scalar operations to conformable arrays and records (B7)
. explicit conversion operators from and to floating point representations (B8)
. explicit conversion operators from and to character string representations of numeric data (B8, D2)
* run time exception handling for range errors (B9)
* input/output operations (B10)
. power set data type and operations (B11, E6)
. left to right evaluation order for operands, subscripts, and subroutine arguments (C1)
. explicit requirement for parentheses within same precedence level for non-associative operators (C2)
. extend INTR, BIT, and BYTE operators to accept expressions and constants, if these functions are not deleted from the language (C3)
. extend constant expression computations to INTR, BYTE, BIT, SHIFTR, SHIFTL, relational expressions, and exponentiation, for those operators retained in the language (C4)
. evaluate constant expressions with target machine range and precision (C4)
. change BIT and BYTE notation to reflect potential modifiability of argument (C5)
* parameterized type definitions (C5)
. permit formal integer parameters to be declared without explicit size (C5)
. permit different array and table bounds for actual subroutine parameters (C6)
. ability to pass subroutines as parameters (C7)

* exception handling control structures (C7, G7)
  * range exception (B9)
  * overflow exception (B9)
  * subscript range exception (D4)
* generic subroutines (C8)
* variable number of parameters (C9)
* provide for input of literals (D2)
. initialization of data local to a subroutine (D3)
. initialization of typed tables (D3)
. run time testing option for accessing uninitialized variables (D3)
. range declarations (D4)
. permit arrays as components of records (D5, E6)
. enumerations types (D5, E6)
* prohibit parameter assignments to variables whose lifetime is longer than that of the object pointed to (D6)
* ability to define new infix and prefix operators (E1)
. permit arrays of records to be accessed with subscripts for user-defined record types (E2)
* ability to extend existing operations to new data types (E4)
* explicit support for data abstractions (E5)
* ability to define initialization and finalization procedures associated with variables of a user-defined type (E8)
. augment ability to define scope of allocation (F1)
. provide directive for redefining names of reusable program components (F2)
* hierarchies of COMPOOL scopes (F2, F6)
* extension libraries (F4)
* permit subroutine definitions in COMPOOLs (F5)
* ability to restrict COMPOOL references to particular programs or subsystems (F6)
. provide case statement (G3)
* equivalent of Zahn's device (G3, G7)
. provide new iterative control structure (G4)
* parallel control paths, real time clocks, asynchronous hardware interrupt language features (G6, G8)
. permit labeled END statements (H1)
. provide a break character for literals (H4)
. require separate quoting of long string literals (H4)
* uniform referent notation (H9)
. provide a new assignment symbol (H10)
. provide environmental enquiries (I3)
. provide for conditional compilation of declarations (I4)
. specify translator restrictions (I6)
* programmer control over movement between main and backing store (J1)
. provide encapsulated assembly code, including provisions for describing register usage within assembly code encapsulations (J3)
. provide ability to associate source language identifiers with machine addresses (J4)
* separate description of data representation from logical properties (J4)
. provide special comment brackets for assertions, etc. (K5)

Required Deletions (J3B)

- fractional fixed point data type (A4)
- character and bit string data types (A2, B11, I5)
- floating point equivalence definition (maximum instead of minimum precision) (B2)
- relational comparisons on pointer data (B3)
- implicit truncation in fixed and floating point arithmetic and assignments (instead of rounding) (B5)
- implicit conversion from real to integer in assignments (B8)
- untyped table data type and its use in assignment and parameter passing (B7, B8, C6)
- implicit conversion from integer to real in assignments and arithmetic operations (B8)
- implicit lengthening and shortening of character and bit strings in assignments and comparisons (B8)
- unsigned integer data type (B9)
- range matching between formal and actual parameters (B9)
- bit string data type as opposed to power set data type (B11)
- evaluation of constant expressions with host machine range and precision (C4)
- untyped pointer parameters (C6)
- default variable initialization (D3)
- distinction between tables and arrays (D5)
- overlay capability (E7)
- delete COPY capability (and extend COMPOOLS to include COPY) (F6)
- SWITCH statements (G2)
- IF-THEN without an ELSE statement (G3)
- FOR statement (G4)
- eight character rule for identifiers (H1)
- do not permit lexical units to continue across line boundaries (H5)
- do not reserve ABS, BIT, BYTE, ENTRY, FIX, INTGR, INTR, LBOUND, NENT, POINT, SCALE, SHIFTL, SHIFTR, and UNBOUND (H6)
- delete quote comment form, or change use of " in macro definitions and invocations to use % or some other delimiter, or delete macros from the language (H7)
- do not permit comments to extend across line boundaries (H7)
- substring assignment (H10)
* implementation dependencies (I1)
- character string data type as a special type rather than as an array of characters (I5)
- macro definitions and invocations (I5)
- object machine restrictions (see I7)
- provision for implementation dependent built-in functions (L1)
- require fixed point implementation (L2)

PASCAL Evaluation Summary

SofTech's evaluation of PASCAL with respect to the Department of Defense requirements for high order computer programming languages ("TINMAN, June 1976) is summarized here. Detailed evaluations of PASCAL with respect to each

TINMAN requirement are contained in separate sections of this report. These sections begin with an overall assessment of the extent to which the requirement is satisfied, followed by a justification of this assessment referencing sections of the PASCAL defining document and other pertinent publications. The following documents are cited throughout the PASCAL evaluation discussions (the citation format used in these discussions is specified below in square brackets for each document):

K. Jensen and N. Wirth, PASCAL User Manual and Report, Springer-Verlag, 1974 [p. xx].

N. Wirth, "An Assessment of the Programming Language PASCAL," IEEE Trans. on Software Engineering, 1, 192-198, (1975) [Wirth, p. xx].

N. Wirth, "The Design of a PASCAL Compiler," Software -- Practice and Experience, 1, 309-333, (1971) [Wirth, SPE, p. xx].

Other references to the PASCAL Newsletter and to two formal definitions of the language are given in full citation form where they appear.

The principal reference is the User Manual and Report. Pages 133-168 comprise the revised Report defining standard PASCAL. (Pages 3-87, 105-118 are the portions of the User Manual explaining standard PASCAL, and pages 88-104, 119-125 deal with PASCAL 6000.) Standard PASCAL is to be supported by all implementations to provide program portability. Standard PASCAL is the language evaluated and called PASCAL. The language PASCAL 6000 is a particular implementation for the Control Data 6000 series of computers. It is sometimes cited to illuminate the relationship between defining and implementing a language feature.

Summary of Changes Required for PASCAL

We list here the additions and deletions that must be made to PASCAL to bring it into conformance with the TINMAN requirements. An addition is considered major (and marked with an asterisk) when it potentially interacts with many language features or when it significantly changes the character of the language and its use. Each addition or deletion is cross-referenced to a TINMAN requirement justifying the modification. A more detailed description of the nature of the required change will be found in the discussion of the referenced requirement.

Required Additions (PASCAL)

* require explicit discrimination among record variants, with complete programmer control over how the discrimination condition is tested (A1, A7, G3)
. fixed point real arithmetic with specifiable precision and step size (A2, A4, D4)
. problem-oriented floating point precision specifications (A3)
. global default floating point precision specifiable (A3)
* ability to define new character sets (A5)
* permit non-constant array upper bounds (A6)
* assignment and access functions definable for new types (B1)

* permit functions to return array and record values (B1, E1, E2)
* programmer-definable equivalence operator for new types (B2)
. unordered enumeration types (B3)
. exponentiation operator (B4)
. define rules for truncation and rounding (B5)
. short circuit Boolean expression evaluation (B6)
* scalar operations on conformable arrays (B7)
. explicit conversion operators (B8)
* run-time exception for range errors (B9, G7, C7)
* dynamically assign/reassign I/O devices (B10)
. set complement operator (B11)
. specify left-to-right evaluation order of operands, subroutine arguments, array indices, and assignment operands (C1)
. require parentheses within the same precedence level for non-associative operators (C2)
. allow constant expressions where constants can appear (C4)
. constant expressions evaluated at compile time (C4)
* variable number of parameters (C5, C9)
* parameterized types (C5)
* parameterized array bounds (C6)
. require parameter declarations for procedures passed as parameters (C6)
* exception handling capability and control structure (C7, D4, G7)
* generic subroutines (C8)
. provide for input of Boolean constants (D2)
. require identical conversion of program and data literals (D2)
. run time testing option for accesssing uninitialized variables (D3)
. require specifying range of numeric variables (D4)
* prohibit assigning pointers to variables whose lifetime is longer than that of the object pointed to (D6)
* define new infix operators (E1)
* specification of a type's internal representation (E2)
* ability to extend existing operators to new types (E4)
* encapsulated type definition with initialization, finalization, allocation-time, and deallocation-time procedures (E5, E8)
* limit access scope (F2)
* external scope for separately defined modules (F2)
* ability to provide new local names for external identifiers (F2)
* support TINMAN library concept (F4, F5, F6)
. labels as identifiers (rather than numbers) (G2)
. restrict GOTO to same scope level and forbid jumping into loop bodies (G2)
. equivalent of Zahn's device (G3, G7)
. iteration with loop exit anywhere (G4)
* (pseudo) parallel processing capability with specifiable priorities (G6, G8)
. access to real-time clock (G8)
* synchronization, mutual exclusion, and delay primitives (G8)
. specify translation from publication language to ASCII (H3)
. internal break character for identifiers and literals (H4)
. separate quoting of each line of long literals (H4)
. uniform referent notation (H9)
. ability to specify routine is not reentrant or recursive (I2)

- provide environmental enquiries (I3)
- conditional compilation (I4)
- specify translator restrictions (I6)
* programmer control over movement between main and backing store (J1)
- machine language insertions (J3)
- associate identifiers with machine addresses (J4)
- specify development tools (K3)
- specify translator options (K4)
- special comment brackets for assertions and units of measure (K5)
- specify that syntax checking is required (L6)
- specific errors to be detected and error messages (L7)
- specify type matching rules for formal and actual parameters (M2)

Required Deletions (PASCAL)

- special parameter format for output procedures (C5)
- assignment to value parameters (C7)
- subrange data type (D4, E7)
- dynamic storage allocation procedures (F1)
- GOTO's across scope levels and not forbidding a jump into loop bodies (G2, G4)
- IF-THEN control structure (G3)
- not fully partitioned case statement (G3)
- for loop control variable accessible outside loop (G4)
- nested recursive procedures (G5)
- abbreviation of identifiers to first eight characters (H1)
- lexical units (including comments) crossing line boundaries (H5, H7)
- syntax sharing for up-arrow (H10)
- implementation dependencies (I1)

SIMULA 67 Evaluation Summary

SofTech's evaluation of SIMULA 67 with respect to the Department of Defense requirements for high order computer programming languages ("TINMAN," June 1976) is summarized here. Detailed evaluations of SIMULA 67 with respect to each TINMAN requirement are contained in separate sections of this report. These sections begin with an overall assessment of the extent to which the requirement is satisfied, followed by a justification of this assessment referencing sections of the SIMULA 67 defining document and other pertinent publications. The following documents are cited throughout the SIMULA evaluation discussions (the citation format used in these discussions is specified below for each document):

O.-J. Dahl, B. Myhrhaug, and K. Nygaard, SIMULA 67 Common Base Language, Norwegian Computing Center Publication No. S-22, 1970. ("CBL 2.2.1" refers to Section 2.2.1 in this defining document. As CBL 20 notes, the SIMULA 67 languge definition uses the revised ALGOL 60 report's syntax and semantics for features that are the same, so the following reference is also cited in this evaluation.)

P. Naur (Ed.), "Revised Report on the Algorithmic Language ALGOL 60," Comm. ACM, January 1963, vol. 6, no. 1, pp. 1-17. ("AR 3.3.5.2" refers to Section 3.3.5.2 of this document.)

G. Birtwistle, O.-J. Dahl, B.Myhrhaug, and K. Nygaard, SIMULA BEGIN, Auerbach Publishers Inc., Philadelphia, 1973. ("B 150" refers to page 150 in the SIMULA BEGIN book, which clarifies and illustrates a number of SIMULA 67 features.)

J. Palme, "SIMULA as a Tool For Extensible Program Products," ACM SIGPLAN Notices, February 1974, vol. 9, no. 2, pp. 24-40. ("Palme, p. 38" refers to page 38 of this paper)

NCC System Programming Group, "The Structure of the NCC SIMULA Compilers and Bench Mark Comparisons with other Major Languages," Norwegian Computing Center Publication No. S-43, 1972. ("Structure, p. 6" refers to page 6 of this report.)

Summary of Changes Required for SIMULA 67

We list here additions and deletions that must be made to SIMULA 67 to bring it into conformance with the TINMAN requirements. An addition is considered major (and marked with an asterisk) when it potentially interacts with many language features or when it significantly changes the character of the language and its use. Each addition or deletion is cross-referenced to a TINMAN requirement justifying the modification. A more detailed description of the nature of the required change will be found in the discussion of the referenced requirement.

Required Additions (SIMULA 67)

- problem-oriented floating point precision specification (A3)
- global default floating point precsions specifiable (A3)
- enumspecifiable character sets (A5)
- arrays indexable by enumeration types (A6)
* programmer control over how discrimination among record variants is tested (A7)
* assignment and access functions definable for new types (B1)
* return arrays and records directly as values of functions (B1, E1, E2)
* can extend domain of existing operator symbols (including equivalence) and can define new infix operators (B2, E1, E2, E4)
- same equivalence symbol for Boolean equivalence and for other forms of equivalence (B2)
- unordered enumeration types (B3)
- explicit truncation and rounding requirements defined (B5)
- exclusive-or Boolean operator (B6)
- assignment and scalar operations on arrays (B7)
- explicit conversion operators for arithmetic types (B8)
- run-time exception condition for numeric range errors (B9)
- power set of enumeration types with set operations (B11, E6)
- operand, subscript, and subroutine argument evaluation order specified as left-to-right (C1)

. require parentheses within same precedence level for non-associative operators (C2)
. evaluate constant expressions before run time (C4)
. extend class parameter capabilities to match procedure parameters (C5)
. type checking on arguments to subroutines passed as parameters (C6, C7)
* exception handling control structure plus a parameter class for specifying the control action when exception conditions occur (C7, D4, G7)
* generic procedures of more than one argument (C8)
. variable number of parameters (C9)
. constant value identifiers (D1)
. permit Boolean and NONE constants on input and output (D2)
. require identical conversion of program and data literals (D2)
. optional variable initialization instead of automatic (D3)
. run time test for uninitialized variables (D3)
. range declarations (D4)
. enumeration types (D5, E6)
* direct support for arrays of records (D5, E6)
* ability to define new infix operators (E1)
* extend existing operators to new types (E4)
* provide means of limiting the automatic inheritance of operations applicable to a data type in terms of which a new type is being defined (E5, F2)
* user-defined finalization, initialization, and deallocation actions associated with type definitions (E8)
. limit access scope both at definition and at use (F2)
. require support for application libraries (F4)
. support routines written in other source languages (F5)
. support Compool-like data sharing (F5, F6)
. more than one compool (F6)
. general case statement (G3)
. loops with exit tests in the middle and at the end (G3, G4)
. equivalent of Zahn's device (G3, G7)
* true parallel processing capability -- can create and terminate processes and have critical section control primitives (G6, G8)
* asynchronous interrupt handling (G8)
. access to real time clocks (G8)
. specify translation into ASCII character set (H3)
. break characters for internal use in identifiers and numeric literals (H4)
. separate quoting of each line of a long TEXT literal (H4)
. permit system defined procedures as parameters (H10)
. optional control over specifying object representations (I2)
. specification that a routine is not recursive or reentrant (I2)
. conditional compilations and compile time variables (I3, I4)
. make translator restrictions a part of the language definitions (I6)
* programmer control of movement between main and backing store (J1)
. machine language insertions (J3)
. ability to associate identifiers with object machine addresses (J4)
. specify object representation for composite data structures (E2, J4)
. can specify open or closed implementation for routine calls (J5)
. require support for separate compilation (K2)

- software development tools (K3)
- standard translator options (K4)
- provide for special brackets to enclose assertions (K5)
- smaller compilers for the language (L5)

Required Deletions (SIMULA 67)

- default REAL for arrays declared with no type (A1, E3)
- TEXT data type as built-in type for strings (B2, I5)
- array lower bounds being an arithmetic expression (instead of a constant) (A6)
- integer to real and real to integer conversion in assignment (B8)
- label parameters (C5, C7, G2)
- by-value parameters should be "read-only" instead of "read-write" (C7)
- by-name parameter transmission mode (C7)
- type specifications for formal parameters as a means of realizing generic procedures (C8)
- implementation restrictions on the use of the SIMSET and SIMULATION classes (E2)
- heap storage allocation (F1)
- GOTOs across scope levels (G2)
- switches (G2)
- designational expressions (G2)
- IF-THEN statements (with no required matching ELSE) (G3)
- non-fully partitioned INSPECT statement (G3)
- allowing GOTOs into loop bodies (G4)
- access to loop control variable from outside loop (G4)
- being able to define procedures within recursive procedure bodies (G5)
- allowing lexical units to cross line boundaries (H5)
- comment conventions as presently defined (H7)
- brackets (instead of parentheses) for array references (H9)
- substring assignment (H10)
- references to "operating systems" in the CBL definition (K1)
- implementation dependencies (I1)
- features that make the language non-minimal, e.g., the TEXT type for strings (I5)
- requiring garbage collection (J2)
- permitting superset implementations (L1)

TACPOL Evaluation Summary

SofTech's evaluation of TACPOL with respect to the Department of Defense requirements for high order computer programming languages ("TINMAN," June 1976) is summarized here. Detailed evaluations of TACPOL with respect to each TINMAN requirement are contained in separate sections of this report. These sections begin with an overall assessment of the extent to which the requirement is satisfied, followed by a justification of this assessment referencing sections of the TACPOL defining document:

TACPOL Reference Manual, USACSCS-TF-4-1, 15 January 1972.

Summary of Changes Required for TACPOL

We list here the additions and deletions that must be made to TACPOL to bring it into conformance with the TINMAN requirements. An addition is considered major (and marked with an asterisk) when it potentially interacts with many language features or when it significantly changes the character of the language and its use. Each addition or deletion is cross-referenced to a TINMAN requirement justifying the modification. A more detailed description of the nature of the required change will be found in the discussion of the referenced requirement.

Required Additions (TACPOL)

* a discriminated union record capability with programmer defined discrimination tests (A1, E7)
. floating point real numbers with specifiable precision (A2, A3)
. global default floating point precision specification (A3)
. specifiable step size for fixed point numbers (A4, D4)
. modified scaling rules for fixed point arithmetic (A4)
* ability to define new character sets (A5)
. array upper bounds fixed on entry to allocation scope, instead of at compile time (A6)
. arrays indexable by enumeration types (A6)
. programmer specifiable array lower bounds (A6)
* assignment and access functions definable for new types (B1)
. return arrays and records as values of functions (B1, E1, E2)
. exact fixed point equivalence operator (B2)
. programmer-specified equivalence operator for comparing user defined types (B2)
. unordered enumeration types (B3)
. integer division with its own symbolic operator (B4)
. rounding during expression evaluation (B5, I1)
. short circuit evaluation of Boolean expressions (B6)
. scalar operations on conformable arrays (B7)
. assignments applicable to arrays and structures (B7)
. capability to dynamically assign and reassign I/O devices (B10)
. power set data type and operations (B11)
. operand, subscript, and subroutine parameter evaluation done left-to-right (C1)
. parentheses required at same precedence level for non-associative operators (C2)
. explicit statement of where expressions can occur (C3)
. allow constant expressions wherever constants are allowed (C4)
. evaluate constant expressions before run time (C4)
. permit procedures passed as parameters to take arguments (C5)
* variable number of procedure parameters (C5, C9)
* exception handling parameter class (C7)
* a generic procedure capability (C8)
. require identical conversion of program and data literals (D2)
. permit records as comonents of records (D5, E6)
. initialization facility (D3)
. generalized range declarations (D4)
. enumeration types (D5, E6)

. pointer variables (D6)
* user definition of new types and infix operators (E1)
. consistent use of types (E2)
. require declaration of loop control variable (E3)
* user can extend existing operators (E4)
* data types with clustered operations definable (E5)
* type initialization and finalization (E8)
. provide for separating variable allocation from variable access (F1)
* ability to limit access to separately defined structures (F2)
. associate new local names with separately defined components (F2)
. application oriented libraries (F4, F5)
. routines written in languages other than TACPOL (F5)
. Compools at all levels of programming activity (F6)
. more than one Compool (F6)
. case statement (G3)
. loops with exit tests in middle (G3, G4)
. equivalent of Zahn's device (G3, G7)
. recursive procedures (G5)
* parallel processing capability (G6)
* fuller exception handling (G7, B9, D4)
* asynchronous interrupt handling (G8)
. access to real-time clocks (G8)
* synchronization and real-time capability (G8)
. break character for use internal to identifiers and literals (H4)
. separate quoting of each line of a long literal (H4)
. require key words to be reserved (H6)
. compile time variables (I3)
. conditional compilation capability with encapsulated code (I4, J3)
. ability to associate identifiers with object machine addresses (J4)
. greater control over composite object representation (J4)
. specify procedure call as open or closed implementation (J5, I2)
. more extensive translator options and available tools (K3, K4)
. special comment brackets for assertions (K5)
. suggested set of diagnostic messages (L7)
. language documentation, especially a formal in-depth definition (M3)

Required Deletions (TACPOL)

. character and bit string types (A2, B11, I5)
. MOVE operation to be replaced by a type checking structure assignment (A6, B1, B7)
. overlay with free union capability (B8, E7)
. partitioned data sets and file labels (B10)
. label parameters (C7, G2)
. write access to value parameters (C7)
. requiring specification of parameter type and dimension (C8)
. restrictions on nesting arrays and structures (D5)
. GOTOs out of same scope (G2)
. switches (G2)
. if-thens (with no else) (G3)
. DO control variable being non-local to DO loop (G4)
. GOTO into loops (G4)
. abbreviation of identifiers (H1, I1)

- continuation of lexical units across line boundaries (H5)
- size of key word list (H6)
- comments crossing line boundaries (H7)
- SUBSTR assignable pseudo-variable (H10)
- syntax sharing of equal sign for both equality test and assignment (H10)
- implementation dependencies (I1)
- multiple target assignments (I5)

A1. Typed Language (TINMAN)

The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable and component of composite data structures be explicitly specified in the source programs.

By the type of an object is meant the set of objects themselves, the essential properties of those objects, and the set of operations which give access to and take advantage of those properties. Compile time determination of types does not preclude the inclusion of language structures for dynamic discrimination among alternative record formats (see A7) or among components of a union type (see E6). Where the subtype or record structure cannot be determined until run time, it should still be fully discriminated in the program text so that all the type checks can be completed at compile time.

A1. Typed Language (ALGOL 68)

Satisfied.

Algol 68 is a typed language [Section 2.1.1.2, 1.2.1, 4.4, 4.6]. All identifiers must be declared [Section 4.4] with their types (modes). Labels are implicitly declared by their placement (prefixing a statement) [Section 3.2]. Field selectors are part of the mode of composite structures [Section 1.2.1]. The type of each language element is known at compile time, except for union types, which are fully discriminated in the program text, using case conformity clauses [Section 3.4.1q].

A1. Typed Language (J3B)

Partially Satisfied.

Explicit declarations for variables and components of composite data structures are always required, even for loop control variables [Sections 3.2 and 4]. Declarations of labels are required when necessary to override a macro declaration declared in a higher scope [Section 5.4.2]. However, there is no language construct that enforces and controls run-time discrimination among the subtypes of a record structure in the program text, so access to

records with alternative structures cannot be validated fully at compile-time. Even so, the requirement that fields of records be accessed only through pointers of an appropriate (possibly union) type [Section 6] means that compile-time checks on access to record alternatives are required to some extent.

A1. Typed Language (PASCAL)

Partially Satisfied.

"Every variable occuring in a statement must be introduced by a variable declaration" [p. 137]. Each component of a record has its type specified in the program text [p. 144]. Even labels must be explicitly declared [p. 158-9]. However, PASCAL provides no language construct for discriminating among record variants whose discrimination condition is not determined by a field in the record (see E7).

A1. Typed Language (SIMULA 67)

Partially Satisfied.

Explicit type declarations of variables are required except in one case: in an array declaration "if the type is omitted, the array is assumed REAL by default" [B 373]. "The structure of the SIMULA language is such that almost all the error checking of a program can be done at compile time" [Palme, p. 38]. Type checks can be performed at compile time in the sense required by A1, i.e., in the case where mismatching between the type required to perform an operation correctly, and the type actually needed, type checks can either be performed at compile time or must be specified in the program text via the INSPECT statement [B 150].

A1. Typed Language (TACPOL)

Partially Satisfied.

"All names must be declared" [Section 10-1]. Moreover, "the type of a quantity and the type of a value are explicitly defined" [Section 2-6.c]. However, the TACPOL CELL capability [Section 4-7.a] permits alternative record structures to be overlayed without requiring or providing constructs permitting the programmer to fully discriminate among these alternatives in the program text.

A2. Data Types (TINMAN)

The language will provide data types for integer, real (floating point and fixed point), Boolean, and character, and as type generators, will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e.,

composite data structures with labeled components of heterogeneous type).

A2. Data Type (ALGOL 68)

Partially Satisfied.

Algol 68 has integer, floating point real, Boolean, and character types, along with arrays and structures [Section 2.1.1.2, 1.2.1, 4.6]. It does not have a fixed point real type. Real constants [Section 8.1.2] and I/O formats [Section 10.3.4.3] are available. Algol 68 also has the following types: pointer (to another type), procedure (with typed parameters and results), (variable length character) string (flexible size array of character), bits (fixed size array of Boolean), and bytes (fixed size array of character).

A2. Data Types (J3B)

Partially Satisfied.

J3B provides integer, floating point (single and double precision), fixed point, character, array and record data types [Sections 4.2-4.5 and 4.8]. Only type generators for records are supported [Section 4.2]. J3B does not directly support a Boolean data type. Instead, bitstrings are given a Boolean interpretation in appropriate contexts [Sections 5.5 and 5.6]. Fixed length character strings are supported directly as a built in data type, although I5 implies character strings are to be supported as arrays of characters.

A2. Data Types (PASCAL)

Partially Satisfied.

PASCAL provides integer, real, Boolean, and character as standard types [p. 142]. Arrays and records are available as structured types [p. 143]. However, floating point is the only real representation supported [p. 93]. PASCAL does not have a fixed point real data type. PASCAL supports (fixed length) character strings as packed arrays of characters [p. 141].

A2. Data Types (SIMULA 67)

Partially Satisfied.

SIMULA 67 has REAL, INTEGER, BOOLEAN [AR 5.1.1], CHARACTER [CBL 3.1], pointer or REFerences to objects of a CLASS, which effectively realizes records [CBL 3.1], and ARRAY types [CBL 3.1w]. But fixed point is not supported [B 71].

The SIMULA text data type provides fixed length character strings (with programmer-defined upper limit on length defined at run time) and several associated operators [CBL 10, B 178].

A2. Data Types (TACPOL)

Partially Satisfied.

TACPOL has fixed point reals, integers (i.e, fixed reals with scale of zero), character strings, and bit strings as basic data types [Section 3-2, 3-3, 3-4, 2-6.h.(2)]. Arrays [Section 4-5, 4-6] and a form of records [Section 4-6, 4-7] are available, but not as type generators. There are restrictions on arrays and records that limit their generality (see D5, A7). TACPOL's character and bit string types exceed A2's requirement for Boolean and character (non-string) types. TACPOL does not support a floating point real type.

A3. Precision (TINMAN)

The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

Precision specifications will not change the type of reals nor the set of applicable operations. Precison specifications apply to arithmetic operations as well as to the data and therefore should be specified once for a designated scope. This permits different precisions to be used in different parts of a program.

A3. Precision (ALGOL 68)

Not Satisfied.

Algol 68 has variable precision in that integral and real types may be declared as real, long real, long long real, short real, etc. indicating more or less precision than real. However the number of bits (digits) corresponding to real, long real etc. is implementation dependent.

There is no way to specify the precision of arithmetic operations in some scope.

A3. Floating Point Precision (J3B)

Not Satisfied.

Although J3B supports single and double precision arithmetic [Sections 1.3.3.1], it does not support any more specific form of precision specification. Nor does it provide a means of specifying global (to a scope) precision for floating point arithmetic operations.

A3. Precision (PASCAL)

Not Satisfied.

PASCAL has only one representation for real variables [p. 93, 142]. There is no way for the programmer to specify precision, not even the common single or double precision.

A3. Precision (SIMULA 67)

Not Satisfied.

SIMULA 67 has an ALGOL 60 form of reals [B 71, AR 5.1.3]. Not even FORTRAN-like double precision can be specified. Accuracy is not explicitly definable. In fact, it is classified as "implementation defined" [B 72]. Global specification of precision is not supported.

A3. Precision (TACPOL)

Not Satisfied.

TACPOL does not support a floating point real data type.

A4. Fixed Point Numbers (TINMAN)

Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.

Integers will also be treated as exact quantities, with a step size equal to one.

A4. Fixed Point Numbers (ALGOL 68)

Not Satisfied.

Algol 68 does not have a fixed point real type [Section 1.2.1, 4.6].

A4. Fixed Point Numbers (J3B)

Not Satisfied.

Although J3B has a fixed point data type, it is a "fractional" rather than exact representation [Section 1.3.3.1, p. 1-14]; the declaration format does not permit an implementation independent specification of a step size, but instead requires the programmer to specify the assumed position of the binary point with respect to a computer word [Section 4.8]. For example, the

fixed point specification A 4 implies the binary point is 4 binary digits to the right of the sign bit. All fixed point values occupy the same number of bits, i.e., they all have the same precision.

Scaling of fixed point computations is performed automatically by the compiler [Section 7.1.4], with a programmer option for performing the scaling explicitly [Section 9.3]. The default scaling rules are not defined such that operations on fixed point values types having (in effect) a step size of one yield the same result as operating on integer value types. Moreover, the default scaling rules do not preserve fractional bits in arithmetic and comparison operations, demonstrating J3B's support of a "fractional" rather than an "exact" concept of fixed point values.

A4. Fixed Point Numbers (PASCAL)

Not Satisfied.

PASCAL does not support computation with fixed point real numbers [p. 93].

A4. Fixed Point Numbers (SIMULA 67)

Not Satisfied.

Fixed point reals are not supported [B 71].

A4. Fixed Point Numbers (TACPOL)

Partially Satisfied.

"In TACPOL, fixed point decimal numbers are converted and manipulated as their binary fixed point equivalents" [Section 3-2], meaning that only integer multiples of powers of two are represented exactly. (In terms of the TINMAN requirement, only step sizes that are powers of two are supported.) Numeric literals that are not integer multiples of powers of two are permitted, however, and are implicitly converted to an approximation of the decimal value. In this sense, TACPOL does not support an "exact" fixed point data type, since a programmer is not constrained to use literals exactly representable as integer multiples of a particular step size.

"During the evaluation of expressions, the scale factors of the values are automatically adjusted to the scale factor of the value containing the largest fractional part" [Section 6-2]. Presumably this implies that the product of two fixed point values with a scale of, say, 3 is also 3, so three fractional bits are lost. Hence, the scaling rules for multiplication and division do not conform to those of an "exact" fixed point data type.

Only fixed point precisions of 8, 15, 31, and 62 are supported [Section 3-2] by the implementation (although precisions of 1 to 62 can be specified in

declarations). Negative values cannot be represented if values are represented with a precision of 8. The precision of the result of arithmetic operations is either 31 or 62 bits. The precision is 62 if one of the operands has a specified precision of 32 to 62, or if both operands have a specified precision of 1 to 31 and the long precision multiply or long precision exponentiation operator is used [Section 6-2].

The scaling rules for comparison of fixed point values with different scales [Section 6-1.1] insure that no fractional bits are truncated before the comparison is performed (which is consistent with "exact" fixed point concepts), but no check is made on the possible loss of significant bits, so incorrect results can occur [Section 6-1.b(1)].

A5. Character Data (TINMAN)

Character sets will be treated as any other enumeration type.

Like any other data type defined by enumeration (see E6), it should be possible to specify the program literal and order of characters. These properties of the character set would be unalterable at run time. In general, unless all devices use the same character code, run-time translation between character sets will be required. Widely used character sets, such as USASCII and EBCDIC will be available in a standard library.

A5. Character Data (ALGOL 68)

Not Satisfied.

The collating sequence (mapping from char to int) is implementation defined [Section 2.1.3.1.g] in Algol 68. The language does not have enumeration types. However, one can construct other collating sequences and translate between them since the language allows arrays of characters and has the operators abs and reps [Section 10.2.1.n-o] for converting between char and int types.

A5. Character Data (J3B)

Not Satisfied.

Character strings are supported by the language, but in an implementation defined code [Section 1.3.3.1, p. 1-15]. The programmer has no ability to define new character sets, nor does J3B support an enumeration data type.

A5. Character Data (PASCAL)

Not Satisfied.

The programmer can define a scalar type by enumeration of identifiers [p. 142]. This enumeration defines an ordering to which both a successor and a predecessor function, as well as standard relational operators [p. 151], are applicable. Moreover, a variable of such a programmer enumerated scalar type can be used as an array index [pp. 143, 146]. For the standard scalar type "char" the ordering is implementation determined [pp. 142, 94-95]. However, a non-standard character set cannot be directly treated as a scalar type since only identifiers can be enumerated [p. 142].

The CDC ASCII 64 character set (which differs from the ISO standard) is available [p. 94] as a standard character set. And built-in functions chr (n), giving the n-th character in the standard scalar type char, and its inverse, ord, may be used in programmer defined conversions [p. 164].

A5. Character Data (SIMULA 67)

Not Satisfied.

SIMULA 67 does not support "enumeration type" as in, say, PASCAL. "The set of internal characters is ordered according to an implementation defined collating sequence" [CBL 3.2.2.1]. The built-in function procedure rank and char provide one-one conversion between integer character codes, such as EBCDIC, and characters [CBL 3.2.2.1, B 175-176]. This can facilitate translation between character sets. But the programmer cannot specify the "order of characters" (TINMAN A5).

A5. Character Data (TACPOL)

Not Satisfied.

TACPOL does not support defining new types by enumeration. The collating order of characters is predetermined and used for relational operators [Section 6-1.b(2)]. The programmer cannot specify or change this order.

A6. Arrays (TINMAN)

> The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. The number of dimensions, the type, and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.

The range of subscript values for any given dimension will be a contiguous subsequence of values from an enumeration type (including integers). The preferable lower bound on the subscript range will be the initial element of an enumeration type or zero, because it often contributes to program efficiency and clarity.

A6. Arrays (ALGOL 68)

Partially Satisfied.

In Algol 68 the number of dimensions and the element type are part of the array type [Section 4.6, 2.1.3.4] and are determinable at compile time. The lower and upper bounds may be expressions and hence their values need not be defined until run time. For non-flexible arrays, the bounds remain fixed once the array is allocated. For a variable declared as a flexible array, the bounds may be varied by assigning arrays with differing numbers of elements to the variable. Subscripts must be of type integer; there are no enumeration types. The default lower bound is 1.

A6. Arrays (J3B)

Partially Satisfied.

All requirements are satisfied except that the upper subscript bound is determined at compile time, not on entry to the array allocation scope, and the lower subscript bound is not programmer definable (it is fixed at zero) [Section 4.4]. Note, however, that lower bounds of tables (i.e., arrays of records) can be specified to be a constant other than zero [Section 4.3].

A6. Arrays (PASCAL)

Partially Satisfied.

PASCAL does not meet the crucial requirement that "the upper subscript bound will be determinable at entry to the array allocation scope" since the bounds (both upper and lower) of an array dimension are fixed at compile time [p. 143] even though arrays may be allocated at run time on entry to a procedure [p. 159]. PASCAL is compatible with the other A6 array requirements. The default lower bound for arrays is one.

A6. Arrays (SIMULA 67)

Partially Satisfied.

SIMULA 67 satisfies A6 except that the lower subscript bound is not determinable at compile time. Rather, "each bound is an arithmetic expression" [B 371, AR 5.2.1], for example, the variable N. Further, with no enumeration type facility, array subscripts are restricted to integer values [AR 3.1.4.2].

A6. Arrays (TACPOL)

Partially Satisfied.

In TACPOL, "three dimensions are the maximum allowable within an array declaration" [Section 4-2.c]. The range of subscript values is fixed at compile time. Lower subscript bounds are not programmer-definable, and are assumed to be 1 [Section 4-2.a].

A7. Records (TINMAN)

The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.

This permits hierarchically structured data of heterogeneous type, permits records to have alternative structures as long as each structure is fixed at compile time and the choice is fully discriminated at run time, but it does not permit arbitrary references to memory nor the dropping of type checking when handling overlayed structures. The discrimination condition will not be restricted to a field of the record but should be any expression.

A7. Records (ALGOL 68)

Partially Satisfied.

Component names (field selectors) are part of the structure type, as are the types of the component fields [Section 1.2.1, 4.4, 4.6, 2.1.3.3], and hence must be declared. Alternate structures are obtained by declaring a field to be a union type. Algol 68 union types are fully discriminated, but the programmer has no control over how the discimination condition is represented.

A7. Records (J3B)

Partially Satisfied.

Records with alternative structures are defined in J3B as "typed tables" [Section 4.3]. However, testing to discriminate at run time among record alternatives is not directly supported by a language construct.

A7. Records (PASCAL)

Partially Satisfied.

A PASCAL record type can have several variants with each identified and typed in its declaration [p. 144]. However, the requirement that the "discrimination condition will not be restricted to a field of the record but should be any expression" (TINMAN, p. 21) implies a discriminated union type union facility without necessarily having a discrimination tag as a record field (or even stored within the record representation). In PASCAL the tagfield of a variant record is optional [p. 144]. When it is omitted, the equivalent of a free type union is obtained [Wirth, p. 197].

A7. Records (SIMULA 67)

Partially Satisfied.

SIMULA 67 provides the equivalent of records with alternative structures with its prefixed classes (or subclasses) capability [CBL 2.2.1]. The INSPECT statement with its optional WHEN form [CBL 7.2.1, B 150-151] provides for discrimination between objects (i.e., record equivalents) of alternative structures. The programmer does not, however, have control over the internal representation of the discrimination information.

A7. Records (TACPOL)

Not Satisfied.

TACPOL severely limits the hierarchical structuring of heterogeneous data in that records and arrays of records may not be components of records. Record components are limited to scalars and arrays of elementary types [Section 4-5]. A special construct, CELL, permits records to be overlayed [Section 4-7.a]. This provides a minimal variant record capability. Discrimination and type checking are the programmer's responsibility.

B1. Assignment and Reference (TINMAN)

Assignment and reference operations will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.

The user will be able to declare variables for all data types. Variables are useful only when there are corresponding access and assignment operations. The user will be permitted to define assignment and access operations as part of encapsulated type definitions (see E5). Otherwise, they will be automatically defined for types which do not manage the storage for their data. (See D6 for further discussion).

B1. Assignment and Reference (ALGOL 68)

Partially Satisfied.

The Algol 68 user may declare variables of any type [Section 4.4]. Assignment [Section 5.2.1] and access [Section 4.8, 6.2] are automatically available for any type. However, assignment and access are not operators and cannot be user defined. But the user may define other operators to have the same effect as assignment and access for specific types.

Arrays and records can be specified as values of functions [Section 5.4.1] although the implication of such a specification is that a pointer to an array or record is what is actually returned.

B1. Assignment and Reference (J3B)

Partially Satisfied.

Although assignment and reference operations are automatically defined for all J3B data types, and assignments are permitted among records with alternative structures (which can be used to, in effect, implement a union data type), the user is not permitted to define assignment and access operations as part of record type definitions.

In addition, arrays and records cannot be returned as values of functions [Section 3.3.3], although such a capability is implied by B1, E1, and E2.

B1. Assignment and Reference (PASCAL)

Partially Satisfied.

In PASCAL "direct assignment is possible to variables of any type, except files" [p. 21]. Files are a type in PASCAL [p. 145] but presumably files are not considered to manage their data storage".

Arrays and records cannot be specified as values of functions (p. 162), although such a capability is implied by B1, E1, E2, and B7.

No ability to define assignment and access operation for new types is provided.

B1. Assignment and Reference (SIMULA 67)

Partially Satisfied.

All elementary and character string (i.e., text) values can be assigned. Other data types are essentially objects of declared classes, with object reference assignment automatically defined to cover any user defined classes [CBL 6.1, 6.1.2, 6.1.2.2]. However, the user cannot define assignment and access operations that are automatically invoked by, say, the assignment operator.

Arrays and records cannot be returned as values of functions [CBL 8.1], although such a capability is implied by B1, E1, E2, and B7.

B1. Assignment and Reference (TACPOL)

Partially Satisfied.

The assignment operator applies only to elementary types (i.e, short and long fixed point numerics, character strings, and bit strings) [Section 5-1]. A built-in procedure MOVE can be employed to assign structured types (i.e, arrays, groups, tables, and cells). MOVE effects bit string copying and does no type checking [Section A-6]. Of course, assignment of file types is not supported [Section 13-16]. Encapsulated type definitions are not present in TACPOL (see E5).

Arrays and records cannot be returned as values of functions, although B1, E1, E2, and B7 together imply the need for such a capability.

B2. Equivalence (TINMAN)

The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

There are many useful equivalence operations for some types and a language definition cannot foresee all these for user defined types. Proper semantic interpretation of identity requires that equality and identity be the same for atomic data (i.e., numbers, characters, Boolean values, and types defined by enumeration) and that elements of disjoint types never be identical. For floating point numbers identity will be defined as the same within the specified (minimum) precision.

B2. Equivalence (ALGOL 68)

Satisfied.

The Algol 68 equal operator (=) is defined for integer, real, Boolean, and character types [Section 10.2.3]. The identity relation (:=:) [Section 5.2.2] allows one to test whether two names (addresses) are the same. The programmer may define the = operator for other types, e.g., arrays, records (structures) and mixed precision real or integer.

B2. Equivalence (J3B)

Partially Satisfied.

An equivalence operation is automatically defined for comparing integer, floating, fixed point, bit string, character string, and pointer variables and literals [Section 7.3]. In addition, table entries (i.e., elements of an array of records) can be compared for equality. However, a user cannot define the meaning of equivalence for a record data type (the only user-definable data type permitted in J3B), and floating point values of different precisions are not compared within their minimum precision (e.g., in comparing a single

precision quantity with a double precision quantity, the double precision value is not truncated before comparison; instead the single precision value is compared against the exact double precision value) [Section 7.3, p. 7-14].

B2. Equivalence (PASCAL)

Partially Satisfied.

The built-in equivalence relational operator (=) is only applicable to operands of scalar, subrange, or [packed] array of character (i.e., strings) type [p. 151]. The equivalence operator doesn't encompass the other structured types (e.g., records) and a programmer cannot extend its definition to these types. Since only one floating point precision is supported, the question of how to compare values of differing precisions does not arise.

B2. Equivalence (SIMULA 67)

Partially Satisfied.

EQV is the logical equivalence operator [B 369] and SIMULA 67 inherits the ALGOL 60 equivalence relation for simple arithmetic expressions [AR 3.4.1] including reals. This implies "bit-by-bit comparison" for reals [AR 3.3.6]; this satisfies B2 since only one real precision is supported. Character and text values are compared for being equal (not identical) [CBL 5.1, 5.2]. References are compared for identity (i.e. if they refer to the same object or text value instance) with == and =/= [CBL 5.4]. In this vein if we let T and U be text references, then "relations T=/=U and T=U may both have the value true. Then T and U refer to physically distinct character sequences which are equal" [CBL 5.4.2]. The distinction between EQV and = means SIMULA does not have a single built in equivalence operator for atomic data.

The definition of built in infix operators (in particular, =) cannot be extended to new data types.

B2. Equivalence (TACPOL)

Not Satisfied.

The built-in equivalence operator tests only for equality of elementary (numeric or string) types [Section 6-1.b]. In the comparison of arithmetic values "the possibility of some significant bits being lost" through shifting can produce an unintended true result [Section 6-1.b.(1)]. Comparing structured types must be coded since it is not built-in.

B3. Relationals. (TINMAN)

Relational operations will be automatically defined for numeric data and all types defined by enumeration.

It will be possible to inhibit ordering definitions when uxnrdered sets are intended.

B3. Relationals (ALGOL 68)

Partially Satisfied.

All six relations are defined for integer, real, character, and Boolean types [Section 10.2.3]. They are also defined for the string (flexible size character arrays) and bytes (fixed size character arrays) types. Algol 68 has neither enumeration types nor unordered set types.

B3. Relationals (J3B)

Partially Satisfied.

Relational operations are defined for numeric data in J3B [Section 7.3]. Only equality and inequality comparisons, however, are permitted among character strings [Section 7.3], and since J3B does not support an enumeration data type, there are no relational operations applicable to such data. In addition, J3B permits relational comparisons between pointer variables of compatible type [Section 7.3]. (This is because in J3B, pointers can be used to access table entries, i.e., elements of an array of records, and therefore, operations on index values are extended to pointer values.)

B3. Relationals (PASCAL)

Partially Satisfied.

The six relational operators are applicable to operands of any scalar or subrange type [p. 151]. In PASCAL scalar types include not only number and enumeration but also character and Boolean [p. 151] which, in effect, inhibit any ordering of set elements.

PASCAL does not support unordered enumeration types, however.

B3. Relationals (SIMULA 67)

Partially Satisfied.

All six relational operators are defined for numerics and text data [AR 3.4.1, CBL 5., B 183]. SIMULA 67 does not have types defined by enumeration and, hence, B3 is vacuously satisfied for such types.

B3. Relationals (TACPOL)

Partially Satisfied.

All six relational operators are included in TACPOL and are applicable to numeric and string types [Section 6-1.b]. Defining types by enumeration is not supported.

B4. Arithmetic Operations. (TINMAN)

The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

Floating point operations will be precise to at least the specified precision.

B4. Arithmetic Operations (ALGOL 68)

Partially Satisfied.

Algol 68 has all of the required operations [Section 10.2.3]. However, it has no fixed point type. The exponentiation operator is defined only for real bases and integer exponents, but may be user defined for other combinations of numeric types. Other arithmetic dyadic operators are not defined over operands of differing precision, e.g., there is no built-in definition for adding a real and a long real [Section 10.2.3.4, 10.2.3.5].

B4. Arithmetic Operations (J3B)

Partially Satisfied.

Addition, subtraction, multiplication, division, and negation operators are provided for numeric data [Section 7.1]. Exponentiation is supported, but only yields a floating point result [Sections 7.1.2 and 7.1.3]. In addition, exponentiation cannot be applied to fixed point values [Section 7.1.4]. Implementations that do not support floating point data also do not support an exponentiation operator [Section 7.1].

There is no operation that directly yields the remainder when two integer values are divided [Section 7.1.1].

The result of floating point computations are maintained to a precision equal to the greatest precision of the operands [Sections 7.1.2 and 7.1.3].

B4. Arithmetic Operations (PASCAL)

Partially Satisfied.

PASCAL has all of the required built-in arithmetic operations except exponentiation [p. 150].

B4. Arithmetic Operations (SIMULA 67)

Satisfied.

All of the required built-in arithmetic operators are inherited from ALGOL 60 [CBL 4.1.1, AR 3.3.1]. The programmer cannot specify the precision for floating point operations.

B4. Arithmetic Operations (TACPOL)

Partially Satisfied.

TACPOL has the required built-in functions except for integer division [Section 6-1.a]. But only "exponentiation by positive integers is permitted" [Section 6-1.a]. Integer division can be effected by assigning a fixed real division result to an integer (i.e, scale factor of zero). A division remainder built-in function REM is provided for both short numeric [Section A-2] and long numeric [Section A-3] values.

B5. Truncation and Rounding (TINMAN)

Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

B5. Truncation and Rounding (ALGOL 68)

Partially Satisfied.

Assignment [Section 5.2.1] and access [Section 4.8] never truncate or round in Algol 68. However, truncation and rounding are implementation dependent for the arithmetic operations [Section 10.2.3, 2.1.3.1]. The round operator is only defined for converting a real value to an integer value of equivalent precision [Section 10.2.3.4], while the shorten operator applies to reals, yielding a rounded real result of less precision. The entier operator truncates reals to integer values. There is a widening coercion [Section 6.5] that implicitly converts integers to reals.

B5. Truncation and Rounding (J3B)

Partially Satisfied.

Arithmetic and assignment operations on integer, floating and fixed point quantities do not truncate significant digits as long as overflow does not occur [Section 7.1 and 5.2]. Implicit truncation of low-order digits is permitted, however, in fixed and floating point arithmetic and assignments [Section 7.1 and 5.2]. Rounding is not required and there is no explicit ROUND operator. In assigning fixed or floating point values to integer variables, fractional bits are always truncated implicitly [Section 5.2, p. 5-5].

B5. Truncation and Rounding (PASCAL)

Not Determinable.

It is not specified as part of standard PASCAL how truncation and rounding are done during arithmetic and assignment operations. However, PASCAL does contain standard built-in transfer functions trunc () and round () for converting from real to integer [p. 164].

B5. Truncation and Rounding (SIMULA 67)

Partially Satisfied.

Reals are implicitly rounded [AR 4.2.4]. But "it is understood that different hardware representations may evaluate arithmetic expressions differently" [AR 3.3.6]. SIMULA 67 does not have fixed point reals, nor is precision definable for floating point reals.

B5. Truncation and Rounding (TACPOL)

Not Satisfied.

"During the evaluation of expressions, the scale factors of the values are automatically adjusted to the scale factor of the value containing the largest fractional part" [Section 6-2.a]. This means the most significant bits can be implicitly truncated and that low order bits can be lost on multiplication. TACPOL does provide built-in TRUNC and ROUND procedures for short [Section A-2] and long [Section A-3] numeric values.

B6. Boolean Operations (TINMAN)

The built-in Boolean operations will include "and, " "or, " "not, " and "xor." The operations "and" and "or" on scalars will be evaluated in short circuit mode.

Note that the equivalence and nonequivalence operations (see B2) are the same as logical equivalence and exclusive-or respectively.

B6. Boolean Operations (ALGOL 68)

Partially Satisfied.

Algol 68 has "and", "or", "not", and "exclusive or" operators [Section 10.2.3.2]. However, the order of evaluation of operands is not defined [Section 5.4.2]. Choice clauses (i.e. conditional expressions) must be used to obtain the effect of short circuit evaluation in an implementation independent manner, e.g., short circuit A AND B could be written (A|B|false).

B6. Boolean Operations (J3B)

Partially Satisfied.

J3B does not support Boolean data types, but instead, supports bit strings. AND, OR, NOT, and XOR are available as operations on bit strings [Section 7.3]. Short circuit evaluation of bit string operations is permitted but not required by the language specification (p. 7-14).

B6. Boolean Operations (PASCAL)

Partially Satisfied.

The Boolean operators "and," "or," and "not" are built-in and "xor" for Booleans is a special case of the inequality operator [pp. 149-150].

However, PASCAL does not evaluate Boolean expresssions in short circuit mode. In coding "and" or "or" operators "the programmer must assure that the second factor is well-defined, independent of the value of the first factor" [p. 21].

B6. Boolean Operators (SIMULA 67)

Partially Satisfied.

Exclusive-or is not a built in Boolean operator; only EQV is built-in. The operators AND and OR are defined to be evaluated in short circuit mode (e.g., P AND Q is defined to mean IF P THEN Q ELSE FALSE) [B 369]. (Note: Boolean relations are also defined as ALGOL 60 relations [CBL 5] and the ALGOL 60 and operator definition does not require short circuit mode evaluation [AR 3.4.5].)

B6. Boolean Operations (TACPOL)

Partially Satisfied.

TACPOL has the Boolean operators AND, OR, and NOT built-in [Section 6-1.d]. They apply to bit strings and yield bit string results. The not equal NE relational operator yields a one bit result [Section 6-1.b.(3)]. But exclusive-or (and any other Boolean operation) can be realized using the built-in bit string procedure BOOL [Section A-5]. Short circuit evaluation mode is not required.

B7. Scalar Operations on Composite Data Structures (TINMAN)

The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

Conformability will require exactly the same number of components (although a scalar can be considered compatible with any array) and one for one compatibility in type. Correspondence will be by position in similarly shaped arrays. Although component by component operations will be available for built-in composite data structures which are used to define application-oriented structures, that capability will not be automatically inherited by defined data structures. Component by component operations also allow operations on character strings represented as vectors of characters and allow efficient Boolean vector operations.

B7. Scalar Operations on Composite Data Structures (ALGOL 68)

Partially Satisfied.

Algol 68 allows assignment of arrays and records (structures) [Section 5.2.1]. Each logical structure has only one physical structure since there are no packed or aligned attributes. For array assignment, the bounds must match, but the programmer can use the new lower bound option to force a match. Thus, only the subscript extent (for each dimension) must match. The scalar operators are not defined for arrays and structures, but the programmer can define them for these types. However, if one defines, for example, multiplication of arrays to be component by component, then this definition is inherited by types (e.g., matrix) represented as arrays.

B7. Scalar Operations on Composite Data Structures (J3B)

Partially Satisfied.

Although J3B permits an array element which is a record to be compared for equality with a similar array element (or the value zero) [Section 7-12], and although assignments to such elements are permitted [Section 5.2], a general form of assignment is not supported for arrays and records as a whole.

In addition, permitted assignments of array elements do not necessarily require one-for-one compatibility in type (p. 5-6). Furthermore, no scalar operations can be applied to arrays or records as a whole.

B7. Scalar Operations on Composite Data Structures (PASCAL)

Partially Satisfied.

Scalar operations cannot take arrays as operands [p. 150]. Direct assignment can be applied to arrays or records as long as the transfer is between identical types [p. 21]. This should permit run time conversion of object representations since the prefix packed for arrays or records "has no effect on the meaning of a program" [p. 143]. But, in fact, "a component of a packed structure must not appear as an actual variable parameter" [p. 83].

B7. Scalar Operations on Composite Data Structures (SIMULA 67)

Not Satisfied.

Neither scalar operations nor assignment on conformable arrays are permitted. For example, in arithmetic assignment the left side is restricted to a variable of type real or integer [CBL 6.1.2.1]. That is, an array is not permitted as the left side (receiver) of an assignment. Data transfers for aggregates are not possible. The built-in COPY function for text-objects produces only identical copies, not some conversion or packing [B 179-181].

B7. Scalar Operations on Composite Data Structures (TACPOL)

Not Satisfied.

In TACPOL, assignment [Section 5-1] and scalar operations [Section 6-1] are applicable only to elementary numeric and string data types. The built-in procedure MOVE copies the bits comprising an aggregate's value [Section A-6]. As such, it partially supports array and structure assignment, but no type checking is done.

B8. Type Conversion (TINMAN)

There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversion operations among integer, fixed point and floating point data, between the object representations of numbers and their representations as characters, and between fixed point scale factors.

B8. Type Conversion (ALGOL 68)

Not Satisfied.

Algol 68 has several coercions (implicit type conversions), but since Algol 68's concept of type is somewhat broader than the TINMAN's, only widening violates B8. Widening [Section 6.5] transforms integers to reals in assignments and subroutine calls, but not in loops or array subscripting. Dereferencing [Section 6.2] obtains the value stored at a given address. (Note: the type of an Algol 68 variable is "reference to another type", so a dereferencing coercion is extremely useful.) Deproceduring [Section 6.3] allows a parameterless procedure to be called without having to write an empty argument list "()". Uniting [Section 6.4] converts a component type of a union type to the union type itself. The union type may be fully discriminated by a conformity clause [Section 3.4]. Rowing [Section 6.6] forms a one element array from a single element. Voiding [Section 6.7] discards the value of a unit (expression or statement). (In a serial clause (block body) [Section 3.2], all units (statements) except the last are voided.) Algol 68 has no fixed point type. The language has operators for converting between real and integer values of differing precision [Section 10.2.3]. It also has two real to integer conversion operators, round and entier. There are procedures for converting between numeric and character representations of numbers [Section 10.3.2.1].

Relational operations on strings of different lengths are permitted [Section 10.2.3.10]. Assignment of strings never, however, causes truncation of the string being assigned.

B8. Type Conversion (J3B)

Partially Satisfied.

The J3B parameter passing rules for typed tables and typed pointers (to tables) satisfy the TINMAN parameter matching constraints in that all the fields of a record accessible through a formal parameter must be potentially present in the record (or array of records) serving as the actual parameter [Section 5.3.3, p. 5-12].

On the other hand, since J3B also supports untyped tables and pointers, the TINMAN's parameter matching constraints can be violated. Moreover, the language supports implicit conversion from integer to real and real to integer in assignment statements [Section 5.2, p. 5-4], although such conversions are not permitted between actual and formal parameters [Section 5.3.3]. Explicit conversion operators are provided for converting between integer and fixed point representations [Section 9.4], but no explicit operators are provided for converting to or from floating point representations. There are also no conversion operators to convert between the object representation of numbers and their representations as character strings, although the BYTE and INTR operation [Section 9.2] can be used to program such conversion operations.

An operator is provided for converting between fixed point scale factors [Section 9.3]. However, the operator does not override the default fixed point scaling rules, but instead rescales the result of the default rules. This is not the most appropriate form of rescaling operator.

The intent of B8 is also not satisfied by implicit lengthening and shortening of bit and character strings in assignment operations and relational comparisons [Section 5.2, p. 5-5].

The requirement is satisfied with respect to array indexing, since implicit real to integer conversion is not permitted in that context [Section 6]. In addition, no real to integer conversion is permitted in iteration statements (which are restricted to integer and pointer valued control variables) [Section 5.6].

B8. Type Conversion (PASCAL)

Not Satisfied.

PASCAL has implicit conversion from integer to real [p. 152] (except in for statements [p. 157] and in indexing arrays [p. 147]), no fixed point data [p. 93,143], and none of the explicit conversion operations required here. However, no conversion is necessary if an actual parameter is a variant of the corresponding formal's record type. On input, "data transfer is accompanied by an implicit data conversion operation" [p. 164].

In comparing packed arrays of characters (i.e., strings), it is not specified whether the arrays must be of the same length [p. 151].

B8. Type Conversion (SIMULA 67)

Not Satisfied.

There are no explicit conversion operators for arithmetic data. Integer-to-real and real-to-integer conversions are effected by assignment [CBL 6.1.2.1, B 80]. The function procedures rank and char provide a rudimentary explicit mechanism (i.e., at the individual character level) for converting between the object representations of numbers and their representions as characters. An actual parameter must be of a "compatible type" with its corresponding formal [B 103]. This effectively includes an actual being a constituent of a formal union type [CBL 6.1.2.2, B 151].

SIMULA incorporates the Algol rule that real expressions used as array subscripts are implicitly converted to integer values [CBL 7.1.1, AR 3.1.1]. In addition, implicit conversion between real and integer values is permitted in the evaluation of for loops [CBL 6.2].

B8. Type Conversion (TACPOL)

Partially Satisfied.

TACPOL does not allow implicit type conversions. In general, "the type of data in expressions may not be mixed" [Section 6-2]. Built-in type converters are called "Redefinition Attribute Procedures" [Section A-8]. Naturally enough, they correspond to the elementary types SHORT, LONG, CHAR, and BIT. CHAR, for example, converts object representations of numeric values to their representations as character strings. The built-in SCALE procedure [Section A-2, A-3] is used to rescale numeric values. Though not explicitly stated, the text and examples imply that parameters passed by reference must match exactly in the quantities they specify (i.e, the storage layout and its exact breakdown into elementary types) [Section 11-2]. All of the above TACPOL features satisfy the B8 requirements. However, the overlay capability of CELLs [Section 4-7] permits implicit type transfers. The example depicted in Figure 4-5 illustrates this. Moreover, the MOVE assignment operator does not check that the type of the composite object being copied is compatible with the type of the object receiving the copy.

B9. Changes in Numeric Representation (TINMAN)

Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.

(This requirement is optional for hardware installations which do not have overflow detection).

B9. Changes in Numeric Representation (ALGOL 68)

Not Satisfied.

Algol 68 does not have numerical subranges. It has no fixed point type. It does not define any action for overflow/truncation exceptions. Nor is there any exception condition available to the programmer for defining his own action.

B9. Changes in Numeric Representation (J3B)

Partially Satisfied.

J3B does not require range declarations for numeric data, but for integer variables, it does require the programmer to specify the number of bits occupied by valid data values. In addition, it supports an unsigned integer data type whose values are always supposed to be non-negative. Specifying the number of bits occupied by valid data values provides an implicit range declaration for signed integers of -2**N+1 to 2**N-1, and for unsigned integers, 0 to 2**N-1 [Section 1.3.3.1]. These implicit "range" declarations

are not checked at compile time when assigning to integer variables [Section 5.2], thus satisfying B9's requirement for no "explicit conversion operations ... between numeric ranges." On the other hand, if a formal parameter is declared to be S 5, for example, then the actual parameter must also be an S 5 [Section 5.3.3]. In effect, the implicitly declared "ranges" must match exactly. The programmer can deactivate this matching requirement, however, by specifying the formal to be of type S (without any implicit "range"), to imply that the actual parameter may be any integer value [Sections 4.8 and 5.3.3].

Range validation at run-time is not under programmer control within the language. In addition, J3B provides no exception condition for integer or fixed point overflow occuring as the result of either an arithmetic opration or assignment.

B9. Changes in Numeric Representation (PASCAL)

Partially Satisfied.

PASCAL supports implicit conversion from integer to real and from compatible subranges [p. 152] but does not support fixed point numbers [p. 93]. PASCAL 6000 has run time tests for range errors as a user option [p. 101] with error messsage "element expression out of range" [p. 121]. There is not, however, an exception handling facility in PASCAL.

B9. Changes in Numeric Representations (SIMULA 67)

Not Satisfied.

There is no run time exception condition for numeric range errors. Rather, the prevention and/or detection of conditions such as overflow are the programmer's responsibility. To do this one must take the specific hardware representations into account [CBL 4.1.1, AR 3.3.6].

B9. Changes in Numeric Representation (TACPOL)

Satisfied.

TACPOL provides a FOFL condition for fixed overflow [Section 12-1]. If the programmer declares the FOFL condition with CHECK, a snapshot procedure is invoked.

B10. Io Operations (TINMAN)

> The base language will provide operations allowing programs to interact with files, channels, or devices including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

I/O operations can be made accessible in a HOL in an abstract form through a small set of generic I/O operations (like "read" and "write, " with appropriate device and exception parameters). Note that devices and files are similar in many respects to types, so additional language features may not be required to satisfy this requirement. This requirement, in conjunction with requirement E1, permits user definition of unique equipment and its associated I/O operations as data types within the syntactic and semantic framework provided by the generic operations.

B10. I/O Operations (ALGOL 68)

Satisfied.

Algol 68 has procedures for dynamically associating internal file variables with external data sets and channels/devices [Section 10.3.1.4]. It has procedures for formatless [Section 10.3.3], formatted [Section 10.3.5], and binary [Section 10.3.6] I/O. Support is provided for simultaneous read access by several processes [Section 10.3.1.1(bb)], as well as user control of some exception conditions [Section 10.3.1.3.cc], e.g. end of file.

Formatted I/O provides more elaborate editing capabilities that B10 requires, but Algol 68's I/O capabilities in other respects, however, may provide a useful basis for satisfying B10.

B10. I/O Operations (J3B)

Not Satisfied.

J3B provides no input/output operations.

B10. I/O Operations (PASCAL)

Partially Satisfied.

The data type "file" with associated operations get, put, reset, and rewrite is supported [p. 55]. The correspondence between a program variable of type file and a peripheral device or logical file is implementation dependent and, so, outside of standard PASCAL [p. 55]. External files are made available "as actual parameters in the program call statement" [p. 91]. These are then established for the duration of program execution. Therefore, programs cannot dynamically assign and reassign I/O devices. Only the exception condition end-of-file, plus end-of-line for the line-oriented textfiles (i.e., file with component type "char") are provided for user control [p. 161, 165].

B10. I/O Operations (SIMULA 67)

Satisfied.

The class FILE is defined (as a subclass of BASICIO) with four subclasses predefined: infile, outfile, directfile, and printfile. A subclass of FILE can include implementation code to communicate data with an external device (CL 11, 11.1). A subclass parameter "ties the internal file object to the external medium" [B 192]. Exception conditions (e.g., more, endfile, lastitem) are available, or can be coded as Boolean procedures, for user control.

Given encompassing declarations such as:

```
BEGIN CLASS file ...;
          file CLASS device1 (-) ...;
          file CLASS device2 (-) ...;
          REF (file) source;
```

Then the code fragments

```
source :- NEW device1 (-);
          .
          .
          .
source :- NEW device2 (-);
```

suggest how "to dynamically assign and reassign I/O devices" in SIMULA 67 [CBL 11.-entire section, B 190-207].

SIMULA 67's I/O capabilities may be a good example of how B10 should be satisfied.

B10. I/O Operations (TACPOL)

Partially Satisfied.

TACPOL provides for input/output via READ [Section 13-6] and WRITE [Section 13-7] statements operating on declared FILEs [Section 14-3]. The user can control the subsequent actions on three file processing exception conditions using the ON statement [Section 13-13]. These are ENDFILE [Section 13-13.a] for serial input files, NOKEY [Section 13-13.b] for a key error in a direct access file, and NOPART [Section 13-13.c] for determining "whether or not an attempt was made to open an old partition in a file when that partition no longer exists." It appears that TACPOL programs cannot "dynamically assign and reassign I/O devices" since "file declarations may not appear outside the Compool" [Section 14-1]. Moreover, "the Compool is like a block in which all programs are contained" [Section 14-2]. This means only a single device can be associated with a given program identifier (declared to be a file name) during a program's execution.

TACPOL requires support for partitioned data sets and file labels [Section 14-3]. This is more than a minimal generic set of I/O concepts.

B11. Power Set Operations (TINMAN)

The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

B11. Power Set Operations (ALGOL 68)

Not Satisfied.

Algol 68 does not have enumeration types. It does not have power set types. It does have a bits type [Section 10.2.2.g] that is an implementation dependent fixed size array of Booleans. This provides for sets of integers. There are union, intersection, complement, and element operators [Section 10.2.3.8] for bits data and operators for converting between bits and "array of Boolean" types. Shift (left and right) operations are also provided. No operator corresponding to set difference is pre-defined, however.

B11. Power Set Operations (J3B)

Not Satisfied.

J3B does not support the concept of enumeration types nor power sets. It does provide a bit string data type, however, with AND, OR, XOR, NOT, and a bit extraction operator that can be used to test whether a given element in the string is a TRUE or FALSE value [Section 9.2]. An operation corresponding to set difference is not part of the language, however.

B11. Power Set Operations (PASCAL)

Partially Satisfied.

PASCAL has a type "set" which defines "the powerset of its so-called base type" [p. 144]. Enumeration types are subsumed as permissible base types [pp. 144, 142]. Union, set difference, intersection, and a membership predicate are built-in operators applicable to all set types [p. 145]. A complement operator is not included.

B11. Power Set Operations (SIMULA 67)

Not Satisfied.

SIMULA 67 does not offer "data types defined as power sets of enumeration types" since it does not even offer enumeration tyes. For, outside of object reference types, SIMULA 67 has only real, integer, Boolean, array, character, and text types [CBL 3.1, AR 5.1.1].

B11. Power Set Operations (TACPOL)

Not Satisfied.

As mentioned in A5, TACPOL does not support enumeration types, so power sets of such types and the associated set operations are absent from the language. However, TACPOL does have a bit string data type [Section 3-3.b] that can directly represent a set. The logical operators AND, OR, and NOT [Section 6-1.d] then realize union, intersection, and complement respectively. The substring operation SUBSTR [Section 6-1.c.(2)] applied to a bit string suffices as an element predicate. No operation equivalent to set difference is built-in, however.

C1. Side Effects (Due to Evaluation Order) (TINMAN)

Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.

This provides a simple rule (i.e., left-to-right) for order of argument evaluation, but allows implementations to alter the order in any way which does not change the effect.

C1. Side Effects (Due to Evaluation Order) (ALGOL 68)

Not Satisfied.

In Algol 68, operands [Section 5.4.2], arguments [Section 5.4.3], and subscripts [Section 5.3.2] are evaluated collaterally, i.e., the order of evaluation is implementation dependent.

C1. Side Effects (Due to Evaluation Order) (J3B)

Not Satisfied.

J3B explicitly does not require that array indices or operands of expressions be evaluated in left-to-right order [Section 7.1, p. 7-3 and Section 5.2, p. 5-3]. Nor is it required that the target of an assignment statement be evaluated before the value to be assigned is determined.

C1. Side Effects (Due to Evaluation Order) (PASCAL)

Not Satisfied.

In expression evaluation "sequences of operators of the same precedence are executed from left to right" [p. 148], i.e., operand evaluation order is not defined. The evaluation order of subscripts [p. 147] and parameters [pp. 151, 152-153] is not defined, nor is the target of an assignment statement specified to be evaluated before the value to be assigned is determined.

C1. Side Effects (Due to Evaluation Order) (SIMULA 67)

Not Satisfied.

SIMULA 67 does not constrain the order of argument evaluation for an operator. This leaves a major ambiguity about side effects in the language which it inherits from ALGOL 60 since SIMULA 67 subsumes ALGOL 60's syntax and semantics for arithmetic expressions [CBL 4.1.1].

C1. Side Effects (Due to Evaluation Order) (TACPOL)

Not Satisfied.

Argument evaluation order is not specified for TACPOL, although "expressions are evaluated from left to right according to the priorities of the operators" [Section 6-2.a]. The explanatory examples seem to imply a left-to-right operand evaluation order.

C2. Operand Structure (TINMAN)

Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.

Care must be taken to ensure that the operator/operand structure of expressions is not psychologically ambiguous (i.e., to guarantee that the parse implemented by the language is the same as intended by the programmer and understood by those reading the program). This kind of problem can be minimized by having few precedence levels and parsing rules, by allowing explicit parentheses to specify the intended execution order, and by requiring explicit parentheses when the execution order is of significance to the result within the same precedence level (e.g., "X divided by Y divided by Z" and "X divided by Y multiplied by Z"). The user will not be able to define new operator precedence rules nor change the precedence of existing operators.

C2. Operand Structure (ALGOL 68)

Not Satisfied.

Algol 68 has ten precedence levels (monadic plus nine dyadic) [Section 4.3]. The precedence of standard operators is essentially conventional [Section 10.2.3.0]. For expressions involving operators *at the same level*, explicit parenthesis may be used to force the association of operands with operators. In their absence, the default is left to right [Section 5.4.2], e.g. (A+B)+C. The programmer may define (redefine) the precedence of new (existing) operators [Section 4.3].

C2. Operand Structure (J3B)

Partially Satisfied.

Arithmetic operations and logical operations satisfy conventional precedence relationships [Sections 7.1 and 7.3]. Explicit parentheses are permitted to specify the intended grouping of operands with operators. But explicit parentheses are not required when the grouping of operands with operators is of significance to the result within the same precedence level [Section 7.1.1-4].

A programmer cannot redefine the precedence of existing operators, nor can new operator precedence rules be introduced.

C2. Operand Structure (PASCAL)

Partially Satisfied.

PASCAL has only 4 levels of operator precedence [p. 149] - not, multiplication, addition, and relational operators - in contrast to ALGOL 60's 9 levels. In retrospect, Wirth believes that this simplification may break too much from tradition [Wirth, p. 194]. Explicit parentheses are not required within the same precedence level. A programmer cannot redefine the precedence of existing operators, nor can new operator precedence rules be introduced.

C2. Operand Structure (SIMULA 67)

Partially Satisfied.

SIMULA 67 has nine levels of operator hierarchy from ALGOL 60 [AR 3.4.6.1, 3.3.5.1] and it does not require explicit parentheses when the execution order is of significance to the result within the same precedence level because "can" [AR 3.3.5.2] rather than "must" qualifies the use of parentheses in such situations.

C2. Operand Structure (TACPOL)

Partially Satisfied.

TACPOL has a standard operator hierarchy [Section 6-2]. The user cannot define new operator precedence rules nor change the precedence of existing operators. However, TACPOL does not require explicit parentheses when the execution order is of significance to the result within the same precedence level. Here, left-to-right evaluation is the determining rule.

C3. Expressions Permitted (TINMAN)

Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.

FORTRAN, for example, has a list of seven different syntactic forms for subscript expressions, instead of allowing all forms of arithmetic expressions.

C3. Expressions Permitted (ALGOL 68)

Satisfied.

This can be verified by checking the syntax. One of Algol 68's design goals is orthogonality [Section 0.1.2].

C3. Expressions Permitted (J3B)

Satisfied.

In general, wherever a variable is to be evaluated for its value at run time, the syntax of J3B permits substituting a constant or expression. There are three exceptions, however. The INTR, BYTE, and BIT functional modifiers [Section 9.2] accept only variables as their arguments. This violates the spirit, if not the letter of C3, since both variables and constants are not permitted as arguments of these functions.

C3. Expressions Permitted (PASCAL)

Satisfied.

This can be verified directly from the formal specification of the language's syntax [pp. 110-118].

C3. Expressions Permitted (SIMULA 67)

Satisfied.

Examination of the defining syntax [CBL] shows constant to be only instances of expressions; they are not used elsewhere.

C3. Expressions Permitted (TACPOL)

Not Determinable.

The Reference Manual is not sufficiently explicit to evaluate TACPOL fully with respect to C3. Expressions are permitted for initial, increment, and final values of iterative DOs [Section 8-3.d]. They also may appear as procedure VALUE arguments [Section 11-3] and as the RETURN value for function procedures [Section 9-3]. This seems to be where both constants and variable references are permitted.

Exponentiation is restricted to positive integer constants, but since variables and expressions containing variables are excluded, this does not violate the letter of C3, although it perhaps violates its spirit.

The SUBSTR operator is described as requiring a string valued identifier [Section 6-1.c.(2)] as its first argument rather than a constant or expression. This does not violate the letter of C3, though it does violate its intent.

C4. Constant Expressions (TINMAN)

Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.

The resulting value should be the same (at least within the stated precision) regardless of the object machine (see D2). Allowing constant expressions in place of constants can improve the clarity, correctness, and maintainability of programs and does not impose any run time costs.

C4. Constant Expressions (ALGOL 68)

Not Satisfied.

Algol 68 allows constant expressions wherever constants are allowed, except in priority declarations [Section 4.3]. However, it does not require them to be evaluated at compile time.

C4. Constant Expressions (J3B)

Partially Satisfied.

The J3B language specification explicitly describes which expressions consisting entirely of constant values will be evaluated at compile time [Section 8]. However, function calls whose arguments are all constant, the absolute value function, the INTR, BYTE, BIT, SHIFTR, and SHIFTL functional modifiers, exponentiation, and relational formulas are never evaluated at compile time. In addition, expressions of the form TRUE OR VAR or FALSE AND VAR (where VAR is a variable or expression) are not evaluated at compile time. Of these restrictions, the most significant is the lack of compile time evaluation of relational expressions, since this limits J3B's conditional compilation capability.

Numeric expressions are specified to be evaluated with host rather than object machine range and precision [Section 8.1.2 and 8.1.3].

C4. Constant Expressions (PASCAL)

Not Satisfied.

PASCAL does not permit constant expressions in the following four contexts:

(1) constant definition part

CONST <identifier> = <constant>; [pp. 110, 118, 159]

(2) CASE labels [pp. 114, 112, 118]

(3) Defining subrange types (i.e., the lower and upper bounds for possible scalar values) <constant> .. <constant> e.g. pp. 111, 116, 143)

(4) GOTO <label> where label is an unsigned integer constant [pp. 110, 153].

C4. Constant Expressions (SIMULA 67)

Not Satisfied.

Constant expressions are allowed anywhere constants are allowed but are not required to be "evaluated before run time".

C4. Constant Expressions (TACPOL)

Not Satisfied.

Some contexts are apparently restricted to constants (rather than constant expressions). For example, all illustrations of declaring array upper bounds [Section 4-2] use only integer constants, although no prohibition of constant expressions is actually stated. In addition, precision and scale specifications must be "numbers" [Section 4-1.a]. Program constants declared in "Value declarations" [Section 4-9] require literal constants in their associated INIT attribute where their values are given.

C5. Consistent Parameter Rules (TINMAN)

There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types, for exception handing, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters.

C5. Consistent Parameter Rules (ALGOL 68)

Partially Satisfied.

Algol 68 does not have exception conditions. Its parallel processes are parameterless, as are its data types. The parameter rules are consistent for operators and procedures [Section 5.4.2, 5.4.3]. However, for arrays there is a special subscripting operation for selecting subarrays [Section 5.3.2], e.g., A[2:4, 3:5].

C5. Consistent Parameter Rules (J3B)

Partially Satisfied.

Although J3B distinguishes read-only from writable parameters in the form of a subroutine call [Section 5.3], the specified syntax is such that array and table indexing is consistent with the form of a subroutine call, since the indices are not writable "parameters". Moreover, consistency is further supported in that functions are permitted to have writable parameters [Section 3.3.3]. However, the use of a colon to separate read-only from writable parameters is not consistent with its use to separate lower and upper bounds of table indices in declarations [Section 4.3] nor is it consistent with the use of POINT (table name, from : to) in table initialization [Section 4.9].

The declaration of subroutines (as opposed to their definition) permits a special type to be specified as a formal parameter to indicate that an integer, bit, or character string argument of any size is deemed to match the corresponding formal parameter. For example,

```
PROC F(S);
```

declares F to be a procedure accepting integer-valued arguments of any size, even though when defining F,

```
PROC F(II); BEGIN
   ITEM II S 31;
```

the formal parameter II must be declared to be of a specific size [Section 4.8].

J3B has no exception handling capability, parallel process control operations, or parameterized type declarations, so C5 is vacuously satisfied for these capabilities.

The BIT and BYTE pseudo variable syntax, however, forms an exception to the parameter notational conventions. In the assignment statement:

```
BIT(STRING, II, 1) = BIT.VAR;
```

the value of the variable "STRING" is modified. To be consistent with the parameter notations elsewhere in the language, the syntax of the BIT and BYTE pseudo variables should require a colon before the first argument, to show that STRING (at least) is a writable parameter.

Another exception to parameterization rules is that ordinary tables that are not declared as "typed tables" may not be passed as parameters (p. 5-12). This is inconsistent with the general ability to pass tables as parameters. However, the provision of untyped ordinary tables conflicts with other TINMAN requirements, so this inconsistency vanishes if J3B is brought into conformity with other TINMAN requirements.

The form and interpretation of actual macro parameters differs from that of subroutine actual parameters (compare Sections 2.2.6 and 3.4 with 3.3.2, 3.3.3, and 5.3). In addition, formal macro parameters are not declared to be of any particular data type.

C5. Consistent Parameter Rules (PASCAL)

Partially Satisfied.

PASCAL does not permit parameterization of type declaration and use; it does not provide for exception handling or parallel processing. The language is consistent in its parameter rules for procedures and functions [pp. 158, 162] in that functions may have output parameters. However, "procedures and functions which are used as parameters to other procedures and functions must have value parameters only" [p. 158].

The standard arithmetic functions abs and sqr are generic [p. 163] (the type of the argument determines the type of result), but a programmer cannot define generic subroutines.

Input and output procedures for textfiles, use "a non-standard syntax for their parameter lists, allowing, among other things, for a variable number of parameters" [p. 164] and for groups of parameters separated by commas. For addressing efficiency reasons the restriction that "components of a packed structure must not appear as actual variable parameters" [p. 53] is imposed.

The form of declaration used for subroutines passed as parameters is the same as the form used when defining a subroutine, i.e., in both cases the parameter types are specified by declaring the type of an identifier serving as the formal parameter [pp. 158, 162].

C5. Consistent Parameter Rules (SIMULA 67)

Partially Satisfied.

SIMULA 67 has a consistent set of rules applicable to parameters and their transmission. However, only a subset of the procedure parameter transmission capabilities are available for class (i.e., type) parameters [CBL 8.2]. Specifically call-by-name mode is not a class parameter option. Further, function procedures, labels, and switches cannot be class parameters but can be procedure parameters [CBL 8.2].

C5. Consistent Parameter Rules (TACPOL)

Not Satisfied.

Procedures and functions satisfy the same parameter transmission rules, including the ability to assign to parameters of functions. Built-in functions have some capabilities denied user-defined functions, specifically, some arguments are optional and some functions, e.g. ABS, are generic. Subroutines passed as parameters may not take any arguments.

C6. Type Agreement in Parameters (TINMAN)

Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

Some notations permit such parameters to be implicit on the call side. Formal parameters of a union type will be considered conformable to actual parameters of any of the component types.

C6. Type Agreement in Parameters (ALGOL 68)

Partially Satisfied.

Algol 68 satisfies all the requirements except that it allows an integer actual parameter to be implicitly converted (widened) [Section 6.5] to real when the formal parameter is real [Section 5.4.3]. Implicit conversions between reals of different precisions, e.g., real and long real, are not allowed, however.

C6. Type Agreement in Parameters (J3B)

Partially Satisfied.

Integer-valued formal parameters can be specified to match actual parameters exactly with respect to the parameter's size declaration (where the "size" specifies the number of bits permitted to hold valid data values) or to ignore the size attribute of actual parameters [Section 4.8 and 5.3.3]. The actual parameter must be integer-valued in all cases, however.

Floating point formal parameters must be matched with actual floating point parameters of the same precision [Section 5.3.3]. Fixed point formal parameters similarly must be matched with actual fixed point parameters of the same scale [Section 5.3.3]. (All fixed point values have the same number of bits of precision.)

Character and bit formal parameters can be specified to match actual parameters exactly with respect to the parameter's length or to ignore the length attribute of actual parameters [Section 5.3.3]. No implicit conversion from character to bit or bit to character values is permitted.

Array and table formal and actual parameters must have the same number of dimensions but, in violation of C6, the subscript range for these parameters is fixed at compile time [Section 4.8].

J3B's type matching constraints for J3B's "typed" tables in effect satisfy C6's constraints on formal parameters of a union type, since the J3B constraints ensure that every record field accesssible through a formal parameter is supported by the structure of the actual parameter.

C6's constraints are violated, however, by J3B's "specified" tables, since when a specified table serves as a formal parameter, the only constraint an actual parameter must satisfy is that the actual parameter be a table with the same number of words per entry as the formal parameter, the same number of dimensions, and the same subscript range (if subscripts are permitted). In particular, the component types of formal and actual parameters need not match [Section 5.3.3].

C6 is also violated by pointer parameters, which may be declared untyped as formal parameters [Section 4.8], meaning that no type check on actual parameters is performed [Section 5.3.3]. Typed formal pointer parameters, however, are checked in a manner that satisfies C6 [Section 5.3.3]. i.e., the fields of a record accessible through a typed formal pointer parameter must be supported by the record accessible through the actual pointer parameter.

C6. Type Agreement in Parameters (PASCAL)

Partially Satisfied.

Though not stated explicitly in the PASCAL User Manual and Report, formal and actual parameters must agree in type, where a formal of a union type is

conformable to an actual of any component type as in assignment. This inference is justified both by current PASCAL implementations and the discussion of formal/actual correspondence [711-72 152-253]. In PASCAL, not only the number of dimensions but also the size and subscript range for array parameters is determined at compile-time. Since formal procedure parameters must specify their type for exact matching, and since the size is part of the type of an array parameter, "a given procedure can only be applied to arrays of one fixed size" [Wirth, p. 194].

For procedures and functions declared as formal parameters, no method is provided for declaring the type and number of their parameters; consequently no type checking is possible when such subroutines are called [p. 79].

C6. Type Agreement in Parameters (SIMULA 67)

Partially Satisfied.

Formals and actuals must "be of compatible types" [B 103]. This means they agree in type or

(1) the actual is a subclass of the formal (i.e., a component type of the formal's union type), or

(2) the formal is a subclass of the actual and the current actual's value is in that subclass (i.e., the actual is of a union type and its current value is "compatible" with the formal's type). Obviously "this must be checked at run time" [B 151] [CBL 8.2, 6.1.2.2]

SIMULA 67 fully satisfies C6's requirements for array parameters. That is, the number of dimensions is compile time determinable and the subscript ranges must be passed as part of the procedure arguments [B 101-105]. Type agreement for procedures passed as parameters means only that the formal and the actual are both procedures [AR 5.4.1], i.e., no type checking is prescribed on the number and type of arguments permitted. Type agreement on functions passed as parameters requires only that "the type associated with the actual parameter must coincide with or be subordinate to that of the formal specification" [CBL 8.2.2].

C6. Type Agreement in Parameters (TACPOL)

Satisfied.

"The parameter attributes must correspond to the attributes of the argument by type ... but not necessarily by size or scale" [Section 11-3]. It is implied that array formals and actuals must match in subscript range [Section 11-2].

Since subroutines passed as parameters cannot take any arguments, no type checking problems arise for such parameters.

C7. Formal Parameter Kinds (TINMAN)

There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.

The first class of data parameter acts as a constant within the procedure body and cannot be assigned to nor changed during the procedure's execution; its corresponding actual parameter may be any legal expression of the desired type and will be evaluated once at the time of call. The second class of data parameter renames the actual parameter which must be a variable. The address of the actual parameter variable will be determined by (or at) the time of call and unalterable during execution of the procedure, and assignment (or reference) to the formal parameter name will assign (or access) the variable which is the actual parameter. A language with exception handling capability must have a way to pass control and related data through procedure call interfaces. Exception handling control parameters will be specified on the call side only when needed. Actual procedure parameters will be restricted to those of similar (explicit or implicit) specification parts.

C7. Formal Parameter Kinds (ALGOL 68)

Partially Satisfied.

Algol 68 has the (non-assignable) constant and variable renaming kinds of parameters, along with procedure parameters [Section 5.4.3, 4.4, 4.6]. However, it does not support exception conditions per se.

C7. Formal Parameter Kinds (J3B)

Partially Satisfied.

J3B satisfies C7 in that it supports read-only input parameters (called by value) and writable output parameters (called by reference) [Section 5.3.3]. Input parameters (including components of tables or arrays passed as input parameters) cannot be assigned to nor can they be used as output parameters in a subroutine call (p. 5-7 and p. 5-14).

C7 is not satisfied, however, in that subroutines cannot be passed as parameters [Section 4-8], nor is there any provision for exception handling. Labels cannot be passed as parameters [Section 4-8].

C7. Formal Parameter Kinds (PASCAL)

Not Satisfied.

PASCAL has essentially three classes of formal parameters: value, variable, and function or procedure parameters [pp. 71, 152]. The variable and procedure parameters satisfy the C7 requirements but there is no formal parameter class for specifying the control action when exception conditions occur. Also, value parameters receive their initial value from the actual in the call, but "the procedure may then change the value of this variable by assigning to it" [p. 72] since it is, in effect, "a local variable of the called procedure" [p. 153]. This violates the TINMAN read-only stipulation that a value parameter acts as a constant within the procedure body and cannot be assigned to nor changed during the procedure's execution.

C7. Formal Parameter Kinds (SIMULA 67)

Not Satisfied.

In SIMULA 67 there is no "formal parameter class for specifying the control action when exception conditions occur" although a label or a switch may be passed as a parameter [CBL 8.2] for control transfer upon program discovery of an exception condition. There are three parameter transmission modes: by value, by reference, by name. The "by name" mode is excluded by C7 and the "by value" mode for SIMULA 67 is not sufficiently restrictive since a by-value actual parameter can be assigned to and "changed" during the procedure's execution". Procedures can be passed as parameters [CBL 8.2], but no type checking is possible when the procedures are invoked.

C7. Formal Parameter Kinds (TACPOL)

Partially Satisfied.

TACPOL has parameter call by reference [Section 11-2] and parameter call by value [Section 11-3], along with procedure and label parameters [Section 11-4, 11-5]. "Any changes to a value parameter in the invoked procedure do not affect the value in the invoking procedure" [Section 11-3]. This violates C7's requirement that value parameters are read only. Label parameters [Section 11-5] are excluded by C7. TACPOL does not support exception handling control parameters.

C8. Formal Parameter Specifications (TINMAN)

> Specification of the type, range, precision, dimension, scale and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time.

Optional formal parameter specification permits the writing of generic procedures which are instantiated at compile time by the characteristics of their actual parameters. It eliminates the need for compile time "type" parameters.

C8. Formal Parameter Specifications (ALGOL 68)

Not Satisfied.

The types of parameters must be specified for procedure declarations [Section 5.4.1] in Algol 68. However, operators are generic in the sense that one can have different routines associated with an operator token, depending on the operand types, and these routines may be programmer defined. But this limits ALGOL 68's generic routine capability to routines of one or two arguments invoked syntactically as operators.

C8. Formal Parameter Specifications (J3B)

Not Satisfied.

J3B does not support the concept of generic procedures.

C8. Formal Parameter Specifications (PASCAL)

Not Satisfied.

In PASCAL "formal procedure parameters specify their types" [Wirth, p. 194]. The specification is not optional. As a consequence, generic procedures cannot be written in PASCAL. Admittedly this "seriously impairs the highly desirable flexibility of procedures" [Wirth, p. 194].

C8. Formal Parameter Specification (SIMULA 67)

Partially Satisfied.

"Specification is required for each formal parameter" [CBL 8.2] where specification "gives the type of each formal parameter" [B 103]. Procedures that are generic in more than one argument cannot be written. The virtual concept, however, provides a form of generic procedure [CBL 2.2.3, B 154-161] wherein a procedure identifier can be associated with procedure bodies belonging to different type definitions.

C8. Formal Parameter Specifications (TACPOL)

Not Satisfied.

TACPOL requires specification of all parameter types and dimensions [Section 11-2, 11-3]. Generic procedures cannot be written.

C9. Variable Numbers of Parameters (TINMAN)

There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

There are many useful purposes for procedures with variable numbers of arguments. These include intrinsic functions such as "print," generalizations of operations which are both commutative and associative such as "max" and "min," and repetitive application of the same binary operation such as the Lisp "list" operation.

C9. Variable Numbers of Parameters (ALGOL 68)

Satisfied.

In Algol 68, the number of parameters is fixed [Section 5.4.3]. However, an actual parameter may be a flexible array. The Algol 68 row display [Section 3.3] allows one to explicitly specify the elements of an array, e.g. (1, 2, 3) is a three element array of integers. Thus, a variable number of parameters is obtained by using a row display as an array parameter, e.g. get (f, (X, Y, Z)) reads values for variables X, Y, Z from file f.

C9. Variable Numbers of Parameters (J3B)

Not Satisfied.

J3B supports only a fixed number of parameters.

C9. Variable Numbers of Parameters (PASCAL)

Not Satisfied.

In PASCAL there must be an exact match both in number and in type between actual and formal parameters. "Upon an activation of the procedure statement, an actual quantity has to be indicated for each parameter which can be referenced from within the procedure through the formal parameter" [p. 13)]. Only the standard input and output procedures read, write, readln, writeln for textfiles allow a variable number of parameters [p. 164-166]. And this variable number is in each case compile-time determinable from the individual call statements.

C9. Variable Number of Parameters (SIMULA 67)

Not Satisfied.

The actual parameter list in a procedure call "must correspond to the formal parameter list in number, order and be of compatible types" [B 103]. In fact, even the standard input and output procedures do not permit a variable number of arguments [CBL 11].

C9. Variable Number of Parameters (TACPOL)

Not Satisfied.

TACPOL does not support variable numbers of procedure parameters. "The argument list appearing in the invoking statement or expression must have the same number of arguments as there are identifiers in the parameter list of the invoked procedure" [Section 11-1.a].

D1. Constant Value Identifiers (TINMAN)

The user will have the ability to associate constant values of any type with identifiers.

D1. Constant Value Identifiers (ALGOL 68)

Satisfied.

Identifiers are declared as constants in Algol 68 by identity declarations [Section 4.4], e.g., real pi = 3.14159. Algol 68 also permits the run time initialization of a constant identifier, i.e., such a "constant" may be given a different value (by an identity declaration) on each entry to the scope in which it is declared, but it may not otherwise be assigned to within its scope. In effect, such an identifier is a "read-only" variable.

D1. Constant Value Identifiers (J3B)

Satisfied.

Constant values of any scalar data type can be given a mnemonic name [Section 4.7]. Such names may not be assigned to [Section 5.2]. The value of such constants must be known at compile time.

D1. Constant Value Identifiers (PASCAL)

Satisfied.

Each procedure may have a constant definition part that "contains all constant synonym definitions local to the procedure" [p. 159]. The value of the constant must be known at compile time (unlike ALGOL 68, for example).

D1. Constant Value Identifiers (SIMULA 67)

Not Satisfied.

SIMULA 67 does not have a facility for letting identifiers represent constants.

D1. Constant Value Identifiers (TACPOL)

Satisfied.

TACPOL supports constant value identifiers. They are declared in value declarations using the particle INIT. "Once a name has been declared in a value declaration, the value assigned that name may never be changed" [Section 4-9]. The value of such constants is determined at compile time.

D2. Literals (TINMAN)

The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output).

Literals are needed for all atomic data types and should be provided as part of the language definition for built-in types.

D2. Literals (ALGOL 68)

Partially Satisfied.

Algol 68 denotations provide constants for the primitive types: integer, real, Boolean, and character [Section 8.1]. Row and structure displays [Section 3.3] provide array and record constants. Routine texts [Section 5.4.1] provide procedure constants. However, pointer constants other than nil depend on the compiler's ability to detect constant addresses. Algol 68 does not satisfy the requirements because it does not require real program constants to yield the same internal representation as real I/O data.

D2. Literals (J3B)

Partially Satisfied.

Integer, floating, fixed, character, and bit literals are defined in J3B [Section 2.2.3]. The value NULL is defined as a pointer constant [Section 8.4]. So all atomic data types have literals defined for them.

J3B provides no capability for converting between numeric values and their printable representation, and so does not address the issue of ensuring that program and data literals are converted identically.

D2. Literals (PASCAL)

Partially Satisfied.

If v is an integer or real variable, then read (v) invokes reading from the standard input file "of a sequence of characters which form a number according to the syntax of PASCAL and the assignment of that number to v" [p. 165]. No constraints are imposed to ensure the program and data literals are converted identically. Similar remarks apply if v is of type char. Boolean, real, integer, char and even string constants can be written directly [p. 166]. However, no provision is made for the input (via the standard procedure read) of Boolean constants [pp. 84-85, 165]. No constraints are imposed to ensure that program and data literals are converted identically.

D2. Literals (SIMULA 67)

Partially Satisfied.

Boolean constant (FALSE and TRUE) and the no-object reference constant NONE cannot be read as input constants or written except as TEXT values, although they can appear as constants within the program code. For example, only character, text, integer, fixed point, or real constants may be read as input [CBL 11.2]. Further, the programmer cannot specify or control the accuracy of conversion of literals from external to internal form.

D2. Literals (TACPOL)

Partially Satisfied.

TACPOL provides a straightforward syntax for numeric and string constants [Section 3-5]. The reference manual does not detail the input representation for constants [Section 13-6], but one wonders if long numeric inputs have a final L as do long numeric program literals [Section 3-5.6]. No restriction is stated on conversion accuracy of program and data literals.

D3. Initial Values of Variables (TINMAN)

The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

There will be provision (at user option) for run time testing for initialization.

D3. Initial Values of Variables (ALGOL 68)

Partially Satisfied.

Initial values are optional in Algol 68 variable declarations [Section 4.4], e.g.

```
int i;
real x := 0;
```

There are no default initial values. Algol 68 declarations are executed in sequence [Section 3.2], so that in

```
real X :=  Y;
real Y := 3.5;
```

the value of Y is not defined when X is initialized, even though Y is accessible. Algol 68 does not define any run time test for uninitialized variables.

D3. Initial Values of Variables (J3B)

Partially Satisfied.

Programmer-defined initialization is supported only for data allocated prior to program execution [Sections 4.4, 4.5, and 4.9]. If such data are not explicitly initialized, a default initial value represented by all zero bits is provided [Sections 6, p. 6-3], even if such a value is not legally permitted according to a variable's type [e.g., see Section 8.4].

It is not possible to initialize variables declared local to a subroutine [Sections 4.3, 4.4, and 4.5], and typed tables can only be initialized if they are declared in a block [Sections 4.3 and 4.1].

There is no support provided in either the language or the implementations for run time testing of access to uninitialized variables.

D3. Initial Values of Variables (PASCAL)

Not Satisfied.

"Variable declarations consist of a list of identifiers denoting the new variables, followed by their type" [p. 146]. No mechanism or syntax for initialization in the declaration part is provided. Consequently for locally declared variables "their values are undefined at the beginning of the statement part" [p. 159].

D3. Initial Values for Variables (SIMULA 67)

Not Satisfied.

In SIMULA 67 "any declared variable is initialized at the time of entry into the block to which the variable is local. The initial contents depends on the type of the variable" [CBL 3.2.4]. Default initial values violate requirement D3, and no programmer-defined initial values can be specified in declarations.

D3. Initial Values of Variables (TACPOL)

Not Satisfied.

The TACPOL user cannot initialize variables as part of their declaration.

D4. Numerical Range and Step Size (TINMAN)

The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

Range specifications offer the opportunity for the translator to insert range tests automatically for run time or debug time validation of the program logic. With the ranges of variables specified in the program, it becomes possible to perform many subscript bounds checks at compile time. These bounds checks, however, can be only as valid as the range specifications which cannot in general be validated at compile time.

D4. Numeric Range and Step Size (ALGOL 68)

Not Satisfied.

Algol 68 has neither subranges nor fixed point type. The range of real, int, long real etc. is implementation dependent [Section 2.1.3.1].

D4. Numeric Range and Step Size (J3B)

Not Satisfied.

The range of integer data can only be specified approximately by declaring a variable to be a signed or unsigned integer occupying a specific number of bits [Section 1.3.3.1 and 4.5]. No range specifications are possible for fixed or floating point data [Section 4.5].

Fixed point data is not considered exact, and so there is no provision for specifying a fixed point "step size" [Section 1.3.3.1].

No subscript range exception condition is supported by the language, although a compile time check for subscript range errors is performed when constant subscript values are used [Section 6].

D4. Numerical Range and Step Size (PASCAL)

Not Satisfied.

PASCAL does not have fixed point variables (see A2). The range of numeric variables - integer and real - plus the range of other scalar varables (e.g., character and enumeration) may be specified using subrange types. However, this is not required and "subrange" is regarded as a type in the language [p. 143].

D4. Numeric Range and Step Size (SIMULA 67)

Not Satisfied.

Individual ranges for numeric variables cannot be specified in SIMULA 67. Rather, a numeric variable can have either integer whole number range or real number range (which are "implementation defined") [B 72, B 375]. Fixed point variables, and hence their step sizes, are not available.

D4. Numeric Range and Step Size (TACPOL)

Partially Satisfied.

TACPOL only supports fixed point numerics [Section 3-2]. Their precision and scale must be declared [Section 4-1]; in effect, this gives a numeric variable's range, although ranges are limited to powers of two, as are step sizes.

D5. Variable Types (TINMAN)

The range of values which can be associated with a variable, array, or record component will be any built-in type, any defined type, or a contiguous subsequence of any enumeration type.

This permits arrays to be components of records or arrays and permits records to be components of arrays.

D5. Variable Types (ALGOL 68)

Partially Satisfied.

Algol 68 allows variables, array elements, structure fields (record components), procedure parameters and results, and pointed-to values to be of any type [Section 1.2.1, 4.4, 4.6]. However, the language does not have enumeration types.

D5. Variable Types (J3B)

Partially Satisfied.

A variable, array, or record component can be defined to be an built-in data type [Sections 4.2, 4.3, 4.4, 4.5, and 4.8]. Tables (i.e., arrays of records) are, however, limited to a single dimension, whereas arrays can have up to three dimensions, so arrays of records are not supported in the full generality implied by D5. In addition, arrays are not permitted as components of records (although parallel tables gives limited support to this concept).

D5. Variable Types (PASCAL)

Satisfied.

The component type of an array [p. 143], the type of a record section [p. 144], and the type of a variable [p. 146] can be any type in the language. This includes the built-in, the programmer defined, and the subrange types. For the implemented version PASCAL 6000 "it is not possible to construct a file of files; however, records and arrays with files as components are allowed" [p. 97].

D5. Variable Types (SIMULA 67)

Partially Satisfied.

Object attributes (i.e., record components) can be any built-in or defined type (or class) [CBL 2.2]. Array components are either value type (i.e., built-in) or reference type (i.e., pointer to defined type) [CBL 3.1], i.e., arrays of records are not supported. As remarked under A5 and B11, SIMULA 67 does not support enumeration types.

D5. Variable Types (TACPOL)

Partially Satisfied.

Arrays are restricted to components of scalar type [Section 4-2]. Groups may contain only elementary components or arrays [Section 4-5.a]. Tables are just one dimensional arrays of groups [Section 4-6]. Thus, TACPOL's array and

record features are severely restricted as to the structure of their components.

D6. Pointer Variables (TINMAN)

The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

Assignment to a pointer variable will mean that the variable's name is to act as an additional label (or reference) on the datum being assigned. Assignment to a nonpointer variable will mean that the variable's name it to label a copy of the object being assigned. For data without alterable component structure or alterable component values, there is no functional difference between reference to multiple copies and multiple references to a single copy. Consequently, pointer/nonpointer will be a property only of variables for composite types and of composite array and record components. Because the pointer/nonpointer property applies to all variables of a given type, it will be specified as part of the type definition. The use of pointers will be kept safe by prohibiting pointers to data structures whose allocation scope is narrower than that of the pointer variable.

Such a restriction is easily enforced at compile time using hierarchical scope rules providing there is no way to dynamically create new instances of the data type. In the latter case, the dynamically created data can be allocated with full safety using a (user or library defined) space pool which is either local (i.e., own) or global to the type definition. If variables of a type do not have the pointer property then dynamic storage allocation would be required for assignment unless their size is constant and known at the time of variable allocation. Thus, the nonpointer property will be permitted only for types (a) whose data have a structure and size which is constant in the type definition or (b) which manage the storage for their data as part of the type definition. Because pointers are often less expensive at run time than nonpointers and are subject to fewer restrictions, the specification of the nonpointer property will be explicit in programs (this is similar to the Algol-60 issue concerning the explicit specification of "value" (i.e., nonpointer) and "name" (i.e. pointer). The need for pointers is obvious in building data structures with shared or recursive substructures; such as, directed graphs, stacks, queues, and list structures. Providing pointers as absolute address types, however, produces gaps in the type checking and scope mechanisms. Type and access restricted pointers will provide the power of general pointers without their undesirable characteristics.

D6. Pointer Variables (ALGOL 68)

Satisfied.

Algol 68 provides pointer types, variables, array and record components [Section 1.2.1, 4.4, 4.6]. It permits definition of recursive, sharable, and dynamically alterable data structures (and substructures). The type of a pointer includes the type of the value pointed to, so pointers are compile time type restricted. The language prohibits assigning pointers to variables whose lifetime is longer than the object pointed to [Section 5.2.1].

D6. Pointer Variables (J3B)

Partially Satisfied.

J3B provides fully type-checked pointers. No data value can be accessed through a pointer without qualifying the pointer value with a data type, either in the pointer's declaration or at its point of use [Section 6]. J3B pointers can only be associated with records and arrays of records, i.e., tables, (in which case the pointer values can be associated with each record of the array) [Section 1.3.3.1, p. 1-16]. Although J3B supports pointer arithmetic and pointer comparisons [Sections 7.3 and 9.7] these operations are type-constrained to guard against error. The language does not, however, support any restriction against assigning a pointer value to a variable whose storage allocation lifetime may be longer than that of the object pointed to.

D6. Pointer Variables (PASCAL)

Not Satisfied.

PASCAL has a pointer type for building shared and recursive data structures [p. 145]. Assignment to a pointer variable is very restrictive in that only a pointer of the exact type may be assigned [p. 152]. The standard procedures new and dispose for explicit dynamic allocation mean pointer variables are not as safe in their use as are any other variables, since it is possible to deallocate record storage that is referenced by a still accessible variable.

D6. Pointer Variables (SIMULA 67)

Satisfied.

There is no implicit deferencing in SIMULA 67. In general, "the type of the value or reference obtained by evaluating the right part, must coincide with the type of the left part" (CB1 6.1.2). However, SIMULA 67 does have an extremely safe and flexible pointer mechanism because "an object-reference-assignment is always checked for legality (mainly at compile time)" [B 152]. This guarantees that a pointer is either NONE or references an object of the specified class.

D6. Pointer Variables (TACPOL)

Not Satisfied.

TACPOL does not have a pointer mechanism.

E1. User Definitions Possible (TINMAN)

The user of the language will be able to define new data types and operations within programs.

Definitions will have the appearance and costs of features which are built into the language while actually being catalogued accessible application packages. The operation definition facility will include the ability to define new infix operators (but see H2 for restrictions).

E1. User Definitions Possible (ALGOL 68)

Satisfied.

Algol 68 allows the programmer to define new data types [Section 4.2], new operators [Section 4.3, 4.5], and new procedures [Section 5.4.1].

E1. User Definitions Possible (J3B)

Not Satisfied.

Although record data can be defined as a data type, J3B provides no capability for defining new infix and prefix operators, nor for extending existing operations to new data types (as required by E4). In addition, programmer-defined functions may not return arrays or records [Section 3.3.3].

E1. User Definition Possible (PASCAL)

Partially Satisfied.

PASCAL provides an elegant "mechanism, type definition, for creating new types" [p. 18]. However, the only operations definition facility is the procedure or function. In PASCAL there is no ability to define new infix operators nor may programmer-defined functions return arrays or records.

E1. User Definitions Possible (SIMULA 67)

Partially Satisfied.

The SIMULA 67 class concept permits user definition of new types together with operations upon the data [CBL 2]. In a strict sense there is no ability

to define new infix operators. But one can write expressions of the form RAND1.OPERATOR(RAND2) where OPERATOR is a procedure defined with the class containing RAND1 as an object and applicable to objects of the class to which RAND2 belongs [B 128-134]. Effectively this is an infix operator definition capability.

E1. User Definitions Possible (TACPOL)

Not Satisfied.

TACPOL has no type or operator definition facility.

E2. Consistent Use of Types (TINMAN)

The "use" of defined types will be indistinguishable from built-in types.

Whether a type is built-in or defined within the base will not be determinable from its syntactic and semantic properties. There will be no ad hoc special cases nor inconsistent rules to interfere with and complicate learning, using, and implementing the language. Type definitions should be processed entirely at compile time and the language should allow full program specification of the internal representation.

E2. Consistent Use of Types (ALGOL 68)

Satisfied.

Algol 68 does not distinguish the use of newly defined types from built-in types. This is part of its orthogonal design [Section 0.1.2].

E2. Consistent Use of Types (J3B)

Not Satisfied.

User-defined operator extensions are not supported by J3B. In addition, arrays of records defined as a new J3B data type can only be accessed via pointer variables; normal subscripting is not allowed [Sections 4.2, 4.3, 4.4, and 6], nor can these types be returned as values of functions. Furthermore, matching rules are different for parameters declared with user-defined type names [Section 5.3.3].

E2. Consistent Use of Types (PASCAL)

Not Satisfied.

Type definitions are processed at compile time but PASCAL, by deliberate design, does not allow full program specification of the internal representation. Also, the language distinguishes between structured and non-structured types (whether built-in or user defined) in that a function's result type "must be a scalar, subrange, or a pointer type" [p. 162]. Thus, none of the structured types - array, record, set, or file - can be returned as a function value.

E2. Consistent Use of Types (SIMULA 67)

Partially Satisfied.

In fact, SIMULA 67 is extended from a common base language to a simulation language by providing the system classes (i.e., built-in types) SIMSET and SIMULATION exactly as user defined types available for prefixing. However, in contrast to user defined classes, the classes SIMSET and SIMULATION are distinct in that "an implementation may restrict the number of block levels at which such prefixes or block prefixes may occur in any one program" [CBL 14]. This is a small but real distinction between two built-in types and user defined types. Secondly I/O facilities are defined using a class called BASICIO, but this "class is not explicitly available in any user's program" [CBL 11]. User defined types cannot be hidden in this manner.

E2. Consistent Use of Types (TACPOL)

Not Satisfied.

As noted in E1, the user cannot define new types.

E3. No Default Declarations (TINMAN)

Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

This is a special case of requirement I1.

E3. No Default Declarations (ALGOL 68)

Satisfied.

Algol 68 has no default declarations. The two level syntax of the Revised Report carries the list of defined identifiers, modes, operators, etc. and thus allows only the defined ones to be applied (used) [Section 4.8, 7.2].

E3. No Default Declarations (J3B)

Satisfied.

All variables, including loop control variables, must be explicitly declared. There are no rules for implicitly defining undeclared identifiers [Section 4].

E3. No Default Declarations (PASCAL)

Satisfied.

"Every variable occurring in a statement must be introduced by a variable declaration" [p. 137]. Some types, procedures, and functions may be predefined (e.g., the standard input and output files - p. 164).

E3. No Default Declarations (SIMULA 67)

Partially Satisfied.

Explicit declarations are required for all program components except, for arrays where "if the type is omitted, the array is assumed REAL by default" [B 371].

E3. No Default Declarations (TACPOL)

Partially Satisfied.

"All names must be declared" [Section 10-1]. Specifically, names are declared in data or procedure declarations. Labels are declared by appearance and "file declarations may not appear outside the Compool" [Section 14-1]. Examination of actual TACPOL shows that loop control variables are implicitly declared to be FIXED (15), and are local to the loop in which they are used.

E4. Can Extend Existing Operators (TINMAN)

The user will be able, within the source language, to extend existing operators to new data types.

The translator will not assume that commutativity of built-in operations is preserved by extensions, and any assumptions about the associativity of built-in or extended operations will be ignored by the translator when explicit parentheses are provided in an expression.

E4. Can Extend Existing Operators (ALGOL 68)

Partially Satisfied.

In Algol 68, the programmer can extend existing operators to new types [Section 4.5]. However, a new type is represented in terms of the base types and the base type properties are inherited by the new type [Section 2.1.1.2, 7.3]. Algol 68 does not have generic procedures although it does support generic user defined operators.

E4. Can Extend Existing Operators (J3B)

Not Satisfied.

User-defined operator extensions are not supported by J3B.

E4. Can Extend Existing Operators (PASCAL)

Not Satisfied.

The permissible operand types for existing operators are completely circumscribed and are not extendable to new data types [pp. 149-151].

E4. Can Extend Existing Operators (SIMULA 67)

Not Satisfied.

Existing operator symbols cannot have their domain of operands extended. For example, "the constituents of simple arithmetic expressions must be of types real or integer" [AR 3.3.4].

E4. Can Extend Existing Operators (TACPOL)

Not Satisfied.

The TACPOL user cannot extend the domain of an existing operator symbol.

E5. Type Definitions (TINMAN)

> Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

Types define abstract data objects with special properties. The definable operations will include constructors, selectors, predicates, and type conversions.

E5. Type Definitions (ALGOL 68)

Not Satisfied.

Algol 68 defined types inherit the properties and applicable operators of the component base types [Section 2.1.1.2, 7.3]. For example, if

mode matrix = [1:m, 1:m] real

and if multiplication '*' is already defined to be component-wise for two-dimensional arrays of real, matrix inherits the component-wise definitions of '*'. The syntax does not require a type's associated operator definitions to be lexically grouped with the type definition.

E5. Type Definitions (J3B)

Partially Satisfied.

J3B does not enforce the clustering of data type definitions with corresponding operations on these data types, although since a compilation unit can consist of several function definitions, user-defined operations on a user-defined data type can be grouped together [Section 3.1].

The rules for assignment and parameter passing of user-defined types require matching type names (rather than type representations) [Section 5.3.3], so different types having the same representation cannot be treated interchangeably by a set of operators. On the other hand, if a record type, D, is defined by adding new fields to a record type C, functions accepting objects of type C as arguments will also accept objects of type D. In this sense, a defined type inherits operations of the data with which it is represented. This kind of "inheritance" is also supported by SIMULA 67.

E5. Type Definitions (PASCAL)

Partially Satisfied.

A type definition in PASCAL has the form

<identifiers> = <type>

where a type is either a simple, a structured, or a pointer type [p. 111]. The set of applicable operators cannot be defined as a part of the type definition for any such types. Of course, procedure and functions may be separately specified to have parameters with newly defined types [p. 112].

A defined type does not inherit the operations of the data with which it is represented.

E5. Type Definitions (SIMULA 67)

Partially Satisfied.

SIMULA 67's class concept first introduced the definition of types as data objects together with a set of applicable operations [CBL 1.3, 2]. Selectors, predicates, and type converters can be programmed as procedures associated with a class. The object generator NEW serves as a constructor [CBL 4.3.2.2].

"A class declaration with the prefix 'C' and the class identifier 'D' defines a subclass D of the class C. An object belonging to the subclass ... is itself an object of the class C" [CBL 2.2.1]. This means operations defined in class C can also be applied to objects declared to be of class D, i.e. objects of a defined type D do, in this sense inherit operations of the type (C) in terms of which the definition is given. This does not actually conflict with the intent of E5, which takes a "bottom up" view of type definition, while SIMULA takes a "top down" view. It does however violate the letter of E5.

E5. Type Definitions (TACPOL)

Not Satisfied.

TACPOL does not have a type definition capability.

E6. Data Defining Mechanisms (TINMAN)

The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.

E6. Data Defining Mechanisms (ALGOL 68)

Partially Satisfied.

Algol 68 has array and record (structure) types, along with fully discriminated union types [Section 1.2.1, 4.6]. However, it has neither enumeration types nor power set types. The language allows full compile time type checking [Section 0.1.4.1], but does not require it. Algol 68 provides both fixed and dynamic storage allocation [Section 5.2.3].

E6. Data Defining Mechanisms (J3B)

Partially Satisfied.

J3B supports a form of type definition limited to records and one dimensional arrays of records.

J3B does not support enumeration types, power sets, or array and record classes in their full generality (arrays and records cannot themselves be defined as array and record components).

Garbage collection and dynamic storage allocation are not supported by J3B.

E6. Data Defining Mechanisms (PASCAL)

Satisfied.

PASCAL permits defining types by enumeration, as arrays, records, discriminated unions (a record type with variants and a tag field), or sets [pp. 142-144]. Garbage collection, or at least dynamic storage allocation, is needed to support the new and the dispose standard procedures [p. 161].

E6. Data Defining Mechanisms (SIMULA 67)

Partially Satisfied.

SIMULA 67 supports neither enumeration types nor their power sets. A garbage collector is a standard, though optional [CBL 9.1], part of the runtime system [Structure, p. 6]. Arrays of records (i.e., classes) cannot be defined, although arrays of refs to classes can be.

Union (i.e., classes with prefixes) are fully discriminated [CBL 2.2.1], with necessary run time checks initiated through use of the INSPECT statement [B 150].

E6. Data Defining Mechanism (TACPOL)

Not Satisfied.

TACPOL does not support enumeration types; limited forms of arrays and record types are provided (see A7, D5). Cells, which "are used to overlay areas of storage" [Section 4-7], give a rudimentary free union facility. No discriminated union facility is provided. The bit string data type [Section 3-3.b] is TACPOL's substitute for the power set of an enumeration type.

E7. No Free Union or Subset Types (TINMAN)

Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.

Range and subset specifications on variables are useful documentation and debugging aids, but will not be construed as types. Subsets do not introduce new properties or operations not available to the superset and often do not form a closed system under the superset operations.

E7. No Free Union or Subset Types (ALGOL 68)

Satisfied.

Algol 68 unions are fully discriminated [Section 1.2.1, 2.1.1.2, 3.4, 4.6]. The language does not have subranges or subsets.

E7. Free Union or Subset Types (J3B)

Not Satisfied.

J3B permits the storage for one variable to overlay the storage occupied by another variable [Section 4.6]. Moreover, discrimination among record variants cannot be specified in the language, i.e., there are no compile time checks to enforce explicit type discrimination of records with variant structures.

E7. No Free Union or Subset Types (PASCAL)

Not Satisfied.

As indicated in A7, the tagfield of a variant record is optional [p. 144]. "If it is omitted, we obtain the equivalent of a free type union, and a compiler has no chance of checking consistency in its application" [Wirth, p. 197]. Range and subset specifications apppear in PASCAL as subrange types [p. 143]. Indeed, a run-time test to check "all assignments to variables of subrange types to make certain that the assigned value lies within the specified range" [p. 101] is an option for PASCAL 6000.

E7. No Free Union or Subset Types (SIMULA 67)

Satisfied.

Type unions are effected in SIMULA 67 by class prefixing [CBL 2.2.1]. Such type unions can be discriminated using the INSPECT statement [B 150].

E7. No Free Union or Subset Types (TACPOL)

Not Satisfied.

As noted in E6, CELLS [Section 4-7] provide a form of free union.

E8. Type Initialization (TINMAN)

When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

It is often necessary to do bookkeeping or to take other special action when variables of a given type are allocated or deallocated. The language will not limit the class of definable types by withholding the ability to define those actions. Initialization might take place once when the type is allocated (i.e., in its allocation scope) and would be used to set up the procedures and initialize the variables which are local to the type definition. These operations will be definable in the encapsulation housing the rest of the type definition.

E8. Type Initialization (ALGOL 68)

Not Satisfied.

Algol 68 does not directly associate procedures with type declarations [Section 4.2] or with storage allocation [Section 4.4, 5.2.3]. One can write allocation procedures, but they must be explicitly called, e.g.

```
                                        heap [m, n] real;
mode matrix = [m, n] real;
ref matrix a = newmatrix  (3, 5);
```

E8. Type Initialization (J3B)

Not Satisfied.

J3B does not support the type definition concept implied by E8, nor does it provide a means of defining subroutines associated with the allocation and deallocation of variables.

E8. Type Initialization (PASCAL)

Not Satisfied.

Type definitions do not admit any procedural specifications [pp. 142-146]. The procedure statement new (p, t1, ...,tn) "can be used to

allocate a variable of the variant [record] with tag field values t1, ..., tn" [p. 161], but no initialization of components can acccompany this allocation.

E8. Type Initialization (SIMULA 67)

Partially Satisfied.

When an object is generated (i.e., a type instance is allocated) the actual parameters are evaluated and passed; then "control enters the object through its initial begin" [CBL 4.3.2.2]. Executing a detach statement completes the initialization actions [CBL 4.3.2.2, B 265]. Upon executing the final code of the CLASS-body and encountering the final end of an object, it becomes "terminated". It may still be accessible via references and consequently not deallocated [B 123-124, CBL 4.3.2.2]. So, in SIMULA 67 code is executed upon type allocation, but is not necessarily associated with deallocation.

E8. Type Initialization (TACPOL)

Not Satisfied.

For TACPOL's closest approximation to a type definition capbility, namely groups, tables, and cells [Section 4-4], the user cannot specify initialization and finalization procedures for the type, nor can he define procedures to be invoked when variables of a type are allocated and deallocated.

F1. Separate Allocation and Access (TINMAN)

The language will allow the user to distinguish between scope of allocation and scope of access.

The scope of allocation or lifetime of a program structure is that region of the program for which the object representation of the structure should be present. The allocation scope defines the program scope for which own variables of the structure must be maintained and identifies the time for initialization of the structure. The access scope defines the regions of the program in which the allocated structure is accessible to the program and will never be wider than the allocation scope. In some cases the user may desire that each use of a defined program structure be independent (i.e., the allocation and accessing scopes would be identical). In other cases, the various accessing scopes might share a common allocation of the structure.

F1. Separate Allocation and Access Allowed (ALGOL 68)

Partially Satisfied.

Algol 68 has two storage allocation classes [Section 5.2.3]: location (LIFO stack) and heap (global to whole program, dynamic allocation, garbage collected automatically). The access rules for identifiers, mode indicators, operators, etc. are the standard Algol 60 block structure rules [Section 4.8, 7.2]. Algol 68 prohibits the accesss scope from exceeding the allocation scope [Section 5.2.1], although the use of heap storage implies that a variable's allocation scope is not a function of a program's static block structure.

F1. Separate Allocation and Access Allowed (J3B)

Partially Satisfied.

J3B supports ALGOL 60 block structure [Section 3.2], although blocks cannot be smaller than subroutines. In addition, variables declared local to a subroutine will be allocated storage on each subroutine entry if the subroutine is defined as a reentrant subroutine [Section 3.3.2, p. 3-16]. Since nested subroutine definitions are not permitted [Sections 3.2.2, 3.3.2, and 5.1], the notion that a nested subroutine's storage might be allocated when a containing subroutine is invoked cannot be supported.

F1. Separate Allocation and Access (PASCAL)

Partially Satisfied.

Variables declared at the procedure or function level "are accessible by their identifier. They exist during the entire execution process of the procedure (scope) to which the variable is local" [p. 145]. Such variables "are not known outside their scope" [p. 159]. "No conflict arises from a declaration redefining the same identifier within a program" [p. 160]. So, declared variables are allocated on function/procedure entry and freed upon exit. They are accessible only from statements in the statement part and from any procedure or function in the declaration part which does not redeclare the variable [pp. 159-106]. That is, scope of access can be distinguished as a subset from scope of allocation for declared variables. This is a restricted form of ALGOL 60 block structure. However, "variables may also be generated dynamically, i.e. without any correlation to the structure of the program" [p. 145]." These variables are allocated by the standard procedure new. "Access is achieved via a so-called pointer value" [p. 145]. In conjunction with the procedure dispose, run-time checks (which are not specified in the defining documents) would be required to ensure that the access scope is never wider than the allocation scope.

F1. Separate Allocation and Access Allowed (SIMULA 67)

Partially Satisfied.

Using classes the user can distinguish between scope of allocation and scope of access. A class object is allocated by the generator new [CBL

4.3.2.2]. Attributes of the object can be accessed from within (i.e., during the execution of its code) or remotely via object expressions referencing the object [CBL 7]. That is, an object is accessible whenever a pointer to it is accessible. "It is a consequence of the language structure that an object, at the time of its deletion, cannot be referenced by any computable <object reference> expression" [CBL 9.1]. The separation of allocation and access is essential to SIMULA 67's coroutine capability [CBL 9.2]. The user does not have control over deallocation, since if pointers to a class object are in use external to it, then the object's lifetime is not associated with the class's executable code.

F1. Separate Allocation and Access Allowed (TACPOL)

Partially Satisfied.

TACPOL has adopted standard block structure for access scope (and automatic storage for allocation) [Section 7-1]. There are no variables declared to be STATIC except, presumably, those declared in COMPOOLS. There is no way to imply that a variable declared in a nested subroutine is to be allocated when a containing subroutine is to be invoked.

F2. Limiting Access Scope (TINMAN)

The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

Limited access on the call side provides a high degree of safety and eliminates nonessential naming conflicts without limiting the degree of accessibility which can be built into programs. The alternative notion, that all declarations which are external to a program segment should have the same scope, is inconvenient and costly in creating large systems which are composed from many subsystems because it forces global access scopes and the attendant naming conflicts on subsystems not using the defined items.

F2. Limiting Access Scope (ALGOL 68)

Partially Satisfied.

Algol 68 does not allow access to a structure to be limited at the point of definition. Access to indicators declared in outer ranges (blocks) is limited at the point of use only by redeclaring the indicator in an inner range [Section 4.8, 7.2]. The language does allow new local names to be associated with global components, e.g.

```
struct (real re, im) X;
begin
        ref real rx = re of X;
        rx := 3.5;
end
```

F2. Limiting Access Scope (J3B)

Partially Satisfied.

Subroutine names can be specified to be accessible only from within a compilation unit, i.e., they can be declared internal rather than external [Sections 3.3.2 and 3.3.3]. Identifiers associated with variables can be declared external by declaring the identifiers in a block declaration [Sections 3.3.2 and 4.1]. Otherwise, such identifiers are local to either the compilation unit or subroutine in which they are declared.

The use of COMPOOLS permits declarations to be separated into groups made selectively available to requesting compilation units [Section 3.1], but J3B does not support a hierarchy of scopes external to compilation units, because COMPOOLs cannot be referenced from within a COMPOOL declaration file [Section 3.1.3].

To some extent, new local names can be associated with separately defined program components through appropriate use of J3B macro definitions [Section 2.2.6], but there is no language capability directly supporting the redefinition of names when reusable program components are integrated.

F2. Limiting Access Scope (PASCAL)

Not Satisfied.

Specifying access capabilities and limitations is not part of a type definition in PASCAL [pp. 142-146]. Similarly, limited access (e.g. read-only) cannot be specified with the use of a defined structure [pp. 148-152]. Separately defined program components (i.e., functions, data types, etc.) cannot be given new local names by the programmer to avoid naming conflicts.

F2. Limiting Access Scope (SIMULA 67)

Partially Satisfied.

The set of operations (i.e., procedures) applicable to a defined type (i.e., a class object) cannot be controlled. All declared class operations are equally accessible [CBL 7]. For example, with the INSPECT statement, code will be executed as if it "were written inside the body of the inspected object" [B 150]. That is, internal representations and operations cannot be hidden. For implementations permitting separate compilation, new local names

can be associated with the procedure or class declarations; naming conflicts can thus be automatically avoided [CBL 15].

F2. Limiting Access Scope (TACPOL)

Partially Satisfied.

The user cannot specify limited access to separately defined structures where they are used. New local names cannot be associated with separately defined program components.

TACPOL does however, provide the ability to limit access to files declared in the Compool. The authorization list and access mode specification AUTH "specifies the program(s) which are authorized to open a file and the access which they are allowed... The access to the file will be one of the following: INPUT, OUTPUT, UPDATE" [Section 14-3.i].

F3. Compile Time Scope Determination (TINMAN)

The scope of identifiers will be wholly determined at compile time.

Identifiers will be declared at the beginning of their scope and multiple use of variable names will not be allowed in the same scope. Except as otherwise explicitly specified in programs, access scopes will be lexically embedded with the most local definition applying when the same identifier appears at several levels.

F3. Compile Time Scope Determination (ALGOL 68)

Satisfied.

In Algol 68, the access scope of identifiers, type indicators, operators, etc. is completely determined by the static program text and hence at compile time [Section 3.2, 4.8, 7.2]. The most local (lexically) definition is used for an applied occurrence of an indicator [Section 4.8]. Multiple declarations of the same identifier in the same lexical level are prohibited [Section 4.8, 7.1]. However, Algol 68 allows identifiers to be declared after their first lexical applied occurrence. Thus, in

```
real X := Y;
real Y := 2.7;
```

the value of Y is not defined when the first line is executed.

F3. Compile Time Scope Determination (J3B)

Satisfied.

Static block structure scoping rules are used to establish lexically embedded scopes for identifiers [Section 3.2]. Different definitions of variable names are not allowed within the same scope ("A name has one and only one meaning in a given scope" [Section 2.2.1, p. 2-5]). In particular, components of different record types cannot have the same name. However, variables of the same table type will necessarily have identically named components. In J3B, the potential ambiguity implied here is resolved by permitting typed tables to be accessed only with pointer values, which inherently specify which table variable is being referenced. If index values were permitted, then a qualified name capability similar to PASCAL's would be required.

Although the scope of a macro definition is completely determined at compile time, macro identifiers can be given alternative definitions within the same block structure scope [Section 3.4]. In addition, since macro invocations appearing in a macro body are not processed until the macro body containing them is invoked [p. 3-25], these embedded macros use the definition active in the context of the macro call rather than the definition applying in the context of the definition. This means the definition of a macro is not necessarily determined by the lexical structure of a program at the point where the macro is declared.

F3. Compile Time Scope Determination (PASCAL)

Satisfied.

Identifiers are declared local to procedures and functions [pp. 159, 162]. Non-local accessible identifiers are determined by the static program text (i.e., at compile-time). "A procedure may reference any variable global to it, or it may choose to redefine the name" [p. 69].

F3: Compile Time Scope Determination (SIMULA 67)

Satisfied.

The class concept is an extension to block structure that carries over its notion of local identifiers [CBL 1.3.3, CBL 2.2, B 122-124]. Virtual quantities provide a means for binding a name in one scope to a definition in another scope at compile time [CBL 2.2.3].

F3. Compile Time Scope Determination (TACPOL)

Satisfied.

A TACPOL program is organized into blocks that "define the scope of names" [Section 7-1]. This scope is wholly determined at compile time.

F4. Libraries Available (TINMAN)

A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

There will be broad support for libraries common to users of well-recognized application areas. Application libraries will be developed as early as possible.

F4. Libraries Available (ALGOL 68)

Not Satisfied.

Algol 68 says nothing about libraries but allows an implementation to use pragmats (compiler directives) [Section 9.2] to access libraries.

F4. Libraries Available (J3B)

Not Satisfied.

Application-oriented language extensions cannot be made available in J3B, and so libraries of them cannot be made available.

F4. Libraries Available (PASCAL)

Not Satisfied.

PASCAL does not support the library concept as defined by the TINMAN.

F4. Libraries Available (SIMULA 67)

Partially Satisfied.

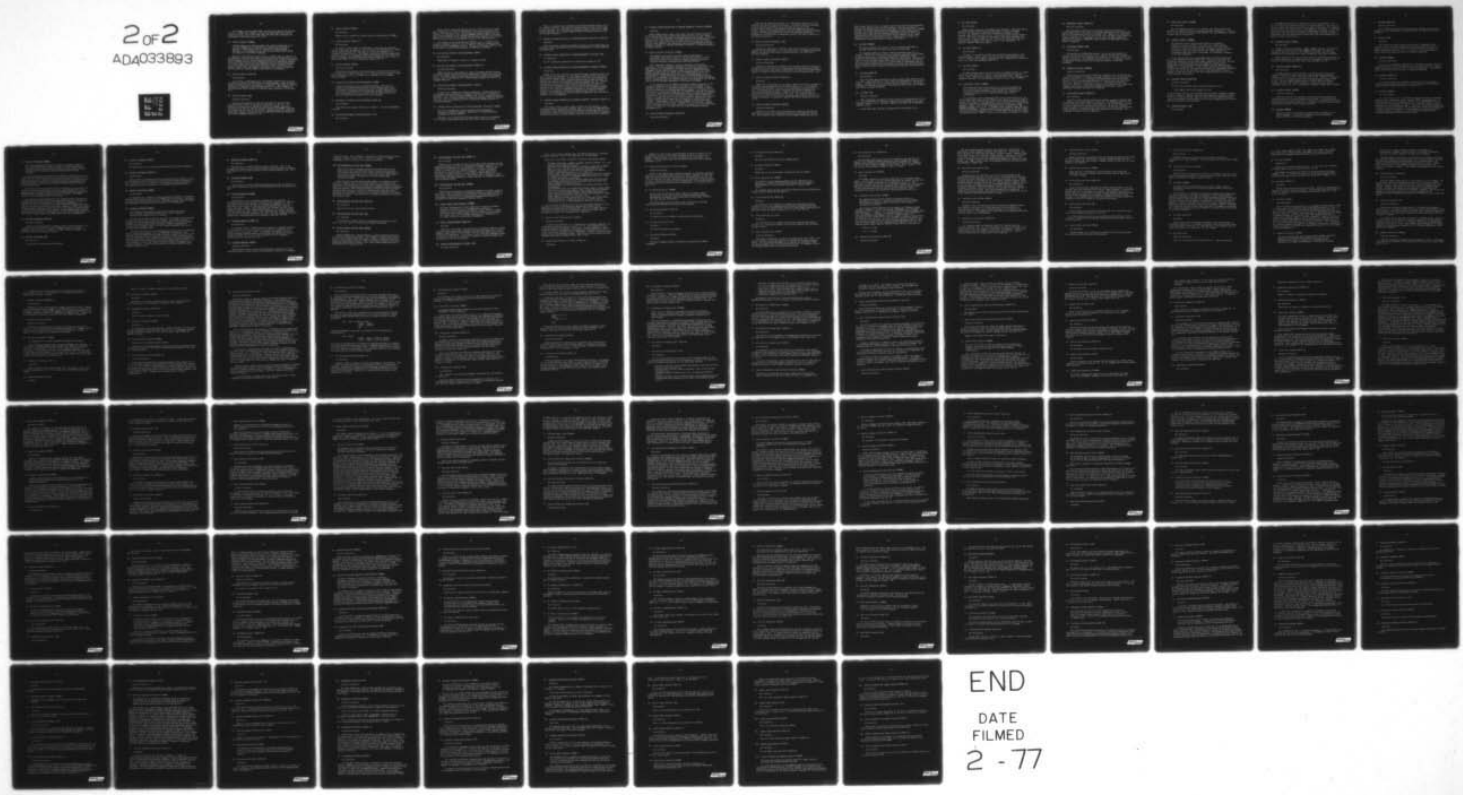
The defining documents do not require broadly supported libraries for a variety of application-oriented data and operations. However, the system classes SIMSET and SIMULATION [CBL 14] along with I/O classes [CBL 11] constitute a primitive library prototype. The effective equivalent of application libraries has arisen in several simulation areas. "A main characteristic of SIMULA 67 is that it is easily structured towards specialized problem areas, and hence will be used as a basis for Special Applications Languages" [CBL PREFACE].

F4. Libraries Available (TACPOL)

Not Satisfied.

AD-A033 893 SOFTECH INC WALTHAM MASS F/G 9/2
EVALUATION OF ALGOL 68, JOVIAL J3B, PASCAL, SIMULA 67, AND TACP--ETC(U)
DEC 76 J B GOODENOUGH, C L MCGOWAN, J R KELLY DAAB07-75-C-0373
UNCLASSIFIED ECOM-75-0373-F NL

2 OF 2
AD4033893



END

DATE
FILMED
2 -77

The language reference manual makes no mention of application oriented libraries. TACPOL does, however, offer built-in procedures for standard numeric operations such as EXP, SQRT, REM, ABS, MAX, MIN, logarithm, and trigonometric functions [Section A-2, A-3].

F5. Library Contents (TINMAN)

> Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

Whether a library should contain source or object code is a question of implementation efficiency and should not be specified in the definition of the source language, but the source language description will always be available. Library routines written in other languages will not be prohibited provided the foreign routine has object code compatible with the calling mechanisms used in the Common HOL and providing sufficient header information (e.g., parameter types, form, and number) is given with the routine in Common HOL form to permit the required compile time checks at the interface.

F5. Library Contents (ALGOL 68)

Not Satisfied.

Algol allows an implementation to use pragmats [Section 9.2] to access libraries and provide for code written in other languages. Algol 68 does not define the internal syntax and semantics of pragmats. Algol 68 also allows an implementation dependent library prelude [Section 10.1] that augments the standard prelude (which defines the standard types, operators, I/O routines, etc.).

F5. Library Contents (J3B)

Partially Satisfied.

Subroutine declarations, record type declarations, and other data declarations can be kept in J3B compools [Sections 3.1 and 4.1]. The subroutine declarations are checked at compile time against subroutine definitions and calls in compiled programs accessing the COMPOOL containing the declarations [Sections 3.3 and 5.3.3]. Compools cannot hold subroutine definitions, however, and may not hold routines whose bodies are written in other source languages [Section 3.1.3].

F5. Library Contents (PASCAL)

Not Satisfied.

PASCAL does not support the library concept as defined by the TINMAN.

F5. Library Contents (SIMULA 67)

Not Satisfied.

The system classes SIMSET and SIMULATION - along with the subclasses linkage, link, head, and process - must, in effect, be maintained in a compile time accessible library since they are not in the base language or in individual programs [CBL 14]. The SIMULA 67 definition does not require such libraries, nor does it accommodate routines whose bodies are written in other source languages. Defined procedures and class declarations (i.e., what is definable in SIMULA 67) can be separately compiled [CBL 15].

F5. Library Contents (TACPOL)

Partially Satisfied.

"Procedures, declarations, data declarations, and file declarations may be included in the Compool" [Section 14-1]. Whether routines written in other source languages can also be included is not addressed in the language reference manual.

F6. Libraries and Compools Indistinguishable (TINMAN)

Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

F6. Libraries and Compools Indistinguishable (ALGOL 68)

Not Satisfied.

Algol 68 does not support libraries or compools. See also requirements F4 and F5.

F6. Libraries and Compools Indistinguishable (J3B)

Not Satisfied.

Source code for inline subroutines (see J5) or other text to be copied into a compilation unit must be contained in files that are separate from compool files [Section 2.2.5]. This distinction between compool files and text files is supported by the language COMPOOL directive (used to reference compool files) [Section 3.1.2] and the COPY directive [Section 2.2.5] (used to insert text files into source programs in place of the COPY directive).

J3B compools are not hierarchically definable, i.e., a compool cannot reference a compool [Section 3.1.3], so compools cannot be associated with various levels of programming activity. The use of particular compools cannot be restricted to any particular program or subsystem of programs.

F6. Libraries and Compools Indistinguishable (PASCAL)

Not Satisfied.

PASCAL does not support a library or a compool concept.

F6. Libraries and Compools Indistinguishable (SIMULA 67)

Partially Satisfied.

SIMULA 67 does not distinguish (in terms of accessing methods) between separately compiled classes and classes represented in source text terms [CBL 15.2]. But classes cannot be associated with various levels of programming activity, nor is the operation of a library/compool specified as part of the language definition.

F6. Libraries and Compools Indistinguishable (TACPOL)

Partially Satisfied.

As noted in F5, TACPOL has a Compool capability. Anything definable in the language "may be included in the Compool" [Section 14-1]. However, as described in the language reference manual, there is just a single Compool per execution. Thus, it is not possible in TACPOL to associate different Compools "with any level of programming activity from systems through projects to individual programs."

F7. Standard Library Definitions for Machine-Dependent Interfaces (TINMAN)

The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

The idea is not to provide all the many special cases in the language, but to provide a few general cases which will cover the special cases.

There is currently little agreement on standard operating system, I/O, or file system interfaces. This does not preclude support of one or more forms for the near term. For the present the important thing is that one be chosen and made available as a standard supported library definition which the user can use with confidence.

F7. Standard Library Definitions for Machine-Dependent Interfaces (ALGOL 68)

Satisfied.

Algol 68 provides standard procedures [Section 10.3] for establishing the interface to data sets/devices and for device independent I/O (both formatted and unformatted).

F7. Standard Library Definitions for Machine-Dependent Interfaces (J3B)

Not Satisfied.

No I/O interfacing capabilities are explicitly provided by J3B.

F7. Standard Library Definitions for Machine-Dependent Interfaces (PASCAL)

Satisfied.

PASCAL's file data type with its associated buffer variable and get, put, reset, and rewrite procedures [p. 161] hides a sequential file allocated on some secondary storage media. In PASCAL 6000 "files that exist outside the program (i.e., before or after program execution) may be made available" if they are "declared as file variables in the main program" [p. 91]. However, "if PASCAL is to serve for system construction purposes, then the file facility might be dropped entirely, because the very purpose would be the description of possible file mechanisms in terms of more primitive concepts" [Wirth, p. 195]. Machine independent interfaces to the machine dependent character set ordering [p. 15] and the maximum integer value [p. 13] are provided. File handling more elaborate than for sequential files and special hardware are not accommodated.

F7. Standard Library Definitions for Machine Dependent Interfaces (SIMULA 67)

Satisfied.

Four types of files - input, output, direct I/O, and printfile - have predefined machine independent interfaces [CBL 11.1.2] but "an implementation may restrict in any way the use of these classes for prefixing" [CBL 11.1.2]. No mention is made of special hardware interfaces, but they would certainly be handled using classes, as with standard input and output devices [B 190-205].

F7. Standard Library Definitions for Machine Dependent Interfaces (TACPOL)

Satisfied.

TACPOL supports open, close, read, write, and other natural operations for both serial and direct access files [Section 13-4, 13-5, 13-6, Table 13-1]. The files may optionally be declared partitioned, blocked, or buffered [Section 14-3]. Most important, the "media" field in a file declaration "specifies the type of device that the file will be allocated to" [Section 14-3.g]. Thus, TACPOL's file declaration and associated operations provide a reasonable standard for "machine independent interfaces" to machine dependent peripheral devices.

G1. Kinds of Control Structures (TINMAN)

The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling, and asynchronous interrupt handling.

These mechanisms, hopefully, provide a spanning set of control structures. The most appropriate operations in several of these areas is an open question. For the present, the choice will be a spanning set of composable control primitives each of which is easily mapped onto object machines and which does not impose run time charges when it is not used. Whether parallel processing is real (i.e., by multiprocessing) or is synthesized on a single sequential processor, is determined by the object machine, but if programs are written as if there is true parallel processing (with no assumption about the relative speeds of the processors) then the same results will be obtained independent of the object environment.

It is desirable that the number of primitive control structures in the language be minimized, not by reducing the power of the language, but by selecting a small set of composable primitives which can be used to easily build other desired control mechanisms within programs. This means that the capabilities of control mechanisms must be separable so that the user need not pay either program clarity or implementation costs for undesired specialized capabilities. By these criteria, the Algol-60 "FOR" would be undesirable because it imposes the use of a loop control variable, requires that there be a single terminal condition and that the condition be tested before each iteration. Consequently, "FOR" cannot be composed to build other useful iterative control structures (e.g., FORTRAN "DO"). The ability to compose control structures does not imply an ability to define new control operations, and such an ability is in conflict with the limited parameter passing mechanisms of C7.

G1. Kinds of Control Structures (ALGOL 68)

Partially Satisfied.

Algol 68 has sequential [Section 3.2], conditional [Section 3.4], and iterative [Section 3.5] control structures. It allows recursive procedures. It has a parallel processing control structure [Section 3.3] and synchronization operators (up and down on integer semaphores) [Section 10.2.4]. However, the language does not have an exception handling mechanism. (Certain I/O conditions such as end of file are handled by providing procedures to be called when the condition occurs). Nor does it provide for asynchronous interrupts.

G1. Kinds of Control Structures (J3B)

Partially Satisfied.

Sequential, conditional, iterative, and recursive control structures are supported by J3B [Sections 5.1, 5.5, 5.6, and 3.3.2]. No support is provided, however, for parallel processing, exception handling, and asynchronous interrupt handling.

G1. Kinds of Control Structures (PASCAL)

Partially Satisfied.

Sequential, conditional, iterative, and recursive control structures are provided [pp. 153-157,159, 162]. Asynchronous interrupt handling, (pseudo) parallel processing, and exception handling are not present in standard PASCAL except, of course, for end of file and end of text line predicates [p. 163].

G1. Kinds of Control Structure (SIMULA 67)

Satisfied.

SIMULA 67's FOR-statement is the same as ALGOL 60's [AR 4.6] extended "to facilitate the processing of list structures" [CBL 6.2.2]. The iteration mechanisms, FOR and WHILE, only test at the beginning of loops [B 80-81, B 92]. The language has no control structures for exception handling and asynchronous interrupt handling. SIMULA 67 does have a co-routine or quasi-parallel sequencing capability [CBL 9.2]. But programs cannot be written as if there is true parallel processing since explicit "detach" and "resume" statements are required to specify the precise sequencing [CBL 9.2.1, 9.2.2].

G1. Kinds of Control Structures (TACPOL)

Partially Satisfied.

TACPOL has structured control mechanisms for sequential [Section the BEGIN block, Section 7-2], conditional [Section the IF statement, Section 8-1], and iterative control [Section the DO statement especially with the

WHILE option, Section 8-3]. One cannot easily build other desired sequential control forms without using the GOTO statement. Parallel processing is available in limited form for input/output where "by specification of a RETURN attribute for certain operations, the transmisssion occurs concurrently" [Section 13-2]. For just the exception conditions zero divide (ZDIV) and fixed overflow (FOFL) the user can specify merely "whether or not a snap procedure is to be called" [Section 12-1]. TACPOL does not provide for asynchronous interrupt handling or recursive procedures.

G2. The GOTO (TINMAN)

The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition.

The language should not, however, impose unnecessary costs for its presence. The "GO TO" will be limited to explicitly specified program labels at the same scope level. Switches, designational expressions, label variables, label parameters and numeric labels are not desired. Switches here refer to the unrestricted switches which are generalizations of the "GO TO" and not to case statements which are a general form for conditionals(see G3). This requirements should not be interpreted to conflict with the specialized form of control transfer provided by the exception handling control structure of G7.

G2. The GOTO (ALGOL 68)

Not Satisfied.

Algol 68 allows jumps from inner to outer lexical scopes [Section 5.4.4, 4.8, 7.2]. Jumps from outer to inner levels are prohibited (by the two-level syntax). Although label is not an Algol 68 type, procedure is. Thus label variables are obtained via procedure variables and label values via procedures whose bodies are *goto* statements.

G2. The GOTO (J3B)

Partially Satisfied.

Label definitions are limited in their scope to a subroutine body in J3B, i.e., it is impossible to jump into or out of a procedure body using a goto statement [Section 5.4.1]. Moreover, labels are not permitted as parameters [Section 5.3.3].

On the other hand, switches are supported by J3B [Section 5.4.3].

G2. The GOTO (PASCAL)

Not Satisfied.

PASCAL permits jumps out of procedures and functions. Each label must be explicitly declared in the procedure block where it marks a statement [p. 153]. Any statement in the block (including one within an inner procedure) may GO TO the label [p. 31]. Thus, scope levels can be crossed by a GO TO. "The effect of jumps from outside of a structured statement into that statement is not defined" [p. 32]. Technically, this means jumps into loops are not forbidden; their effect is left to the implementor. In addition, PASCAL's labels are numeric.

G2. The GOTO (SIMULA 67)

Not Satisfied.

The GOTO is not limited to explicitly specified program labels at the same scope level. Rather, GOTO's can leave a class scope completely [CBL 9.2.4]. SIMULA 67 also supports switches [CBL 8.1], designational expressions [CBL 4.1.1], and label parameters [CBL 8.2].

G2. The GOTO (TACPOL)

Not Satisfied.

"A GOTO statement may be used in any block to transfer control to a point within the block itself or to a point in any containing block" [Section 9-2.b]. Thus, GOTOs are not limited to the same scope level. Moreover, label parameters [Section 11-5] and switches [Section A-7] are a part of TACPOL.

G3. Conditional Control (TINMAN)

The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

The conditional control operations will be fully partitioned (e.g., an "ELSE" clause must follow each "IF THEN") so the choice is clear and explicit in each case. There will be some general form of conditional which allows an arbitrary computation to determine the selected situation (e.g., Zahn's device [Donald E. Knuth, "Structured Programming with go to Statements," ACM Computer Surveys,r Vol. 6, No. 4, December 1974] provides a good solution to the general problem). Special mechanisms are also needed for the more common cases of the Boolean expression (e.g., "IF THEN ELSE") and for value or type discrimination (e.g., "CASE" on one of a set of values or subtype of a union).

G3. Conditional Control (ALGOL 68)

Partially Satisfied.

Algol 68 permits selection of alternatives based on the value of a Boolean expression, on the value of an integer expression, and on the current type of a discriminated union value [Section 3.4]. However, the conditional control structures are not fully partitioned since the else or out clauses are optional (and default to else skip), nor are case statement alternatives labeled when selection is based on integer values. No equivalent of Zahn's device is supported.

G3. Conditional Control (J3B)

Not Satisfied.

J3B supports an IF-THEN-ELSE construct, but it does not require the presence of an ELSE clause [Section 5.5]. J3B does not directly provide a control structure for discriminating the subtype of a value from a union type, nor does it provide a case statement control structure. In addition, there is no equivalent of Zahn's device for exiting from loops or conditional statements.

G3. Conditional Control (PASCAL)

Partially Satisfied.

PASCAL permits if-then-else conditional statements [p. 154] but the else clause is not required. Similarly the case statement is not constrained to be fully partitioned [p. 155], although case constitutents must be explicitly labeled. If the case expression evaluates to a label not in the case label list, then "the effect is undefined" [p. 31]. Discriminated unions are realized in PASCAL by records with alternative structures, and the PASCAL case statement can be used to discriminate among these variants. Loop and conditional statement exiting is not supported.

G3. Conditional Control (SIMULA 67)

Not Satisfied.

SIMULA 67 permits if-then conditionals without requiring a matching else in accordance with ALGOL 60 [CBL 6., AR 4.5.1]. The INSPECT statement is a general case-like construct that discriminates on type [B 150-151]. It allows any object expression to determine the selection. An optional OTHERWISE clause can make it fully partitioned [CBL 7.2.1], but its use is not required. A case construct for other than type discrimination is not provided. No equivalent of Zahn's device is supported.

G3. Conditional Control (TACPOL)

Not Satisfied.

The IF statement [Section 8-1] is TACPOL's only conditional control structure and "the ELSE clause is sometimes omitted" [Section 8-1]. That is, the IF is not required to be fully partitioned. No case statement or variant of Zahn's device is supported.

G4. Iterative Control (TINMAN)

The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables to be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run time execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).

In its most general form, a programmed loop is executed repetitively until some computed predicate becomes true. There may be more than one terminating predicate, and they might appear anywhere in the loop. The most common case is termination after a fixed number of iterations and a specialized control structure should be provided for that purpose (e.g., FORTRAN "DO" or Algol-60 "FOR"). Loop control variables are by definition variables used to control the repetitive execution of a programmed loop and as such will be local to the loop body, but at loop termination it will be possible to pass their value (or any other computed value) out of the loop, conveniently and efficiently.

G4. Iterative Control (ALGOL 68)

Partially Satisfied.

The Algol 68 iterative control structure [Section 3.5] is

for i from e1 by e2 to e3 while b do s od

The termination condition is tested before each iteration instead of anywhere during an iteration. However, the other requirements are satisfied (control variable is local to the loop, entry is only at the head (labels defined in loops are inaccessible from outside the loop), for and while loops are special cases of the general loop structure).

G4. Iterative Control (J3B)

Not Satisfied.

Although J3B provides an iterative control structure [Section 5.6], it does not permit the termination condition to appear anywhere in the loop, it does not require the control variable to be local to the loop, and it does not provide special case notations for a fixed number of iterations. In addition, the value of the loop control variable is implementation defined on loop exit and is directly accessible from outside the loop body. J3B does, however, insure that loops are entered only at their head, and not by jumping into the body of the loop [Section 5.6].

G4. Iterative Control (PASCAL)

Not Satisfied.

PASCAL's repetitive statements - while, repeat, and for - do not permit the termination condition to appear anywhere in the loop. In addition, although G4 does not require it, the PASCAL for statement does not permit incrementing or decrementing the control variable by steps other than unity.

Entry should only be at the loop head since the repetitive statements are structured statements. But jumping into such statements is undefined (i.e., it can vary with implementations) rather than forbidden [p. 32].

G4. Iterative Control (SIMULA 67)

Not Satisfied.

The FOR-statement control variable is not local to the loop [B 92]; SIMULA 67's iterative control structures, the FOR and WHILE statements, terminate "only at the head of the loop" [B 80, B 92]. GOTOs are restricted to lead only "to a program point within an operating block instance" [CBL 9.2.4]. The CBL defining document does not forbid a GOTO to a labeled statement within a FOR or WHILE construct. And the AR simply makes it "undefined" [AR 4.5.6] (i.e., implementation dependent) rather than forbidden.

G4. Iterative Control (TACPOL)

Partially Satisfied.

TACPOL does not have an iterative control structure that permits the termination condition to appear anywhere in the loop. The DO statement's control variable is local to the DO loop [Section 8-3], although the Reference Manual does not state this explicitly. In addition, the loop control variable is implicitly declared to be an integer variable.

G5. Routines (TINMAN)

Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

G5. Routines (ALGOL 68)

Partially Satisfied.

Algol 68 has both recursive and non-recursive routines [Section 5.4.1]. However, it allows procedures to be defined within the bodies of recursive procedures.

G5. Routines (J3B)

Satisfied.

Both recursive and non-recursive subroutines are supported by J3B [Section 3.3.2]. (Recursive subroutines are those routines declared to be reentrant.) Because no J3B subroutine can be defined within the scope of another subroutine [Sections, 3.2, 3.3.3, and 5.1], it is not possible to define subroutines within the body of a recursive subroutine.

G5. Routines (PASCAL)

Partially Satisfied.

"The use of the procedure identifier in a procedure statement within its declaration implies recursive execution of the procedure" [p. 159]. Similar remarks apply to functions [p. 162]. As with any procedure it is possible to define a procedure within the body of a recursive procedure.

G5. Routines (SIMULA 67)

Partially Satisfied.

Recursive procedures are permitted (164-173) but it is possible to define procedures within the body of a recursive procedure since SIMULA 67 subsumes ALGOL 60 declarations [CBL 2.1, AR 5.4.1].

G5. Routines (TACPOL)

Not Determinable.

It is not clear whether or not TACPOL supports recursion; the Reference Manual does not address this issue. However, TACPOL is characterized as "a modified subset of the PL/I language" [Section 1-1] and RECURSIVE is not a reserved word [Section 2-6.g.(3), Table 2-1]. Since PL/I requires explicit declaration of the RECURSIVE attribute, it could be concluded that TACPOL does not permit recursive procedures. On the other hand, since local variables are allocated on procedure invocation it would appear that procedures could be called recursively, although it is uncertain whether nested recursive procedures would execute correctly in all cases.

G6. Parallel Processing (TINMAN)

The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possibly pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

The parallel processing capability will minimally provide the ability to define and call parallel processes and the ability to gain exclusive use of system resources in the form of data structures, devices, and pseudo devices. This latter ability satisfies one of the two needs for synchronization of parallel processes. The other is required in conjunction with real time constraints (see G8).

The parallel processing capability will be defined as true parallel (as opposed to coroutine) primitives, but with the understanding that in most implementations the object computer will have fewer processors (usually one) than the number of parallel paths specified in a program. Interleaved execution in the implementation may be required.

The parallel processing features of the language should be selected to eliminate any unnecessary overhead associated with their use. The costs of parallel processes are primarily in run time storage management. In particular, it will not be possible to define a parallel routine within the body of a recursive routine and it will not be possible to define any routine, including parallel routines, within the body of those parallel routines which can have multiple simultaneous activations. If the language permits several simultaneous activations of a given parallel process then it might require the user to give a upper bound on the number which can exist simultaneously.

G6. Parallel Processing (ALGOL 68)

Partially Satisfied.

Algol 68 has a true parallel processing control structure [Section 3.3]. It provides for synchronization via up and down operators on integer semaphores [Section 10.2.4]. However, it allows recursive procedures and parallel clauses to be arbitrarily nested.

G6. Parallel Processing (J3B)

Not Satisfied.

J3B provides no process control primitives.

G6. Parallel Processing (PASCAL)

Not Satisfied.

Standard PASCAL has no (possibly pseudo) parallel processing capability.

G6. Parallel Processing (SIMULA 67)

Not Satisfied.

SIMULA 67 does not support parallel processing with the ability to create and to terminate a process along with critical section control primitives. It does, however, provide a co-routine or quasi-parallel sequencing capability with appropriate creation, detaching, and resuming primitives [CBL 9.2].

G6. Parallel Processing (TACPOL)

Not Satisfied.

As mentioned in G1, TACPOL's parallel processing capability is limited to READ and WRITE statements where "if the RETURN option is present it specifies concurrent operation" [Section 13-6.f]. Otherwise processes cannot be created and terminated; consequently mutual exclusion is not provided for.

G7. Exception Handling (TINMAN)

The exception handing control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

The user must be able to specify the action to be taken on any exception situation which might occur within his program. The exception handling mechanism will be parameterized so data can be passed to the recovery point. Exception situations might include arithmetic overflow, exhaustion of available space, hardware errors, any user defined exceptions, and any run time detected programming error.

The user will be able to write programs which can get out of an arbitrary nest of control and intercept it at any embedding level desired. The exception handling mechanism will permit the user to specify the action to be taken upon the occurrence of a designated exception within any given access scope of the program. The transfers of control will, at the users option, be either forward in the program (but never to a narrower scope of access or out of a procedure) or out of the current procedure through its dynamic (i.e., calling structure). The latter form requires an exception handling formal *parameter class* (see C7).

G7. Exception Handling (ALGOL 68)

Not Satisfied.

Algol 68 has no exception handling control structure. Since it has procedure data types, routines may be assigned to data structures and passed as parameters and hence may be used for some exception handling, e.g., end of file.

G7. Exception Handling (J3B)

Not Satisfied.

J3B provides no exception handling capability other than the ability to write into a procedure or function's output parameters before returning from a call.

G7. Exception Handling (PASCAL)

Not Satisfied.

PASCAL does not have an exception handling control structure. It provides only end of file and end of text line text predicates [p. 163] and the capability of having functions or procedures as parameters [p. 152]. These can, in some cases, be used for exception handling that transfers control and data. However, such "procedures and functions must have value parameters only" [p. 158]. PASCAL 6000 provides as a user option, run-time tests for division by zero, array or case indexing error, integer overflow, and violation of subrange types [pp. 101, 121]. But such exception conditions produce only a run-time error and an informative dump [pp. 102-103].

G7. Exception Handling (SIMULA 67)

Not Satisfied.

As remarked under G1, SIMULA 67 does not support an exception handling control structure. Function procedures that test for a condition can be declared with a class (i.e., type). For example, the INFILE class has a Boolean procedure LASTITEM usable each time before attempting to read from INFILE [B 193-201, CBL 11.2.1]. Run time errors occur on arithmetic exceptions such as overflow [B 70-71].

G7. Exception Handling (TACPOL)

Partially Satisfied.

TACPOL gives the user control with ON statements [Section 13-13] over endfile conditions, trying to open a nonexisting file partition, or making a

record key error. One can CHECK for zerodivide or fixed overflow and get a diagnostic snapshot [Section 12-1]. This is the extent of a TACPOL programmer's control over error or exception situations.

G8. Synchronization and Real Time (TINMAN)

There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.

When parallel or pseudo parallel paths appear in a program it must be possible to specify their relative priorities and to synchronize their executions. Synchronization can be done either through exclusive access to data (see G6) or through delays terminated by designated situations occurring within the program. These situations should include the elapse of program specified time intervals, occurrence of hardware interrupts and those designated in the program. There will be no implicit evaluation of program determined situations. Time delays will be program specifiable for both real and simulated times.

G8. Synchronization and Real Time (ALGOL 68)

Not Satisfied.

Algol 68 has none of the required features.

G8. Synchronization and Real Time (J3B)

Not Satisfied.

J3B provides no language features for dealing with parallel control paths, real time clocks, or asynchronous hardware interrupts.

G8. Synchronization and Real Time (PASCAL)

Not Satisfied.

Standard PASCAL has no synchronization or mutual exclusion primitives. It also has no DELAY-like statement. With respect to source level access to real-time clocks PASCAL 6000, for example, provides a predefined procedure time (), which assigns the current time as a character string [p. 98], and a predefined parameterless function, clock, which yields elapsed job time in milliseconds as an integer [p. 99].

G8. Synchronization and Real Time (SIMULA 67)

Not Satisfied.

SIMULA 67 does not admit "access to real time clocks" and does not have exception handling facilities for "asynchronous hardware interrupts" or other computational interrupts such as zerodivide or overflow. However, with respect to simulated time a comprehensive set of coroutine scheduling procedures are provided [CBL 14.2.4, B 286-288]. Delays and relative priorities are easily specified. Jacob Palme in "Making SIMULA into a Programming Language for Real Time" (1974 SIMULA Users' Conference) claims "that a small and simple addition to SIMULA can make the language suitable for real time applications".

G8. Synchronization and Real Time (TACPOL)

Not Satisfied.

TACPOL does not provide for asynchronous hardware interrupts, access to real time clocks, specification of relative priorities, or specification of time delays. After executing a WAIT statement, "the continued execution of the program is delayed until all operations requested pertaining to the designated file have been completed" [Section 13-14].

H1. General Syntax Characteristics (TINMAN)

The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

H1. General Characteristics (ALGOL 68)

Satisfied.

Algol 68 is free format, uses semicolon as a statement separator, allows arbitrary length identifiers, is based on conventional forms, has a uniform grammar, avoids special case notations, does not allow abbreviations, and is syntactically unambiguous. It is harder to parse than some languages because it allows the first use of a new type or operator to lexically precede the declaration of the type of operator.

H1. General Characteristics of Syntax (J3B)

Partially Satisfied.

J3B is a free format language, with the semicolon acting as a statement terminator [Sections 4 and 5]. Key words may not be abbreviated.

There are at least eight violations of syntactic uniformity, however:

- The ENDs terminating a compound statement cannot be labeled, i.e., one cannot write AA: END; [Section 5.1].
- Although a null compound statement is permitted (signified by BEGIN END;), a null set of table item and block declarations is not permitted, even though it also would be signified by the keyword sequence, BEGIN END; [Sections 4.1, 4.2, and 4.3];
- The BIT, BYTE, and INTR functions are permitted to take only variables as arguments [Section 9.2], although semantically there is no reason to forbid expresssions or constants.
- The syntax for defining macros differs unnecesarily from the syntax for defining subroutines in that a semicolon does not delimit the formal parameter list [Section 3.4];
- Arrays may have up to three dimensions, but tables can have at most one dimension;
- Typed tables can only be accessed with pointer values; untyped tables can only be accessed with integer values; arrays and array elements cannot be accessed through pointers, only through index values;
- Block items are initialized separately from their declaration, unlike other variables [Section 4.1, 4.4, and 4.5];
- Typed tables can only be initialized in a block, but untyped tables declared global to a compilation unit (or in a block) can be initialized as part of their declaration [Section 4.3 and 4.1].

H1 is also not satisfied in that only the first eight characters of an identifier are used to distinguish two identifiers [Section 2.2.1]. Consequently, identifiers can be abbreviated. The use of single letters to specify built-in data types in declarations [Section 4.8] might also be considered to violate H1's requirements for readability.

H1. General Syntax Characteristics (PASCAL)

Partially Satisfied.

In PASCAL "the semicolon (;) is considered as a statement separator, not a statement terminator (as e.g., in PL/I)" [p. 7]. The language is free format and permits long mnemonic identifiers. "Standard PASCAL will always recognize the first 8 characters of an identifier as significant" [p. 9]. In effect, this means identifiers can be abbreviated. Syntactic simplicity was a guiding idea in the language's design. PASCAL has an easily parsed one-token lookahead deterministic grammar [Wirth, SPE, p. 320].

H1. General Characteristics of Syntax (SIMULA 67)

Satisfied.

SIMULA 67 is free format, following ALGOL 60, which it extends [B 10]. The semicolon is used as a statement separator [B 79, B 363, AR 4.1.1]. Mnemonic identifiers [AR 2.4.1] with no abbreviations are allowed. The language is easily parsed using "the well-known transition table technique" [Structure, p. 4].

H1. General Characteristics of Syntax (TACPOL)

Partially Satisfied.

TACPOL is free format with a semicolon serving as a statement terminater [Section 2-2] and mnemonic identifiers can be used. "However, if more than eight characters are used for an identifier, the compiler will use only the first five and the last three characters for the identifier" [Section 2-6.g]. In effect, this means identifiers can be abbreviated. The key word SUBSTR for the string operator can also be written (abbreviated ?) as $ [Section 6-1.c, Table 6-3].

H2. No Syntax Extensions (TINMAN)

The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms, or define new infix operator precedences.

This requirement does not conflict with E4 and does not preclude associating new meanings with existing infix operators.

H2. No Syntax Extensions (ALGOL 68)

Not Satisfied.

In Algol 68, one can redefine existing operator precedences.

H2. No Syntax Extensions (J3B)

Satisfied.

No syntax extensions are permitted.

H2. No Syntax Extensions (PASCAL)

Satisfied.

No source language syntactic extensions are permitted the PASCAL programmer.

H2. No Syntax Extensions (SIMULA 67)

Satisfied.

The user cannot modify the source language syntax.

H2. No Syntax Extensions (TACPOL)

Satisfied.

TACPOL has none of the extension capabilities that H2 forbids.

H3. Source Character Set (TINMAN)

The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

The language definition will specify the translation from the publication language into the restricted character set.

H3. Source Character Set (ALGOL 68)

Satisfied.

The character set of Algol 68 is suitable for publication purposes [Section 9.3, 9.4]. The language defines alternate representations using characters in the 64-character ASCII subset for those symbols, operators, etc. which are outside the subset [Section 0.1.4.5, 9.3, 9.4, 10.2.3].

H3. Source Character Set (J3B)

Satisfied.

All J3B symbols can be composed using the 64 character ASCII subset [Section 2.1]. No special character set has been specified for publication purposes, however.

H3. Source Character Set (PASCAL)

Partially Satisfied.

A CDC ASCII 64-character set version of PASCAL exists [pp. 94-95]. And the ASCII subset is more than adequate for expressing the source language syntax. But the defining documents do not specify the translation from a publication language into a restricted character set. For example, there is no representation dictum for the non-ASCII up-arrow.

H3. Source Character Set (SIMULA 67)

Not Satisfied.

The defining document does not specify translation into the ASCII 64 character set although a mapping between the formal language description symbols and the equivalent in sample print-outs is given [B 11]. SIMULA 67 uses the non-ASCII turnstile symbol for NOT [B 360]. Also, non-ASCII lower case letters are permitted in SIMULA 67 [AR 2.1, B 360].

H3. Source Character Set (TACPOL)

Satisfied.

All TACPOL programs can be written using just the 64 character ASCII subset. In fact, only 50 of the 64 ASCII characters are needed [Section 2-3]. There are even TACPOL conventions for representing a colon (..) or a semicolon (.,) when they are not in the available character set [Section 2-4]. Wherever non-ASCII characters are permitted (e.g, vertical bar for logical inclusive OR and double vertical bar for string catenation), ASCII alternatives are given (e.g, OR and CAT [Section Tables 6-3 and 6-4]).

H4. Identifiers and Literals (TINMAN)

The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

Some possible break characters are the space(E. W. Dijkstra, coding examples in Chapter I, "Notes in Structured Programming," in Structured Programming by O.-J. Dahl, E. W. Dijkstra and C.A.R. Hoare, Academic Press, 1972; and Thomas A. Standish, "A Structured Program to Play Tic-Tac-Toe," notes for Information and Computer Science 3 course at Univ. of California-Irvine, October 1974) (i.e., any number of spaces or end-of-line), the underline, and the tilde. The space cannot be used if identifiers and user defined infix operators are lexically indistinguishable, but in such a case the formal grammar for the language would be ambiguous (see H1). The language should require separate quoting of each line of a long literal:

"This is a long"

"literal string".

H4. Identifiers and Literals (ALGOL 68)

Partially Satisfied.

The Algol 68 Revised Report defines the formation of literals (8), identifiers, tokens, symbols, operators, key words, etc. [Section 9]. The break character for identifiers and numeric literals is space. However, there is no break character for character string literals. They must be continued at the start of the next line since separate quoting of each line is not allowed. Although one can use the concatenation operator to express the same effect, e.g. "ABC" + "DEF", there is no guarantee that the concatenation will be performed at compile time.

H4. Identifiers and Literals (J3B)

Partially Satisfied.

Formation rules for identifiers and literals are specified in the language definition [Sections 2.2.1 and 2.2.3]. The period and dollar sign may be used as break characters within identifiers [Section 2.2.1]. However, no break character is provided for use with numeric literals [Section 2.2.3]. Although separate quoting of each line of a long character or numeric literal is not required, the character literal format makes special provision for detecting a missing closing quote bracket, namely, semicolons can only be included in character literals by writing ';;' [Section 2.2.3.5]. Writing a single semicolon is an error. Since all statements are terminated by semicolons, a missing closing quote bracket is detected no later than the end of the statement containing the character literal.

H4. Identifiers and Literals (PASCAL)

Partially Satisfied.

Formation rules for identifiers and for literals are explicitly given [pp. 140-141]. But there is no break character for use internal to identifiers and literals, nor is separate quoting of each line of long literals required. The readability of identifiers can be enhanced using upper and lower cases, e.g., RealTimeClock.

H4. Identifiers and Literals (SIMULA 67)

Partially Satisfied.

The formation rules for identifiers [AR 2.4.1] and for literals - character constants [CBL 4.2.1, B 174], Boolean or logical values [AR 2.2.2], numbers [AR 2.5.1], text constants [B 178-179] - are explicit. There is no break character for use internal to identifiers and literals. Separate quoting of each line of a long literal is not required.

H4. Identifiers and Literals (TACPOL)

Partially Satisfied.

Though indicating the formation rules for identifiers [Section 2-6.g] and literals [Section 3-5], TACPOL does not permit a break character for use internal to identifiers and literals. Separate quoting of each line of a long literal is not required.

H5. Lexical Units and Lines (TINMAN)

There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

H5. Lexical Units and Lines (ALGOL 68)

Not Satisfied.

Algol 68 defines where typographical display features (space, new line, new page) may occur [Section 9.4.d], which is almost anywhere. In particular lexical units may be continued across lines. Some lexical units such as string literals or operations (:=) cannot contain arbitrary display features. The end of line character can be included in string literals as a data character if the implementation extends the production for other-string-item [Section 8.1.4.1.d] to include it, as stated in the Report.

H5. Lexical Units and Lines (J3B)

Not Satisfied.

Lexical units are specifically defined in J3B to be continuable across input record boundaries [Section 2.1, p. 2-3].

The end-of-line character can be included in literal strings if this character is supported by the implementation-defined character set [Sections 2.2.3.5 and 2.1].

H5. Lexical Units and Lines (PASCAL)

Not Satisfied.

Standard PASCAL does not mention the end-of-line for source programs. Thus, lexical units could cross line boundaries.

H5. Lexical Unit and Lines (SIMULA 67)

Not Satisfied.

SIMULA 67 programs are expressed in free format form with no consideration for line boundaries. Lexical units are not restricted to single lines.

H5. Lexical Units and Lines (TACPOL)

Not Satisfied.

Lexical units can continue across lines. For example, lines may contain up to 80 characters [Section 2-2], but "an identifier can be from one to any number of characters in length" [Section 2-6.g].

H6. Key Words (TINMAN)

Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

By key words of the language are meant those symbols and strings which have special meaning in the syntax of programs. They introduce special syntactic forms such as are used for control structures and declarations, or they are used as infix operators, or as some form of parenthesis. Key words will be reserved, that is unusable as identifiers, to avoid confusion and ambiguity. Finally, there will be no place in a source language program in which a key word can be used in place of an identifier. That is, functional form operations and special data items built into the language or accessible as a standard extension will not be treated as key words but will be treated as any other identifier.

H6. Key Words (ALGOL 68)

Satisfied.

Algol 68 key words are reserved, informative, and are not usable as identifiers [Section 9]. The language has about ninety bold face key words (counting syntactic words like if, types like real , and operators like le). Algol 68 key words are concise, e.g. int for integer.

H6. Key Words (J3B)

Partially Satisfied.

There are 52 key words in J3B [Section 2.2.1]. They are reserved.

Of the 52 key words, ABS, BIT, BYTE, ENTRY, FIX, INTGR, INTR, LBOUND, NENT, POINT, SCALE, SHIFTL, SHIFTR, and UBOUND can be used in syntactic contexts where a programmer-defined identifier can be used [Section 9].

H6. Key Words (PASCAL)

Satisfied (?)

PASCAL has 35 reserved keywords [p. 9]. Is this "very few in number"? The keyword mod is used as an infix operator [p. 13].

Requirement E1 includes "the ability to define new infix operators." (TINMAN, p. 34). If these new operators can be identifiers, then PASCAL's infix mod would appear "where an identifier can be used" (TINMAN, p. 49).

H6. Key Words (SIMULA 67)

Satisfied.

SIMULA 67 has 50 reserved key words [B 359]. They are a reasonable set that does not impair syntax recognition. The key words TRUE, FALSE, NONE, and NOTEXT can appear as constants in appropriate expressions [AR 2.2.2., CBL 4.3.1, CBL 4.4.1].

H6. Key Words (TACPOL)

Not Determinable.

This is an area of inconsistency in the language reference manual. Table 2-1 gives a list of 97 "reserved words" including such words as: B, CELL, E, L, S, and MOVE. And the rules for identifier formation state that "no identifier may be identical to any of the reserved words shown in Table 2-1" [Section 2-6.g.(2)]. However, this restriction on identifier names is ignored in Chapter 10 where one is cautioned that "care should be taken not to inadvertently redeclare names of intrinsic procedures" [Section 10-2.a]. An example is given wherein MOVE is used as a label; it is observed that "MOVE would no longer be accessible as an intrinsic procedure in that block or in any blocks contained within that block" [Section 10-2.a]. This contradicts the injunction against using MOVE as an identifier in Chapter 2. Similarly an example in Chapter 13 uses CELL as an identifier [Section 13-6.f.(2)], again contradicting Chapter 2.

H7. Comment Conventions (TINMAN)

The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear,

will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

Comments anywhere reasonable in a program will not be taken to mean that they can appear internal to a lexical unit, such as, an identifier, key word, or between the opening and closing brackets of a character string. One comment convention which nearly meets these criteria is to have a special quote character which begins comments and with either the quote or an end-of-line ending each comment. This allows both embedded and line-oriented comments.

H7. Comment Convention (ALGOL 68)

Partially Satisfied.

Algol 68 comments begin/end with a cents symbol, #, co, or comment. The report defines where they may appear [Section 9.1, 9.2], which is almost anywhere, but not inside certain lexical units such as key words and string literals. Comments may contain any character except the opening comment symbol. Automatic reformatting is not prevented. The requirement not satisfied is that end of line does not terminate a comment. Algol 68 also has pragmats, delimited by pragmat or pr, which are comments that an implementation may interpret as directives to the compiler, e.g., pr copy file xxx pr.

H7. Comment Conventions (J3B)

Partially Satisfied.

Comments are enclosed in either quote (") or percent (%) characters [Section 2.2.4]. Since quote characters are also used to delimit macro bodies and (possibly) actual macro parameters [Section 2.2.6] comments delimited with quote characters can potentially be confused by readers with source code not serving as a comment.

Any combination of characters can appear in a comment except a semicolon [Section 2.2.4]. The position of comments in programs is not unreasonably restricted [Sections 2.2 and 2.2.6]. However, comments do not automatically terminate at the end of a line. (Note however that a missing end of comment symbol is detectable because semicolons, i.e., statement delimiters, are not permitted in comments.)

H7. Comment Conventions (PASCAL)

Partially Satisfied.

Comments in PASCAL are demarked by curly brackets, { and }. On systems where they are not available, "the character pairs (* and *) are used in their place" [p. 9].

A comment "may be inserted between any two identifiers, numbers, or special symbols" [p. 9]. But end-of-line in the program text does not automatically terminate a comment.

H7. Comment Conventions (SIMULA 67)

Not Satisfied.

SIMULA 67 has two, rather than one, comment convention [B 360]. Comments are introduced by either the COMMENT or the END keyword and are terminated by a semicolon [B 360]. After the END keyword certain character combinations are forbidden, specifically: END, ELSE, OTHERWISE, and WHEN [B 360]. Comments do not automatically terminate at end-of-line if not otherwise terminated.

H7. Comment Conventions (TACPOL)

Partially Satisfied.

"Comments are permitted wherever blanks are allowed or required in a program... The character pair /* indicates the beginning of a comment and the same characters reversed */ indicate its end" [Section 2-5]. However, comments do not automatically terminate at end-of-line.

H8. Unmatched Parentheses (TINMAN)

The language will not permit unmatched parentheses of any kind.

Some programming languages permit closing parentheses to be omitted. If, for example, a program contained more "BEGINs" than "ENDs" the translator might insert enough "ENDs" at the end of the program to make up the difference. The language will require full parentheses matching. This does not preclude syntactic features such as "case x of s1, s2, ..., sn end case" in which "end" is paired with a key word other than "begin." Nor does it alone prohibit open forms such as "if-then-else-."

H8. Unmatched Parentheses (ALGOL 68)

Satisfied.

Algol 68 requires every opening symbol (e.g., if, begin, (, case, do) to have a closing symbol (e.g. fi, end), esac, od). Multiple closure is not allowed.

H8. Unmatched Parentheses (J3B)

Satisfied.

Implicit closure of compound statements is not permitted in J3B.

H8. Unmatched Parentheses (PASCAL)

Satisfied.

PASCAL does not support multiple closures. This can be verified by direct inspection of the language's syntax charts [pp. 116-118].

H8. Unmatched Parenthesis (SIMULA 67)

Satisfied.

SIMULA 67 does not support multiple closure.

H8. Unmatched Parentheses (TACPOL)

Satisfied.

All parentheses are matched including an END paired with the "four types of blocks; BEGIN, DO, PROC, and CODE" [Section 7-1]. Interestingly TACPOL, through essentially a subset of PL/I, does not permit labeled ENDs and multiple closures.

H9. Uniform Referent Notation (TINMAN)

There will be a uniform referent notation.

There will be no language imposed syntactic distinction between function calls and data selection. This does not preclude the inclusion of more than one referent notation.

H9. Uniform Referent Notation (ALGOL 68)

Partially Satisfied.

Algol 68 uses parentheses for procedure argument lists [Section 5.4.2, 5.4.3] and square brackets or parentheses for array subscript lists [Section 5.3.2, 9.4.1.f]. However, record components (structure fields) are selected by writing [Section 5.3.1] field of structure. The proceduring coercion (see B8) permits a parameterless routine to be called when an empty parameter list is not applied. Hence a variable can be replaced with a function implementation.

H9. Uniform Referent Notation (J3B)

Partially Satisfied.

The extent to which a language supports the uniform referent concept can be determined by answering two general questions: 1) can references to variables be replaced with invocations of subroutines merely by changing the declaration of the identifier representing the variable; and 2) can composite data structures be physically (but not logically) reorganized in a declaration without having to change any references in a program to these data structures or their components (i.e., a data structure's physical organization should not constrain the notation used to access it). J3B partially satisfies these criteria. Although array (table) and subroutine references use a uniform syntactic notation, parameterless function calls must use an empty parameter list enclosed in parentheses [Section 5.3.2], meaning that scalar variable references cannot be replaced with function calls simply by changing the declarations of an identifier. In addition, the uniform referent concept is also violated by J3B's inability to invoke functions as the target of assignment statements; this also implies references to variables cannot uniformly be replaced with function invocations.

H9 is also violated in that although a J3B indexed table is semantically similar to a one dimensional array of records, the method of accessing table entries (i.e., a particular record in the array of records) is not syntactically equivalent to the method of accessing array elements, since the keyword ENTRY must be used when accessing tables [Sections 9.5 and 4.3, p. 4-9], and this keyword cannot be used to access array components. In addition, J3B array elements cannot be accessed with pointers [Section 6], so although a table with a single table item is logically similar to an array, different accessing restrictions are imposed depending on whether the name being referenced is declared as a table or as an array.

Any "serially" organized table [Section 4.3] cannot be organized as a "parallel" table, since multiple word items may not be included in parallel tables [Section 4.3]. Consequently, serial and parallel tables cannot be used interchangeably with full generality. However, for tables not containing multiple word items, the serial/parallel options support the uniform referent concept, since the notation used to access components of such tables does not have to be changed when the organization is changed from serial to parallel (or vice versa).

Similarly, since ordinary tables cannot be passed as parameters, ordinary and specified tables cannot be used interchangeably (even though the only essential difference between them is the degree of control a programmer has over the physical layout of the table). This also violates the uniform referent principle.

On the other hand, a uniform notation is used whether table or table components are accessed with pointer or index values.

H9. Uniform Referent Notation (PASCAL)

Not Satisfied.

Array references use square brackets to enclose the index expressions [p. 147] while function designators have their actual parameter list demarked by parentheses [p. 151]. On the other hand, since parameterless functions can be invoked without supplying an (empty) argument list, a variable can be replaced with a function call. However, an identifier associated with a field of a record cannot be implemented as a function. For example, a tag field's value cannot be computed by a user -defined function as would be possible if the notation Record.tag, could be interpreted as a function call, tag (Record).

The ability to physically reorganize data structures without affecting how they are referenced notationally is also not fully supported in PASCAL. For example, BOX(I).WEIGHT would be written to access an array of records declared as:

```
box:  array [1..100] of record
                        weight:  integer;
                        girth :  integer;
                        end
```

whereas BOX.WEIGHT(I) would have to be written if the structure were reorganized as:

```
box:  record
                        weight:  array [1..100] of integer;
                        girth :  array [1..100] of integer
```

This physical reorganization of elements is supported uniformly, in contrast, by various JOVIAL dialects, where the first organization is called a "serial" table and the second, a "parallel" table. In JOVIAL, WEIGHT(I) would suffice to access components of BOX, whether its organization were serial or parallel, thus supporting the uniform referent concept.

H9. Uniform Referent Notation (SIMULA 67)

Not Satisfied.

SIMULA 67 takes its syntax for arrays and procedures from ALGOL 60. That is, arrays are declared [AR 5.2.1] and referenced [AR 3.1.1] using square brackets. Procedures are declared and invoked using parentheses (AR 3.2.1). For representation using standard printers, parentheses may serve as array brackets but the belief is that "A(I, J) loses much of the clarity of the formal A[I, J]" [B 11].

H9. Uniform Referent Notation (TACPOL)

Satisfied.

Array [Section 4-2], group [Section 4-5], table [Section 4-6], and cell [Section 4-7] component references all have the same form as function procedure calls [Section 9-3, 11-1.) .

H10. Consistency of Meaning (TINMAN)

No language defined symbols appearing in the same context will have essentially different meanings.

In particular, this would exclude the use of = to imply both assignment and equality, would exclude conventions implying that parenthesized parameters have special semantics (as with PL/1 subroutines), and would exclude the use of an assignment operator for other than assignment (e.g., left hand side function call. It would not, however, require different operator symbols for integer, real or even matrix arithmetic since these are in fact special cases of the same abstract operations and would allow the use of generic functions applicable to several data types.

H10. Consistency of Meaning (ALGOL 68)

Partially Satisfied.

In general, Algol 68 is uniform and consistent in its use of symbols with the exception that = is used to represent both the equality operator and to specify the value being assigned an identifier in an identity declaration. Although left hand side procedure calls are allowed (if they return a ref value) [Section 5.2.1], this is entirely consistent with the notion of assignment as supported by Algol 68.

Some of the operators have different meanings, depending on the type of the operand [Section 10.2.3.5.g, 10.2.3.7.5, 10.2.3.8.g, 10.2.4.e]. For example, + denotes concatenation when applied to strings; abs applied to a character converts it to an integer value; the interpretation of up depends on whether it is applied to a bits variable (when it means, shift left) or a semaphore variable.

H10. Consistency of Meaning (J3B)

Not Satisfied.

The = symbol is used for both assignment and equality in J3B [Sections 5.2 and 7.3.].

Real and integer division are both represented by / [Section 7.1], although the results of integer and real division are sufficiently different that different symbols should be used, as in ALGOL.

The colon is used to separate upper and lower bound specifications in table declarations [Section 4.3], to declare labels [Section 5.4.2], and to separate input from output parameters [Section 3.3.2, 3.3.3, 5.3.1 and 5.3.2].

The quote symbol (") is used to delimit comments [Section 2.2.4], macro bodies [Section 3.4], and actual macro parameters [Section 2.2.6].

Assigning to the name of a function is the method used to specify the value to be returned by the function, i.e. a special interpretation of the assignment operation is imposed when the target of the assignment is the name of a function. (Note that this makes it impossible for a compiler to ensure that a function always returns a well defined value at run time [Section 5.2 and 5.3.2].) The letter S has different meanings depending on its context. For example, the first occurrence of S in the declaration below specifies that TT is a serial table; the second occurrence specifies that II is a signed integer variable:

```
TABLE TT (5) S 8;
BEGIN
   ITEM II S 15;
   ...
END;
```

One-origin indexing is used to specify the number of words per table entry [Section 4.3], but zero-origin indexing is used elsewhere in the language, including the syntax for table packing [Section 4.5].

H10. Consistency of Meaning (PASCAL)

Partially Satisfied.

PASCAL substantially satisfies this requirement, but it does have at least one notable form of syntax sharing, namely, X followed by up-arrow denotes a buffer variable if X is of type file and it denotes the variable referenced by X if X is a pointer [p. 148].

H10. Consistency of Meaning (SIMULA 67)

Not Satisfied.

SIMULA 67 supports a PL/I-like "pseudo-variable" facility. For example, if T is a TEXT variable, then T.SUB(7, 2) can be used to denote a substring receiving field or a substring value depending on whether it appears on the left or the right of an assignment [B 183]. Further, in contrast to user defined procedures, "systems defined procedures may not be transmitted as parameters" [CBL 16].

H10. Consistency of Meaning (TACPOL)

Not Satisfied.

TACPOL permits a "substring operation on the left of an assign statement" [Section 6-1.c.(2)(b)]. Thus, the SUBSTR operation delivers a receiving field or a value. Further, the = symbol can stand for the equal relational operator [Section Table 6-2] and also the assignment operator [Section 5-1]. The keyword INIT is used to indicate both that an identifier has a constant value and to define the value [Section 4-9].

I1. No Defaults in Program Logic (TINMAN)

There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

The only alternative is implementation dependent defaults with the translator determining the meaning of programs. What a program does, should be determinable from the program and the defining documentation for the programming language. This does not require that binding of all program properties be local to each use. Quite the contrary, it would, for example, allow automatic definition of assignment for all variables or global specification of precision. What it does require is that each decision be explicit: in the language definition, global to some scope, or local to each use. Omission of any selection which affects the program logic will be treated as an error by the translator.

I1. No Defaults in Program Logic (ALGOL 68)

Satisfied.

I1. No Defaults in Program Logic (J3B)

Not Satisfied.

The following situations are defined to yield undefined effects by the J3B language specification, and these situations are such that the effects may differ for different translators for the same object machine:

- access the value of a loop variable from outside a loop after the loop has been exited [Section 5.6];
- returning from a function without assigning a value to the function [Section 5.4.1];
- executing a switch statement with an out of range value [Section 5.4.1];
- calling a subroutine with a negative integer actual parameter when the corresponding formal parameter is declared to be unsigned [Section 5.3.3];

- comparing a pointer value with NULL in a relational formula using <, <=, >, >= or comparing two pointer formulas that both point outside the table for which they were constructed [Section 7.3];
- using a BIT, BYTE, or INTR function with a negative index value, or specifying an index value and length whose sum is greater than the length of the value being extracted [Section 9.2];
- specifying a negative shift [Section 9.6].

Furthermore, the precision of fixed and floating point data is implementation defined, as is the character code used for character strings.

I1. No Defaults in Program Logic (PASCAL)

Not Satisfied.

The first character of a textfile sent to a printer is used as a printer control character "in some installations" [p. 60]. If a case selector has a value that does not equal any of the case labels, the effect is undefined [p. 31]. "The final value of a [loop] control variable is left undefined upon normal exit from the FOR statement" [p. 24]. "The predecessor of a character ... is undefined if one does not exist" [p. 15]. The effect of jumping into a loop from outside the loop is undefined rather than forbidden [p. 32].

I1. No Defaults in Program Logic (SIMULA 67)

Not Satisfied.

"The effect of text assignment is implementation defined if the left part and right part of the assignment refer to overlapping texts" [CBL 10.6].

I1. No Defaults in Program Logic (TACPOL)

Not Satisfied.

On assignment, "low order bits lost due to conforming to the quantity allocation of an identifier are truncated" [Section 5-2.a]. Thus, assignment of 40960 (= 2**16 + 2**14) to a short numeric identifier gives an actual value of 8192 (= 2**14) even though "the value is said to be undefined" [Section 5-2.a.(2)].

The effect of executing a switch statement with an out of range value is not specified in the reference manual [Section A-7], nor is it specified whether the value of a loop control variable is defined on loop exit.

I2. Object Representation Specifications Optional (TINMAN)

Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does

not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

Defaults will be allowed, in fact, encouraged in don't care situations. Such defaults will include data representations (see J4), open vs. closed subroutine calls (see J5), and reentrant vs. nonreentrant code generation.

I2. Object Representation Specifications Optional (ALGOL 68)

Partially Satisfied.

Algol 68 does not define any specific way for the programmer to specify object representations or override the defaults. However, one can use pragmats [Section 9.2] as implementation dependent compiler directives.

I2. Object Representation Specifications Optional (J3B)

Satisfied.

The J3B "specified" table capability permits programmers to describe how the fields of a record are arranged within the space occupied by the record [Sections 4.3 and 4.5]. A different form of table declaration permits the arrangement of fields to be determined by the compiler, although the programmer may specify general directives defining how space-efficiently the fields are to be packed [Section 4.3]. In addition, the physical arrangement of fields in an array of records (a table) is normally "serial"; this arrangement can be changed to "parallel" *without* changing the logical properties of the table [Section 4.3].

Whether a subroutine is compiled *as open* is *not* determined by default; the programmer must specify [Section 3.3.4]. In addition, reentrant code generation must be specified explicitly [Sections 3.3.2 and 3.3.3].

The physical organization of arrays in storage is prescribed explicitly in the language specification [Section 4.4], but there is no capability for programmers to specify a different physical organization.

The physical ordering in storage of variables, arrays, and tables is normally determined in an implementation dependent manner. The OVERLAY statement, however, may be used to physically reorder the arrangement of these objects to insure, for example, that certain addressing optimizations may be performed.

I2. Object Representation Specifications Optional (PASCAL)

Partially Satisfied.

Structured types, such as arrays and records, can be optionally designated as packed. "This has no effect on the meaning of a program, but it is a hint to the compiler that storage should be economized een at the price of some loss in efficiency of access" [p. 143]. However, object representation specifications for open vs. closed subroutine calls or for reentrant code are not available to the PASCAL programmer. And "a component of a packed structure must not appear as an actual variable parameter" [p. 83]. Interestingly, PASCAL 6000 has an option (and default) for using registers to pass parameters which improves code quality but runs the risk of a "running out of registers" error message [p. 101].

I2. Object Representation Specifications Optional (SIMULA 67)

Not Satisfied.

The SIMULA 67 user cannot optionally override the default representation of class objects.

I2. Object Representation Specifications Optional (TACPOL)

Partially Satisfied.

With arrays "the coder may change the normal aligned allocation for arrays to a packed allocation by using the PACKED option in the array declaration" [Section 4-3]. Groups, tables, and cells are normally PACKED, but this can be changed "by using the ALIGNED option" [Section 4-5.6].

However, open subroutines cannot be specified.

I3. Compile Time Variables (TINMAN)

The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

When a language has different host and object machines and when its compilers can produce code for several configurations of a given machine, the programmer should be able to specify the intended object machine configuration. The user should have control over the compile time variables used in his program. Typically they would be associated with the object computer model, the memory size, special hardware options, the operating system if present, peripheral equipment or other aspects of the object machine configuration. Compile time variables will be set outside the program, but available for interrogation within the program (see I4 and C4).

I3. Compile Time Variable (ALGOL 68)

Partially Satisfied.

Algol 68 does provide a variety of environment enquiry capabilities [Section 10.2.1], e.g. max real is defined as the largest value of type real. Implementations may provide additional environment information in a library prelude [Section 10.1]. The specified set of environmental enquiries does not, however, include object computer model, memory size, etc.

I3. Compile Time Variables (J3B)

Not Satisfied.

There is no provision for environmental enquiries in J3B, although a variety of machine dependent parameters are specified in the language specification [Section 1.3.1].

I3. Compile Time Variables (PASCAL)

Not Satisfied.

PASCAL was designed and defined "without reference to any particular machine in order to facilitate the interchange of programs" [p. 168]. Nothing about the intended object machine configuration is programmer specifiable. An implementation standard identifier, maxint, can be used within a program [pp. 13-14] and is the only such environmental enquiry available in the language.

I3. Compile Time Variables (SIMULA 67)

Not Satisfied.

No provisions for environmental enquiries exist.

I3. Compile Time Variables (TACPOL)

Not Satisfied.

TACPOL as defined in the language reference manual has a single object computer, the AN/GYK-12 [Section 7-5]. So, the language does not have compile time variables.

I4. Conditional Compilation (TINMAN)

The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and

other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.

An environmental inquiry capability permits the writing of common programs and procedures which are specialized at compile time by the translator as a function of the intended object machine configuration or of other compile time variables (see I3). This requirement is a special case of evaluation of constant expressions at compile time (see C4). It provides a general purpose capability for conditional compilation.

I4. Conditional Compilation (ALGOL 68)

Not Satisfied.

Algol 68 does not define any conditional compilation capability. An implementation could use pragmats [Section 9.2] to provide it.

I4. Conditional Compilation (J3B)

Partially Satisfied.

If the predicate in a conditional statement is composed of bit constants [Section 8.3], the predicate is evaluated at compile time and object code is only generated for the THEN branch of the statement or the ELSE branch, if one is present [Section 5.5]. However, any macro definitions contained in the THEN or ELSE branches are processed regardless of the value of the predicate, i.e., conditional compilation of macro definitions is not possible [Section 5.5].

It should be noted, however, that the generality of this form of conditional compilation is limited in J3B since relational expresssions are not evaluated at compile time (an oversight in the language definition) [Section 8.3] and, since declarations must appear before executable statements [Section 3.1.1], declarations cannot be conditionally compiled. When subroutines are compiled in-line, conditional compilation is possible if some of the subroutine's actual parameters are constant [Section 3.3.4].

The term "conditional statements" in the TINMAN statement of I4 presumably rules out the conditional non-compilation of loop bodies when the predicate controlling loop iteration is a constant evaluable at compile time. J3B does not support this form of conditional compilation, although there appears to be no reason why it should not be supported.

I4. Conditional Compilation (PASCAL)

Not Satisfied.

Conditional compilation is not a PASCAL capability.

I4. Conditional Compilation (SIMULA 67)

Not Satisfied.

SIMULA 67 contains no conditional compilation capability.

I4. Conditional Compilation (TACPOL)

Not Satisfied.

TACPOL does not support a conditional compilation capability.

I5. Simple Base Language (TINMAN)

The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capablity not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

Only the base need be implemented to make the full source language capability available.

Base features will provide relatively low level general purpose capabilities not yet specialized for particular applications. There will be no prohibition on a translator incorporating specialized optimizations for particular extensions. Any extension provided by a translator will, however, be definable within the base language using the built-in definition facilities. Thus, programs using the extension will be translatable by any compiler for the language but not necessarily with the same object efficiency.

I5. Simple Base Language (ALGOL 68)

Partially Satisfied.

One of the design goals of Algol 68 is orthogonality [Section 0.1.2], i.e. a small number of independent concepts that may be arbitrarily combined. The Revised Report defines a base language and a standard environment [10] that defines the specific types, operators, and procedures generally available. For example, the addition operator is defined by extension, although implementations will of course treat addition as a built-in operator, to improve "object efficiency." Many of these procedures can be implemented using a subset of Algol 68, e.g., arithmetic functions like sin and some of the I/O procedures.

Despite the orthogonality and consistency of Algol 68, it contains several features not needed to satisfy TINMAN requirements (other than I5). Among these are: subroutine variables, alternate notations for case and conditional statement, formatted I/O, complex coercion rules, conditional and case expressions, array trimscripts (e.g., A(i:j@k)), valued assignment statements (e.g., X[(j := i)]), flexible arrays, the bits and bytes data types, etc. In short, Algol 68 contains more generality than the TINMAN requires (or permits).

I5. Simple Base Language (J3B)

Partially Satisfied.

Although J3B does not satisfy the concept of a kernel language with user-definable extensions, it is a simple language in terms of its variety of data types, control structures, and assumptions about its run-time environment. But to conform to the TINMAN requirements for simplicity, the absolute value function would have to be removed from the language (and be reintroduced via extension capabilities not presently available, since ABS is a generic function; i.e., it takes different types of arguments as its input parameter and returns a corresponding result type. Such properties of functions cannot be specified within J3B.), and the character string data type would have to be removed (so character strings could be supported as arrays of characters). It may also be that the J3B macro facility [Sections 2.2.6 and 3.4] should be dispensed with to satisfy I5. These, however, appear to be the only J3B capabilities that need to be deleted because of simplicity considerations. Other features, of course, need to be modified to conform to other requirements, and many features need to be added. But the fact that relatively few features violate I5's requirements for simplicity confirm the evaluation that J3B essentially satisfies I5.

I5. Simple Base Language (PASCAL)

Satisfied.

PASCAL, unlike ALGOL 68, is not usually thought of in kernel language/extended language terms. But its modest size (e.g., a self compiler consisted of only 4000 PASCAL source statements - Wirth, SPE, p. 321) and its data type definition facility permit it to be regarded as a reasonable base language. Certain procedures and functions are, of course, standard and are in effect predeclared in every implementation as are the two textfiles, input and output [pp. 160-167]. Particular definitions/extensions can be oriented to a specific environment and yet can be fully expressible within standard PASCAL. For example, PASCAL 6000 uses a predefined type "alfa" (which is a packed array of 10 characters) because on CDC 6000 series machines "a value of type alfa is representable in exactly one word" [p. 97].

I5. Simple Base Language (SIMULA 67)

Partially Satisfied.

The language is designed as a Common Base Language that defines the features common to all implementations. This base or "substrate" can be extended by defining new types with their accompanying operations [CBL 1.2]. That is, "SIMULA 67 may be oriented towards a special application area by defining a suitable class containing the necesary problem-oriented concepts. This class is then used as a prefix to the program by a user interested in the problems in question" [CBL 1.3.4]. However, the base language is not minimal. For example, the built-in type text provides for character strings and their manipulation [CBL 10]. This is done rather than synthesizing character strings from arrays of single characters. Indeed, character in SIMULA 67 is a distinct type from text [CBL 3.1].

I5. Simple Base Language (TACPOL)

Not Satisfied.

TACPOL is a reasonably small language and does have most numeric functions such as absolute value, maximum, minimum, and remainder as built-in procedures [Section A-2, A-3] instead of as language operators. But TACPOL has a character string data type [Section 3-3.a] with an infix catenation operator CAT [Section 6-1.c.(1)] rather than synthesizing these features out of arrays of single characters. Multiple target assignments are supported [Section 5-1.d]. This too violates I5's simple base requirement.

I6. Translator Restrictions (TINMAN)

Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

Limits on the number of array dimensions, the length of identifiers, the number of nested parentheses levels in expressions, or the number of identifiers in programs are determined by the translator and not by the object machine. Ideally, the limits should be set so high that no program (save the most abrasive) encounters the limits. They should be small enough that they do not dominate the compiler and large enough that they do not interfere with the usefulness of the language. If they were set at say the 99 percent level based on statistics from existing DoD computer programs the limits might be a few hundred for numbers of identifiers and less than ten in the other cases mentioned above.

I6. Translator Restrictions (ALGOL 68)

Not Satisfied.

Algol 68 does not define any translator limits. In Algol 68, identifier lengths, number of dimensions, nesting levels, number of identifiers, etc. may be arbitrarily large.

I6. Translator Restrictions (J3B)

Partially Satisfied.

Limits on the number of array dimensions are specified explicitly in the J3B language definition [Section 4.4.]. (Up to 3 dimensions are permitted.) Limits on identifier length, character string length, and fixed point scale factors are also specified [Sections 2.2.1 and 1.3.1]. Other limits on the size of program that can be compiled, the number of identifiers permitted, etc., are not specified, however.

I6. Translator Restrictions (PASCAL)

Not Satisfied.

Translator restrictions such as identifiers "must differ over their first 8 characters" and labels have "at most 4 digits" are stated as part of the definition for standard PASCAL [p. 168]. But some more important translator restrictions are not explicitly specified in the defining documents. For example, PASCAL 6000 produces error indicators for: too many cases in case statement, too many nested scopes, too many forward references, procedure code is too long, too many local files, and expression too complicated [pp. 120-121]. Standard PASCAL addresses none of these constraints.

I6. Translator Restrictions (SIMULA 67)

Not Satisfied.

The SIMULA 67 language definition does not specify such restrictions except for "not allowing an IF-statement to follow a THEN" [B 361]. For system classes the definition merely acknowledges that "an implementation may restrict the number of block levels at which such prefixes or block prefixes may occur in any one program" [CBL 14].

I6. Translator Restrictions (TACPOL)

Partially Satisfied.

Separating translator and object machine restrictions is not easy because the language reference manual defines the language as it is implemented on a single object machine. Only some translator restrictions, such as the number of array dimensions [Section 4-2] and identifier abbreviations [Section 2-6.g] besides maximum size for bit and character strings [Section 3-5.c,d], are made explicit. For example, maximum permissible nesting depth for blocks and expressions are not stated.

I7. Object Machine Restrictions (TINMAN)

Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

Limits on the amount of run time storage, access to specialized peripheral equipments, use of special hardware capabilities and access to real time clocks are dependent on the object machine and configuration. The translator will report when a program exceeds the resources or capabilities of the intended object machine but will not build in arbitrary limits of its own.

I7. Object Machine Restrictions (ALGOL 68)

Partially Satisfied.

Algol 68 does not define any object machine restrictions [Section 2.2]. However it does not prevent an implementation from doing so.

I7. Object Machine Restrictions (J3B)

Not Satisfied.

Language restrictions dependent on the object machine are explicitly specified as part of the J3B language definition [Section 1.3.1]. These include the maximum range of representable integer, floating, and fixed point values, the number of bits occupied by floating and fixed representations, the floating and fixed point precisions supported, the word length of the object computer (which affects the maximum bit string length permitted), the number of bits occupied by a pointer value, the number of bits occupied by a character, and the maximum number of words per record.

I7. Object Machine Restrictions (PASCAL)

Partially Satisfied.

No PASCAL restrictions are inherently dependent only on the object environment. But all program file processing (including all input and output) is restricted to sequential files [pp. 145, 161, 164-167]. So, a major restriction on the range of flexibility in using the object environment is defined into the language.

I7. Object Machine Restrictions (SIMULA 67)

Partially Satisfied.

SIMULA 67 does not provide for "access to real time clocks" nor for any parallel processing capability. These language restrictions limit the range

or the use of possible object environments. The class concept could be used to encapsulate access to real time clocks [CBL 1.3.4].

I7. Object Machine Restrictions (TACPOL)

Satisfied.

The TACPOL reference manual defines TACPOL as it is to be implemented for a particular computer, the AN/GYK-12. No concern is given for making TACPOL efficient on other computers. If efficiency considerations are ignored, TACPOL is quite free of object machine restrictions.

J1. Efficient Object Code (TINMAN)

The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

The language should not force programs to require greater generality than they need. When a program does not use a feature or capability it should pay no run time cost for the feature being in the language or library. When the full generality of a feature is not used, only the necessary (reduced) cost should be paid. Where possible, language features (such as, automatic and dynamic array allocation, process scheduling, file management, and I/O buffering) which require run time support packages should be provided as standard library definitions and not as part of the base language. The user will not have to pay time and space for support packages he does not use. Neither will there be automatic movement of programs or data between main storage and backing storage which is not under program control. Language features will result in special efficient object code when their full generality is not used. A large number of special cases should compile efficiently. If the base language components are easily composable they can be used to construct the specialized structures needed by specific applications, if they are simple and provide a single capability they will not force the use of unneeded capabilities in order to obtain needed capabilities, and if they are compatible with the features normally found in sequential uniprocessor digital computers with random access memory they will have near minimum or at least low cost implementation on many object machines.

J1. Efficient Object Code (ALGOL 68)

Not Satisfied.

Most of Algol 68 can be efficiently implemented[Section 0.1.4]. For example, the I/O and file management is all done by standard procedures [Section 10.3]. However, Algol 68 provides dynamic storage allocation (stack and garbage-collected) [Section 5.2.3] as part of the base language, along with nested recursive procedures [Section 5.4.1] and parallel processes [Section 3.3]. Garbage collection requires that additional information about

pointers and types be maintained at run time. In addition, string data (flexible arrays of characters) cannot be efficiently stored on a stack; even if a programmer wishes to specify an upper bound on string lengths, he cannot do so. Furthermore, because procedures can exit to a containing scope via a GOTO, with the possibility of terminating an unknown (at compile time) number of intervening subroutine calls, an efficient means of terminating a routine with an exit to its caller cannot be realized (i.e., implementing label parameters as procedures complicates an already complicated problem).

J1. Efficient Object Code (J3B)

Partially Satisfied.

J3B imposes a minimal burden in terms of run-time support packages, which consist primarily of subroutines for manipulating character and bit strings, exponentiation, and manipulation of table entries. J3B in its use for producing avionics applications programs has proven capable of producing sufficiently efficient object code for these applications. J3B compilers provide a number of register allocation and subroutine linkage conventions to permit efficient code to be generated.

There is no support provided for programmer control of program and data movement between main and backing store.

J1. Efficient Object Code (PASCAL)

Partially Satisfied.

The file handling and the dynamic (heap) allocation language features are provided as standard predeclared procedures [pp. 160-161]. Hence they can be implemented as run time support packages which are included only if needed. PASCAL does not support process scheduling or dynamic arrays. There is no program control of the automatic movement of programs or data between main store and backing store. PASCAL "was designed so that implementation of most of its constructs should be possible without a need for extensive optimization processes on conventional computers" [Wirth, SPE, p. 316].

J1. Efficient Object Code (SIMULA 67)

Not Satisfied.

SIMULA 67 does not provide for programmer control over overlays. Garbage collection occurs automatically and cannot be invoked by the programmer [CBL 9.1]. The SIMULA 67 language and compilers have been designed so that "only those runtime routines needed are included" [Structure, p. 6]. Class concatenation yields, in effect, variant record structures that occupy more space than if the programmer could specify the record structure directly, because type discrimination fields are included in the record and their position and coding are not under program control (Palme, J., "List structures

in SIMULA and PL/I -- a comparison." Software- Practice and Experience, volume 4, no. 4 (Oct.-Dec. 1974), pp. 379-399). Moreover, since records can only be heap allocated, time and space must be taken at run time to create class instances that could, as far as the programmer is concerned, be allocated and initalized prior to run time. Text data must be allocated on the heap (even for what would be local variables in another language) because of the ability to assign strings of arbitrary length to the same text variable.

J1. Efficient Object Code (TACPOL)

Partially Satisfied.

"TACPOL has been designed to restrict the use of those source language elements which could easily generate large volumes of machine language code" [Section 1-6]. As a result, the programmer "does not have to be concerned with implicitly generating large volumes of machine language code." TACPOL does make it possible to generate efficient object code. It provides partial program control over the movement of programs or data between main and backing store with the LOAD statement [Section 13-15] whose execution causes a designated program to be loaded into memory.

J2. Safe and Effective Optimization Possible (TINMAN)

Any optimizations performed by the translator will not change the effect of the program.

The number of applicable safe optimizations can be increased by making more information available to the compiler and by choosing language constructs which allow safe optimizations. This requirement allows optimization by code motion providing that motion does not change the effect of the program.

J2. Safe and Effective Optimization Possible (ALGOL 68)

Partially Satisfied.

The Algol 68 Revised Report defines explicitly and precisely what action, effects, and properties must be satisfied by an implementation [Section 2.2]. All else is left open to possible optimization. But collateral elaboration of expressions (see C1) permits optimizations to change the result of executing a program, and lack of adequate information about a subroutine's possible side-effects prohibits some optimizations that would be safe if a compiler had additional information. On the other hand, the requirement that pointers be typed permits some optimization that would otherwise be impossible.

J2. Safe and Effective Optimization Possible (J3B)

Partially Satisfied.

Because the order in which operands of arithmetic expresssions are evaluated is not fully specified [Section 7.1], optimizations can change the effect of a program and still satisfy the language specification. Similarly, since short circuit evaluation of Boolean expresssions is explicitly permitted, but not required [Section 7.3], the results of performing or not performing short circuit evaluation can change the effect of program executions in some cases and still satisfy the language specification.

The provision of typed pointers, on the other hand, permits more effective code optimization because an assignment through a pointer can modify only a restricted set of locations. Consequently, the compiler can determine that the contents of certain registers do not need to be updated after the asssignment. Similarly, the unsigned integer data type provides implicit range information to a compiler that can be used in optimizing programs.

J2. Safe and Effective Optimization Possible (PASCAL)

Satisfied.

Wirth has co-authored with C.A.R. Hoare an axiomatic definition of PASCAL which requires that language features have their defined meaning regardless of implementation and optimization approaches. The packed prefix [p. 43], the subrange type [p. 143], and the dispose procedures [p. 161] are examples of language features which present opportunities for, but do not require, (storage) optimizations. But this does not ensure that an optimized program will produce unchanged effects if the difference in effect is permitted by the language definition. In this sense, PASCAL does not satisfy J2 completely, since (for example) operands in expressions may be evaluated in different orders, even if some operands are functions having side effects, and this can lead to different (but valid) results. On the other hand, code motion eliminating some function calls cannot generally be safely performed because a compiler does not have adequate information about the subroutine's possible side effects.

J2. Safe and Effective Optimization Possible (SIMULA 67)

Partially Satisfied.

"The Common Base comprises the language features required in every SIMULA 67 compiler" [CBL Preface]. Optimizations are not directly addressed in the defining document, but by inference any optimizations are required to conform to the Common Base Language semantics. But since SIMULA does not constrain the order in which operands of expressions are evaluated (see C1), optimizations yielding different evaluation orderings can produce different results and still conform to the language specifications. Similarly, lack of adequate information about a subroutine's possible side effects prohibits some optimizations that would otherwise be safe.

J2. Safe and Effective Optimization Possible (TACPOL)

Not Satisfied.

For example, the default packing of structures - groups, tables, cells - [Section 4-4] can be changed by using the ALIGNED option that optimizes access at the expense of storage. This packed or aligned choice does not affect the rest of the program code except if the structure is copied with the MOVE operator, which does not check packing (or other structural attributes) for consistency between the source and target of the MOVE operator.

J3. Machine Language Insertions (TINMAN)

The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

The ability to enter machine language should not be used as an excuse to exclude otherwise needed facilities from the HOL; the abstract description of programs in the HOL should not require the use of machine language insertions. The semantics of machine language insertions will be determinable from the HOL definition and the object machine description alone and not be dependent on the translator characteristics Machine language insertions will be encapsulated so they can be easily recognized and so that it is clear which variables and program identifiers are accessed within the insertion. The machine language insertions will be permitted only within the body of compile time conditional statements (see I4) which depend on the object machine configuration (see I3). They will not be allowed interspersed with executable statements of the source language.

J3. Machine Language Insertions (ALGOL 68)

Not Satisfied.

Algol 68 does not define a mechanism for machine language insertions, nor for compile time conditional statements. An implementation may use pragmats [Section 9.2] for this purpose, however.

J3. Machine Language Insertions (J3B)

Not Satisfied.

J3B makes no provision for incorporating assembly code into programs, either directly or in encapsulated form, although an implementation may support a special set of functions that give access to some machine dependent capabilities, including specific machine instructions [Section 3.2.1]. For example, HOL programmers can be given access to machine level I/O instructions in this way. But this method of providing access to assembly code is not as general or as disciplined as the TINMAN requires.

J3. Machine Language Insertions (PASCAL)

Not Satisfied.

Machine language code insertions and compile time conditional statements are not part of PASCAL and they violate the language's design philosophy [p. 136].

J3. Machine Language Insertions (SIMULA 67)

Not Satisfied.

Not supported in the SIMULA 67 Common Base Language.

J3. Machine Language Insertions (TACPOL)

Partially Satisfied.

TACPOL allows machine language insertions via CODE blocks. The machine code is bracketted by CODE and END particles. "Names defined in TACPOL blocks are not normally known to CODE blocks" [Section 7-5.a]. The USES particle followed by a list of previously defined names "marks the names in the list known in the CODE block" [Section 7-5.b]. However, these CODE blocks do not appear within the body of compile time conditional statements. Moreover, examination of some actual TACPOL program shows that the USES particle is not actually required in at least some cases.

J4. Object Representation Specifications (TINMAN)

> It will be possible within the source language to specify the object representation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

It will be possible to specify the order of the fields, the width of fields, the presence of don't care fields, and the position of word boundaries. It will be possible to associate source language identifiers (data or program) with special machine addresses. The use of machine dependent characteristics of the object representation will be restricted as with machine dependent code (see J3).

If the object representation of a composite data object is not specified in the source program, there will be no specific default guaranteed by the translator.

J4. Object Representation Specifications (ALGOL 68)

Not Satisfied.

Algol 68 does not define a mechanism for specifying object representations of data. The programmer can only specify logical representations [Section 4.6]. There are no packed or aligned attributes. However, an implementation may use pragmats [Section 9.2] for this purpose. Algol 68 leaves the choice of object representation to the translator.

J4. Object Representation Specifications (J3B)

Partially Satisfied.

J3B's "specified" table construct permits programmers to specify the object representation of records and arrays of records (i.e., tables). However, the syntax for specifying this representation is not separated from the description of the structure's logical properties [Sections 4.3 and 4.5].

Similarly, the packing specifications supported for "ordinary" table declarations [Section 4.3] permit a programmer to specify space/time tradeoffs in a general way, leaving the specific organization of a record's fields to the compiler.

Source language identifiers cannot be associated machine addresses.

The BIT and BYTE operators provide access to the object machine representation of any data value. Dependence on object machine representation is indicated when these operators are applied to other than bit or character strings, respectively.

It is not possible to specify the presence of "don't care" fields.

The serial and parallel forms of table structure are generic ways of specifying the object representation of table [Section 4.2].

J4. Object Representation Specification (PASCAL)

Not Satisfied.

The PASCAL programmer cannot specify the object representation of structured types. The prefix "packed" signifies a wish to economize on space at the expense of time [p. 143]. But this wish might be ignored by the implementator.

Object machine addresses cannot be associated with program identifiers.

J4. Object Representation Specification (SIMULA 67)

Not Satisfied.

There is no provision in SIMULA 67 for specifying object representations but the class concept, if extended to include this capability, would serve to encapsulate or "hide" the machine dependencies.

J4. Object Representation Specifications (TACPOL)

Partially Satisfied.

The TACPOL user has the (relatively) machine independent choice of PACKED or ALIGNED for the object representation of composite data structures [Section 4-4]. The choice applies to the entire structure and cannot be used more selectively. Thus, specification at the level of word boundaries for components of PACKED structures is not possible. Nor can special machine addresses be associated with source language identifiers.

J5. Open and Closed Routine Calls (TINMAN)

The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.

Open routine capability is especially important for machine language insertions.

The distinction between open and closed implementation of a routine is an efficiency consideration and should not affect the function of the routine. Thus, an open routine will differ from a syntax macro in that (a) its global environment is that of its definition and not that of its call and (b) multiple occurrences of a formal value (i.e., read only) parameter in the body have the same value. If a routine is not specified as either open or closed the choice will be optimal as determined by the translator.

J5. Open and Closed Routine Calls (ALGOL 68)

Not Satisfied.

Algol 68 does not define a way to specify whether calls of a routine are to be open or closed. However, an implementation may use pragmats [Section 9.2] for this purpose.

J5. Open and Closed Subroutine Calls (J3B)

Satisfied.

The J3B INLINE directive, when applied to the name of a subroutine, specifies to a compiler that that subroutine is to be compiled in open form [Section 3.3.4]. If actual parameters are constants, and the substitution of actual for formal parameters yields expressions that are evaluable at compile time in J3B (see C4) [Section 8], these expressions are evaluated before object code is generated. Except for efficiency effects, the INLINE and closed form of a subroutine are required to have the same semantic effect [Section 3.3.4].

J5. Open and Closed Routine Calls (PASCAL)

Not Satisfied.

The PASCAL programmer cannot specify whether calls will have an open or closed implementation. This important capability is left exclusively to the translator. Adding this feature to the language would not impact the other features.

J5. Open and Closed Routine Calls (SIMULA 67)

Not Satisfied.

The programmer cannot specify an open or closed implementation for routine calls.

J5. Open and Closed Routine Calls (TACPOL)

Not Satisfied.

The TACPOL programmer cannot specify whether procedure calls have an open or closed implementation.

K1. Operating System Not Required (TINMAN)

The language will not require that the object machine have an operating system. When the object machine does have an operating system or executive program, the hardware/operating system combination will be interpreted as defining an abstract machine which acts as the object machine for the translator.

K1. Operating System Not Required (ALGOL 68)

Partially Satisfied.

Algol 68 does not require an operating system. However, some of its features, e.g. I/O, need run time support routines drawn from a library.

K1. No Operating System Required (J3B)

Satisfied.

J3B programs make no demands on operating system capabilities except that the file names referenced in COMPOOL and COPY directives must conform to the length and identifier conventions of the host operating sytem [Sections 3.2.2.5 and 3.1.1]. This requirement, however, does not impact the semantics of object programs, and so does not violate K1.

K1. Operating System Not Required (PASCAL)

Satisfied.

Program input and output using file data types requires buffering and administration of secondary storage [pp. 145, 161, 164-167], but this does not necessitate a full operating system. Interestingly, in the initial compiler implementation effort, interfacing with the operating system was one of the three principal development tasks [Wirth, SPE, p. 323].

K1. Operating System Not Required (SIMULA 67)

Partially Satisfied.

None of SIMULA 67's features per se require an operating system. However, the language definition does assume the compiler exists in an operating system environment. For example, under the separate compilation category it is noted that "an external identifier is an identification with respect to an 'operating system' of a separately compiled declaration" [CBL 15.2].

K1. Operating System Not Required (TACPOL)

Satisfied.

TACPOL has no feature which would require a full operating system to support; no mention of a supporting operating system is made in the language reference manual. TACPOL is intended, in part, to help develop and maintain the operating system for TACFIRE software [Section 1-2.b]. On the other hand, the discussion of I/O mentions partitioned data sets, a capability not supported in all environments TACPOL might be used in. The LABELLED particle [Section 14-3] specifies a "file is to be processed with standard header and trailer labels." This appears to presume the existence of operating system conventions, at least. Moreover, the OPEN statement [Section 13-4] specifies whether a file is NEW, OLD, KEEP, or PASS, which appears to presume the particular file management capabilities of a well known operating system.

K2. Program Assembly (TINMAN)

The language will support the integration of separately written modules into an operational program.

Separately written modules in the form of routines and type definitions are necessary for the management of large software efforts and for effective use of libraries. The user will be able to cause anything in any accessible library to be inserted into his program. This is a requirement for separate definition but not necessarily for separate compilation. The decision as to whether separately defined program modules are to be maintained in source or object language form is a question of implementation efficiency, will be a local management option and will not be imposed by the language definition. The trade-offs involved are complicated by other requirements for type checking of parameters (see C6), for open subroutines (see J5), for efficient object representations (see J1), and for constant expression evaluation at compile time (see C4).

K2. Program Assembly (ALGOL 68)

Not Satisfied.

Algol 68 does not explicitly define a mechanism for integrating separately written modules, although an implementation may use pragmats [Section 9.2] for copying code from a library or for specifying separate compilation, etc. For example,

pr copy filename from libraryname pr

K2. Program Assembly (J3B)

Satisfied.

J3B supports (but does not require) the separate compilation of programs [Section 3.1], and the COPY directive can be used to insert source language modules into programs being compiled [Section 2.2.5]. The COMPOOL directive is used to access files of declarations for use in compiling a program [Sections 3.1.2 and 3.1.3]. The two directives together suffice to cause anything in an accessible "library" to be inserted into a program. In addition, COMPOOL declarations of subroutines are checked against the actual subroutine definition for accuracy [Section 3.3.2, p. 3-18, errors 8 and 9].

K2. Program Assembly (PASCAL)

Not Satisfied.

No language features support the combining of independently written modules. The original PASCAL compiler, which has served as the basis for subsequent ones, used a one-pass scheme and generated absolute code.

Compilation speed was gained by avoiding a relocating loader. Wirth believes that "merging of programs should occur at the source language level" [Wirth, SPE p. 317]. However, PASCAL 6000 allows the integration of object code modules as called procedures or functions [p. 101]. A constraint on separately written type definitions is that a given identifier can appear in at most one enumeration of a scalar type [p. 34].

K2. Program Assembly (SIMULA 67)

Partially Satisfied.

The SIMULA 67 definition makes separate compilation optional rather than mandatory. "If an implementation permits user defined procedure and class declarations to be separately compiled, then a program should have means of making reference to such declarations as external to the program" [CBL 15]. But support for integration of modules is not provided as a mandatory part of the language.

K2. Program Assembly (TACPOL)

Satisfied.

Through the integration of separately defined program modules is not addressed in the TACPOL manual, the language has a Compool capability. The Compool contains programs and data "that are commonly used by many different programs in a system" [Section 14-1].

K3. Software Development Tools (TINMAN)

A family of programming tools and aids in the form of support packages including linkers, loaders and debugging systems will be made available with the language and its translators. There will be a consistent easily used user interface for these tools.

K3. Software Development Tools (ALGOL 68)

Not Determinable.

The Algol 68 Revised Report does not discuss software tools. The language is appropriate for constructing such tools. Some of the current Algol 68 compilers are coded in Algol 68.

K3. Software Development Tools (J3B)

Not Determinable.

No software development tools are specified within the J3B language specification.

K3. Software Development Tools (PASCAL)

Not Determinable.

Defining documents do not address the tools requirement. Compilers generating absolute code (as most present PASCAL compilers do) are not strictly compatible with linkers and loaders [Wirth, SPE, p. 317]. Wirth downplays debugging systems, favoring instead a programming language that facilitates program formation with automatic (largely compile-time) consistency checks [Wirth, p. 197].

K3. Software Development Tools (SIMULA 67)

Not Determinable.

Although a representative SIMULA 67 implementation includes diagnostic routines and additional routines for user convenience [Structure, p. 6], such tools are not part of the language specification. In particular, "a consistent easily used user interface" is not defined.

K3. Software Development Tools (TACPOL)

Not Determinable.

The TACPOL documentation does not specify support tools to be made available with the language. In fact, TACPOL was designed "for use in developing the TACFIRE software" [Section 1-1]. This includes a range of aids and diagnostic programs [Section 1-2].

K4. Translator Options (TINMAN)

> A variety of useful options to aid generation, test, documentation and modification of programs will be provided as support software available with the language or as translator options. As a minimum these will include program editing, post-mortem analysis and diagnostics, program reformatting for standard indentations, and cross-reference generation.

There will be special facilities to aid the generation, test, documentation and modification of programs. Tools and debugging aids will be source language oriented.

Some of the translator options which have been suggested and may be useful include the following. Code might be compiled for assertions which would give run time warnings when the value of the assertion predicate is

false. It might provide run time tracing of specified program variables. Dimensional analysis might be done on units of measure specifications. Special optimizations might be invoked. There might be capability for timing analysis and gathering run time statistics. There might be translator supplied feedback to provide management visibility regarding progress and conformity with local conventions. The user might be able to inhibit code generation. There might be facilities for compiling program patches and for controlling access to language features. The translator might provide a listing of the number of instructions generated against corresponding source inputs and/or an estimate of their execution times. It might provide a variety of listing options.

K4. Translator Options (ALGOL 68)

Partially Satisfied.

Algol 68 does not define any translator options. It does allow an implementation to use pragmats [Section 9.2] to specify options, e.g.

pr crossref, attributes, trace, object list pr

K4. Translator Options (J3B)

Not Determinable.

Translator options are not specified in the J3B language specification, although actual J3B compilers provide cross reference listings and a variety of optimization options, including register allocation strategies and linkage conventions.

K4. Translator Options (PASCAL)

Not Determinable.

Defining documents do not specify standard translator options, but PASCAL 6000, for example, offers options to include important run-time tests, to obtain a complete post-mortem dump that is source code oriented, etc [pp. 100-103]. Users have complained about "the spartan compilation listing produced by the compiler" [PASCAL Newsletter, No. 3, February 1975, SIGPLAN Notices, 11, 2, February 1976, p. 38].

K4. Translator Options (SIMULA 67)

Not Determinable.

No translator options are suggested or required for SIMULA 67's Common Base Language. However, standard SIMULA 67 compilers often produce cross reference listings [Structure, p. 4] and provide "several options to improve the printout of the source program" [Structure, p. 5].

K4. Translator Options (TACPOL)

Partially Satisfied.

TACPOL provides condition declarations as a debugging aid [Section 12-1]. Conditions that arise and have been declared by CHECK produce a diagnostic snapshot. Besides zerodivide and fixed overflow checking, extensive execution tracing can be done for assignments, procedure calls, and GOTOs by specifying the appropriate names with the USAGE particle [Section 12-2]. A TACPOL compiler outputs an attribute and reference list, cross reference listings, and set/used information besides source and object code [Section D-1]. Editors, formatters, and post- mortem programs are not mentioned.

K5. Assertions and Other Optional Specifications (TINMAN)

The source language will permit inclusion of assertions, assumptions, axiomatic definitions of data types, debugging specifications, and units of measures in programs. Because many assertional methods are not yet powerful enough for practical use, nor sufficiently well developed for standardization, they will have the status of comments.

The language will not prohibit inclusion of these forms of specification if and when they become available for practical use in programs. Assertions, assumptions, axiomatic definitions and units of measure in source language programs should be enclosed in special brackets and should be treated as interpreted comments -- comments which are delimited by special comment brackets and which may be interpreted during translation or debugging to provide units analysis, verification of assertions and assumptions, etc. -- but whose interpretation would be optional to translator implementations.

K5. Assertions and Other Optional Specifications (ALGOL 68)

Satisfied.

Algol 68 provides a pragmat mechanism [Section 9.2] which meets this requirement exactly. Pragmats are comments, but are delimited by pr instead of co so that a translator may easily recognize them and optionally treat them as compile time directives, in particular to specify assertions.

K5. Assertions and Other Optional Specifications (J3B)

Not Satisfied.

There is no provision for special-comment brackets for including assertions or units measures, etc. in programs in a manner that can be ignored by translators not prepared to process these constructs.

K5. Assertions and Other Optional Specifications (PASCAL)

Not Satisfied.

There is no provision for special comment brackets that might be needed to delimit assertions. Extending PASCAL to meet requirement K5 would be straightforward, however. Moreover, all of the literature on the language (e.g., Wirth's Systematic Programming book) extensively uses assertions as comments. There is no provision for incorporating or making use of units of measure.

K5. Assertions and Other Specifications (SIMULA 67)

Not Satisfied.

No provision is made for optionally interpretable comments delimited by special brackets.

K5. Assertions and Other Optional Specifications (TACPOL)

Not Satisfied.

TACPOL does not support assertions or any form of interpretable comments.

L1. No Superset Implementations (TINMAN)

No implementation of the language will contain source language features which are not defined in the language standard. Any interpretation of a language feature not explicitly permitted by the language definition will be forbidden.

This does not, however, disallow library definition optimizations which are translator unique.

L1. No Superset Implementations (ALGOL 68)

Not Satisfied.

The Algol 68 Revised Report precisely defines the language and what requirements must be satisfied to have an "implementation" be an implementation of Algol 68. However, it permits features to be added by an implementation as long as all standard Algol 68 programs retain the same syntax and semantics [Section 2.2].

L1. No Superset Implementations (J3B)

Not Satisfied.

J3B permits implementation-dependent "built-in" functions to be defined [Section 3.2.1]. These language constructs have the syntactic appearance of functions, but they may have special properties not definable within the language, or they may provide access to special object machine instructions, such as I/O instructions. Programs written using these functions cannot be compiled by a compiler that does not support "built-in" functions having identical names and properties.

L1. No Superset Implementations (PASCAL)

Satisfied.

The definition of PASCAL establishes a standard that excludes further source language features.

L1. No Superset Implementations (SIMULA 67)

Not Satisfied.

Separate compilation is "an optional part of the Common Base" [CBL 15]. Consequently SIMULA 67 implementations with this facility are permissible supersets.

L1. No Superset Implementations (TACPOL)

Not Satisfied.

The TACPOL manual does not forbid superset implementations.

L2. No Subset Implementations (TINMAN)

Every translator for the language will implement the entire base language. There will be no subset implementations of the base language.

The intended source language product from this effort is a small, simple, uniform, base language with specialized features, support packages, and complex features relegated to library routines not requiring direct translator support. If simple low cost translators are not feasible for the selected language, then the language is too large and complex to be standardized and the goal of language commonality will not be achievable.

L2. No Subset Implementations (ALGOL 68)

Not Satisfied.

The Algol 68 Revised Report precisely defines the language and what requirements must be satisfied to have an "implementation" be an implementation of Algol 68. However, it permits features to be deleted by an implementation as long as all subset programs have the same syntax and semantics as in a full Algol 68 implementation [Section 2.2]. In fact, most existing Algol 68 implementations omit the parallel processing feature and some omit garbage collection of the heap.

L2. No Subset Implementations (J3B)

Not Satisfied.

The language specification permits some implementations to not support the fixed point and double precision floating point data types [Sections 7.1.3 and 7.1.4]. If floating point data are not supported, then the exponentiation operator is also not supported [Sectin 7.1]. In addition, all implementations are not required to support unaligned pointers [Section 1.3.3.1, p. 1-16].

L2. No Subset Implementations (PASCAL)

Satisfied.

The defining documents establish standard PASCAL without allowing for subsets. (The first PASCAL compiler for the ICL 1900 series of computers (SPE, vol. 2, no. 2, pp. 75-76), however, did not implement set operations.)

L2. No Subset Implementations (SIMULA 67)

Satisfied.

The defined Common Base Language "is required to be a part of any SIMULA 67 system" [CBL 1.4].

L2. No Subset Implementations (TACPOL)

Not Satisfied.

The reference manual does not explicitly require a common baseline for all TACPOL implementations. In fact, at a syntactic level, restricted character sets (e.g, without a colon or a semicolon [Section 2-4]) are accommodated.

L3. Low Cost Translation (TINMAN)

The translator will minimize compile time costs. A goal of any translator for the language will be low cost translation.

Where practical and beneficial the user will have control over the level of optimization applied to his programs. The programmer will have control over the tradeoffs between compile time and run time costs. The goal will be effective use of the available machines both in object execution and translation and not maximal speed of translation.

Both the translator and the language design will emphasize low cost translation, but in an environment of large and long-lived software products this will be secondary to requirements for reliability and maintainability. Language features will be chosen to ensure that they do not impose costs for unneeded generality and that needed capabilities can be translated into efficient object representations.

L3. Low Cost Translation (ALGOL 68)

Partially Satisfied.

Algol 68 is more difficult to translate than a small language like PASCAL. Although Algol 68 is comparable in size and features to PL/I, Algol 68's orthogonality and choice of features permits a smaller, simpler compiler than PL/I does. In particular, Algol 68 is easier to optimize than PL/I [Section 0.1.4].

L3. Low Cost Translation (J3B)

Satisfied.

Certain J3B constructs were designed to reduce the cost of translation, in particular, the requirement that declarations be grouped at the beginning of lexical scopes [Section 3.1.1], the requirement that identifiers, in general, be declared before they are used [Sections 3.2.2, 3.2.3, and 3.2.2.4], and the provision for preprocessed compools and lexical (rather than character) macros [Sections 3.1.3 and 3.4].

L3. Low Cost Translation (PASCAL)

Satisfied.

Many language features were deliberately omitted from PASCAL in order to attain compilation speed that eliminated the need for relocatable object code [Wirth, SPE, p. 317]. "A rigorous selection among the immense variety of programming facilities available had to be made in order to keep the compiler relatively compact and efficient and economical for both the user who writes only small programs using few constructs of the language and the user who

writes large programs and tends to make use of the full language" [p. 8]. The syntax supports a one-pass scheme which greatly facilitates compilation speed on computers with large main storage.

L3. Low Cost Translantion (SIMULA 67)

Partially Satisifed.

Bench-mark tests have shown that the SIMULA 67 "IBM system compiles roughly twice as fast as PL/I-F" and on the Univac 1100 series SIMULA 67 compiles "about the same as NU-ALGOL". "I/O is significantly faster in SIMULA than in FORTRAN" [Structure, p. 11]. But no mention is made of user control over the level of optimization applied to programs.

According to J. Palme: "Most of the compilers are very good and efficient. For example, the IBM 360/370 compiler is about as efficient as the FORTRAN G compiler, and the CDC 3300 compiler is regarded as the best compiler existing for that computer" [Palme, p. 25].

L3. Low Cost Translation (TACPOL)

Satisfied.

The TACPOL language features have been chosen so that they have natural counterparts in patterns of assembly code. The costly and difficult-to-translate features of PL/I have been excised.

L4. Many Object Machines (TINMAN)

Translators will be able to produce code for a variety of object machines. The machine independent parts of translators might be built independent of the code generators.

L4. Many Object Machines (ALGOL 68)

Satisfied.

The Algol 68 Revised Report does not prohibit a translator from producing code for several object machines. Pragmats [Section 9.2] may be used to specify the desired object machine. Several Algol 68 translators already exist and/or are under construction.

L4. Many Object Machines (J3B)

Satisfied.

J3B compilers have been developed that generate code for the IBM 360/370, Delco M362F, Litton 516, and Singer SKC-2000.

L4. Many Object Machines (PASCAL)

Satisfied.

PASCAL compilers currently exist for the following computers: CDC 6000 series, IBM 360/370 series, ICL 1900 series, DEC 10, CII IRIS 80, Univac 1100 series, CDC 3300, CDC 3600, B6700, PDP-11/20. A recent survey indicated that PASCAL implementation "projects were underway for the following machines: TI ASC, Data General 840, Raytheon 704, Univac AN/UYK-20, Honeywell G635, Burroughs 4700 and 6700, and PDP-11/45" [PASCAL Newsletter in SIGPLAN Notices, 11, 2, February 1976, p. 37].

L4. Many Object Machines (SIMULA 67)

Satisfied.

Already a number of implementations exist. J. Palme writes: "SIMULA systems are available on a large number of computers: IBM 360/370, Univac 1106-1110, CDC 3200-3500, CDC 3600, CDC 6400-6600, CDC Cyber 70 series and the larger CII and Iris computers. New SIMULA systems are under development for the ... DEC system 10 series" [Palme, p. 25].

L4. Many Object Machines (TACPOL)

Not Satisfied.

In principle, TACPOL could satisfy the L4 requirement. In fact, TACPOL (according to the reference manual) has been built for a particular machine, the AN/GYK-12.

L5. Self-Hosting Possible (TINMAN)

The translator need not be able to run on all the object machines. Self-hosting is not required, but is often desirable.

The size of machines which can host translators should be kept as small as possible by avoiding unnecessary generality in the language.

L5. Self-Hosting Not Required (ALGOL 68)

Not Determinable.

Whether Algol 68 can be hosted on a small computer is not determinable from information available to us.

L5. Self-Hosting Possible (J3B)

Satisfied (?)

Current J3B compilers are large because no design constraints were imposed to insure a small compiler would be produced. There appears to be no reason, however, why a small J3B compiler could not be developed.

L5. Self-Hosting Possible (PASCAL)

Satisfied.

The modest size of a self compiler (i.e., 4000 PASCAL source statements - Wirth, SPE, p. 321) permits a wide range of hosting machines.

L5. Self-Hosting Possible (SIMULA 67)

Partially Satisfied.

SIMULA 67 compilers are not excessively large; nor are they small. "The Univac compiler takes about 30 K words on the 1108" [Structure, p. 3] and the next release of the 360/370 version requires about 93K bytes.

L5. Self-Hosting Possible

Satisfied.

This assumes the L5 requirement constrains the compiler's potential size to fit on small, or at least medium, sized machines. TACPOL meets the L5 requirement in the above sense.

L6. Translator Checking Required (TINMAN)

> The translator will do full syntax checking, will check all operations and parameters for type compatibility and will verify that all language imposed semantic restrictions on the source programs are met. It will not automatically correct errors detected at compile time.

L6. Translator Checking Required (ALGOL 68)

Satisfied.

The Algol 68 Revised Report precisely and completely specifies the syntax and semantics of the language. In particular, an Algol 68 program must have compatible operator-operand types and actual-formal parameter types. The Report requires an implementation to meet these requirements [Section 2.2].

L6. Translator Checking Required (J3B)

Satisfied.

Full syntax checking, including checks on parameter declarations and usage, is supported by every J3B compiler. No errors are automatically corrected v

L6. Translator Checking Required (PASCAL)

Not Determinable.

PASCAL was designed to facilitate full automatic checking for syntactic consistency [Wirth, p. 197], but the defining documents do not require it nor do they specify the kinds of errors to be detected.

L6. Translator Checking Required (SIMULA 67)

Partially Satisfied.

SIMULA 67 is designed for extensive compile time checking, The defining document [CBL] does not specify how much is required. In SIMULA compilers, all language errors "are discovered as soon as they logically can be discovered, and the error message is always phrased in SIMULA terms" [Palme, p. 30]. "The structure of the SIMULA language is such that almost all the error checking of a program can be done at compile time" [Palme, p. 38].

L6. Translator Checking Required (TACPOL)

Satisfied.

In general, "attributes must be explicitly declared. This results in fewer errors because missing specifications are flagged at compile time" [Section 1-6]. The TACPOL compiler is charged with full syntax checking. For example, it must detect and reject expressions with mixed mode.

L7. Diagnostic Messages (TINMAN)

> The translator will produce compile time explanatory diagnostic error and warning messages. A suggested set of error and warning situations will be provided as part of the language definition.

Diagnostic messages will not be coded but will be explanatory and in source language terms. Translators will continue processing and checking after errors have been found but should be careful not to generate erroneous messages because of translator confusion. The translator will always produce correct code; when source program errors are encountered by the translator or referenced program structures omitted, the compiler will produce code to cause

a run time exception condition upon any attempt to execute those parts of the program. Warnings will be generated when a source language construct is exceptionally expensive to implement on the specified object machine. A suggested set of diagnostic messages, provided as part of the language definition, contributes to commonality in the implementation and use of the language.

L7. Diagnostic Messages (ALGOL 68)

Not Satisfied.

The Algol 68 Revised Report does not define translator actions for erroneous programs, nor does it provide a suggested set of error and warning situations.

L7. Diagnostic Messages (J3B)

Partially Satisfied.

Some J3B program errors to be detected by compilers are specified implicitly in the J3B syntax equations, i.e., a program not conforming to the equations is considered to be in error. Error situations not specified in the syntax equations are specifically listed in the language specification. These situations are divided into two classes, "Illegal Constructs" (errors that are detectable at compile time), and "Undefined Effects" (errors detectable only at run time). Specific constructs that are legal, but likely to be indicative of programmer error, are also explicitly cited in the specification as "Warnings". The J3B specification states, "It is expected that all violations of the syntax equations, all 'Illegal Constructs' and all 'Warnings' will be detected by every compiler implementation for JOVIAL/J3B, although strictly speaking, which errors will be detected is defined by the compiler implementation specification" [Section 1.3.1]. Whether or not code is generated to check for "Undefined Effects" is implementation-dependent, and is generally a possible compiler option. The language specification does not demand that such code be generated.

Although error situations are completely documented, the language specification does not specify or suggest a set of error messages, nor does the specification limit a compiler's behavior once an error has been found. In practice, the J3B compilers use a common set of error messages and produce no code if compile-time errors are detected.

L7. Diagnostic Messages (PASCAL)

Not Satisfied.

Such messages are left to the PASCAL implementor. The PASCAL 6000 error messages [pp. 119-121] are a reasonable set of diagnostics that could be suggested as part of the standard language.

L7. Diagnostic Messages (SIMULA 67)

Not Satisfied.

No suggested set of diagnostic messages are provided as part of the language definition.

L7. Diagnostic Messages (TACPOL)

Not Satisfied.

Whereas in practice TACPOL does give diagnostic error and warning messsges, the language reference manual does not list nor suggest any such messages.

L8. Translator Internal Structure (TINMAN)

The characteristics of translator implementations will not be dictated by the language definition or standards.

L8. Translator Internal Structure (ALGOL 68)

Satisfied.

The Algol 68 Revised Report does not define the internal characteristics of translators and/or run time support routines.

L8. Translator Internal Structure (J3B)

Satisfied.

The J3B definition avoids dictating compiler charactistics.

L8. Translator Internal Structure (PASCAL)

Satisfied.

The PASCAL definition avoids dictating compiler characteristics.

L8. Translator Internal Structure (SIMULA 67)

Satisfied.

The language definition does not constrain unduly a translator's internal structure.

L8. Translator Internal Structure (TACPOL)

Satisfied.

The TACPOL manual does not recommend or restrict implementation techniques.

L9. Self-Implementable Language (TINMAN)

Translators for the language will be written in their own source language.

L9. Self-Implementable Language (ALGOL 68)

Satisfied.

Algol 68 is a suitable language for writing an Algol 68 translator. At least one such translator exists.

L9. Self-Implementable Language (J3B)

Partially Satisfied.

J3B's capabilities are sufficient and reasonable for writing a compiler in J3B, but to date, this has not been done for any J3B compiler. (All J3B computers are written in AED and run on IBM 360/370 computers.)

L9. Self-Implementable Language (PASCAL)

Satisfied.

The first PASCAL compiler was written in FORTRAN and then discarded. The second compiler was originally written in PASCAL and then translated by hand. PASCAL has carried self-compilation to the extreme that currently there is no compiler "available in any other language than PASCAL itself" [Wirth, SPE, p. 324].

L9. Self-Implementable Languge (SIMULA 67)

Partially Satisfied.

Though SIMULA 67 has not strictly been self-implemented, a standard "abstract implementation was given in a slightly extended SIMULA language, and covered most of the support routines required during execution of user programs... This abstract definition caused most SIMULA runtime systems to use similar techniques, and simplified both the implementation and the debugging of the different runtime systems" [Structure, p. 2].

L9. Self-Implementable Language (TACPOL)

Partially Satisfied.

TACPOL has not yet been implemented in itself. But TACPOL was designed, in part, to help in developing and in maintaining compilers [Section 1-2.c].

M1. Existing Language Features Only (TINMAN)

> The language will be composed from features which are within the state of the art and any design or redesign which is necessary to achieve the needed characteristics will be conducted as an engineering design effort and not as a research project.

The adoption of a common language can be successful only if it makes available a modern programming language compatible with the latest software technology and is compatible with "best" current programming practice, but the design and implementation of the language should not require additional research or require use of untried ideas. State-of-the-art cannot, however, be taken to mean that a feature has been incorporated in an operational DoD language and used for an extended period, or DoD will be forever tied to the technology of FORTRAN-like languages; but there must be some assurances through analysis and use that its benefits and deficiencies are known. The larger and more complex the structure, the more analysis and use that should be required. Language design should parallel other engineering design efforts in that it is a task of consolidation and not innovation. The language designer should be familiar with the many choices in semantic and syntactic features of language and should strive to compose the best of these into a consistent structure congruous with the needed characteristics. The language should be composed from known semantic features and familiar notations, but the use of proven feature should not necessarily impose that notation. The language must not just be a combination of existing features which satisfy the individual requirements but must be held together by a consistent and uniform structure which acts to minimize the number of concepts, consolidates divergent features and simplifies the whole.

M1. Existing Language Features Only (ALGOL 68)

Satisfied.

The Algol 68 Report was completed in 1968-69. At least one implementation (of nearly the full language) has been operational for several years (at Royal Radar Establishment, England). Several implementations now exist. The features of Algol 68 are within the state of the art. Most of them can be found in a similar form in at least one other implemented language. The revised language is very similar to the original. In particular, certain difficult to translate features have been removed (e.g., proceduring) or corrected (void symbol, separate cast and routine symbols).

M1. Existing Language Features Only (J3B)

Satisfied.

J3B consists of programming constructs which exist either in JOVIAL J3, J73, or other programming languages. The fact that J3B compilers exist and have been used in programing operational military systems means in itself that this requirements is satisfied.

M1. Existing Language Features Only (PASCAL)

Satisfied.

PASCAL was designed in 1969 and has been in practical use since 1970 [Wirth, p. 193]. The language retains the form of its original definition and subsequent changes have been "very few and relatively minor" [p. 133].

M1. Existing Language Features Only (SIMULA 67)

Satisfied.

SIMULA 67 has been implemented on a variety of object machines. Such implementations have existed for several yhears.

M1. Existing Language Features Only (TACPOL)

Satisfied.

TACPOL is an operational language. Consequently all of its features are well within the state of the art.

M2. Unambiguous Definition (TINMAN)

The semantics of the language will be defined unambiguously and clearly. To the extent a formal definition assists in attaining these objectives, the language's semantics will be specified formally.

M2. Unambiguous Definition (ALGOL 68)

Satisfied.

The Algol 68 Revised Report provides a precise, clear (to initiates, at least!), complete, and unambiguous formal definition of the language (both syntax and semantics).

M2. Unambiguous Definition (J3B)

Partially Satisfied.

No formal definition of J3B has been provided, but in practice, when questions have arisen about whether a J3B compiler has correctly implemented the language, it has always been the case that the specification provided a clear answer.

M2. Unambiguous Definition (PASCAL)

Partially Satisfied.

The Revised Report provides a clear, concise definition with only a few gaps (e.g., defining the type matching for parameters and arguments).

There are at least two attempts at a formal language definition:

(1) C.A.R. Hoare and N. Wirth, "An Axiomatic Definition of the Programming Language PASCAL", ACTA INFORMATICA, 2, 335-355 (1973).

(2) L. Aiello, M. Aiello and R. Weyhrauch, "The Semantics of PASCAL in LCF", Stanford University, Tech. Report CS-74-447, (1974).

M2. Unambiguous Definition (SIMULA 67)

Partially Satisfied.

Palme writes of the "exact definition of SIMULA, which is standardized and works exactly in the same way on all computers with SIMULA systems. One reason for this is that the word "undefined" does not occur in the SIMULA definition as in many other language definitions. Therefore there is very little which can be done differently on different computers" [Palme, p. 38]. The definition of SIMULA 67 [CBL] is indeed an admirably complete document when augmented with the Revised ALGOL 60 Report [AR]. But the language definition is not totally free of ambiguities. For example, the order of argument evaluations for operators is not specified leading to side-effect ambiguities (see C1).

M2. Unambiguous Definition (TACPOL)

Not Satisfied.

The language reference manual is inadequate for precisely defining TACPOL. Examples are used throughout that do not resolve the limitations and the permissible generality of a language feature. Further, a feature is only explained in terms of an accompanying example. Apparent contradictions, omissions, and syntax errors compound the difficulty. A more formal definition was made available to us too late for use in our evaluation of TACPOL; it appears that it may provide a satisfactory definition of TACPOL.

M3. Language Documentation Required (TINMAN)

The user documentation of the language will be complete and will include both a tutorial introductory description and a formal in-depth description. The language will be defined as if it were the machine level language of an abstract digital computer.

The action of any legal program will be determinable from the program and the language description alone. Any computation which can be described in the language will ultimately draw only on capabilities which are built into the language. No characteristics of the source language will be dependent on the idiosyncrasies of its translators.

The language documentation will include syntax, semantics and examples of each language construct, listings of all key words and language defined defaults. Examples shall be included to show the intended use of language features and to illustrate proper use of the language. Particularly expensive and inexpensive constructs will be pointed out. Each document will identify its purpose and prerequisites for its use.

M3. Language Documentation Required (ALGOL 68)

Satisfied.

The Algol 68 Revised Report is the formal definition of the language. C.H. Linsey and S.G. van der Meulen have written an Informal Introduction to Algol 68 (North-Holland, 1973) that gives a very informal tutorial overview initially and then gives a good tutorial treatment of the complete language with lots of examples. The Informal Introduction is currently being revised to correspond to the Revised Report.

M3. Language Documentation Required (J3B)

Partially Satisfied.

An introductory Programmer's Guide exists for J3B, but there is no formal definition of J3B's semantics. Moreover, some aspects of J3B semantics are implementation dependent, e.g., the character code used to implement character strings, and hence, the effect of the BIT operator when applied to character data. These implementation dependent properties of the language are pointed out explicitly in the language specification.

The language specification includes syntax and semantics descriptions, a listing of all key words, and a description of language-defined defaults (such as implicit conversion from integer to fixed in assignment statements). However, no examples of J3B constructs are provided.

The purpose and prerequisites for use of both the language specification and programmer's guide are defined in each document.

M3. Language Documentation Required (PASCAL)

Satisfied.

The current documentation for PASCAL is excellent and is certain to be enriched. It includes:

(1) The two formal definitions cited in M2 above.

(2) The Revised Report by Wirth, which defines the language in clear ALGOL-60 like style.

(3) The User Manual by K. Jensen and N. Wirth, which presents each language feature with examples of use in code fragments and in complete programs. Chapters 0 through 12 cover standard PASCAL, while Chapters 13 and 14 are concerned with PASCAL 6000 specific features.

(4) Systematic Programming by N. Wirth (Prentice-Hall, 1973) is an intelligent introduction to programming using PASCAL as the illustrative language.

M3. Language Documentation Required (SIMULA 67)

Satisfied.

The language definition [CBL] is based on and subsumes part of the revised ALGOL 60 Report [AR]. The book SIMULA BEGIN [B] provides "a tutorial introductory description" with many examples.

M3. Language Documentation Required (TACPOL)

Not Satisfied.

TACPOL's documentation is far from complete. The language reference manual is more in the vein of a tutorial introductory description than a formal in-depth one. Such an in-depth formal description is currently lacking and is sorely needed.

M4. Control Agent Required (TINMAN)

The language will be configuration managed throughout its total life cycle and will be controlled at the DoD level to ensure that there is only one version of the source language and that all translators conform to that standard.

All compilers will be tested and certified for conformity to the standard specification and freedom from known errors prior to their release for use in production projects. The language manager will be on the OSD staff, but a group within the Military Departments or Agencies might act as the executive

agent. A configuration control board will be instituted with user representation and chaired by a member of the OSD staff.

M4. Control Agent Rquired (ALGOL 68)

Not Satisfied.

There is no DoD designated control agent for Algol 68. I.F.I.P. is responsible for the formal definition of the language (the Revised Report), but not for ensuring that implementations satisfy the requirements of the Report.

M4. Control Agent Required (J3B)

Not Satisfied.

There is no DoD designated control authority for J3B.

M4. Control Agent Required (PASCAL)

Not Satisfied.

There is no Dod designated control authority for PASCAL.

M4. Control Agent Required (SIMULA 67)

Not Satisfied.

No DoD SIMULA 67 control agent has been designated. However, there does exist a "SIMULA Standard Group, consisting of representatives for firms and organizations having responsibility for SIMULA 67 compilers" [CBL 1.4]. This group enforces strict standardization and controls future language extensions.

M4. Control Agent Required (TACPOL)

Not Satisfied.

No DoD control agent has been designated, although ARTADS serves as the de facto control agent.

M5. Support Agent Required (TINMAN)

There will be identified support agent(s) responsible for maintaining the translators and for associated design, development, debugging and maintenance aids.

Support of common widely used tools and aids should be provided independent of projects if common software is to be widely used. Support should be on a DoD-wide basis with final responsibility resting with a stable group or groups of qualified in-house personnel.

M5. Support Agent Required (ALGOL 68)

Not Satisfied.

There is no DoD designated support agent for Algol 68.

M5. Support Agent Required (J3B)

Not Satisfied.

There is no DoD agent designated for supporting J3B compilers and programming aids. SofTech, Inc. is currently performing these functions for the Air Force.

M5. Support Agent Required (PASCAL)

Not Satisfied.

There is no DoD agent supporting PASCAL.

M5. Support Agent Required (SIMULA 67)

Not Satisfied.

There is no DoD identified support agent for SIMULA 67.

M5. Support Agent Required (TACPOL)

Not Satisfied.

No DoD support agent has been designated.

M6. Library Standards and Support Required (TINMAN)

There will be standards and support agents for common libraries including application-oriented libraries.

In a given application of a programming language three levels of the system must be learned and used: the base language, the standard library definitions used in that application area, and the local application programs. Users are responsible for the local application programs and local definitions

but not for the language and its libraries which are used by many projects and sites. A principal user might act as agent for an entire application area.

M6. Library Standards and Support Required (ALGOL 68)

Not Satisfied.

There is no DoD designated support agent for Algol 68 application-oriented libraries. Algol 68 does not define a library concept, except that an implementation may supply a library prelude [Section 10.1] to augment the standard prelude declarations [Section 10] and may use pragmats [Section 9.2] to access libraries.

M6. Library Standards and Support Required (J3B)

Not Satisfied.

J3B does not support the concept of libraries of language extensions in the sense intended by the TINMAN (see E1, E3, E4, E5, E6, F4, F5, and F6).

M6. Library Standards and Support Required (PASCAL)

Not Satisfied.

PASCAL does not include the library concept; further, there are no DoD agents supporting PASCAL application-oriented libraries.

M6. Library Standards and Support Required (SIMULA 67)

Although SIMULA 67 is designed to accommodate naturally special application oriented packages [CBL 1-3.4], the library concept is not supported as defined by the TINMAN.

M6. Library Standards and Support Required (TACPOL)

Not Satisfied.

TACPOL does not support the concept of application oriented libraries in the sense intended by M6.