DESIGN OF SYSTEMS LANGUAGES

A thesis presented in partial fulfilment
of the requirements for the degree of
Master of Science in Computer Science
at Massey University

Christopher Allen Freyberg

March 1975

Abstract

Systems Languages have often been designed on a rather ad hoc basis. This thesis attempts to formulate and analyse design criteria in a more systematic manner. These criteria are drawn from three major sections: a survey of languages used for systems programming, a discussion of systems programs features, and a discussion of programming language effectiveness. The resulting criteria are then discussed in relation to their application to the language design. A collection of language summaries is included in the appendices.

To the many people who have helped
or encouraged me in the completion of this thesis.

I would like to acknowledge by name the work of the following:

Brian Carpenter, Bob Doran, Prof. G. Tate (Supervisors)

Lloyd Thomas (Algol 68)

Lyn Stening (typist)

Table of Contents

§1 Introduction

- §1.1 Aim
- §1.2 Some Difficulties
- §1.3 Outline

§2 A Brief Survey

- §2.1 The Various Languages
- §2.2 Algol Based Languages
- §2.3 PL/I Based Languages
- §2.4 Other High-Level Languages
- §2.5 PL360 Based Languages and Assemblers
- §2.6 Summary

§3 Systems Programs : Common Features

- §3.1 Systems Programs : Definition and Scope
- §3.2 Breakdown of Processing Features
- §3.3 Collected Features

§4 Effectiveness : Functionality, Error Immunity and Efficiency

- §4.1 Functionality
- §4.2 Error Immunity
- §4.3 Efficiency

§5 Criteria and their Application to the Language Design

- §5.1 Extensibility/Orthogonality
- §5.2 Data Related Criteria
- §5.3 Control Related Criteria
- §5.4 Miscellaneous Critera
- §5.5 Conclusions

§6 Appendices

- §6.1 Language Summaries
- §6.2 A Language Design
- §6.3 Glossary
- §6.4 References

§1 Introduction

§1.1 Aim

In the past, a commonly accepted project for a masterate in the programming languages field was to take an already designed language, suitably modified, to implement on a given machine. The student thereby derived experience in implementation problems. However the usefulness of that type of project is severely limited when the language is a systems programming language, in that most machines to which the student has access already have well developed suites of systems programs. Hence rewriting a part of any suite runs foul of intercommunication problems, and rewriting the whole would be an excessively large task even for a small machine. The project would accomplish little more than an intimate knowledge of <u>one</u> language and <u>one</u> machine.

An obvious alternative, that of designing and implementing a systems language, was discarded for similar reasons; also there is already a plethora of languages of that type. Such a 'home grown' language is, moreover, only likely to gain acceptance and be used in the immediate locality unless it happens to incorporate some startlingly new and useful technique or construct. In other words, it would be little more than an exercise.

The topic finally selected, "Design of Systems Languages", was anticipated to require two reasonably distinct subprojects:

i. a survey of existing systems languages

ii. development of design criteria based on an analysis of their features

and possibly a third subproject developing a language based on those criteria. However it soon became obvious, while surveying the existing languages, that in many cases the criteria employed by their authors were neither explicit nor extensive, so that a somewhat different approach would be required, even if the basic intention remained the same.

§1.2 Some inherent difficulties

A major problem in tackling a topic such as this is that the experience (or lack of it) of the author can lead to large distortions of outlook. He attempts to survey and criticise a group of languages with a wide range of features, when the only features he has experience of are limited to those implemented on the few machines he has worked on. Coupled to this, the machines to which he at present has access colour and in some respects bias his appraisal of anything which applies to other machines, particularly to those to which he has never had access. For example, access to a stack machine leaves him with doubts about the sanity of anyone who uses 360 type parameter passing.

Realising that these two difficulties exist fortunately provides some solutions - abstraction becomes a keyword and generality becomes an overall goal. The survey of languages thus becomes a means to an end, and can be divided into two parts:

i. a gathering of information from separate sources

ii. a criticism of what has been done or not done to date.

Following this, greater emphasis can be given to determining the paramount features and linguistic requirements of the various types of systems program, instead of relying upon other people's opinions about them. Similarly it is preferable to determine for oneself the desirable features of a systems language, particularly in the shadow of considerable disagreement in the literature over machine independence of systems languages, and even over terminology, notably 'efficiency'. These disagreements point to fields of study outside the scope of this thesis.

§1.3 Outline

This thesis therefore attempts, through a survey of existing systems languages and an examination of the characteristics of systems programs, to develop a series of criteria by which a systems programming language may be judged, and through which a new language can be constructed to make it a useful tool.

Section 2 surveys existing languages by grouping them with respect to common base languages. It relies heavily upon §6.1, which is a table of the features of the various languages. The section is summarised in §2.6.

Section 3 addresses the nature of systems programs and what language features are required to write and support them. Emphasis is placed here (as in Section 2) on the language itself and the linguistic criteria, rather than on the compiler or the methodology of construction. A summary is made in §3.3.

Section 4 is an attempt to put some order into arguments about efficiency, commentation, and the general methodology of systems programs.

It also attempts to draw up guidelines for those features the compiler must provide exclusive of the language itself.

Section 5 collects together the criteria from the above three sections and, along with criteria related to extensibility, attempts to order them into a preferential system. This system is then discussed in terms of the limitations it places on, and demands it makes on, the language structure.

The conclusions of the thesis are really contained in Section 5, and for this reason are not given a separate section heading.

§2 A Brief Survey

§2.1 The Various Languages

Sammett [Sam 71] lists 44 programming languages either designed or used for systems programming. At the end of 1974, some 20 more had made an appearance. Bearing in mind that these languages are only the ones of which accounts have been published (and have come to my attention), I consider it likely that at the present time there are at least 80 programming languages (other than assemblers) used for systems programming. From these facts, one can draw two immediate conclusions. Firstly, there is no systems programming language which satisfies the needs of any large group of people; and, secondly, institutions get a certain satisfaction in creating a new language tailored to their own (and often limited) needs.

Dissatisfaction with existing languages will always, in my opinion, be endemic. Any language used for writing an operating system must be machine-dependent, as each machine has instructions which are unique to it, and which often must be coded explicitly. This is not to say that the machine dependence need be extensive; there are various mechanisms to severely limit its scope.

However, the proliferation of new languages for limited purposes is a human problem which could be overcome by greater adaptation of existing languages, rather than by creation of new ones. To this end extensible languages, such as ALGOL68, may provide a partial answer.

It seems fairly obvious that no systems program (in the sense defined in §3.1) is ever truly portable, regardless of whether or not it is coded in some high-level language. In fact, it could be said that systems programs are precisely those programs which, for some reasons, are non-portable. The algorithm effected might be the same but some internal steps will doubtless have to be machine dependent. However, this does not deny that the systems language itself may be largely portable. A great deal of time wasted on complete redesign of languages could be saved by providing either an extensible language, or a language which might easily be modified to accommodate the idiosyncrasies of different machines.

§2.1.1 Language Hierachies

A large number of systems languages are regarded by their creators as an extension or modification of a base language. This progressive modification of existing languages provides, when studied, many examples of what language features are considered necessary for systems programming. It is indeed interesting that three hierachies (based on ALGOL, PL/I, and PL-360) account for more than half the total number of systems languages available - an indication that these base languages contain many useful constructs or ideas. These three trees are dealt with separately in §2.2, §2.3, and §2.5. The remaining high level languages have at most two-node trees, and are discussed in §2.4 under groupings related to their intended usage.

Figures 2.1, 2.2, and 2.3 picture the trees for ALGOL, PL/I and PL-360 respectively while Fig 2.4 pictures any remaining minor trees. Where it is ascertainable that the language has been implemented on more than one machine, a superscript asterisk indicates this. Bracketed elements in Fig 2.1 have to my knowledge never been used for writing systems programs and were not intended for that purpose.

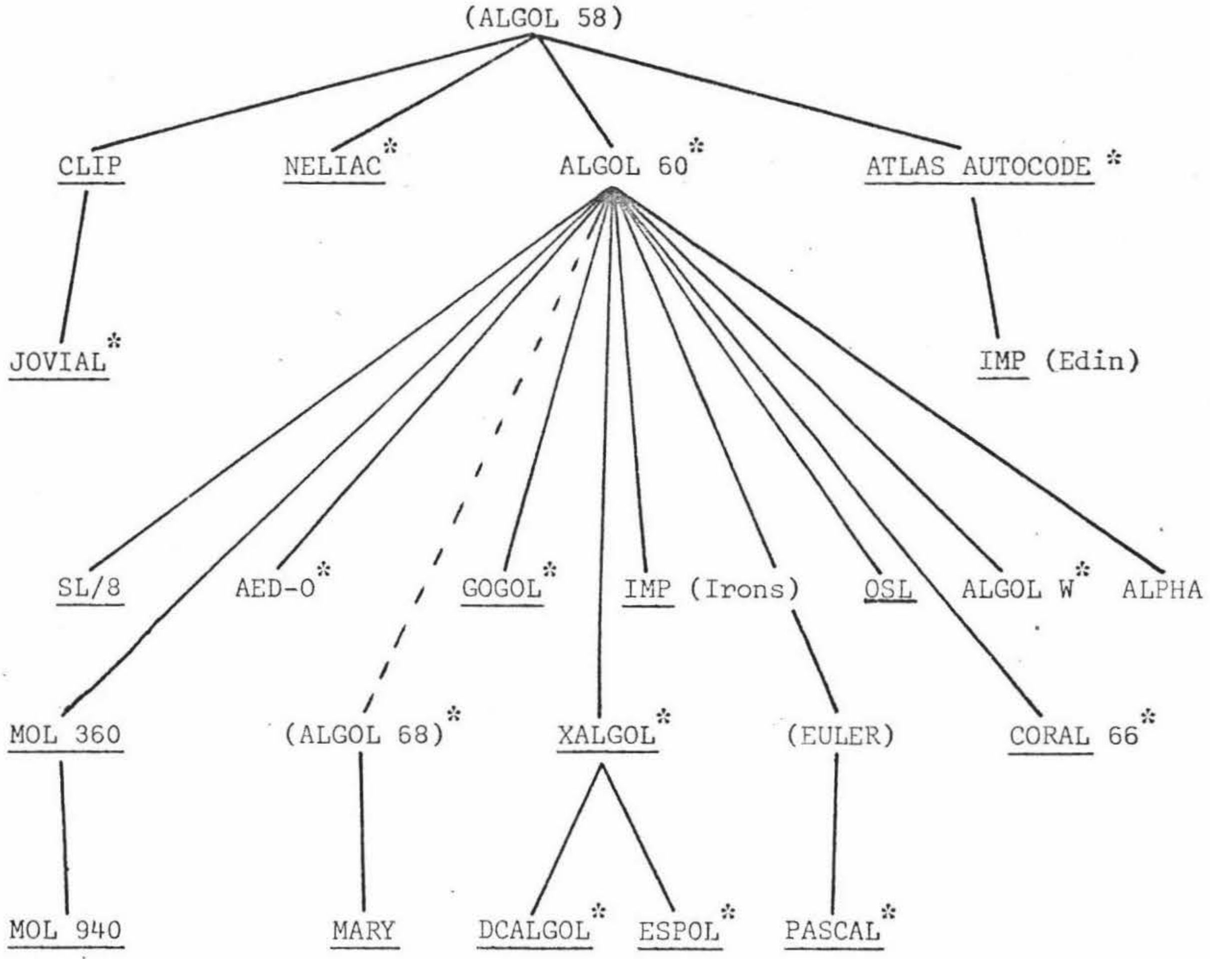
§2.1.2 Assembler Languages per se

In this survey I have avoided discussing assembly languages for two reasons. The first is that there is an extremely large number of assembly languages with varying degrees of structure, and the second is that most people are fully aware of limitations and capabilities of assembler languages (except, perhaps, how 'efficient' assembler language programs are). I will discuss this in greater detail in §4.

It is doubtful that those who still advocate assembler languages for systems programming will ever be convinced that some loss in object-code efficiency can be amply compensated for in other ways. However, some time spent using a system written in a high-level language would soon demonstrate the facilities available which could not or would not have been provided in assembler language. Examples of these are variable length job + program names, free format job control cards, easily understood operator messages, to mention just a few.

Some assembler languages are, nevertheless, quite highly structured. SAC (for the Elliott 503, 803) is an early example, and PL-360, AL, SAL, are more modern varieties. Powerful macro processors such as ML/I can provide considerable structure up to the procedure level in assembler language programming with no less direct source-object code correspondence than the assemblers themselves. The only inefficiencies that can be generated are by using overly generalised macros, and calling them too often. One wonders then why these aids do not seem to be more widely used by advocates of assembler-language programming.

Fig 2.1 ALGOL Hierachy



§2.2 Algol-Based Languages

Figure 2.1 pictures the dependency tree for the ALGOL based languages. Of the 23 nodes, the 16 underlined were designed explicitly for systems programming, while another two have been used for that purpose. The sheer size of the tree, particularly the ALGOL-60 subtree, indicates a high proportion of useful features in the base language.

The fact that ALGOL58 and ALGOL60 were designed by committees illustrates the value of cooperative effort in producing a general design. It does not, however, indicate usefulness of the language, as by far the most heavily used languages are end-nodes. As far as the subsequents of ALGOL58 are concerned, JOVIAL, NELIAC and ALGOL60 were developed all at the same time. ALGOL60 emerged as the prime example of an elegant language design; the others, while both incorporating new features not present in the former (such as primitive macros, initialised arrays, precision specification, new operator declarations) and also enjoying considerable use, failed to generate any further language (principally, I believe, because of their general untidyness).

ALGOL60 is the direct predecessor of the greatest number of systems languages. Of those 4 are machine dependent to an appreciable extent and of the remaining 8 only two were designed specifically for systems programming, although all have been used in this field.

§2.2.1 Data

The three basic single cell modes, INTEGER, REAL, and BOOLEAN occur in almost all the languages in the ALGOL hierachy. Some earlier languages distinguished these types by quite elaborate precision specifications. For example, in NELIAC:

```
ITEM A 000 , B 000.00
```

In the later languages precision is specified in terms of storage units required (that is, bytes or words) as decimal processing hardware has become more rare. Several languages have other single cell modes, the most prominent being POINTER. Some of the earlier languages have no mode <u>real</u>, but as most of the languages are implemented on large machines with hardware floating-point arithmetic nearly all the later languages have this mode.

Arrays are often cut to a single dimension, and sometimes have a fixed lower bound of 0 or 1. More importantly, most languages permit array initialisation at declaration time.

Bit-fields as data occur almost exclusively in the later languages with the exception of JOVIAL (which has variable modifiers that select bits or bytes). XALGOL and its two derivatives have a very marked field syntax that reflects the hardware of the B6700.

Generalised data structures are not evident at all in the earlier languages, although some specific modes such as lists and stacks occur in several. JOVIAL again was ahead of its time with the TABLE declaration, but PASCAL, ALGOLW, ALGOL68, and MARY all have considerably more general structure mechanisms. In these languages the outstanding difference is whether or not a scalar quantity may have purely mnemonic values. To illustrate this consider the following definition of a mode 'person' in PASCAL:

```
type person = record name : string;
                     age  : integer;
                     sex  : (male, female)
              end
```

Here if fred were of mode 'person' then the following is legitimate

```
fred.sex := male;
```

No such mechanism exists in ALGOL68 (for instance), where the mode definition would be

```
mode person = struct (string name,
                      int age
                      bool male);
```

and it would have to be understood that, if jane was of mode 'person', that

```
jane.male = false
```

means jane is female.

Declarations are required for all these languages, and since ALGOL-like languages are typically block structured, scope is predominantly local (JOVIAL has local scope only in procedures). Initialisation of single cell variables is rare, unlike arrays. Storage classes, however, tend to be evenly split between static and dynamic, presumably for reasons of efficiency. Programmer control over these really occurs only in ALGOL68 and MARY. How items are packed in storage seems to be largely outside programmer control, except in JOVIAL, whose TABLE declaration is functionally similar to the PL/I structure mechanism.

Name equating of identifiers is widespread, but address equating occurs only in ESPOL. The later languages have unions - in PASCAL they are embedded in the record mechanism, while in ALGOL68 and MARY they are completely independent of the structure mechanism.

Probably the most obvious difference between the ALGOL60 and its successors is the presence of string or character modes.

§2.2.2 Operators

There is little difference among the languages in the arithmetic, logical, or relational operators although some languages have special double precision arithmetic operators. ALGOL68 and MARY have a special group of conformity testing relations for unions.

Referencing operators exist in almost all of the languages which have reference modes, ALGOL68 being the exception. The most common is the deref operator, but where dereferencing is implicit (as in ALGOL68, XALGOL, ESPOL, DCALGOL), there is sometimes a ref-to operation, particularly for string pointers.

String operations tend to be machine dependent, particularly as to whether only single character or whole string operations are available. The basic operations are relational but move, size, and substring functions are evident where strings may be manipulated as a whole. Translate operations are not common, but table membership appears in several languages.

The assignment operation generally holds for all modes in a language although the method of determining the mode to be assigned may become complex (as in ALGOL68).

Priority among operators is generally standard, with the exception of operators peculiar to a given language (such as a shift operator). In cases where new operators may be declared care has to be taken to ensure that the priorities of binary infix operators are not unusual.

Coercions vary considerably from language to language. Throughout this family (unlike PL/I) the implicit coercions are those normally expected, such as real→integer and dereferencing. Machine dependent or unexpected coercions are nearly always explicit. With the exception of real→integer the implicit coercions are normally 'widening' operations.

§2.2.3 Control

One of the identifying features of this language family is the block and procedure structure. These blocks serve (in almost all cases) as a method of obtaining local identifier scope as well as statement grouping. Declarations are most often limited to the heads of blocks, with the exception of ALGOL68 and MARY. These two languages are expression languages rather than statement languages, and blocks (closed clauses) may have values. No languages have named blocks.

Most of the languages have gotos, although their use tends to be more restricted in the more recent languages. PASCAL, for instance, limits their scope to within procedures, and has only integer labels. The computed goto is sometimes absent as well.

Selection in the earlier languages was limited to conditional and biconditional forms, but none of the later languages omits some form of numeric selection (or case statement). Probably the most advanced of these is that of PASCAL where the labels may be scalar value mnemonics (see §2.2.2). The exception condition in most numeric selections is equivalent to an invalid array subscript.

Procedures are always permitted to return a value, though the mode of this value may be severely limited, generally to a scalar. Parameters are passed as a mode and, with one exception (ALGOL68), their formal use is indicated separately (value, name in ALGOL60; const, var in PASCAL). Generally procedures may be recursive without further specification, but they never have multiple entry points.

Software and hardware interrupt mechanisms tend to be present only in the more machine-oriented languages such as XALGOL, DCALGOL, ESPOL, and IMP (Edin).

Process control only appears in the ALGOL68 subfamily, and the XALGOL subfamily. In the former, parallel elaboration is part of the language and sema's are used to control such elaboration, while in the XALGOL family the task control mechanism (coroutines, independent and dependent asynchrous tasks, events for semaphores and resource control, interrupts, locks) is quite machine-oriented.

The basic ALGOL60 looping construct is for statement, where the loop is tested at the top, and optionally stepped. Most implementors have found it necessary to extend looping constructs to include tested-at-bottom, and occasionally a next clause. In these cases the syntax has been altered to include while <cond> do <stmt> and do <stmt> until <cond>, or by using repeat as a keyword (PASCAL, IMP).

§2.2.4 I/O

Stream I/O was indicated in the ALGOL60 revised report and many implementations of ALGOL60 reflect this. Because the designers of ALGOL60 left the I/O mechanisms undefined, the implementations of I/O processes vary considerably from machine to machine (and the syntax from language to language). Most (if not all) of the successor languages have a record form of I/O, including some mechanism for formatting. In most cases there appears to be no distinction between buffered and unbuffered forms of I/O.

§2.2.5 Machine Dependency

There is a large variation in the machine dependency of the languages in this family. Languages designed for general use are the least so afflicted, while those designed for systems programming are the most afflicted, as one might expect. At least one general-purpose language was also designed for systems use, and this has a considerable proportion of its syntax machine dependent (and as might be expected, this is heavily used [Bro 74]). Overall, the machine dependence of the general purpose languages tends to be limited to those areas (such as I/O and string handling) which were left undefined by the revised report. On the other hand, in languages intended for systems programming (such as ESPOL, MOL 360, MOL 940), machine dependence takes such forms as access to machine registers, pseudo-procedures for peculiar instructions and shift operators.

§2.2.6 Extensibility

The earliest attempts at extensibility are in JOVIAL which had both macro, table, and operator declarations. While most of the successors to ALGOL60 have some form of macro available in them, only the more recent have made any real advance into extensibility. The prime reason for this seems to have been two papers in the middle 1960's [Gal 67, Wir 66], which made suggestions for extensions to ALGOL60. Since then new languages rarely do not have a data-structuring mechanism. This is in several cases generalised to allow new modes to be declared, and in these cases declaration of operators is always available. Unions (which exist in PASCAL, ALGOL68, and MARY) are the latest development in extensibility.

Fig 2.2 PL/I Hierachy

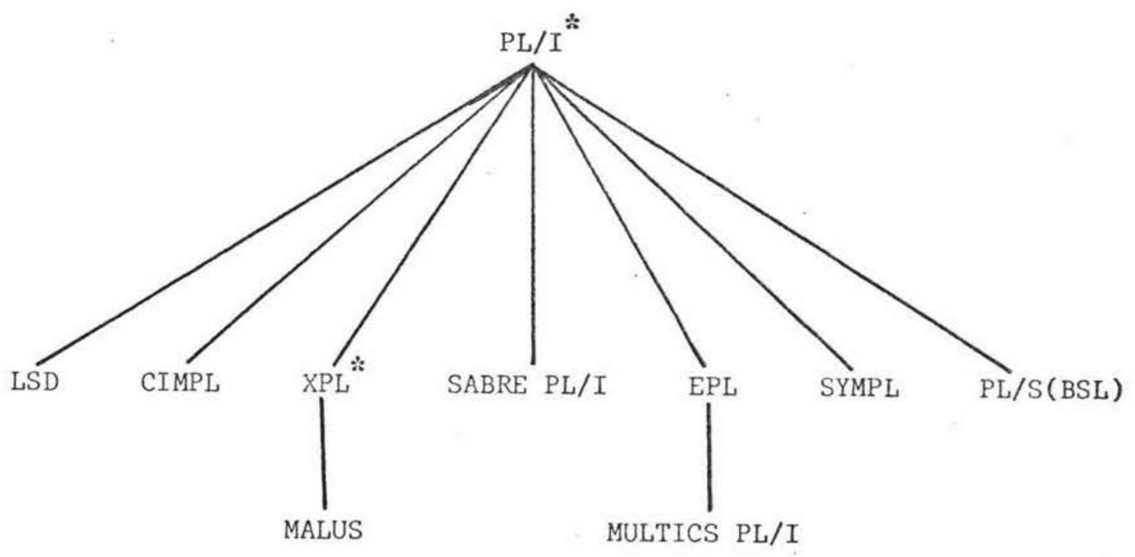
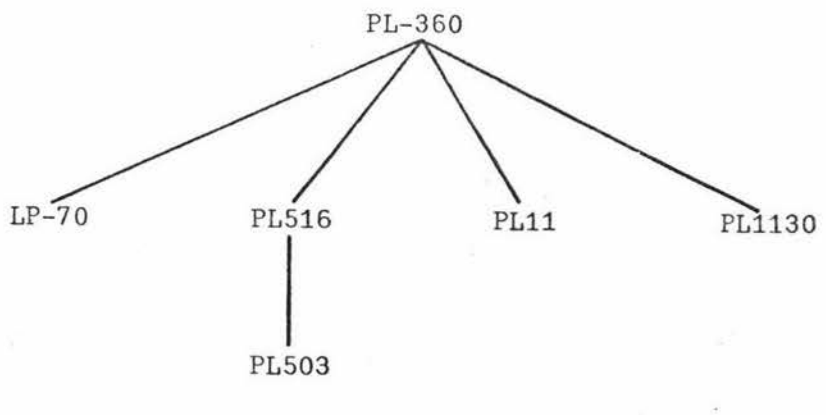


Fig 2.3 PL-360 Hierachy



§2.3 PL/I - Based Languages

The PL/I - based languages form the second-largest family in the systems programming area (Fig 2.2). All the languages (other than PL/I itself) were designed for use in an area of systems programming, and the authors are generally explicit as to which one. The most interesting difference between this family and the ALGOL family is that, while the latter are generally built up from the base language by inclusion of additional features, the PL/I based languages are generally an extended - subset languages. That is, the language designers have cut back, rather than built on, the base language. Some authors look upon the complexity of PL/I as a prominent fault. [Ber 72]

§2.3.1 Data

In the main, the data types and data structuring mechanism of PL/I have been carried into the successor languages. XPL and MALUS are exceptions to this. XPL, for example, does not permit declaration of data structures (except one dimensional arrays) and has only fixed, character, and bit-string modes. In the remainder, there are many minor variations. Strings may have a static maximum length as in PL/S, LSD, and SABRE PL/I. Fields may be available as bit-strings with a maximum length, or as multiple bytes. This may serve to indicate the packing required. There is a distinct tendency away from specifying precision in terms of digits, as for example in LSD where numeric data may be half word, word, or double word. The major area of divergence is in the handling of the various storage classes. LSD has the STATIC, AUTOMATIC, and BASED classes from PL/I (although the BASED mechanism is a little different), has STACKED instead of CONTROLLED, and additionally has ENTRY and CONSTANT. Multics PL/I and SABRE PL/I both have different BASED mechanisms, and PL/S appears only to have AUTOMATIC, STATIC, and BASED. In all cases (including XPL and MALUS), there is initialisation available within declarations. Most of the languages flag undeclared variables.

§2.3.2 Operators

As before, most of the operators from PL/I have been carried into the succeeding languages. Of course, where a data type has been eliminated, operations on it have been eliminated also. Significantly, left and right logical shifts, and logical exclusive - or operators have been added to several of the languages (PL/S, XPL, MALUS, LSD). Referencing operations have undergone some minor alterations, generally to make them easier to use. Although string handling operations vary considerably between the languages within this family, (in the case of PL/S they have almost vanished) they appear to be attempts to make use of available hardware (particularly the 360/370 series hardware) and because of this they often perform similar functions (for example FIND in LSD and INDEX in SABRE PL/I). There seems to be little or no alteration of the assignment operator, or the standard priority of operators. However, the amount of implicit coercion has been considerably decreased. There is no implicit coercion in LSD or PL/S, and in XPL it is limited to some widening coercions. Explicit coercions, however, are provided for almost every case.

§2.3.3 Control

Whereas PL/I itself was a block-and-procedure-oriented language, several of its successors have become procedure-oriented languages. This means that the begin-end combination serves only to group statements and is not used to give local scope, or automatic storage to variables. XPL, MALUS, LSD, MULTICS PL/I are examples. The goto remains the same as standard PL/I in all the languages, but selection (which is standardly conditional, and biconditional in PL/I) has been extended in some cases (XPL, LSD) to include numeric selection. In LSD the exception condition is trapped to a special statement rather than forcing a runtime error. Procedures may have values (though this value may not be a structured mode) and parameters are almost exclusively call-by-reference. In several cases procedures may have to be explicitly declared recursive, if this facility is desired. LSD also provides for omitted parameters. The PL/I ON statement provides for handling of software and hardware interrupts, although this mechanism is not available in at least one language (XPL). Because the original specification of process control (in PL/I) was quite loose, there is considerable variation in the languages which have some control of this sort. The exact form appears to depend on the operating system which is intended to run behind the compiled program. (cf LSD, MULTICS PL/I). Loops, basically tested-at-top and stepped, have been extended in the case of LSD to include tested-at-bottom, and an exit clause.

§2.3.4 I/O

In nearly all cases, the complex PL/I I/O and formatting routines have been omitted or replaced by a smaller set of simpler routines. These appear to include both stream and record I/O but without formatting.

§2.3.5 Machine Dependency

XPL, MALUS, LSD, PL/S permit the introduction of inline code through a code statement. In XPL and MALUS it is limited to one instruction. Several of the languages make use of operations which are machine dependent, such as logical shifts. XPL, PL/S, and LSD at least have register declarations, and both LSD and PL/S give the programmer considerable control over segmentation of code and data.

§2.3.6 Extensibility

The extensibility facilities of standard PL/I include extensible data types (through the BASED storage class), and the compiletime macro facilities. The latter mechanism has been replaced in almost all of the successor languages. XPL, for example, has only a simple unparametered macro facility, while LSD has considerable mechanism for extension at each of the pre-parse, post-parse, and code generation phases. This is described in somewhat greater detail in [Ber 72].

Fig 2.4 Other Languages

String and List Processing Languages:-

SNOBOL* LISP* TRAC APL AMBIT/L

Compiler Generators:-

COGENT CWIC360 APAREL GARGOYLE TRANDIR FSL META TMG

Algorithmic Languages:-

SYSL BCPL BLISS* SIMPL-X COBOL PROTEUS SUE
GPL FORTRAN-LRLTRAN

§2.4 Other High-Level Languages

Figure 2.4 lists the remaining high-level languages. They are divided for discussion purposes into three groups, according to usage. It should be noted that within the compiler-generator group it is difficult to distinguish an algorithmic language from a compiler-generating system. This is not considered important in this context.

§2.4.1 String and List Processing Languages

Languages in this group, because they were designed for a specific purpose, are not easily described in the manner used previously. An attempt to do so might give a deceptive appearance of the individual languages. What follows is a brief general description of the languages and their capabilities with little regard for specifics.

Apart from APL, the prime objective of these languages is the processing of runtime-dynamic data structures such as strings and lists. With one exception data items are considered as strings or lists, although they may be operated on, at times, as if they were of other modes. Thus, the mode of a data item depends largely on the context. Array-like modes are available in some of the languages, and some languages have a general data-structuring mechanism.

The basic operations within the languages (those on strings and lists) may be implied by the structure of the language statements. Priority of infix operators varies from language to language. These languages are generally interpreted rather than compiled, and often this is used to give a program the ability to extend itself at runtime. With the exception of AMBIT/L, the only program structuring tool available in the group is the procedure. Control generally passes from statement to statement via gotos or an equivalent explicit construct. Mechanisms for passing parameters to procedures vary considerably, and a number of standard functions are provided for the programmer in three of the languages.

I/O is basically stream, and the languages are all machine independent.

§2.4.2 Compiler-generators

The compiler-generating languages, like the string and list processing languages, are highly specialised. This means, as it did in §2.4.1, that their modes, operations, and control structures are limited to those required for the specific application. Because of this, there is little to be gained by close study of this group. The group is discussed only in order to make this chapter complete.

The basic data modes are strings, stacks, and integers. Identifiers are generally undeclared and typeless. Operations available include those necessary for string manipulation and parsing. Program structure virtually does not exist for these languages - control passes from statement to statement through implied goto mechanisms. I/O is generally stream, and since most of these languages would normally be run interpretively they are not machine dependent. Unlike the string and list processing group, they do not have any self-extension capabilities.

§2.4.3 Algorithmic Languages

Seven languages in this group were designed explicitly for systems programming, and another two (FORTRAN and COBOL) are languages in general use. LRLTRAN is based on FORTRAN, but the remainder do not acknowledge a particular base language. The intention in this section is to identify and discuss the areas in which the systems languages in this group differ from other languages.

Three of these languages (BCPL, BLISS, and SIMPL-X) have only one primary data mode. In each case, this mode corresponds to a standard storage cell of twenty four to thirty six bits. These modes are weakly typed, and for arithmetic operations behave like integers. The only additional mode available in SIMPL-X is the single dimensional array, but BLISS and BCPL have considerable facilities for building structured modes. The concept behind both these facilities is a distinction between the 'value' a name possesses, and the value of the cell to which that name refers. This distinction is not available in most languages.

BCPL has two methods of evaluating expressions containing names, address (lmode) evaluation and data (rmode) evaluation. The left hand side of an assignment symbol is evaluated in lmode, while the right hand side is evaluated in rmode. The method of evaluation can be forced by use of two unary infix operators, lv and rv. The following example shows how this may be used to build a simple data structure

```
let V = vec(10)
rv(V+3) := rv(V+4) + 10
```

The first statement declares and initialises a variable V to point to an area of 10 adjacent cells. The second statement assigns the value of the fourth of these ten cells, plus 10,to the third. V is effectively the name of a vector. The general form rv(x+y) is found so useful it is abbreviated to x↓y, and so the second statement may be written V↓3 := V↓4 + 10. This mechanism is combined with a simple but effective procedure mechanism to provide arbitrary data structures. However data protection is non-existent.

The BLISS structuring mechanism is similar but clearer. A name always stands for the address that name possesses. An explicit dereferencing operator (.) is provided and must be used to obtain a value. Thus .A means 'the value of A'; and x←.x + 1 adds one to the value of x. It is possible to define a mode by giving its accessing mechanism. Thus

```
structure matrix34[e1,e2]=(.matrix34 + .e1*4 + .e2)
```

defines the accessing mechanism for a three by four matrix. This is used by mapping the structure onto a chunk of storage. For example

```
local space [12];
map matrix34 space;
space [2,1] := .space [2,2] + 6;
```

Bliss storage classes are local, global, own, and register, and a controlled class. The method of structuring data varies slightly between these.

It can be seen that these structuring mechanisms are quite general, but that their use may become tedious.

LRLTRAN extends the standard FORTRAN data types by (a) use of a sophisticated macro system to give a structured appearance to data, and (b), permitting bytes of variable length to be mapped into part of a whole word.

A prominent feature of the operations available in this group of languages is the extensive set of operations for partial cells. In addition to the standard logical operations, there are shift, exclusive or, and masking operators. Other than these, operators are only provided for the primary data modes (which in several cases means no real arithmetic),and for 'address of' and dereferencing operations.

The most significant difference between this group of languages and any others lies in the control area. Three of the languages in this group (SUE, BLISS, SIMPL-X) do not have explicit goto statements. These are the only languages in this survey to omit this control feature and replace it with other more readable constructs. Basically what has been done is firstly, to include a full set of standard control statements (conditional, biconditional, numeric selection, loops tested at top and bottom, stepped loops, procedures), and secondly, to add extra statements to exit any closed environment such as procedures, loops, and statement groups. These exits may be conditional or unconditional and may carry a value with them. Such attempts to eliminate the goto appear to be at least partially successful [Ber 72].

The more modern languages in this group also provide for process control in the form of coroutines (dependent synchronous tasks). These generally take the form of an invocation of a procedure (all of whose parameters are evaluated at invocation) indicating that it is to be run as a coroutine, and giving other system-dependent information. For example, in BLISS

create run (a,3) at 100 length 1000 then exit

invokes the procedure 'run' as a coroutine with stack beginning at location 100 of length 1000, and exit's on termination of the coroutine. Semaphore equivalent constructs are also provided in some of these languages.

The languages in this group are not exceptional in the remaining discussion areas. They have provision for inline assembler language but are otherwise machine-independent, and with the exception of SIMPL-X, their extensibility is limited to data structuring and macro facilities.

§2.5 PL360 - Based Languages and Assemblers

Towards the end of the last decade there was a major development in the field of assembly languages, the structured assemblers. The first of these was PL360, and since then some five or six further structured assemblers have appeared whose indebtedness (stated or otherwise) to PL360 is obvious. (Fig 2.3 page 12). The basic intention of the authors of these languages was two-fold - firstly to improve the readability of the assembly language, and secondly to allow full access to hardware of the machine. The way in which these intentions have been implemented is of major importance, since no one could deny their possible advantages.

The basic tools which have been used to perform the implementation are these. One, make the assembler free format. Two, include considerable program structuring through the use of blocks, procedures, conditional clauses, and looping clauses. Three, allow the use of meaningful symbolic names for all data items, including registers. Four, where possible permit groups of assembly language instructions to be replaced by an expression-type phrasing.

It can be seen from this that structured assemblers offer considerable advantages over conventional assembly languages while adding no major disadvantages. For this reason, the discussion here is confined to structured assemblers.

§2.5.1 Data

Each language has scalar modes corresponding to (a) the cell (smallest addressable unit of storage), and (b) any multiples of a single cell that are recognised by the hardware, and (c) mnemonic register names. Thus PL360 has BYTE, SHORT INTEGER, INTEGER(LOGICAL), REAL, and LONG REAL (corresponding to one, two, four, four, and eight bytes respectively), while PL11 has BYTE, LOGICAL, and INTEGER (one, two, two bytes), and PL503 has only INTEGER (one word). Machine registers are known by mnemonics (RO, FO, F01) but in most cases there is a synonym mechanism for making these more meaningful. For example, in PL360 <u>integer</u> <u>register</u> exp <u>syn</u> RO.

Arrays of all non-register scalar modes may be declared. These are one dimensional with a fixed base value which depends on the machine. Strings are normally treated as an array of bytes, but where byte addressing is not possible, as on the ELLIOT 503, the characters are packed into cells, The basic provision for strings appears to be the storage of literal constants.

There are no attempts to provide for fields, structures (other than arrays) or reference variables.

Identifiers must be declared, and in some cases may be made local or global. In all cases variables may be initialised at declaration, and in all but one case (PL503) they may be either name equated or address equated through a syn clause (or similar).

§2.5.2 Operators

Infix operators are restricted to those whose function is accomplished by a single machine instruction. Commonly, mnemonics are provided for some unary operators such as shift operators. Most of these languages do not have an operator priority mechanism. The main reason for this is that there should be a direct correspondence between the expression forms and machine instructions. Thus the assignment statement normally contains only two distinct identifiers, in the form a = a + b or a = a + 3 or a = b. Similarly logical expressions contain two operands. The form of these assignments is directly dependent on the addressing structure of the machine.

As with conventional assemblers, the typing of variables is very weak. However, in the PL360 family typing does determine which one of a family of infix operators (represented by the same symbol) will be used to evaluate an expression. For instance, on a 360, there are generally nine machine operators corresponding to any arithmetic operation. Coercions do not exist in these languages.

§2.5.3 Control

The constructs for control are the only place where the programmer is not directly aware of machine instructions being used. These languages are more procedure-oriented than block-oriented; blocks generally only serving to group statements. Simple goto statements exist in all these languages.

Selection mechanisms vary from language to language. All languages have conditionals and biconditionals but only one or two have a form of numeric selection. PL360 for example has a case statement.

While procedures occur in all the languages, their implementation, particularly the parameter passing mechanism, is very machine-oriented. The only parameter of a PL360 procedure is a register, while PL503 permits passing of integers, constants, and strings. The procedures do not return values (except through parameters), are not recursive, and do not have multiple entry points.

Looping constructs are generally extensive. In most of this family they are equivalent to those which are available in ALGOL60, that is tested-at-top and stepped, although the syntax varies considerably.

§2.5.4 I/O

None of these languages have I/O facilities exceeding those available in assembler.

§2.5.5 Machine Dependency

This is obviously total. Only the basic skeleton of the language applies to most machines. Some of these languages have facilities for controlling segmentation of programs.

§2.5.5 Extensibility

None of these languages are extensible.

§2.6 Summary

It is evident now that this survey has not accomplished what was originally intended. There are two reasons for this: firstly, there are a large number of languages spanning many machines and language areas, and secondly, it is difficult to estimate the extent of use and success of any particular language in the systems programming area.

Thus, while the survey has not made it clear exactly what language features are desirable, it has shown which parts of systems languages are undergoing the most development. Hence it has also shown the deficiencies (and by implication the adequate parts) of systems languages. In particular, data structuring and goto replacement are two areas undergoing major change, and thus it may be said that these areas have in the past been deficient.

What has not been made evident however is the inadequacy of facilities for string manipulation, or task control, or partial-cell manipulation. The only indication that the survey gives of this is that these facilities are different in almost every language. It is for this reason that I have made the independent survey of systems programming requirements in §3 and §4.

§3 Systems Programs : Common Features

A systems language is used to implement systems programs; hence in order to decide what features a systems language should support, it is necessary first to define the term 'systems program' and then to examine the various types of systems programs so as to elicit the features which must be present in any implemented systems language. Two things must be made clear at the outset. Firstly, a large number of the features may be representative of a larger class of programming languages although we are really interested in the differences between systems programming languages and general purpose languages. Secondly, a fairly substantial part of the features needed are reflected by the implementation rather than the language itself; for example merge-editing, efficiency or sectional compilation.

§3.1 Systems Programs - Definition and Scope

§3.1.1 Definition of the term 'Systems Program'.

While most programmers are aware of the meaning of the term 'systems program', it proves almost impossible to get a working definition onto paper. Although there is no widely accepted analytic definition of the term, Bergeron [Ber 72] does attempt such a statement. Wulf [Wul 72] defines systems programs much more clearly in terms of their discernable properties. "They :

(1) must be efficient on a particular machine

(2) are large, probably requiring several implementors

(3) are 'real' in the sense that they are widely distributed and are used frequently (perhaps continuously)

(4) are rarely 'finished', but rather are elements in a design/ implementation feedback cycle."

These form a reasonable working definition with which other authors agree to a large extent ([Ber 72], [Sam 72], [Don 72]). However, I would add one more property: -

(5) can be distinguished from applications programs in that they are rarely directly productive to the user.

This identifies one major property that is inherent in the nature of systems programs. That is, they are used to control, support, and define a user's algorithm without actually performing the algorithm itself. This is obviously a distinction of degree rather than kind. Several people have made remarks which glance upon the property. For example Donovan [Don 72]: "Systems programs . . . . . were developed to make computers better adapted to the needs of their users" and Cox [Cox 71] while discussing operating systems said that they make up the difference between the hardware that was designed and the 'virtual machine' that the salesmen sell.

This method of definition, however, does not provide much insight into the language features which systems programs require, but rather gives an indication of the nature of programming involved. Hence, like Sammet [Sam 71], I prefer to define a 'systems program' by listing the classes within which it must fall. Systems programs, then, belong to one of four primary categories.

(1) Language Implementing Programs :- all those programs which manipulate or run an algorithm coded in some source language. Includes compilers, interpreters, macro processors.

(2) Supervisory Programs :- all those programs which supervise the running of a machine, or manipulate machine-code programs. Includes supervisors, monitors, loaders.

(3) Runtime Support Routines :- subroutines which are used by a program at runtime to support features which are not available in the hardware. Includes device handling routines, I/O routines, floating-point arithmetic where this is not available in the hardware.

(4) Special Applications Programs :- those applications programs which by nature require greater access to hardware features than a 'high-level language' permits. Includes dump routines, dump analysers, matrix manipulation routines, data base management.[1]

Although these classes include instances which would not under any circumstances be called systems programs, they do include virtually all cases of systems programs. Those which are truly systems programs are those which further have the five properties as listed above.

§3.1.2 Processing Levels in Systems Programs

The primary classes of program just mentioned provide a convenient division through which the features of systems programs can be examined. Within any program there exist separable secondary processing types, for example table searching, scanning, sorting,or stack manipulation. These secondary processing types, hereinafter referred to as forms, are generally bounded in context: that is, they are confined to several closed areas of program, and within these areas or subalgorithms they are locally salient features of data and processing. For example, a compiler may have string scanning as one of its forms. This may occur in several parts of a compiler, but in each of these places the data types are similar, the primitive operations are the same, and flow of control passes along similar paths. These similarities directly indicate the requirements of systems programs. Thus section §3.2 attempts to reduce the four primary classes to these forms for detailed study. Note, however, that certain forms are well integrated into the structure of the algorithm (for example error recovery), and some programs retain a global unity which makes this sort of top down analysis difficult.

[1] Efficiency considerations often force a program into this category.

Identifying these forms and noting the frequency and/or necessity of their occurrence within the whole realm of systems programming provides one objective means of studying the requirements of systems programs. It is my contention that many authors of systems languages have not made a sufficiently objective study, but rather have decided what would constitute 'nice' (as opposed to necessary) elements of systems language design. This fact is amply evidenced by the plethora of systems languages presently available.

While realising that languages must change somewhat to accommodate different and new hardware, and more advanced software techniques, it is not obvious that this need be much more than superficial. (For instance, the apperance of array processing hardware has not necessitated new general purpose languages). In fact the reverse is true.

§3.2 Breakdown of Processing Features

§3.2.1 Language Implementing Programs

There are four main varieties of language implementing programs:- compilers, interpreters, assemblers, and macro-processors. These are listed below together with their component parts.

Compilers :-

which generally comprise

(a) lexical analysis

(b) parsing and reduction

(c) code production and optimisation

(d) error recovery over (a) and (b)

(e) input/output at a high level.

Interpreters :-

which generally comprise

(a) lexical analysis

(b) parsing (but not necessarily reduction)

(c) interpretation

(d) error recovery over (a), (b) and (c)

(e) input/output at a high level.

Assemblers :-

which generally comprise

(a) lexical analysis

(b) code production

(c) error recovery over (a) and (b)

(d) input/output at a high level.

Macro Processors :-

which generally comprise

(a) lexical analysis

(b) simple parsing

(c) input/output at a high level.

Thus the salient components of language-implementing programs are

(a) lexical analysis (occurs in 4)
(b) high-level I/O (occurs in 4)
(c) parsing (occurs in 3)
(d) error recovery (occurs in 2)
(e) code production (occurs in 2)
(f) interpretation (occurs once)

Here types (a) - (c) typify this category of systems programs.

Lexical analysis, which occurs in all four types of program, involves the scanning of text, breaking it up into terminal symbols, checking on symbol formation, recognising identifiers and reserved words, and conversion of text to internal data types. The following sub-processing forms are used in performing these tasks

(a) scanning, comparison and manipulation of variable length strings
(b) accessing table structures containing differing modes (symbol + reserved word tables)
(c) conversion of data (e.g. string to various internal modes)
(d) decision making (either through tables or program structure)

Input/Output used in these programs is generally at a high level, that is, fully buffered single statement transfer operations. The operations need not be formatted, but considerable facilities must be available for string manipulation, particularly concatenation, and conversions from binary to string and vice versa. Both random access (for code files), and serial access (text files) are necessary. The forms required are

(a) single statement, buffered, random and serial access I/O
(b) format conversion
(c) string manipulation
(d) and conversion to and from string representations of internal data items.

The parsing process is generally accomplished in one of two ways - the decision-making process is built into the program through subroutine structure (as in recursive descent), or it is accomplished through the use of stacks and a decision making structure (as in precedence analysis, or list structure form of syntax). The forms required are

- (a) extensive program structuring facilities including recursive procedures
- (b) stack manipulation
- (c) list structure manipulation
- (d) decision tables.

Error recovery is one of the most important components of any systems program, but because of the messy nature of standard techniques, effective language support is scarce. This area is correspondingly more important than component areas where standard language techniques are adequate. Basically, what is required in error recovery is access to both data and program across distinct component area boundaries. Thus the error recovery process can both know of and adjust the state of what might otherwise be entirely self-contained components. The following forms are basic to this function

- (a) inter-component communications
- (b) inter-component passage of control (software interrupts or an equivalent)

Code production involves basically a manipulative knowledge of all machine instructions and cell formats, and use of tables (for register allocation, decision making, external symbols etc), while code optimisation requires retention of code and program flow images in graphs, tables, stacks etc. If folding (evaluation of constant sub-expressions) is to be performed, then the optimisation process must be able to perform operations on all the data types in the language being compiled. The forms used are

- (a) a decision making structure
- (b) manipulation of table structures containing mixed modes
- (c) operations on modes not standardly available in the language in which the compiler (assembler) is written
- (d) stack and list manipulation
- (e) manipulation of partial cells.

The forms from the above paragraphs are collected into the table below.

decision making	(occurs 9 times)
table structures	(9)
string manipulation	(8)
conversions	(8)
stack and list structure manipulation	(5)
extensive program structuring	(4)
single-statement buffered I/O	(4)
inter-component communications	(2)
inter-component control	(2)
manipulation of non-standard modes	(2)
manipulation of partial cells	(2)

Note that this table does not purport to be complete; rather it is a gathering of processing forms selected because they are peculiar to or relatively more important to the nature of systems programs. Nor do the frequency counts purport to be an absolute measure of the importance of the various processing forms; rather they provide an adequate measure of the relative importance of these forms.

It must also be born in mind that what is being provided here is a basis for the discussion in §3.3, not a complete analysis of systems programs.

§3.2.2 Supervisory Programs

In this and the following two subsections, the procedure followed is identical to that of §3.2.1 above, but is abbreviated in order to avoid considerable repetition.

There are three types of supervisory program to be considered; supervisors and monitors; binders and linkage editors; and loaders. While the overall functions of these are quite distinct, they have some large common component areas. The salient components are:

(a)	manipulation of code files	(3)
(b)	access to system libraries	(2)
(c)	hardware aligned processes	(I/O and interrupt handling)
(d)	task control	(1*)
(e)	resource management	(1*)
(f)	user and operator interfacing	(1*)

The components (c) to (f) belong entirely to the first of the three subclasses of program, and hence have an importance far greater than their frequency count would indicate. This is indicated with a star (*) throughout.

The following is the table of forms derived

manipulation of partial cells	(6*)
manipulation of non-standard modes	(6*)
single-statement I/O	(6*)
manipulation of table structures	(5*)
direct two-statement I/O (unbuffered, random access)	(4*)
intercomponent communications	(4*)
use of restricted or special machine instructions	(3*)
decision making	(3*)
inter-component passage of control	(2*)
extensive program structuring	(2*)
stack and list manipulation	(2*)
conversions	(2*)
string manipulation	(1*)
task initiation	(1*)

§3.2.3 Runtime Support Routines

Runtime or intrinsic routines include hardware extension routines (such as format handling, floating point arithmetic), device handling routines, file handling intrinsics. These routines do not normally have common components, so the forms (derived directly) are

direct two-statement I/O	(2)
non-standard modes	(1)
conversions	(1)
string manipulation	(1)
partial-cell manipulation	(1).

The distinction between programs in this class, and parts of the operating system is not at all clear. I have treated this class as comprising those programs which are loaded or link-edited as an integral part of a user's program. Even this method of discrimination fails for some machines (for example the B6700).

§3.2.4 Special Applications Programs

Programs in this class are regarded as systems programs for one or both of the following two reasons: they manipulate non-standard or sub-cell modes, or they must be more efficient than would be possible in a general purpose language. Such programs as dumpanalyzers, dump programs, directory listers, compacters, emulators might belong to this class. The essential components here are the following

(a) manipulation of code files (1)

(b) access to system libraries (1)

However, this is a gross oversimplification of the situation, as there is a wide variety of programs in this category. The forms involved are

partial cell manipulation	(1)
single-statement I/O	(1)
direct two-statement I/O	(1)
non-standard modes	(1)

This section also serves to emphasize that systems programs must above all be effective (efficient, functional, error immune), since programs may be in this class only for reasons of efficiency.

§3.3 Collected Features

This subsection contains a brief discussion of the various processing forms isolated in §3.2 in the context of systems programs. These forms are collected together in the following list:

table structure manipulation	(14*)
decision making	(12*)
conversions	(11*)
single statement buffered I/O	(11*)
string manipulation	(10*)
manipulation of non-standard modes	(10*)
manipulation of partial cells $^{\Delta}$	(10*)
stack and list structure manipulation	(7*)
direct two-statement I/O (unbuffered) $^{\Delta}$	(7*)
extensive program structuring	(6*)
inter-component communications	(6*)
inter-component control $^{\Delta}$	(4*)
use of restricted or special machine instructions$^{\Delta}$	(3*)
task initiation $^{\Delta}$	(1*)

$^{\Delta}$ indicates that the form would normally be found only systems programs.

Table structure manipulation. Systems programs require or retain large amounts of information in tabular structures (that is structures, whose composition is static at runtime). This includes such items as peripheral status, error messages, reserved words, symbol tables, task status, to name but a few. Systems programs specifically require two features in such structures; firstly that they may contain mixed modes; and secondly that they are initialisable. However, the basic requirement is for a general data-structuring mechanism.

Decision making. Systems programs spend a large proportion of their time making decisions as against manipulating data. (This assertion can easily be verified by comparing pages selected at random from systems programs and applications programs). This decision making is sometimes of an extremely complicated nature, for example the parsing process. Simple conditional and numeric selection statements are not sufficient in the systems programs context. Generalised statements for multiway choice are desirable; similarly recursive procedures (if practicable).

Conversions. Hardware normally provides few facilities for conversions between modes. Normally, runtime support routines are used to make up this deficiency in the hardware (e.g. formatted I/O routines). These systems programs must be coded at the lowest possible level, and if, as in a minicomputer, all conversions are performed by the software, a one-to-one correspondence between source and object code may be the only feasible method. Larger machines, even though they may have some string to binary (and vice versa) conversion in the hardware, still require software to perform the bulk of generalised format conversions. This provides an excellent case for having inline code available in some places, and also for a rationalised syntax for every form of explicit coercion.

Single statement buffered I/O. This raises three important points as far as systems programs are concerned. Firstly, standard I/O should not be omitted from systems languages for, although it is 'high-level', many systems programs can make effective use of it. (Note that this comment applies to several other features of high-level languages). Secondly, if it is provided in a systems language, provision must be made for trapping associated faults (such as disk parity errors). Thirdly, it is common practice for systems programs to utilise other systems programs for performing various functions. While this plagiarism is commonly contained within operating systems, it should not be inhibited elsewhere, and to adequately achieve this may require either an extensible language, or a series of languages (Burroughs ESPOL,DCALGOL, and XALGOL for example).

String manipulation. A considerable amount of the total data manipulated by systems programs is in string form. These strings are either manipulated as complete entities (for data communications, or error messages), or are created or broken down for manipulation of substrings. There is obviously a considerable case for a string mode in its own right, and it is one of the peculiarities of this mode that the subparts of the mode are of the same mode. If syntax for string manipulation is forced to conform to the usual tight keyword-and-bracketed-context form, string manipulation can appear very clumsy indeed (see §5.2.5). Ideally, the syntax for scanning, parsing, and concatenation should be simple, unique, and at as high a level as is practicable.

Manipulation of non-standard modes. A predominant feature of systems programs in general and operating systems in particular is that they manipulate special purpose modes, that is, modes which would not normally be found as a standard type in a general purpose language. Such modes are generally peculiar to the place in which they are used. They are not always static at runtime. If these are to be described adequately at compiletime (to enable compiletime checking), a completely general data structuring mechanism is required, with an extension to permit creation of multiple copies, either through modes, or based variables. It is obviously necessary for the programmer to have total control over the storage allocation of many of these structures.

Manipulation of partial cells. 'Bit fiddling' is generally confined to systems programs. It has two uses: either to pack data within cells to save space, or more importantly to manipulate parts of cells which are directly recognised by the hardware. This latter use may require some explanation. Many machines have hardware to operate directly on composite modes (floating point numbers for example). The individual parts of these modes (e.g. exponent, mantissa) often are not directly addressable because they do not fall on cell boudnaries. These partial cells (or fields) must also be available to the software for manipulation, and would normally be treated as unsigned integers.

Access to fields requires that a mode must be able to be treated as either a complete unit, or as a low-level structure with component fields. That is, the mode must be to some extent weakly typed. The natural extension of the data structuring mechanism to include sub-cell structure may be machine dependent. (See §5.2.4).

Stack and List structure manipulation. While stacks and lists should perhaps have been included in the discussion of non-standard modes, they appear sufficiently frequently in systems programs to be discussed separately. Their provision in the language is a necessity, either through a comprehensive data structuring mechanism, or as standard modes. Their presence highlights two aspects of data structuring. Both structures are dynamic, and because of this manipulating them requires the use of references, either explicitly or implicitly. The current state of the art is such that explicit manipulation of references is necessary. I am not suggesting that this is a good thing, merely that it is currently necessary. Also, instances of list elements must be created at runtime, and some functions of lists may return a list (or a list element), thus further reinforcing the case for generalised modes.

Direct two-statement I/O. Direct unbuffered, or core I/O, is used only by systems programs. It is necessary in order to gain efficiency by overlapping processing with input or output. This type of I/O also gives the programmer greater control over a peripheral device and a decrease in I/O overheads (by avoiding formatting and buffering). The two statements of the I/O are, naturally, 'initiate' and 'wait if I/O not finished'.

Extensive program structuring. Systems programs tend to be large and complicated and often messy. This indicates a requirement for a comprehensive set of program structuring facilities. It is essential to have block and procedure structuring, and it is just as essential for identifiers to have local scope. Considerable attention must be paid to aiding the programmer in laying out the program's structure, and as this is made easier in an extensible language, serious consideration should be given to this. In particular, a simple macro facility (if not abused) can considerably improve program structuring.

Inter-component communications. Under some circumstances, normally error conditions, there may be a requirement for two mutually exclusive components of a program to have access to local information in the other. This is used for such operations as task information, recovery from errors and mutual exclusion with respect to resources. While the normal method of accomplishing this might be to make such information global to both, this may be neither desirable nor necessary (cf Simula 67). Provision should be made for mutual exclusion with respect to data, and also for data to be passed between components via queues or some other mechanism. This applies particularly to information about independent tasks.

Inter-component control. One of the main problems of an operating system is that of neatly terminating or changing the flow of control in another task. Under normal conditions, this can be accomplished through the standard inter-component communications system, although, of course, semaphores or their equivalent must be present to obtain mutual exclusion in critical sections of processing. However, for the handling of error conditions, this mechanism may be ineffectual (for example terminating a task which is waiting indefinitely on a semaphore). The requirement here is for a software interrupt mechanism, so that a task can be forced to change its state. Note that this may be equivalent to forcing a task to execute a bad goto (cf premature termination because of a fault). This problem is really a special case of the much larger problem of multiple exits from controlled environments.

Use of restricted or special machine instructions. An inevitable problem of writing an operating system in a high-level language is that sooner or later the programmer will wish to include inline code. This can occur for several reasons - all perfectly valid. Firstly, the source to machine code mapping may be such that a considerable loss in efficiency is achieved by never inserting inline code. Secondly, the use of some machine instructions may be restricted, either because of the physical state of the machine, or because their use is dangerous under normal circumstances. Thirdly, the systems programmer may require access directly to machine registers, or absolute addresses.

These may be separated into two separate requirements. Initially, there is a need to provide access for the programmer to any machine instruction he desires, while at the same time minimising the chances of his making errors. Subsequently, there is a need to provide for the more common of these dangerous instructions to be provided directly in the language. (Weak typing is one of the best ways of accomplishing this. See §5.2.3.).

Task initiation. One of the most important aspects of a multiprogramming operating system is the running of tasks, that is, processes which compete with each other for the available system resources. As this process is obviously different from the normal procedure calls, it generally requires separate syntax. Language constructs must be available to allow programmers to initiate different types of task. These types are discussed in §5.3.1. Post-initiation communication and control of these tasks has been discussed in the paragraphs on inter-component control and inter-component communications.

The foregoing paragraphs have discussed the requirements of some of the forms of systems programs. Meeting these requirements is discussed in §5.

§4 Effectiveness : Functionality, Error Immunity, and Efficiency

Almost every set of criteria for systems languages contains the term 'efficiency'. Almost none of these explain the meaning that is attributed to the term but rather treat 'efficiency' as somehow self-defining. On reading the arguments concerning these criteria, it is evident that the uses of 'efficiency' differ considerably. It may be used in its broadest sense to indicate overall efficiency in use of a language, or in its narrowest sense to indicate some relative measure of the speed or size of code compiled from a language (presumably against the optimal machine code). Because most glib uses of 'efficiency' are in the narrow sense, and because it is evident that criteria based on this sense are inadequate (in that they ignore other aspects of overall language efficiency), I have instead made use of the term as described below.

The general criterion is for language <u>effectiveness</u>. (This term has been used before with similar meaning [Ard 70].). Effectiveness has three components: functionality (of the language and of the compiler), error immunity (of the language), and efficiency (of the compiled code). Effectiveness thus may be regarded as a measure of the relative cost of developing, using, and maintaining a piece of software (coded in that language, as against others). Two components of effectiveness (functionality, and error immunity) unfortunately have no real measure, and so 'effectiveness' may only be 'real' in its relation to cost of software.

§4.1 Functionality

§4.1.1 Functionality of the Language

A language may be functional in two ways, writeability and readability. That is, a programmer must be able to easily and correctly write the language, and with correct understanding read the language. Writeability and readability naturally depend to a large extent on how suitable the language is for the purpose for which it is being used. This is the first and perhaps most important criterion for functionality of the language; that the language is suited to its use. §3 concentrates on satisfying this criterion.

What reflects on writeability? Firstly, the mapping from what the programmer envisages into how it is realised in the language must be as simple as possible. Thus the language should concentrate on describing algorithms rather than their implementation, and similarly the programming language description of a data structure should not differ significantly from the natural language description of the same data structure (this is probably the greatest criticism of the BLISS data structuring mechanism. See §5.2.2). The mapping must be as nearly one-to-one as possible, in that there should be the minimum number of ways of writing the same thing; thus avoiding the possibility of the programmer making an arbitrary decision as to the 'best' way of writing something. This is most important, as even good systems programmers can mistake the 'prettiest' path for the 'best' path.

Secondly, there must exist a reasonable mapping for almost everything the programmer may want to do, and moreover, exactly what form this mapping has must be obvious. That is, the programmer should not have to struggle to express himself, as this may result in unnecessarily devious code.

Thirdly, the programmer must not be misled by finding that constructs he attempts to use because of similar constructs already used, are either not in the language or have an effect other than that which he expects. This implies that a language must obey the Law of Least Astonishment (itself a proper part of language orthogonality), which states the programmer should be the least astonished as to the function or presence of an arbitrary construct in the language.

for example

```
real procedure p(r,q);
    real r; procedure q;
    p := q(r);
.
.
.
.
z := p(if b then x else y, if c then sin else cos);
```

This is not valid ALGOL60, but it would seem reasonable to be able to be able to write this.

This law also significantly reflects on readability and error immunity.

Readability is reflected in three things. Firstly, constructs must not be deceptive or obscure in their function (Law of Least Astonishment).

for example

```
     in PL/I          DO WHILE  '1' B;                 is obscure

while in ALGOL60,     within a real procedure p
                      p := 3;
                      if p = 3 then . . . . . .       is deceptive
```

Secondly, the language must be such that the structure of a program is superficially apparent. This may be inhibited by the language (as in COBOL, FORTRAN or fixed format assemblers), and may be considerably aided by the compiler (begin-end counts in ALGOL, automatic indentation of blocks).

Thirdly, the language must not contain arbitrary restrictions, such as limiting identifiers to 6 letters.

§4.1.2 Functionality outside the Language

How a program is written may depend quite significantly upon things only vaguely related to the language. The large and complex nature of systems programs makes certain demands on the construction of the compiler and other programs associated with getting a program running.

There are two important functions of a compiler which need to be taken into account here: commentation support, and optimisation. The manner in which a compiler supports these drastically affects program readability.

Comments do not naturally form part of the language structure and, while it is obvious that they are most likely to be needed around some particular parts of the language, permitting them only at fixed points in the language may be seen as an arbitrary restriction. It might be argued that high-level languages are self-documenting and thus do not need extensive commentation. However, since a program does not specify what is being done but rather how it is being done, this argument is deceptive. (The reader may verify this for himself by selecting an algorithm at random from Collected Algorithms of CACM, getting somebody to transcribe the algorithm omitting comments, and then attempt to say what the algorithm does and how it is used). It is essential that comments may be inserted in a program between any two terminal tokens. Perhaps the easiest way of achieving this is to have a single character which denotes the end of the line of program (as in Burroughs ALGOL, ESPOL, DCALGOL, and many assembly languages).

Redundancy in a program often provides an extemely effective means of making that program more readable. This applies particularly to redundancy of load operations. Consider the following single statement from the Burroughs B6700 2.4 MCP.

```
RETURN (M[W := (W := M[MYF+1].PIRF + (IF BOOLEAN(W.[13:1])
       THEN WORDSTACK[R := FINDD1STACKNUM(MYF-BOSR),SEGDICTMSCW
       + W.MYSDIF] ELSE M[DOSETTING + W.MYSDIF]).ADDRESSF]
       .[16-M[MYF+1].PSRF)x8-1:8] & (W)[47:19:20] & R[27:19:20]);
```

This statement, which may only be regarded as programming pornography, in fact probably saves four or five memory references. (In all fairness, the latest MCP has a slightly different representation of this statement, with comments). The point being made here is that proper optimisation might obviate statements of this sort. Machines with cache stores would not even require this optimisation.

Separate and optional compilation are both very important aids in the construction of systems software. The former requires that procedures or blocks can be compiled separately and bound (link-edited), thus saving time by recompiling only what is necessary, and allowing members of a programming team to work much more independently than would otherwise be possible.

Optional compilation can be extensively used for introducing debugging code, extended features which may not always be desired, and so on. For an example, both the DCALGOL and ALGOL compilers on the B6700 are compiled from the same source code, the difference being the setting of a compiletime option.

§4.2 Error Immunity

Dijkstra once made a request for "intellectually manageable programs, which can be understood and for which we can justify, without excessive amounts of reasoning, our belief that they will operate properly under all conditions ..... ". It is a prime prerequisite for such programs that the language used does not contain error-prone constructs. There are three levels on which these constructs become apparent, purely syntactic, mixed syntactic and semantic, and purely semantic.

At the purely syntactic level, it is most important to retain the maximum lexographic distance between all terminal symbols. Thus the distance between GEQ and LEQ is one, and a single miskeyed character can produce another valid symbol; while the distance between GREQ and LSEQ is two, and it takes two miskeyed characters to produce another valid symbol. This is somewhat analogous to the case of the programmer who declares a variable D0 in ALGOL60 (and subsequently finds DO on his listing).

The greatest confusion can arrive at the next level, when syntax and semantics interact. For instance, in ALGOL68, the declaration real r may mean just that, or be a contraction for ref real r = loc real, depending on the context. If the declaration occurs as a formal parameter specification then the declared item is a real, but if it occurs on its own as variable declaration then it is a ref real. Again, the distance between real r := 6 and real r = 6 is one, but the difference in meaning is considerably greater. Thus the distance between any two constructs at the syntactic level should be proportional to the distance between those constructs at the semantic level. This is another form of the Law of Least Astonishment.

There are several rules that can guide the language designer here. Firstly, to ensure a minimum distance of two between keywords in the language. Secondly, to avoid having the meaning of any group of two or more symbols context dependent. Thirdly, to avoid having two similar terminals syntactially valid in the same place (for example | and /, - and _, : and ;). And lastly, to maintain the relation between syntactic and semantic distances.

One particular item arising from the third of these rules is that there should be no implicit declarations, since implicit declaration permits a mispunched identifier to be syntactically valid where it otherwise would not be.

At the purely semantic level, there are only two major items, closely related, to be avoided. These are gotos and reference variables.

Because gotos inhibit program validation procedures, and because they may in their unrestricted form cause scope violations (which have to be fixed up by a runtime intrinsic), their use should either be abolished or at least contained within limited-scope areas. There seems little doubt that unrestricted use of gotos is error-prone and, while the arguments for and against are too long-winded to expand here, §5.3 contains my views and a partial solution.

Hoare [Hoa 73] points out that reference variables are very closely related to gotos in that they can cause scope violations. Checking against this is time consuming and, as with gotos, the best solution would seem to be eliminating them entirely from the language. This problem is no less difficult than that of eliminating gotos from the language, particularly because reference variables may be used to greatly increase efficiency by avoiding repeated use of some addressing mechanism. At the present time, complete removal of reference variables appears unrealistic.

§4.3 Efficiency

Efficiency, in its narrow sense, is a relative measure of two things: firstly, the amount of time a section of program takes to execute, and secondly, the amount of space that section of program takes in core. These should be measured relative to time-optimal and space-optimal machine code programs which perform the same function, but because this is unrealistic they can only be measured relative to a single assembler language program which performs the same function.

The range of values quoted for efficiency varies considerably between languages (as might be expected), and even within a language. In one project it was found that code written by programmers in experience in PL/I was five to ten times worse than that which they could have written in assembly language [Cor 69]. Another study [Ard 70] found that code written by experienced PL/I programmers using the F-compiler was four to five times worse than assembler language. These figures are perhaps quite atypical, and probably result from the generality of PL/I. For instance Terashima [Ter 74] quotes a run-time efficiency of 1.11 times and a space efficiency of 1.17 times for programs written in SYSL, a language specifically designed for systems programming. Efficiency obviously <u>tends</u> to increase with the level of the language, but the reason this is so marked may be due to the bad design of the higher-level languages and their compilers.

It is worthy of mention that Terashima estimates the 'productivity' (≡ functionality) of his language at approximately twice that of assembler language. This should increase with the level of the language, but may not always balance out the decrease in efficiency.

There are several ways of aiding programming efficiency. Firstly, each section of source must compile into the most efficient object code. This might be regarded as totally a function of the compiler, but the language designer can make sure that only the most efficient constructs are available for any particular processing form. There is obviously some trade-off point where a language construct may be too inefficient to use, and unfortunately this often will depend on the machine on which the language is being implemented. Thus the compiler should make the programmer aware of how much code each statement generates, and even what code is generated.

Secondly, where suitable language constructs are unavailable, or grossly inefficient, the programmer should be able to recode them into assembler language. Bergeron et al [Ber 72] have made such a succinct justification for this that I quote it here in its entirety.

"In a few cases, it is impossible for the user to describe a peculiarity of his system. Furthermore, every compiler will have features which restrict some systems programmers. These unsatisfactory conditions can be relieved by allowing the insertion of assembler language as in-line "open subroutines". The section of code thus produced will contradict some of the rules of a systems programming language, especially syntactic clarity, but at times the advantages of low level coding are great enough to compensate for this loss.

When an available facility is expensive in the systems language, judicious recoding of critical portions in assembler language may also be valuable. By taking the machine environment into consideration, a programmer may modify the compiler's target code to raise the overall efficiency of his system appreciably. The following example will show that a tight algorithm at the source code level does not necessarily ensure an efficient program:

> *A linguistics group at Brown University was coding in an early version of PL/I which did not include the TRANSLATE and VERIFY functions. In a key routine, they needed to scan a character string for the first occurrence of one of several eight-bit configurations, returning the index of that character in the string. To implement this, they coded a very tight PL/I loop (which compiled into a large amount of machine code). Finally, a systems programmer realized that they had just simulated the single /360 TRT instruction. By recoding that small, key routine in assembler language, he was able to reduce the execution time of the whole program two orders of magnitude. It was only through low level knowledge of the machine that the program could be made more efficient.*

The trade-off between the efficiency of assembler language and its lack of clarity is easily resolved since such inserts should occur only infrequently when the relevant section of the program is completely thought out. Furthermore, documentation should be provided in great detail to explain the meaning of the code (in fact, the original code in the systems language provides excellent documentation in such a case)."

The most important effects of permitting some form of inline assembler are to allow the language designer to raise the level of other language constructs (thus decreasing machine-dependency), and to increase significantly the efficiency of critical portions of programs. These are both extremely worthwhile.

However, in order that the critical portions be identified accurately (a more difficult task than it would appear [Wul 72]) there should be available to the programmer, through the compiler, a method of timing execution of program modules.

The most important point to emerge is that any application of criteria related to efficiency in particular, and effectiveness in general, will require a compromise between opposing arguments.

§5 Critieria and their Application to the Language Design

The previous three sections have concentrated on determining, for language-related aspects (§2 and §3) and effectiveness-related aspects (§4), the most important criteria. This section introduces criteria related to aspects of extensibility, and for these and the abovenmentioned discusses their interrelationships and application to the language design. In subsection §5.5 an attempt is made to sum up and order the criteria overall, and to provide in part a guide to what aspects the intending designer/implementor of a systems language should consider important.

A large proportion of the criteria previously derived fall into two distinct groups - those related to data and those related to control. The effects of the criteria in these groups are discussed in §5.2 and §5.3 respectively. The remaining criteria are discussed in §5.4.

It must be emphasised that not all criteria indicate concrete requirements of the language (such as provision for task initiation and control, and string manipulatives) but rather form guidelines for the overall design of the language (such as efficiency, orthogonality). These guidelines are by far the most important of the criteria, but they do not themselves construct the language. They have a governing effect in that they may force a choice between two equivalent constructs, but they never create the constructs themselves. Thus they are not considered in isolation but rather when they affect the application of other criteria.

§5.1 Extensibility/Orthogonality

§5.1.1 Extensibility versus Universality.

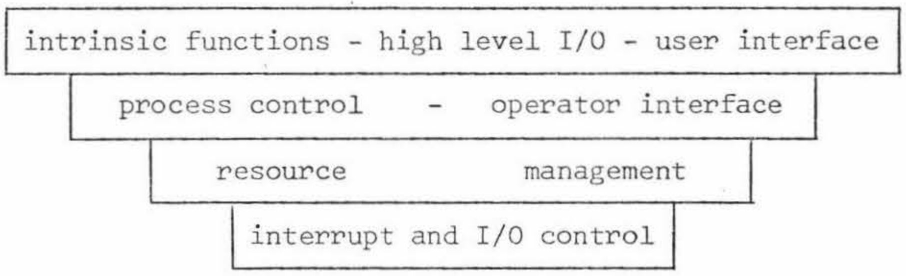
Hardware today presents the systems programmer with great divergence in functional appearance. Because of this, there is general acceptance of the belief that systems programs cannot be made portable without a significant loss in efficiency [Sam 71]. However, this has not reduced a belief that systems programming languages may be made portable. There are basically two ways of achieving this: either by including all features that may be required on all machines or at least a large proportion of them (the universal language approach); or by providing a relatively low-level portable language kernel and means to develop extensions to this (the extensible language approach).

Universal languages have several disadvantages. A programmer may not in fact find the language feature he requires, or he may find it too inefficient to use on his particular machine. The language itself becomes large and correspondingly clumsy, making it both difficult to learn and difficult to read. The definition of the language may make it difficult for a construct to be implemented in the most natural way, resulting in either a slight deviation from the definition, or a loss in efficiency. The language may prohibit utilisation of some hardware feature on an existing or future machine. All of these may be small and detourable items but they combine to provide a sizeable argument against this approach.

Extensible languages, on the other hand, have several advantages, and only one major drawback.

Firstly they promote better structuring of a program or a suite of programs. This is of particular significance to operating systems, where the program structuring is otherwise provided through procedures or macros. For example, an overview of an operating system may have the appearance of an inverted wedding cake as in Fig 5.1, where each layer is involved in building data structures and operations for the next and subsequent tiers, while at the same time retaining a functional unity within the one level.

Fig 5.1


intrinsic functions - high level I/O - user interface
process control - operator interface
resource management
interrupt and I/O control

In a non-extensible language a programmer may have to work hard to make this structuring neat or even apparent. This is not so in an extensible language.

Secondly, constructs can <u>nearly</u> always be built to provide anything the programmer requires, and generally at an efficient low level of implementation. This is because, in describing an extended structure, a programmer is not only specifying the structure but also its implementation. The 'nearly' above is the major disadvantage, or rather deficiency, of extensibility. It might be argued that an extensible language is not worth the effort involved in implementing it if the extensibility is not completely general. This may pose some practical difficulty (see §5.1.2).

The third advantage of extensibility is that since extensive data structuring facilities are required anyway, a large amount of duplication of definition may be avoided by allowing modes to be defined and then associated with different names. Modes also combine extremely well with other language features required, such as weak typing and linked-list manipulation. Having made the language extensible in this respect the jump to more general extensibility is easier and shorter.

§5.1.2 Extensibility Features

Extensibility, then, is a neat and useful tool, as far as systems languages are concerned. Can it be justified in terms of the increased implementation effort? There are three directions in which a language can be extended: extension of data (modes and unions); extension of operations (operator declaration); and extension of control (e.g. by macros). Each of these varies in usefulness according to the environment to be considered and providently, each is relatively independent of the other two, so that only the most useful may be chosen for implementation.

Extension of data is the most useful of the three. It has already emerged that good data structuring is an essential part of systems programming (§3). Non-extensible methods of describing data structures have several drawbacks. A significant amount of duplicated source code may be necessary to declare multiple copies of the same structure, (and methods for avoiding this, such as based variables in PL/I, result in semantic confusion). Programatically imposing a mode on a weakly typed section of storage becomes clumsy, as does the manipulation of cells whose internal layout is determined by hardware rather than software. There are two documented approaches to data extensions, the syntactic approach of ALGOL68, and the semantic approach of BLISS.

BLISS defines the mode through a STRUCTURE declaration, which associates an identifier, possibly with parameters, with an accessing algorithm. This algorithm may then be "MAP"ed on to any available piece of storage. This is a very flexible approach although a little tedious.

ALGOL68 permits the definition of modes, declaration of identifiers as declared modes, or unions of modes. This syntactic approach is a mixed blessing however, for,in dealing with low level modes, some confusion exists between the functions of the definition of the mode and the functions of the declaration of an identifier having that mode. However, a small syntactic rearrangement could avoid most of this confusion (see §5.2). Definition of modes does have a singular advantage in that it permits a consistent way of referring to sub-cellular items such as bits, bytes, and fields. This avoids both the packing problem of PL/I,and the extra declarations for word layouts of ESPOL. On the contrary, I find no such justification for declaration of unions. They appear to have little use in systems programming and require considerable extra effort in implementation. Conformity relations then become essential, and as there is little hardware available for conformity testing, these may require significant amounts of software, included behind the programmer's back. This should be avoided (§4.3).

Extensibility of operations is normally accomplished through procedures or macros. The declaration of infix operators has two semantic pitfalls. Firstly, if a standard operator name is used then care must be taken not to alter implied meaning of that operator, for example, using "-" for concatenation. Secondly, setting the priority of an operator may virtually eliminate readability of the program. If a standard operator name has a different priority for different operands, or a new operator has an unexpected priority,this may prohibit rapid understanding of source code involving those operators. Even having a fixed priority scale as in ALGOL68 may cause confusion until the programmer is totally aware of the priorities of existing operators. Priority settings are too complex to be used with impunity. Thus it is suggested that if infix operators may be declared, then they have a fixed priority (probably the highest) if they are non-standard, or the priority of the standard operator whose name they use. Considering binary and unary operations are only a small part of any set of operator extensions, they could easily be dispensed with as infix operators.

Extensibility of control is one of the least developed aspects of extensible languages. Because control is so embedded in the syntax of the language, extensibility with respect to control requires the ability to drastically alter the syntax and semantics of a language during a compilation. This is such a difficult task that it does not seem ever to have been accomplished successfully. Rather, most extensibility of control is provided through macro processors which, of course, work totally at the pre-syntax stage. Simple parametered macros such as those in Burroughs B6700 ALGOL provide a procedure-like appearance upon invocation which may be confusing. Avoiding this requires a much more complex system which would not normally be embedded in the language, for example ML/I. Perhaps if a language has adequate control primitives, this sort of extensibility might be better avoided completely.

§5.1.3 Orthogonality

Orthogonality is one of the criteria worth bearing in mind as a guideline. Its real value is as a contribution to making a language more functional - that is, more writeable, more readable, and less error prone. It helps a language to conform to the 'Law of Least Astonishment'. A programmer can form a construction more easily from experience with similarly formed constructions. Conversely, the programmer is more likely to correctly interpret the function of a construction he sees before him because of his experience with similar constructions.

Thus the real value of orthogonality is as a means of maintaining internal consistency of the language.

§5.2 Data-Related Criteria

Criteria in this section fall into two broad groups: firstly, those which merely predicate the presence in the language of a data type and operations thereon and where there is little or no choice of implementation involved; and secondly, those which for some reason are peculiar to the nature of systems languages or which have considerable flexibility of implementation. The former group is dealt with for convenience in §5.2.1 and in arbitrary order (since it deals with necessities). The latter are dealt with individually in §5.2.2 and succeeding subsections, and in my order of decreasing importance. That is

Data Structuring
Weak Typing
Fields
Strings
Storage Allocation
Array Processing.

It should be noted that the ordering does not imply that one criterion should be satisfied before others, but rather that all should be satisfied to an extent dependent on their prominence.

§5.2.1 General Data Requirements

There are several single cell data modes required. Integer or fixed point arithmetic variables are evidently necessary, as are the basic arithmetic operations. However, as floating-point hardware may not be available, providing a floating-point data mode may lead to the impression that the corresponding operations may be used with impunity. If a floating point mode is provided, as I think it should, then all operations on that mode requiring a procedure call should be flagged as such.

Logical values must be provided, and may be considered either as a one bit long field, or an integer value (e.g. 0 or 1). This choice is normally not hardware dependent, and since the one bit field cannot be confused with a floating point variable of value 0 (as may occur in PL/I) the former is preferable.

Precision on modern machines is limited to multiples of the basic storage cell, and therefore different precisions should be considered as different modes rather than as variants on a single mode (cf NELIAC).

Arrays of all single cell modes are also essential items. These should be multidimensional if no data structuring facility is provided (see §5.2.2). There is no necessity for a flexible lower bound, as a fixed lower bound improves efficiency and requires little adjustment for an experienced programmer (although it may reduce readability). However since whether the indexing base is one or zero largely depends on the hardware, it would be necessary to have the base value made clear to any intending programmer. (Operations on arrays are discussed in §5.2.7.)

Strings and string operations are discussed in §5.2.5.

Fields and partial words are discussed in §5.2.4.

Structures are discussed in §5.2.2.

For the sake of efficiency, reference variables are necessary. Of late there has been some discussion [Hoa 73] as to ways of eliminating references from programming languages, since it is indeed clear that they are in effect analagous to the goto - that is, they may obscurely connect two otherwise unrelated pieces of data. However, hardware currently makes extensive use of such modes, as does software (cf linked lists, dynamic tables). I feel that current software engineering has not yet developed adequate mechanisms for hiding the reference mode from the programmer, particularly in systems work. It is difficult to create a general data structuring mechanism without them, and almost impossible to perform efficient string manipulations (see §5.2.5). Unfortunately little protection can be built into the syntax against deallocation of a referenced area (dangling pointer problem), but some can be gained by typing reference modes as in ALGOL68. This is consistent with generally strong typing, and although most machines do not distinguish in the hardware between reference variables referring to different modes, some (e.g. Burroughs B6700) definitely do.

Declaration of all data should be mandatory. There is little room in systems programming for the type of errors that may be generated by allowing implicit declaration. Identifiers should be local in scope, but should be able to name variables whose storage class is either local dynamic or controlled (see §5.2.6). Initialisation at declaration should be made available for all data, since this normally causes little overhead on storage allocation, particularly in a stack machine. Some provision could be made for setting default initialisation values (e.g. zero for arithmetic modes).

§5.2.2 Data Structuring

Data Structures are perhaps the most important single facility in either a general programming language or a systems programming language. It might be argued that, in a language designed for implementing an operating system, that facility is second in importance to a facility for handling task (or program) structures. The proportion of an operating system which deals with data structures is far greater than the proportion dealing with task structures. In any case I hope to demonstrate in the next few paragraphs that the difference between task and data structures is not great.

The definition (declaration) of a data structure consists basically of four parts

- (a) the syntax of the data structure (how parts of the structure are named)
- (b) the storage layout of the data structure (how and where parts of the structure are allocated)
- (c) the addressing mechanism (how a part of the structure is obtained, given its name)
- (d) the operations on the data structure (the things which may be done to parts of the structure).

In practice these parts are combined in some way. The syntax and the storage layout are nearly always combined, often with the address mechanism. Although then the operations are not explicit in the declaration, they are defined by separate procedures or macros. Also (c) and (d) are sometimes combined (cf PL/I, COBOL, ALGOL68, BLISS, ESPOL).

Tasks as set up in most multiprogramming machines are little more than a specific sort of dynamic data structure and should be considered as such. They comprise a static part (code, instruction counter, task control information) and a dynamic part (working storage - stack or register space, environment control). They are subject to operations (initialisation, execution (both senses of the word), status enquiry) by the operating system, and even allocation and deallocation. XALGOL contains a good example of this. Thus systems programs really spend most of their time dealing with some form of data structure.

In fact there are extremely few objects of manipulation by programs that could not be included under the classification of arbitrary data structures. All this forces the conclusion that, in the past, particularly in systems programming languages, data structuring facilities have been seriously neglected.

Current systems languages contain predominantly four different methods of describing data structures, and these vary as to whether the structures may reference one another (and themselves), or are dynamic as regards storage allocation or linking, and so on.

COBOL and PL/I provide the best examples of the first method. The structure is represented as a two dimensional table, the "associated" ("concatenated") dimension running down the program listing and the "derived" ("nested") dimension being indicated by a level prefix. PL/I data structuring, which is by far most commonly used, is complete in that arbitrary data structures may be declared and used, and the set of permissible end nodes is more than sufficient for system programming needs [Cor 69]. However, the omission from the language of mode declarations has made the manipulation of multiple copies of an element unnecessarily awkward [Ber 72].

BLISS contains the most concise form of the second method. Data structures are created by allocating an amorphous piece of storage, defining an accessing algorithm, and "MAP"ing or imposing the algorithm onto the storage. The main advantage of this relatively low level technique is that the accessing algorithm can be tailored to suit the resource tradeoff requirements of the particular user's data structure at compile time, thus gaining efficiency. Consequently, hashing, linking, stacking and other accessing methods are associated directly with the data structure rather than being set up through procedures or subroutines. Also, since a piece of storage may have two different structures mapped onto it, weak typing is inherent.

Thirdly, ALGOL68 has perhaps the most comprehensive set of data structuring facilities. As in PL/I the structure is represented as a two dimensional table, the associated dimension being indicated by sequential separation with commas, the derived dimension by nested brackets. This tends to overwork brackets. These data structures may be self referencing but not self embedded and, as with PL/I data structuring, linked lists and other varying structures require an independent description of the method of access.

ESPOL contains the prime example of the fourth, or poor man's, method. The structures are defined by a series of macros which become more nested as one goes deeper into the derived dimension. This shares advantages with both the BLISS and PL/I method but relies heavily on the programmer keeping associated macros together, and not misusing them since compiletime checking is here limited to type checking. Thus this method is dangerously error-prone particularly in sections of program where weak typing is used extensively.

Hoare [Hoa 73] has described yet another possibility, recursive data structuring, in which the need for explicit references is eliminated. While this currently appears a little academic, the nature of this new method indicates that data structuring techniques are not at all complete or static.

The basic requirements of systems languages as far as data structuring is concerned are firstly to be able to refer without ambiguity to any part of any data structure that may be directly or indirectly manipulated, and secondly to have complete control over how parts of a data structure are stored and accessed. Currently available methods, such as those described above, each handle static data structures adequately, but with varying degrees of ease. However, only one (BLISS) appears to recognise the existence of dynamic data structures, mainly because of the low-level nature of its data structuring technique.

Thus, for the one element of data structuring which has greater importance in systems programming (dynamic structures), inadequate techniques are available. The systems programmer currently must implement most of the four parts described above through subroutines or macros.

§5.2.3 Weak Typing

Weak typing is the ability to treat an element of data as different modes at different times. This is not the same as unions - unions allow one to treat a name as referring to different modes at different times. That weak typing is necessary is made evident in §3.

There are at least seven different ways of implementing weak typing. These are outlined in Fig 5.2. They are often implemented in some combination, but will now be considered individually.

Redefinition is normally used in structure declaration and has one main advantage; the two names by which the data may be referenced are associated in the declaration. Equivalencing, a close relation, suffers badly from lack of this association (the equivalencing statement may be far from the declarations in the program) making the program much less readable. It also requires an extra statement. A very similar concept is name equating, and of course its counterpart address equating. These two methods are normally used for single cell data items, and if used for multicell data items, have the disadvantage compared with redefinition in that very little compile time checking can be done of the compatibility of data usage. Address equating is probably the worst method possible. Variable names referring to the same data cell are not related in any cross-referencable way, and since this method is normally used in assemblers to reference core locations outside the address range of the assembly, thus avoiding link-editing, programs using this may have the variable misplaced through relocation of the referenced module. Implicit coercions are also defective for two reasons. Firstly, they may include large amounts of code behind the programmer's back, and secondly the coercion may not have been intended at all but was rather a lexicographical error which might have been picked up by type checking. This has been remarked on as one of the great failings of PL/I as a systems programming language [Boo 74, Ber 72]. Explicit coercions, however, have one minor disadvantage and several advantages. They preserve the one-to-one mapping between identifiers and data items and so avoid some of the drawbacks of reference variables, and strong typing can be enforced at compile time thus reducing error-proneness. The minor disadvantage is that their use may become tedious (for example fixed-to-floating coercion) and under such circumstances it may be desirable to declare a coercion implicit. The last, but by no means least, common way of obtaining weak typing is the typeless variable. Typeless variables normally correspond to the storage elements of the machine, that is words, bytes, registers, and their use is accompanied by two rules. Firstly they implicitly coerce into and from ordinary typed variables, and secondly they provide a means of identifying when an overrride of a hardware function, such as memory protect, is desired.

It seems then that the choice should be between explicit coercions and typeless variables. Each has its own advantages, and therefore a combination of the two is the most advantageous. Implicit coercions may be required as well to avoid tiresome use of explicit coercions.

Note that typeless variables may also be utilised as a base mode for single cell mode declarations, and in some circumstances redefinition may be used as a way of avoiding reference variables. It is worth pointing out again that any method of using two names to refer to one data item increases susceptibility to the same sort of errors as accompany reference variables and unlimited gotos (cf above and below).

Fig 5.2 Weak Typing Methods

a) redefinition (as in COBOL)

```
01    BLOCK1.
   02   REALS OCCURS 20 TIMES PIC 999.
01    BLOCK2 REDEFINES BLOCK1.
   02   CHARS OCCURS 20 TIMES PIC XXX.
```

b) equivalencing (as in FORTRAN)

```
REAL A, B(20)
INTEGER I(5)
EQUIVALENCE (A,B(1)),(I(1),B(10))
```

c) name equating (as in ESPOL)

```
REAL A:
POINTER P=A;
```

d) address equating (as in most assemblers)

```
SAVE EQU   /123
```

e) coercion (implicit) (as in PL/I)

```
DCL A FLOAT, B FIXED;
B=A;
      /*hidden coercion code*/
```

f) coercion (explicit) (as in XALGOL)

```
REAL A;
BOOLEAN B;
B := BOOLEAN(A);
```

g) typeless variables (as in ESPOL, assemblers)

```
WORD W;
REAL A;
POINTER P;
W := P;
A := W;
```

§5.2.4 Fields

One requirement largely peculiar to systems programs is the ability to manipulate fields, or parts of addressable storage cells. The reasons for this are two-fold. Firstly, it is often necessary to conserve space by packing information at a greater density than can be achieved through standard storage addressing, and secondly systems programs, especially an operating system, must be able to recognise any field which the hardware recognises - for example, the exponent and mantissa fields of a floating point operand.

Orthogonality of design requires that the method of specifying fields be a natural extension of the data structuring definitions. This results in a conflict. There are basically two types of machines to be considered, the byte-oriented machine (e.g. IBM 360-370), and the word-oriented machine (e.g. Burroughs B6700). This orientation really refers to the smallest directly addressable storage cell. On the 360 the closest packing for numeric items is at the byte level, and the next step down is a single bit. There is no practical means of manipulating a three bit long value. This means that any numeric field recognisable by the hardware as such is at least a whole byte long, for example, the exponent field. Compare this with the B6700 where the smallest addressable unit is 48 bits (51 including the tag). Special hardware is provided to manipulate, as unsigned integers, fields of any length from one to forty eight bits, at any position within the word. Therein lies the problem . To describe generalised fields in terms of bits and bytes, and conversely bits and bytes in terms of generalised fields, does not adequately reflect the true hardware circumstances. For the language to be portable, both schemes must be included in the syntax, and only one implemented. This is unpleasant but necessary.

Once the fields possess names, manipulating them becomes relatively easy, as each size of field can be treated by the compiler as a basic mode, embedded within another mode. For example, a floating point operand could be expanded (or redefined) as four or six eight-bit bytes. (cf §5.2.2 and §5.2.3). This ability to treat non-terminal nodes of a data structure as possessing values independently of the terminal nodes does not exist in all languages.

§5.2.5 Strings

Strings and string manipulation are a weak point of most system languages. There is little evidence of rationalisation of the syntax and semantics of string processing (only recently has pattern matching taken on a more mathematical basis). This is in spite of the fact that hardware operations for dealing with alphanumeric data differ little in function between machines. That is, most machines which have hardware for this type of data have similar operations differing only in complexity and side effects.

String manipulation in systems programs can be split into two classes. In the operating system, strings are created, moved around, and translated. Thus the first class contains only operations where the string content is not significant. The second class comprises scanning and comparisons such as is found in the parsing of strings. The first class requires only the implied use of pointers. Syntax for concatenation, replacement, and translation can be adequately constructed (as in XPL, PL/I) to hide the use of pointers from the programmer. However, with the second class of manipulations, this is much more difficult. Consider for instance the simple problem of isolating the first non-blank substring in a string. This might be written

```
                TEMP = SUBSTR (SOURCE, VERIFY (SOURCE,' '));
in PL/I         DEST = SUBSTR (TEMP, 1, INDEX (TEMP, ' ')-1);


                SCAN SOURCE : SOURCE WHILE EQL " ";
in XALGOL       REPLACE DEST BY SOURCE UNTIL EQL " ";


in ALGOLW       i := 0; while source (i|1)=" " do i := i + 1;
                j := i; while source(i|1)¬=" " do
                        begin dest(i-j|1):=source(i 1);i:=i+1; end;

while
in SNOBOL       SOURCE SPAN(' ') BREAK(' ') . DEST
```

Then consider the even simpler problem of <u>locating</u> (but not isolating) the first non-blank substring (for comparison purposes). It is paradoxically more difficult in the above languages.

The difficulties involved in creating usable syntax are three-fold. Firstly, string operations often alter more than one operand, which is not easy to express in an assignment-oriented language. Secondly, the operands may not be assigned to, or used purely as a source, but rather are altered in nature (compare A←A+3), meaning add 3 to A). And thirdly, if an attempt is made to eliminate explicit use of pointers (as in PL/I) through substrings these substrings have to be identified by a pointer and a length, rather than their composition.

It is apparent that at the state of the art, clear and concise syntax for parsing strings is not available. Considering the large amount of string manipulation performed in any compiler, this area has been severely neglected, at least as far as systems programming languages are concerned. Until an adequate high level syntax for string manipulation becomes available, string pointers must remain as the only completely workable scheme.

§5.2.6 Storage Allocation

The applications programmer is concerned only with the surface (syntactic) appearance of his data, and where or how it is stored is immaterial. For the systems programmer, concerned with efficiency of usage, explicit control of storage allocation is in some cases essential.

Unfortunately for the designer of systems languages, the types and relative efficiencies of the various storage classes vary from machine to machine. On a 360 or 370 the most efficient classes are static and explicit dynamic, although implicit dynamic is perhaps the most convenient to use. On a B6700, static does not exist, and must rather be considered as outer block dynamic storage. Implicit dynamic, on the other hand, is allocated in a hardware stack, and thus is very efficient. Both machines provide the explicit dynamic class of storage, at approximately the same efficiency. The B6700 also has provision for read only data.

Thus the storage classes which must be provided are machine dependent. On a machine operating with virtual memory, some mechanism must be provided whereby heap storage segments can be marked as overlayable or nonoverlayable. (Provision must be made in the language for the use of defaults which may be overridden at the programmer's request).

§5.2.7 Array Processing

There are two excellent reasons why array processing should feature in a system language. Firstly, a number of machines now have some form of vector processing capabilities (e.g. B6700, CDC STAR, MU5), and such hardware should for the sake of efficiency, be utilised if it is available. Secondly, syntax for array processing significantly abbreviates an algorithm which might otherwise have to use clumsy looping structures. This holds regardless of whether the operation can be performed in special hardware.

Several languages have array processing embedded (ALGOL68, PL/I, XALGOL), and this practice should be extended. Providing that such operations may be defined at the lowest level, there is no reason why the syntax should be machine dependent; the natural extensions to single value operations appear adequate.

§5.3 Control-Related Criteria

Control in a systems programming language differs from that in a general purpose language in two areas, process control and exception condition processing. These are discussed in §5.3.1 and §5.3.2 respectively. The four basic types of control remaining

a) parallel elaboration
b) conditional or selective elaboration
c) repetitive elaboration
d) subroutine elaboration

are here discussed only briefly as the arguments differ little from those concerned with general purpose languages.

Parallel elaboration syntax is unjustifiable in the context of present day hardware. Although some special purpose machines exist (e.g. ILLIAC IV), a large multiprocessor machine spends a very small amount of time engaged on parallel processing within a single task. Such a machine when running normally would have each processor performing a different task. The only times that multi-processors would be coordinated enough to parallel-process on a statement basis would be during operating system initialisation and such fault conditions as power failure. However even in these circumstances processing would be more feasible on a multi-tasking basis.

The three remaining types of control comprise forms of implicit gotos. Enough has been written about gotos [Dij 68, Knu 71, Wul 71] for the topic to be avoided here, but I will state and justify my viewpoint in the systems language context.

Unconditional gotos should be retained with their range limited to within any controlled environment. This means that no goto should result in an implicit procedure call (a bad goto). The unconditional goto is a highly efficient construct. Elimination of the goto completely in any language which is not rich enough in control constructs with embedded gotos results in a serious loss in efficiency through code duplication, artificial variables, or proceduring. There are relatively few languages which even approach the variety of control constructs required. A maximum amount of effort therefore should be put into making a language rich with a wide variety of constructs containing embedded goto's. Some in common use are illustrated in Fig 5.3.

Fig 5.3

Form	Common Syntax	Embedded Gotos		
		Unconditional	Conditional	Multiway
conditional	if .... then	-	1	-
biconditional	if .... then .... else	1	1	-
numeric selection	case .... of (....,....,....)	n-1	-	1
multi conditional	select .... in (c_1:....,c_2:....,c_3:....)	n-1	n	-
loop tested at top	while .... do	1	1	-
loop tested at bottom	do .... until	-	1	-
loop tested elsewhere	repeat (.... exit ....)	1	1	-
subroutine	call doit (....)	2	-	-

All commonly used constructs for conditional or selective elaboration should be available. Where the number of choices exceeds two, the condition for each choice of elaboration should accompany the choice more or less as a label. For example

case n of (0:....., 1:2:.....,) else .....

Note that a choice may be accompanied by more than one condition, thus avoiding a goto to get between choices, as in

case n of (0:....., 1:....., 2:goto 1, ...) else .....

It is particularly important to have an exception choice. In particular non-numeric conditions should be as expansive as possible so that any Boolean expression may be used as a label. There is one additional form of the conditional whose use is almost peculiar to systems programs. Commonly, evaluation of Boolean expressions proceeds to completion regardless of whether this complete evaluation is necessary. For example, only one of the operands in a conjunctive needs to be false to ensure the value of the conjunctive. Thus what might more naturally have been written

if b_1 andif b_2 andif b_3 then . . . . .

would have to be written

if b_1 then if b_2 then if b_3 then . . .

This is particularly important if the operands have side effects on elaboration. For example if n < upperbound (A) and A[n] = 0 then ...

Constructs for repetitive execution provide probably the most difficult place for eliminating the goto, because loops may be exited on more than one condition, and then to different places. For this reason the exit as in BLISS gained some acceptance, but still falls short of a complete solution. Zahn [Zah 73] describes an effective but clumsy construct for such loops. In general, constructs in which the key words are naturally distributed over several lines seem awkward, and therefore probably appear difficult to understand (to the language designer). Nevertheless, some construct similar to that of Zahn is necessary if a serious attempt is to be made in elimination of the goto.

Subroutines, like blocks, are one of the best tools a programmer has for creating an understandable and well structured program. As systems programs tend to be large by nature, subroutines should be used extensively, and therefore extra attention should be paid to the convenience of their use. In particular the parameter and returned value mechanism should cater for every defined mode. There has to be a choice between ALGOL68 parameter passing, and ALGOL60 or ALGOLW parameter passing. That is, either the formal parameters are completely specified modes (and ref to modes, and proc modes) or some formal cases are indicated (such as value, name etc.). If the language would include references as being strongly typed (that is ref to mode, or ref to ref to mode) then the former is obviously desirable. On the other hand, a language like ALGOL60 or ALGOLW does not restrict programmers to a great extent and, although it is not the neatest solution to the problem, should not be condemned on grounds of impracticality.

One of the better justifications for unions lies in parameter passing. Often in systems programming one comes across a procedure of which the type and number of parameters,in reality, varies as a function of one other parameter. Some existing languages permit actual parameters to be omitted in the call (AED,LSD), and others use untyped variables (ESPOL). Weakly typed reference variables afford another solution. All of these rely on runtime checking, and so the use of these mechanisms should perhaps be avoided as far as possible. For the remaining unavoidable cases, I favour the solution which introduces the least additional features into the language being designed.

§5.3.1 Process Control

Process or task control is the most prominent of the functions peculiar to systems programming. Through it multiprogramming systems make efficient use of I/O devices and other system resources. Adequate provision for all aspects of this type of control is essential. Discussion here is centred around the three aspects of task control that are most apparent in a systems language - dependency, communication, and mutual resource control.

Spawning a task from an initiating process differs from calling a procedure or subroutine in that it is used when the execution of the initiating or parent task is desired to be overlapped with the initiated or son task. In order to achieve this, the son task must maintain a degree of independence or life of its own. The degree of independence may vary, however, since a son task may require access to information found in the parent. Such tasks are called dependent, and for example user jobs running under an operating system, or tasks run from a batch controller are of this type. Other tasks are called independent, and might include such things as the compiled program being initiated by a compiler.

In some circumstances, dependent tasks may wish to obtain mutual exclusion with respect to a resource not by avoiding simultaneous execution but by passing control from one to the other by explicit program command. Such tasks are called coroutines. Each of these three types of task should be provided, although coroutines might be omitted if some other method of mutual exclusion was always preferable. This is simply a matter of providing three different initiating statements and a statement for explicit passage of control, such as those in XALGOL. It is also necessary to have a means of indicating that the son-task's code is independent of the initiating task, since an independent task cannot use code of the initiating task but a dependent task often may.

As a son task is nested deeper than the initiating task, most of its housekeeping properties (such as status, cpu time used etc) would not normally be available to the initiator. Access to these by the initiator is essential. This can be accomplished in two ways: either the operating system can be used to obtain the information (through its knowledge of all tasks); or such information can be put in a commonly accessible pool, for which the normal rules of scope do not apply. I favour the information pool technique for two reasons. Firstly, it may appear in each task as a normal data structure, and accessed as such, although some information may be read-only when the task is executing. Secondly, some of this task information needs to be available to the parent before and after execution (such as use of resources, limits on resources). This task information mode would of course depend on any given implementation, and thus might be included in a prelude.

While most communication between tasks could go through the information mode, processing of an independent nature also requires mutual exclusion with respect to resources, and synchronisation of execution. The way in which these are obtained depends largely on the hardware available, but suitable high level primitives can easily be designed for both procuring and releasing locks, and waiting on or causing events (cf XALGOL, PL/I).

§5.3.2 Exception Condition Processing

Some of the major omissions from current systems languages are facilities for recognizing and dealing with hardware or software conditions which are unexpected. Systems software can ill afford the luxury of not attempting to recover from software or hardware failures. The basic requirements are twofold. Firstly, a mechanism must be provided whereby control enters a given section of program after such an exception condition occurs. The PL/I and XALGOL on-statements perform this function. Secondly, since it is not desirable or even possible to return from some exception conditions (particularly those detected by hardware) exit from this fault code must be made by the equivalent of a goto. Quite often this goto is a bad goto, that is, one passing through declaration of dynamically allocated variables. If gotos are to be eliminated some acceptable mechanism must be found to replace it, such as treating these conditions as a coroutine call.

This problem is evidently a special case of a much larger one, namely multiple exits from controlled environments. See §5.3 on loop exits for discussion of another special case. However, for the purposes of <u>error</u> handling the mechanisms can be enormously inefficient, as one would not normally expect them to be used frequently. In reality, implementations of software interrupts tend to be just that.

§5.4 Miscellaneous Criteria

Two remaining topics which affect the language itself (provision for inline code, and separate compilation) are discussed in §5.4.1 and §5.4.2.

The remaining criteria, mentioned here for completeness, are the managerial (or guideline) criteria. The reasons for these, and the effects of them were discussed fully in §4. They are, in decreasing order of importance

Functionality
Error Immunity
Code Efficiency.

While these criteria reflect little on the language structure, they nevertheless form the most important overall group to be considered, effectiveness. A language must be effective in its purpose.

§5.4.1 Inline Code

For reasons given elsewhere (§4) I have considered the provision of inline assembler language or machine language coding. The main argument for this facility is efficiency, and the main argument against is error susceptibility. Therefore an attempt must be made to satisfy both as far as possible!

There appear to be basically three ways of introducing the facility - code statements, pseudo procedure calls, and code-bodied procedures. Code statements are undesirable because of their error susceptibility. Pseudo procedure calls are of two types: one has each instruction field as a separate parameter (as in Elliott 503 Algol) allowing inline use of any individual instruction; and the other has a separate procedure for each useful instruction (as in ESPOL). Both form a very long-hand way of obtaining a single machine instruction, and while the former is singularly error prone, the latter restricts the machine instructions available. Code-bodied procedures (possibly inline) allow reasonable error checking, and when used in a language with a simple macro processor, can be surprisingly effective. All operands can be type checked at compile time, and any use of absolute addresses could be flagged. In effect the procedure provides a closed environment for the code, and increases the ability of the compiler to detect errors.

As an example consider the commonly used 'disable external interrupts' instruction on the B6700:

code proc disallow : DEXI ; inline

If an extensibility mechanism is available, operations on new modes can be defined in terms of a code-bodied procedure, thus providing the most efficient means of performing that operation. Note that these procedures may be inline or not, depending on whether the programmer considers inclusion of the code desirable at each invocation.

§5.4.2 Separate Compilation

Large programs can often be fixed or modified by changing only a small part of the whole program. Thus it is extremely convenient to separately compile procedures (or even blocks) of a program and then link edit or bind them into the whole program. This mechanism may also be used to aid in the production of a large program by a team. While this facility does not reflect greatly on the language design, it does have a very pronounced effect on the compiler, and is worthy of separate mention. (Further discussion on this and related topics exists in §4.)

Towards this end, procedures and blocks need as far as possible to be logically isolated, and code physically isolated into separate segments. This may well reflect on the overall system as much as on the language design.

§5.5 Conclusion

The present trend of both computer companies and computer users towards packaged systems - hardware and software combined - has made necessary a reappraisal by computer companies of the cheapest way to produce reliable system software. It is apparent that the cost/performance ratio of a large systems software project depends more on issues which some would call managerial (as against technical) [Wul 72], and although these issues by and large are global to a language design and its implementation, the design and in particular the implementation influence and are influenced by managerial issues. This happens to an extent which I believe is not fully understood.

This thesis set out to investigate design criteria for systems languages, and to a lesser extent systems language implementation. There are three aspects from which a language and its implementation can be viewed. They are: Effectiveness, purely linguistic criteria, and extensibility/orthogonality. Ignoring the fact that these aspects overlap, they can be placed into descending order of importance as follows

(1) Effectiveness
(2) Linguistic
(3) Extensibility/Orthogonality

The overlap is shown in diagramatic form in figure 5.4. Note that although the ordering is implicit in the way the criteria are written down, this ordering is a general one applying to systems languages intended for all types of systems software projects. If the language were intended to deal only with one aspect of systems programming, the order might change slightly. For example a language intended for writing an operating system only would probably require greater consideration of code efficiency and process control, and less of control structures and string manipulation facilities. In general a criterion will be positioned higher in the diagram if it is required for all aspects of systems programming, not just for one. It is desirable to satisfy all the criteria indicated to a level depending roughly on their position in the diagram. It is certainly not suggested that some be considered to the exclusion of others. Thus the best results are achieved by striking a balance between opposing criteria.

Fig 5.4

EFFECTIVENESS	LINGUISTIC	EXTENSIBILITY
Functionality (read and writeability)		
	Data Structuring	
Error Immunity	Sub-Cell Layouts	Modes
Code Efficiency	Complete Control Structures	Macro Facility
	String-Manipulation Facilities	
Modularity of Compilation and Execution	High-Level Unformatted I/O	
	Weak Typing	Orthogonality
	Dynamic Storage Allocation	
	Process Control	
Runtime Debugging Aids	Array Processing	
		Operator Declaration
		Unions

Some criteria appear to be duplicated. This is because a specific function to which a specific criterion is addressed may be encompassed by a more general concept whose wholesale implementation is perhaps not so necessary: for example, data structuring and modes.

There are five ways to approach a language design. They are:

(1) a new and machine dependent language } non portable
(2) an old language with machine dependent extensions }
(3) an extensible base language (with minimum implementation) }
(4) new machine independent language } portable
(5) old machine independent language }

Not all these, however, may permit sufficient criteria to be fulfilled to the desired level. I think it is generally accepted that languages in categories 4 and 5 are unsuitable for systems programming (for reasons of efficiency). The machine independent structures which could describe some machine functions, notably direct I/O functions, and interrupt handling operations, are so 'high-level' that their inefficiencies become completely unacceptable under some circumstances. Either that or so 'basic' that the use of large groups of these constructs becomes tedious and time consuming, and the overall function is obscured. This is because there is no list of 'intermediate level' primitives that can adequately describe each of the varieties of ways of controlling the same overall functions on different machines. For example, an I/O operation on a PDP11 is done by moving data to a reserved apparent core address, while on a B6700, the same function is performed through a SCAN OUT (a channel operation).

The three remaining approaches are all used to a greater or lesser extent (see §2) and are justified in various ways. The most widely used type is of course 1, and is most often justified by the catch phrase ". . . . . other languages were examined and found unsuitable . . . ." or similar [Wul 71a, Cla 71], which is to my mind the greatest condemnation of that variety of language. The second type has gained a lot of acceptance recently, and in spite of its shortcomings [Bro 74], it is certainly a useful approach. Two very real advantages are that a completely new language does not have to be designed or learnt, and that an algorithm may be passed between machines with possibly only minor recoding. The third type has advantages also. Not only may programs be passed between machines with only minor recoding, but the language may be able to be implemented as originally defined on several machines.

My conclusion, then, is that systems programming languages should aim to be generally more effective (in the sense of §4), and to that end should provide better data structuring facilities and control constructs. Paying greater attention to these points should decrease the cost and improve the standard of systems software.

The most careful attention must be paid to separating those features which have only a minor effect on the effectiveness of the language (such as choice of implementation approach), from those which have a major effect (such as elimination of the goto). Performing this separation is not easy. In the past too great an emphasis has been placed on those things affecting the surface appearance of the language, and too little on the basic underlying requirements.

§6 Appendices

§6.1 Language Summaries

§6.1.1 Structure

A brief resumé is provided for all languages surveyed. Where no adequate description of the language was available only the design factors are given. Note that as these summaries are often derived from informal descriptions of the languages, they are liable to be incomplete (and possibly inaccurate).

Each language is summarised under six headings: design, data, operators, control, I/O, machine dependency, and extensibility. Under each heading there is a list of major items (in capitals and abbreviated to the underlined sections, as below), and each of these may further be modified by bracketed information. Since many languages have common components BASEPLUS is used to indicate that, for the particular heading, only differences between the language and the base language are itemised. The following is more or less a combination of all possibilities.

Design:
- AUTHOR (<name>)
- BASE LANGUAGE (<name>)
- YEAR OF APPEARANCE (<name>)
- PURPOSE (general/systems/compiler/applications/special...)
- CRITERIA (<stated criteria>)

Data:
- BASEPLUS
- INTEGER REAL BOOLEAN ALPHA
- ARRAY (single dim/multi dim, based <n>, int, real, bool, any ...)
- STRING (bounded/varying)
- FIELDS (bit/<n> bit bytes/variable)
- PRECISION (digits/multiple cell)
- STRUCTURES (general/restricted, self referencing)
- REFERENCE (string, array, general, arbitrary)
- DECLARED (required/optional, static/dynamic, local/global, initialisable, equateable)
- OTHER (......)

Operators:	BASEPLUS
	ARITHMETIC (+ - x / ÷ ↑ mod .....)
	LOGICAL (∧ ∨ ¬ ⊃ ≡ .....)
	RELATIONAL (> < ≥ ≤ = ≠ .....)
	REFERENCING (ref to, deref, address of, value of)
	STRING (relational, size, substring, translate, move, table occurrence)
	ASSIGNMENT (arith, boolean, reference, any, array slices)
	PRIORITY (std/left-right/right-left)
	COERCION (implicit ..., explicit ...)
	OTHER (.....)
Control:	BASEPLUS
	CLOSED CLAUSES (value, declarations, named, ...)
	GOTO (conditional, numeric)
	SELECTION (conditional,biconditional, numeric, logical)
	PROCEDURES (value, parameters ref/value/name, recursive, multiple entry)
	INTERRUPT TRAPS (software, hardware)
	PROCESS CONTROL (events, locks)
	LOOPS (tested at top-bottom-middle, exitclause, stepped, nextclause)
	OTHER (.....)
I/O:	STREAM RECORD
Machine Dependency:	CODE (inline, pseudo procs, proc bodies)
	SEGMENTATION (code, data)
	OTHER (.....)
Extensibility:	BASEPLUS OPERATORS DATA CONTROL MACRO

AED-0

Design:	AUTHOR (MIT, Ross?) BASE (ALGOL60) YEAR (early sixties) PUR (general) CRIT (intended for machine independent programming on register machines)
Data:	BASEPLUS ALPHA? ARRAY (single dim, based 0, fixed range) FIELD (bit) STRUCT (restricted) REF (arbitrary) DECL (required, dynamic, global, initialisable) OTHER (stack, lists)
Operators:	BASEPLUS STRING (?) REF (address of, value of) ASSIGN (reference, embedded) OTHER (infix stack ops)
Control:	BASEPLUS CLAUSES (value, named) PROC (parameters ref, recursive only if declared)
I/O:	RECORD (through package) STREAM (through package)
Dependency:	None
Extensibility:	MACRO (?)

AL

Design:	AUTHOR (Haines) YEAR (1971) PUR (systems) CRIT ("...all the capabilities of the basic assembler language of the System/360 yet offers a dramatic improvement in intelligibility", "The language is implemented as a preprocessor to BAL")
Data:	DECL (registers)
Operators:	LOG ($\wedge$,$\vee$,$\neg$)
Control:	SEL (conditional, biconditional) PROC (value, parameters ref) LOOPS (tested at top, tested at bottom, stepped) GOTO (conditional)
I/O:	None
Dependency:	Extreme
Extensibility:	None

ALGOL 60

Design:	AUTHOR (committee ed Naur, P) BASE (ALGOL58) YEAR (1960) PUR (scientific) CRIT ("a language suitable for expressing a large class of numerical processes in a form sufficiently concise for direct automatic translation into the language of programmed automatic computers")
Data:	INT REAL BOOL ARRAY (multi dim, int-real-bool) STRING (parameter only) DECL (required, dynamic, local)
Operators:	ARITH (+, -, x, /, ÷) LOG (∧, ∨, ¬, ⊃, ≡) REL (<, ≤, >, ≥, =, ≠) PRI (std) ASSIGN (single value only) COER (implicit int-real)
Control:	CLAUSES (declarations) GOTO (conditonal, numeric) SEL (conditional, biconditional) PROC (value, parameters value-name, recursive) LOOPS (tested at top, stepped)
I/O:	Undefined, but STREAM indicated
Dependency:	None
Extensibility:	None

ALGOL 68

Design:	AUTHOR (committee ed Van Wijngaarden) YEAR (1968) BASE (ALGOL60?) PUR (general) CRIT (machine independent general purpose language)
Data:	INT REAL BOOL ALPHA ARRAY (multidim, anytype, flexible bounds) STRING (bounded (array of alpha)) FIELDS (bit arrays, 8 bit bytes) PREC (multiple cell) STRUCT (general) REF (general) DECL (required, dynamic or static, local initialisable, equateable)
Operators:	ARITH (+, -, x, /, ÷, ↑ etc) LOG (∧, ∨, ¬, ⊃, ≡ etc) REL (>, <, ≥, ≤, =, ≠ etc) STRING (relational, size) ASSIGN (all modes) PRI (std, declarable) COERC (implicit)
Control:	CLAUSES (value, declarations) GOTO SEL (conditional, biconditional, numeric) PROC (value, parameters, recursive) LOOPS (tested at top-end, stepped) OTHER (parallel elaboration)
I/O:	STREAM RECORD
Dependency:	None
Extensibility:	OPER,DATA

ALGOL W

Design:	AUTHOR (Wirth) BASE (ALGOL60) YEAR ( ) PUR (general) CRIT (suitable for teaching)
Data:	BASEPLUS STRING (bounded) STRUCT (restricted) REF (ARBITRARY)
Operators:	BASEPLUS STRING (single character relational, move)
Control:	BASEPLUS SEL (numeric) PROC (parameters value-name-result) LOOPS (tested at top-bottom, stepped)
I/O:	STREAM
Dependency:	None
Extensibility:	DATA (restricted data structures)

ALPHA

Design:	AUTHOR (Yenshov A.P.?) BASE (ALGOL60) YEAR (1964) PUR ("scientific")
Data:	BASEPLUS STRUCT (restricted?) OTHER (complex)
Operators:	BASEPLUS ASSIGN (multivalue)
Control:	PROC (not recursive)
I/O:	STREAM?
Dependency:	None
Extensibility:	DATA (restricted data structures)

AMBIT/L

Design:	AUTHOR (Christensen C) YEAR (1970) PUR (symbol manipulation)
Data:	STRING (varying) OTHER (various substring forms) DECL (required)
Operators:	LOG ($\wedge$, $\vee$, $\neg$)
Control:	CLAUSES (delcarations) GOTO SEL (conditional, biconditional)
I/O:	None
Dependency:	None
Extensibility:	None

APAREL

Design:	AUTHOR (Balzer RM) BASE (PL/I) YEAR (1968) PUR (Compiler construction)
Data:	BASEPLUS
Operators:	BASEPLUS OTHER (specialised string parsing operators)
Control:	BASEPLUS OTHER (non-sequentially processed groups of parsing statements with a special syntax)
I/O:	BASEPLUS
Dependency:	None
Extensibility:	BASEPLUS

APL

Design:	AUTHOR (Iverson) YEAR (1961) PUR (machine description language)
Data:	INT REAL ALPHA ARRAY (two dimensional, based 0, any) DECL (optional, static, global/local) OTHER (variables are typeless)
Operators:	ARITH (+, -, X, /, ÷, ↑, mod, many more) LOG (∧, ∨, ¬, ⊃, ≡, many more) RELATIONAL (<, >, ≤, ≥, =, ≠) ASSIGN (any, array slices) PRI (right-left) COER (implicit int-real) OTHER (a consierable number)
Control:	GOTO (conditional, numeric) PROC (value, parameters ref, recursive)
I/O:	STREAM
Dependency:	None
Extensibility:	None

B

Design:	AUTHOR (Johnson and Kernighan?) YEAR (1972) BASED (BCPL) PUR (systems programming) (Implemented on HIS 6070)

LANGUAGE REFERENCE MATERIAL NOT AVAILABLE

BCPL

Design:	AUTHOR (Richards) BASE (CPL) YEAR (1968) PUR (compiler writing) CRIT (linquistic elegance)
Data:	ARRAY (single dim, base 0) DECL (required, static, local, equateable) OTHER (cell data only)
Operators:	ARITH (+, -, x, /) REL (>, <, ≥, ≤, =, ≠) LOG (∧, ∨, ≡, ≢) REF (address of, deref) ASSIGN PRI (std) OTHER (shift left, shift right)
Control:	CLAUSES (declarations) GOTO (conditional, numeric) SEL (conditional, biconditional, numeric) PROC (value, parameters untyped) LOOPS (tested at top-bottom, stepped, exit clause)
I/O:	None
Dependency:	None
Extensibility:	DATA (through data structuring mechanism)

BLISS

Design:	AUTHOR (Wulf et al) YEAR (1970) PUR (systems) CRIT ("so as to be especially suitable for use in writing production software systems for a specific machine")
Data:	ARRAY (single dim, based 0) DECL (required, local/global, static/dynamic) OTHER (only cells and registers declared)
Operators:	ARITH (+, -, x, /, ÷, abs, ↑) REL (<, >, ≤, ≥, =, ≠) LOG (∧, ∨) REF (deref) ASSIGN PRI (std) OTHER (shift left, shift right)
Control:	CLAUSES (value) SELECTION (conditional, biconditional, numeric, logical) PROC (value, parameters ref, recursive) PROCESS (coroutines) LOOPS (tested at top-bottom, exit clause, stepped) OTHER (expression language)
I/O:	None
Dependency:	CODE (inline) OTHER (pointer mechanism)
Extensibility:	MACRO DATA (general data structuring)

CIMPL

Design:	AUTHOR (MIT) BASE (PL/I) YEAR (1970) PUR (teaching systems programming)

LANGUAGE REFERENCE MATERIAL NOT AVAILABLE

CLIP

Design:	AUTHOR (Book et al) BASE (ALGOL58) YEAR (1960) PUR (information processing)
Data:	BASEPLUS STRING (fixed length) DECL (initialisable, equateable) STRUCT (restricted to static forms)
Operators:	BASEPLUS STRING (substring, relational)
Control:	BASEPLUS
I/O:	STREAM?
Dependency:	None
Extensibility:	None

COBOL

Design:	AUTHOR (Committee) YEAR (1959) PUR (business data processing)
Data:	INT REAL ARRAY (single dim, based 1, any) STRING (bounded) PREC (digits) STRUCT (restricted) DECL (required, static global, initialisable, equateable)
Operators:	ARITH (+, -, *, /, rem) REL (>, <, =) LOG (∧, ∨, ¬) STRING (relational, move, translate) ASSIGN (any, substructures) PRI (std) COER (explicit)
Control:	CLAUSES (named) GOTO (numeric) SEL (conditional, biconditional) LOOPS (tested at top, stepped)
I/O:	RECORD
Dependency:	None
Extensibility:	None

COGENT

Design:	AUTHOR (Reynolds) YEAR (1965) PUR (symbol manipulation) CRIT ("primarily for use as a compiler-compiler")
Data:	DECL (optional, initialisable) OTHER (Cogent operates directly on tree structures through productions. Variables may be used to hold production trees, procedures simple data)
Operators:	OTHER (Cogent has a parse operation similar to SNOBOL but operatong on tree structures)
Control:	SEL (conditional PROC (value, parameters ref?) OTHER (statement fixture)
I/O:	STREAM
Dependency:	None
Extensibility:	MACRO (simple)

CORAL 66

Design:	AUTHOR (?) BASE (ALGOL60) YEAR (1966) PUR (real time systems)

LANGUAGE REFERENCE MATERIAL NOT AVAILABLE

However, the language has greater data-structuring ability, and bit manipulation facilities as well as permitting inline machine code.

DCALGOL

Design:	AUTHOR (Burroughs Corp) BASE (XALGOL) YEAR (1970) PUR (systems programming CRIT (specifically designed for writing parts of the B6700 operating system, especially data communications)
Data:	BASEPLUS OTHER (messages, queues)
Operators:	BASEPLUS OTHER (functions provided for many operations on queues and messages)
Control:	BASEPLUS OTHER (wait on queue)
I/O:	BASEPLUS
Dependency:	All extensions to XALGOL (which is machine dependent) are machine dependent.
Extensibility:	None

EPL

Design:	AUTHOR (Bell Labs & MIT) BASE (PL/I) YEAR (1966) PUR (systems programming)
	LANGUAGE REFERENCE MATERIAL NOT AVAILABLE
	However, the language is basically a subset PL/I and does not have the following PL/I functions: I/O, PICTURE, complex arithmetic, controlled storage, ON statements, recursive procedures, based storage, pointers, and compiletime facilities.

ESPOL

Design:	AUTHOR (Burroughs Corp) BASED (XALGOL) YEAR (1966) PUR (writing operating systems)
Data:	BASEPLUS ARRAY (single dim, based 0) FIELDS (variable) STRUCT (one level only) DECL (address equateable, initialisable) OTHER (registers, typeless variables, controlled storage allocation)
Operators:	BASEPLUS
Control:	BASEPLUS OTHER (exit statement)
I/O:	BASEPLUS
Dependency:	extreme CODE (pseudo-procedures)
Extensibility:	MACRO (parametered) DATA (specification of bit layout of typeless variables)

FORTRAN IV

Design:	AUTHOR (IBM Corp) YEAR (1964) PUR (general)
Data:	INT REAL BOOL ALPHA ARRAY (two dim, based 1, any) DECL (optional, static, local, initialisable, equateable)
Operators:	ARITH (+, -, x, /, $\uparrow$) LOG ($\wedge$, $\vee$, $\neg$, $\equiv$) ASSIGN (arith, logical)
Control:	GOTO (numeric) SEL (conditional) PROC (value, parameters ref, multiple entry) LOOPS (tested at top, stepped)
I/O:	RECORD (formatted)
Dependency:	None
Extensibility:	None

FSL

Design:	AUTHOR (Feldman) YEAR (1964) PUR (writing compilers)
Data:	ALPHA OTHER (symbol tables, stacks, typeless variables)
Operators:	ARITH (+,-) REL (<, >, =, ≠) LOG (∧, ∨, ¬) ASSIGN OTHER (stack ops, symbol table searches)
Control:	GOTO SEL (biconditional) OTHER (parse requests are part of the language, each statement has a named successor)
I/O:	None
Dependency:	FSL is part of a system implemented on some particular machine. The language becomes tied to the machine.
Extensibility:	None

GARGOYLE

Design:	AUTHOR (Garwick) YEAR (1963) PUR (writing compilers)
Data:	ARRAY (single dim, based 1) DECL (required, local/global, static) OTHER (typeless variables, Gargoyle operates on a string of tokens)
Operators:	ARITH (+, -, x, /) REL (>,<, =, ≠) LOG (∧, ∨, ¬) OTHER (shift, mask, and array search operations)
Control:	GOTO SEL (conditonal, biconditional) PROC () OTHER (each statement has a named successor)
I/O:	STREAM (tokens only)
Dependency:	Moderate
Extensibility:	None

GOGOL III

Design:	AUTHOR (McKeeman, Sauter) BASE (ALGOL60) YEAR (1967) PUR (writing operating systems)

LANGUAGE REFERENCE MATERIAL NOT AVAILABLE

However, basically the language consists of the integer parts of ALGOL60, inline code capability, simple string handling, and facilities for process control. Arrays are single dimensioned and storage is static only.

GPL

Design:	AUTHOR (Garwick) BASE (ALGOL60) YEAR (1968) PUR (general) CRIT (" a truly general purpose language")
Data:	BASEPLUS ARRAYS (single dim) FIELD (variable) STRUCT (general) REF (any)
Operators:	BASEPLUS REF (value of)
Control:	BASEPLUS OTHER (extended procedure forms)
I/O:	?
Dependency:	None
Extensibility:	DATA (structured modes) OPER MACRO CONTROL

IMP (Irons)

Design:	AUTHOR (Irons) BASE (ALGOL60) YEAR (1965) PUR (systems) CRIT ("a real word tool, particularly useful for systems programming")
Data:	BASEPLUS STRUCT (restricted) REF (abritrary) OTHER (list)
Operators:	BASEPLUS REF (address of, deref) PRI (right-left) OTHER (infix list operations)
Control:	BASEPLUS CLAUSES (without declarations)
I/O:	?
Dependency:	None
Extensibility:	DATA (structures modes) OPER MACRO CONTROL

IMP (EDIN)

Design:	AUTHOR (Stephens?) BASE (ATLAS AUTOCODE) YEAR (1966) PUR (systems programming)
Data:	INT REAL ARRAY (multidim, real int) STRING (bounded) PREC (multicell) STRUCT (general) REF (arbitrary) DECL (required, dynamic, local, equateable)
Operators:	ARITH (+, −, ×, /, ÷, ↑) REL (>, <, ≥, ≤, =, ≠) LOG (∧, ∨, XOR, ¬) STRING (concat, simple pattern matching) ASSIGN (whole structures only) PRI (std) COER (implicit integer-real) OTHER (left shift, right shift)
Control:	CLAUSES (declarations) GOTO (numeric) SEL (conditional, biconditional) PROC (value, parameters recursive) INT (hardware) LOOPS (tested at top, stepped)
I/O:	STREAM
Dependency:	Low
Extensibility:	DATA

JOSSLE

Design:	AUTHOR (White & Presser) YEAR (1973) PUR (the post-syntactic phase of compiler construction)
Data:	INT ARRAY (single dim, based 1) STRINGS (bounded) FIELDS (variable) REF (any mode) STRUCT (one level) DECL (required, local, dynamic?) OTHER (descriptors?)
Operators:	ARITH (?) LOG (?) REL (?) STRING (?) REF (deref) ASSIGN (any mode) COER (all explicit)
Control:	CLAUSES (declarations) GOTO SEL (numeric) PROC (value, parameters value, return) LOOPS (tested at middle, exit clause)
I/O:	?
Dependency:	?
Extensibility:	DATA (structured modes)

JOVIAL

Design:	AUTHORS (System Development Corporation) BASE (CLIP) YEAR (1960) PUR (general)
Data:	INT REAL BOOL ALPHA ARRAY (multidim) FIELDS (bit, 8 bit bytes) PREC (digits) STRUCT (one level only) DECL (optional, static, local, initialisable)
Operators:	ARITH (+, -, x, /, ÷, ↑) LOG (∧, ∨, ¬) REL (>, ≥, <, ≤, =, ≠) ASSIGN (any) PRI (std) COER (implicit int-real)
Control:	CLAUSES (declarations) GOTO (conditional, numeric) SEL (conditional, biconditional, logical) PROC (value, parameters name-value) LOOPS (tested at top-middle, stepped)
I/O:	None
Dependency:	CODE (inline) SEGMENTATION (data)
Extensibility:	MACRO

LISP 2

Design:	AUTHOR (?) BASE (LISP 1.5, ALGOL60) YEAR (1965) PUR (general)
Data:	INT REAL BOOL ALPHA ARRAY (simple dim, based 1) STRUCT (linked lists of other data) DECL (required, local, static) OTHER (procedure reference)
Operators:	ARITH (+, -, *, /, ÷, rem) LOG (∧, ∨, ¬) REL (<, ≤, >, ≥, =, ≠) ASSIGN (any) PRI (std except list operators) COER (implicit int-real-bool) OTHER (head and tail of list operators)
Control:	CLAUSES (declarations, value) GOTO SEL (conditional, biconditional, numeric) PROC (must have value, parameters value-ref, recursive) LOOPS (tested at top, exit clause, stepped) OTHER (expression language)
I/O:	STREAM
Dependency:	CODE (inline)
Extensibility:	Programs can extend themselves

LP-70

Design:	AUTHOR (Rossiensky et al) BASE (PL-360) YEAR (1969) PUR (systems programming) CRIT ("a systems programming language with parallel processes")
	LANGUAGE REFERENCE MATERIAL NOT AVAILABLE

LRLTRAN

Design:	AUTHOR (Mendicino) BASE (FORTRAN IV) YEAR (1966) PUR (systems language) CRIT (ease of writing, portability, efficient ...)
Data:	BASEPLUS FIELDS (variable) ARRAYS (based 0) REF (arbitrary) STRUCT (subword structure of fields, higher levels by equivalencing) OTHER (register)
Operators:	BASEPLUS REF (deref, address of) ASSIGN (multivalued) OTHER (shift left, shift right)
Control:	BASEPLUS PROC (parameters ref-value) LOOPS (may be decremented) OTHER (alphabetic labels)
I/O:	BASEPLUS
Dependency:	CODE (inline)
Extensibility:	MACRO (parametered)

LSD

Design:	AUTHOR (Bergeron et al) BASE (PL/I) YEAR (1970) PUR (systems)
Data:	INT REAL ARRAY (single dim, based 1, any type) STRING (variable) FIELD (bit, byte 8) PREC (halfword, doubleword) STRUCT (general) REF (arbitrary) DECL (optional, dynamic/static, global, initialisable) OTHER (stacked, register)
Operators:	ARITH (+, -, X, ÷, MOD, :) LOG (∧, ∨, ¬, X, <<, >>, ε) REL (>, <, ≤, ≥, ...) STRING (substring, delete, insert, concat, table occurrence) ASSIGN (arith, ref) REF (value of, address of) PRI (std) COER (explicit)
Control:	SEL (conditional, biconditional, numeric,exception) GOTO PROC (value, parameters ref, omitted parameters, recursive) TRAP (software) PROCESS (events) LOOPS (tested at top, exit clause, next clause, stepped) OTHER (coroutines)
I/O:	STREAM RECORD
Dependency:	CODE (inline, pseudo procs) SEG (data) OTHER (register allocation)
Extensibility:	OP, CONTROL, MACRO

MALUS

Design:	AUTHOR (General Motors Corp) BASE (XPL) YEAR (1970) PUR (systems programming)
	LANGUAGE REFERENCE MATERIAL NOT AVAILABLE
	MALUS is a machine dependent (CDC STAR) extension of XPL

MARY

Design:	AUTHOR (Rain) BASE (ALGOL68) YEAR (1972) PUR (systems programming)
Data:	BASEPLUS FIELDS (none) DECL (specify exactly the internal mode required) OTHER (data can be declared read-only, simple row structures, sets, powersets)
Operators:	BASEPLUS REF (value of) COER (no widening)
Control:	BASEPLUS LOOPS (tested at bottom, optimised form)
I/O:	?
Dependency:	None
Extensibility:	BASEPLUS

META II

Design:	AUTHOR (Schone et al) YEAR (1963) PUR (writing compilers)

META II cannot be adequately described in the standard structure. A META II program consists of a representation of a BNF language description. Each production of this description normally contains at least one call on a literal output procedure which would normally be used to produce an intermediate language (e.g. assembler). Each production is considered a recursive procedure definition. The META II program inputs a string of tokens (which can be identifiers, strings, numbers, and special chars), parses this top-down, and produces the indicated output.

MOL-360

Design: AUTHOR (System Development Corp) BASE (ALGOL60) YEAR (1967) PUR (writing an operating system)

LANGUAGE REFERENCE MATERIAL NOT AVAILABLE

However, the language has only arrays for data structuring and has no floating-point or character manipulation facilities. It is machine-dependent (IBM360) and features inline assembler and register declarations.

MOL-940

Design: AUTHOR (Hay and Rulisfson) BASE (ALGOL60) YEAR (1968) PUR (systems programming)

LANGUAGE REFERENCE MATERIAL NOT AVAILABLE

However, the language has no floating-point or string manipulation facilities, arrays are the only data structuring tool, but has bit manipulation facilities. It is machine-dependent (SDS-940) and permits inline assembler.

NELIAC

Design:	AUTHOR (Halstead) BASE (ALGOL58) YEAR (1969) PUR (general)
Data:	INT REAL ARRAY (single dim) FIELD (variable) PREC (digits) DECL (required, local/global, static, initialisable)
Operators:	ARITH (+, -, x, /, ↑, abs) REL (<, ≤, >, ≥, =, ≠) LOG (∧, ∨) ASSIGN (array slices) PRI (none) COER (explicit)
Control:	CLAUSES GOTO SEL (conditional, biconditional) PROC (value, parameters) LOOPS (tested at top, stepped)
I/O:	STREAM (most implementations)
Dependency:	None
Extensibility:	None

OSL

Design:	AUTHOR (Alsberg and Wells) BASE (ALGOL60) YEAR (1968) PUR (writing operating systems)

LANGUAGE REFERENCE MATERIAL NOT AVAILABLE

However, the language is an extended subset of ALGOL60 (including constructs to handle interrupts). It appears to be machine dependent.

PASCAL

Design:	AUTHOR (Wirth) BASE (EULER) YEAR (1966) PUR (teaching)
Data:	INT REAL ALPHA ARRAY (multidim) REF (?) PREC (digits) STRUCT (general) DECL (required, local, static) OTHER (set, powerset, file)
Operators:	ARITH (+, -, x, /, rem) LOG (∧, ∨, ¬) REL (>, ≥, <, ≤, =, ≠, ⊂) REF (deref) ASSIGN (subranges) PRI (std) COER (implicit int-real) OTHER (set union, intersection, and difference)
Control:	CLAUSES GOTO SEL (conditonal, biconditional, numeric, logical) PROC (value, parameters, recursive) LOOPS (tested at top-bottom, stepped)
I/O:	STREAM RECORD
Dependency:	None
Extensibility:	DATA

PL11

Design:	AUTHOR (Russell) BASE (PL360) YEAR (1971) PUR (systems programming)
Data:	INT REAL ARRAY (single dim, based 0) PREC (multicell) DECL (required, equateable, initialisable) OTHER (registers)
Operators:	ARITH (+, -, x, /) REL (<, >, =, ≠) LOG (∧, ∨, ¬) REF (ref to) ASSIGN (arith) PRI (none) OTHER (stack manipulation operators reflecting PDP11 architecture)
Control:	CLAUSES SEL (conditional, biconditional) PROC () LOOPS (tested at top, stepped)
I/O:	None
Dependency:	Extreme
Extensibility:	None

PL1130

Design:	AUTHOR (Doran) BASE (PL360) YEAR (1971) PUR (systems programming)
Data:	INT ARRAY (single dim, based 0) DECL (required, initialisable) PREC (multicell) OTHER (predeclared registers)
Operators:	ARITH (+, -, x, /) LOG (∧, ∨, XOR) OTHER (test accumulator)
Control:	GOTO SEL (conditional, biconditional) PROC (value, parameters ref) LOOPS (exit clause)
I/O:	None
Dependency:	Extreme
Extensibility:	None

PL360

Design:	AUTHOR (Wirth) YEAR (1967) PUR (systems programming) CRIT ("to improve readability of programs which must take into account the specific characteristics and limitations of a particular computer")
Data:	INT REAL BOOL FIELD (8 bit bytes) ARRAYS (single dim, based 1) PREC (multicell) DECL (required, local/global, static equateable, initialisable) OTHER (registers)
Operators:	ARITH (+, -, x, /, ↑, mod) REL (<, ≤, >, ≥, =, ≠) LOG (∧, ∨, ¬) ASSIGN (any) PRI (left-right) OTHER (register tests, shift ops)
Control:	CLAUSES (declarations) GOTO SEL (conditional, biconditional, numeric) PROC () LOOPS (tested at top, stepped)
I/O:	None
Dependency:	Extreme CODE (inline) SEGMENTATION (code, data)
Extensibility:	None

PL503

Design:	AUTHOR (Gordon) BASED (PL516) YEAR (1972) PUR (systems programming)
Data:	INT ARRAY (single dim, based 0) DECL (required, static, local/global, initialisable)
Operators:	peculiar to the Elliott 503, basically integer manipulations which reflect Elliott 503 code
Control:	CLAUSES (declarations) GOTO (conditional) SEL (biconditional) PROC (parameters) LOOP (tested at top, stepped)
I/O:	None
Dependency:	Extreme
Extensibility:	None

PL516

Design:	AUTHOR (Bell and Wichman) BASE (PL360) YEAR (1970) PUR (systems)
Data:	INT ARRAY (single dim, based 1) DECL (required, local, dynamic, initialisable)
Operators:	ARITH (+, -, x, /, mod) LOG (∧, ∨, ¬, XOR) REL (<, ≤, >, ≥, =, ≠) ASSIGN (any) PRI (left-right) OTHER (shift ops, compare)
Control:	CLAUSE GOTO SEL (conditional, biconditional) PROC (value) LOOPS (tested at top, stepped)
I/O:	None
Dependency:	Extreme
Extensibility:	None

PL/I

Design:	AUTHOR (IBM Corp) YEAR (1964) PUR (general)
Data:	INT REAL FIELD (bit, 8 bit byte) ARRAY (multidim, any) STRING (bounded) FIELDS (bit, 8 bit bytes) PREC (digits) STRUCT (general) REF (arbitrary) DECL (optional, static/dynamic, local/global, initialisable, equateable) OTHER (areas)
Operators:	ARITH (+, -, x, /, ↑, mod) LOG (∧, ∨, ¬) REL (<, ≤, >, ≥, =, ≠) REF (ref to, deref) STRING (relational, size, substring, translate, move, table occurrence) ASSIGN (any, array slices) PRI (std) COER (all implicit)
Control:	CLAUSE (declarations, named) GOTO (numeric) SEL (conditional, biconditional) PROC (values, parameters ref, recursive, multiple entry) INT (software, hardware) PROCESS (events) LOOPS (tested at top, stepped)
I/O:	STREAM RECORD
Dependency:	None
Extensibility:	DATA (structured modes) MACRO (compile-time everything) CONTROL (generic procedures)

PL/S

Design:	AUTHOR (IBM Corp) BASE (PL/I) YEAR (1970?) PUR (systems programming)
Data:	INT ARRAY (single dim, based 0) FIELDS (bit, 8 bit bytes) STRING (bounded) PREC (multicell) STRUCT (general, self referencing) REF (arbitrary) DECL (required?, local/global, static/dynamic, based, equateable, initialisable) OTHER (entry, register)
Operators:	BASEPLUS REF (address of) COER (no implicit coercions)
Control:	CLAUSE GOTO (numeric) SEL (conditonal, biconditional) PROC (value, parameters ref, multiple entry) LOOPS (tested at top, stepped)
I/O:	None
Dependency:	CODE (inline) SEGMENTATION (code, data) OTHER (register control)
Extensibility:	MACRO (?) DATA (based data structures)

PROTEUS

Design:	AUTHOR (Bell) YEAR (1968) PUR (systems programming)

LANGUAGE REFERENCE MATERIAL NOT AVAILABLE

However, PROTEUS is an extensible language, and has at least one derived language.

PS440

Design:	AUTHOR (Sapper et al) YEAR (1970) PUR (systems programming)
Data:	ARRAY (single dim) FIELDS (variable) PREC (multicell) DECL (required, local/global, static?, initialisable) STRUCT (single level only) OTHER (typeless variables)
Operators:	ARITH (+, -, x, /) LOG (∧, ∨, ¬) REL (<, ≤, >, ≥, =, ≠) REF (address of, value of) ASSIGN PRI (std) COER (all explicit) OTHER (monadic machine operations, shift and mask ops)
Control:	CLAUSES (declarations) GOTO SEL (conditional, biconditional, numeric) PROC () LOOPS (tested at top-middle, stepped)
I/O:	None
Dependency:	CODE (inline) SEGMENTATION (code)
Extensibility:	None

SABRE PL/I

Design: AUTHOR (Hopkins) BASE (PL/I) YEAR (1968) PUR (systems programming)

LANGUAGE REFERENCE MATERIAL NOT AVAILABLE

However, the language is a large-subset PL/I with restrictions to improve efficiency (e.g. bounded strings)

SAL

Design: AUTHOR (Lang) YEAR (1967) PUR (systems programming) CRIT ("combines freedom and flexibility of assembly code with many facilities normally associated with high-level languages")

Data: INT REAL ARRAY (single dim, based 1) DECL (required, local/global, static) OTHER (registers, entry points)

Operators: ARITH (+, -, x, /) LOG (∧, ∨, ¬) REL (<, ≤, >, ≥, =, ≠) REF (deref, address of, index) ASSIGN (any) PRI (none?) OTHER (shift ops)

Control: GOTO (numeric) SEL (conditonal, biconditional) PROC (?) LOOPS (tested at top, stepped)

I/O: STREAM

Dependency: high CODE (inline) SEGMENTATION (data)

Extensibility: MACRO (simple)

SL/8

Design: AUTHOR (Heidt and Fricks) BASE (ALGOL60) YEAR (1970) PUR (writing operating systems)

LANGUAGE REFERENCE MATERIAL NOT AVAILABLE

A machine dependent language which reflects features of the PDP/8 architecture.

SLANG

Design:	AUTHOR (IBM?) YEAR (1960) PUR (general) CRIT ("it is possible to describe processes in a machine-independent language which are themselves machine dependent")

The SLANG system can not be adequately described within the usual notation. The system accepts as input a problem-oriented-language (POL) and a machine description and produces a machine language program. The machine description is given in terms of a number of machine-independent macros. The POL bears some similarity to ALGOL58.

SNOBOL 4

Design:	AUTHOR (Griswold) BASE (SNOBOL 2) YEAR (1967) PUR (string and list processing)
Data:	ARRAY (multidim, any) STRING (varying) STRUCT (general, self referencing) REF (arbitrary) DECL (array and structures only) OTHER (typeless variables)
Operators:	ARITH (+, -, x, /, ↑, mod) REL (<, ≤, >, ≥, =, ≠) REF (deref) STRING (relational size, substring, move, table occurrence and many more) ASSIGN PRI (left-right) OTHER (parsing implied by statement form)
Control:	Each statement in the language may have named successors. If control does not pass to a named successor it passes to the next statement.
I/O:	STREAM
Dependency:	None
Extensibility:	DATA CONTROL (programs may extend themselves)

SUE

Design:	AUTHOR (Clark and Ham) YEAR (1972) PUR (writing operating systems)
Data:	INT ARRAY (multidim) STRING (bounded) FIELDS (variable) PREC (bits) STRUCT (general) REF (general) DECL (required, local, dynamic) OTHER (register)
Operators:	ARITH (+, -, x, /, mod) LOG (∧, ∨, XOR, ¬) REL (<, ≤, >, ≥, =, ≠) REF (deref) STRING (characters are treated as bytes, i.e. integers) ASSIGN (any, array slices) PRI (std) COER (explicit) OTHER (set union, intersection, difference and powerset operations, succ, pred)
Control:	SEL (conditional, biconditional, numeric, logical) PROC (value, parameters value, recursive) PROCESS (events) LOOPS (tested at middle, stepped, exit clause)
I/O:	STREAM
Dependency:	CODE (inline)
Extensibility:	MACRO DATA (any modes)

SIMPL X

Design:	AUTHOR (Basili) YEAR (1973) PUR (systems programming)
Data:	INT ARRAY (single dim, based 1?) DECL (required, local, dynamic, initialisable)
Operators:	ARITH (+, -, x, /) REL (<, ≤, >, ≥, =, ≠) LOG (∧, ∨, ¬, XOR) ASSIGN PRI (std) OTHER (shift ops, part word ops)
Control:	CLAUSES SEL (conditional, biconditional, numeric) PROC (value, parameters value-ref, recursive) LOOPS (tested at top, exit clause)
I/O:	STREAM
Dependency:	Low
Extensibility:	Compiler is extensible.

SYMPL

Design:	AUTHOR (Computer Sciences Corporation) BASE (PL/I) YEAR (?) PUR (systems programming)

LANGUAGE REFERENCE MATERIAL NOT AVAILABLE.

SYSL

Design:	AUTHOR (Terashima?) YEAR (1972) PUR (systems) CRIT (runtime efficiency and space efficiency less than 1.15 times that of assembler language)
Data:	INT REAL? ALPHA ARRAY (single dim, based 1) int, real) STRING (variable) FIELDS (bit strings) STRUCT (static) REF (string, program, offset into array) DECL (required?, dynamic, local, initialisable) OTHER (areas)
Operators:	ARITH (?) REL (?) REF (deref) STRING (?) ASSIGN (arith?) PRI (std) COER (implicit?)
Control:	CLAUSES (declarations, named) GOTO (conditional) SEL (biconditional) PROC (value, ?) INTER (software) LOOPS (tested at top, stepped)
I/O:	RECORD
Dependency:	OTHER (pseudo procs)
Extensibility:	None?

TMG

Design:	AUTHOR (McLure) YEAR (1964) PUR (compiler generator)

This language can not be described adequately within the usual notation. TMG performs a top-down parse with backup. Semantic rules may be embedded in the parse rules. The basic TMG statement form is a sequence of actions. Each action may be labelled, and it may indicate a failure exit label. I/O is character oriented.

TRAC

Design:	AUTHOR (Mooers) YEAR (1964) PUR (text manipulation) CRIT (designed specifically for handling unstructured text in an interactive mode).
Data:	STRING (varying) OTHER (other types e.g. integer are considered subclass of strings)
Operators:	ARITH (+, -, x, /) LOG (∧, ∨, ¬) REL (=, >) REF (ref to) STRING (relational, substring) ASSIGN PRI (bracketing) OTHER (string operations, shift and rotate)
Control:	interpretive, statement by statement
I/O:	STREAM (string)
Dependency:	None
Extensibility:	Programs may modify themselves

TRANDIR

Design:	AUTHOR (Massachusetts Computer Associates) YEAR (1964) PUR (compiler generator)
Data:	INT STRING (varying) OTHER (label)
Operators:	A number of builtin functions manipulate a parse tree directly
Control:	GOTO SEL (conditional, biconditional)
I/O:	STREAM
Dependency:	None
Extensibility:	None

XALGOL

Design:	AUTHOR (Burroughs Corp) BASE (ALGOL60) year (1967) PUR (general) CRIT (suitable for compiler writing)
Data:	BASEPLUS ALPHA ARRAY (read only) PREC (multicell) FIELDS (variable) REF (string, array) DECL (equateable)
Operators:	BASEPLUS STRING (relational, translate, move, table occurrence) COER (many explicit coercions) OTHER (field operations)
Control:	BASEPLUS SEL (numeric) INT (software, hardware) PROCESS (events, locks) LOOPS (tested at bottom)
I/O:	STREAM RECORD
Dependency:	High
Extensibility:	MACRO (parameters)

XPL

Design:	AUTHOR (McKeeman) BASE (PL/I) YEAR (1968?) PUR (compiler writing)
Data:	INT ARRAY (single dim, based 0) STRING (varying) FIELDS (variable) DECL (required, static, local, initialisable)
Operators:	ARITH (+, -, x, /, mod) LOG ($\wedge$, $\vee$, $\neg$) REL ($<$, $\leq$, $>$, $\geq$, $=$, $\neq$) STRING (relational, substring, concatenation)
Control:	CLAUSES GOTO SEL (conditional, biconditional, numeric) PROC (value, parameters ref) LOOPS (tested at top, stepped)
I/O:	STREAM
Dependency:	Low CODE (inline)
Extensibility:	MACRO

§6.2 A Language Design

A careful study shows that several languages satisfy most of the criteria for good systems languages. They are SUE, BLISS, MARY and SIMPL-X. Each has some shortcomings: SUE, BLISS, and SIMPL-X do not have adequate string manipulation facilities, for example. SUE and MARY are more suited to larger machines, BLISS and SIMPL-X to medium-size and small machines. In view of the remarks made in the introduction (and elsewhere) regarding proliferation of languages, it would be hypocritical of me to suggest otherwise than that one of these languages be adopted.

§6.3 Glossary

bad goto	-	any goto which cannot be compiled into a simple branch instruction
cell	-	the smallest addressable unit of main storage
coercion	-	the process by which one mode may be converted to another
definition (as opposed to declaration)	-	the specification of a mode, a declaration which does not allocate storage
dynamic	-	runtime changeable
field	-	part of a cell
form	-	secondary processing type, such as string manipulation, error recovery, list processing
heap	-	storage under control of a garbage collector
inline code	-	arbitrary code inserted at the statement level
items (data)	-	any piece of named data
mode	-	formal class associated with identifiers, normally a data type
orthogonality	-	independence of complementary concepts within a language
overlayable	-	able to be overwritten (in main storage) at runtime
prelude	-	standard set of definitions used to extend a base language for a specific purpose
process	-	performed function
task	-	a unit of work treated in an independent manner by the operating system
typeless	-	having no mode or type, being any mode without coercion
union	-	mode possessing one of a limited set of modes
weak typing	-	the ability to treat data as various different modes

§6.4 References

This section has two parts. §6.4.1 is a language ordered cross reference to §6.4.2 (which is the proper table of references).

An attempt has been made to limit the number of references by deleting those concerned directly with languages if and only if (a) another (more accessable) paper discussing the language was available (b) the paper concerned did contain discussion of things other than the language itself. This mainly eliminated programmers guides and language reference manuals. Sammet [Sam 71, Sam 74] gives many of these.

§6.4.1 Language to Papers Cross Reference

AED-0	Ros 69
AL	Hai 73
ALGOL 60	Gal 67, Nau 63, Wir 63, Wir 66
ALGOL 68	Bra 71, Van 74
ALGOL W	Boo 74
ALPHA	Yer 66
AMBIT/L	Chr 71
APAREL	Bal 69
APL	Ive 62
B	Joh 73
BCPL	Ric 69
BLISS	Wul 71a, Wul 71b, Wul 72
BSL	= PL/S
CIMPL	Cla 71a
CLIP	Eng 61
COBOL	ANS 73
COGENT	Rey 65
CORAL 66	BCS 70
DCALGOL	Bur 73
EPL	MIT 66
ESPOL	Bur 72
FORTRAN IV	ANS 66
FSL	Fel 66
GARGOYLE	Gar 64
GOGOL III	Sau 67
GPL	Gar 68
IMP (Irons)	Iro 70
IMP (Edin)	Ste 74
JOSSLE	Whi 73
JOVIAL	Sha 63
LISP	Jen 70, Abr 66
LP70	Ros 70
LRLTRAN	Dub 71, Men 68
LSD	Ber 71, Ber 72
MALUS	GMC 70
MARY	Rai 72
META	Opp 66

MOL-360	Boo 70, Boo 71
MOL-940	Hay 68
NELIAC	Mas 60
NPL	= PL/I
OSL	Als 68
PASCAL	Wir 69, Wir 71a
PL11	Rus 71
PL1130	Dor 72
PL360	Wir 68
PL503	Gor 72
PL516	Bel 71
PL/I	ANS 73, Cor 69, Dod 66, Fre 69, Hop 71, Pes 71
PL/S	Wie 71
PROTEUS	Bel 68
PS440	Sap 71
SABRE PL/I	Hop 68
SAL	Lan 69a, Lan 69b
SL/8	Hei 70
SLANG	Sib 61
SNOBOL 4	Gri 68
SUE	Cla 71b
SIMPL	Bas 73, Bas 74
SYMPL	CSC
SYSL	Ter 74
TMG	McL 65
TRAC	Mop 65
TRANDIR	Che 66b
XALGOL	Bur 74
XPL	McK 70

§6.4.2 Table of References

Abr 68	Abrahams PW The LISP2 Programming Language and System Proc FJCC 29 p661. 1966
Als 68	Alsberg PA and Wells RA OSL, An Operating System Language TP University of Illinois, Urbana, Ill 61801. May 1968
ANS 66	American National Standards Institute American National Standard FORTRAN ANS X3.9 - 1966
ANS 68	American National Standards Institute American National Standard COBOL ANS X3.23 - 1968
ANS 73	American National Standards Institute ECMA ANSI PL/I (BASIS 1-10) working document 1973
Ard 70	Arden B and Hamilton J A Study of Programming Language Effectiveness US Army Safeguard Systems Command, Contract DAH C60-70-C-0036. 1970
Bak 72	Baker FT System Quality through Structured Programming Proc FJCC pp339-343. 1972
Bal 69	Balzar RM and Farber DJ APAREL, A Parse-Request Language CACM 12,11 pp624-631. Nov 1969
Bas 73	Basili VR SIMPL-X: A Language for Writing Structured Programs TR-223 Computer Science Center, University of Maryland. Jan 1973
Bas 74	Basili VR The SIMPL Family of Programming Languages and Compilers TR-305 Computer Science Center, University of Maryland. June 1974
BCS 70	BCS Specialist Group (Online Computers and their Languages) A Language for Real Time Systems Computer Bulletin. Dec 1970
Bel 71	Bell DA and Wickmann BA An Algol-Like Assembly Language for a Small Computer Software Practice and Experience 1,1 p61. Jan 1971
Bel 68	Bell JR The Design of a Minimal Expandable Computer Language Ph.D. Thesis. Dept. of Computer Science Stanford University Stanford Calif. Dec 68
Ber 71	Bergeron RD et al Languages for System Development Sigplan Notices 6,9 p50. Oct 1971
Ber 72	Bergeron RD et al Systems Programming Languages Advances in Computers 12 pp175-284. 1972

Boo 70 Book E et al
The CWIC/360 System, A Compiler for Writing and Implementing Compilers
TR-SD-3510 System Development Corp, Santa Monica, California 90406. April 1970

Boo 71 Book E et al
CWIC Users Guide : The MOL-360 Language
TR-TM-(L)-4185/004/00 System Development Corp, Santa Monica, California 90406. Feb 1971

Boo 74 Boom H
Experience with the use of Algol W as a SIL.
Algol Bulletin 37 pp63-67. July 1974

Bra 71 Branquart P et al
The Composition of Semantics in ALGOL-68
CACM 14,11 pp697-707. Nov 1971

Bro 66 Brooker RA et al
The Main features of Atlas Autocode
Computer Journal 8,4. Jan 1966

Bro 69 Brown PJ
A Survey of Macro Processors
Annual Review in Automatic Programming 6,2. 1969

Bro 69 Brown PJ
Using a Macro Processor to Aid Software Implementation
Proc SJCC 1969 p327-331. 1969

Bro 74 Brown P
Writing Software in ALGOL
Software Practice & Experience 4,2 pp139-144. April 1974

Bur 74 Burroughs Corporation
Burroughs B6700/B7700 ALGOL Language Reference Manual
TR-5000649 Burroughs Corp. May 1974

Bur 73 Burroughs Corporation
Burroughs B6700/B7700 DCALGOL Reference Manual
TR-5000052 Burroughs Corporation. June 1973

Bur 72 Burroughs Corporation
Burroughs B6700/B7700 ESPOL Language Manual
TR-500094 Burroughs Corporation. June 1972

Che 66a Cheatham TE
The Introduction of Definitional Facilities into Higher Level Programming Languages
Proc FJCC 29 pp623-637. 1966

Che 66b Cheatham TE
The TGS-II Translator Generator System
Proc IFIP Congress 65 Vol.2 pp592-593. 1966

Chr 71 Christensen C
An Introduction to AMBIT/L, A Diagramatic Language for List Processing
Proc 2nd Symposium on Symbolic and Algorithmic Manipulation ACM. March 1971

Cla 71b Clark BL and Horning JJ
The System Language for Project SUE
Sigplan Notices 6,9 p79-88. Oct 71

Cla 71a Clark DD et al
The Classroom Information and Computing Service
TR-MAC TR-80, MIT Project MAC, Cambridge, Massachussetts 02139.
Jan 1971

Cor 69 Corbato FJ
PL/I as a Tool for Systems Programming
Datamation 15,5. May 1969

CSC CSC
Systems Programming Language (SYMPL)
Computer Sciences Corporation, El Segundo, California 90245

Dij 68 Dijkstra EW
GO TO Statement Considered Harmful
CACM 11,3 p.147. March 1968

Dod 66 Dodd G
APL, A Language for Associative Data Handling in PL/I
Proc FJCC 29. Nov 1966

Don 72 Donovan JJ
Systems Programming
McGraw-Hill 1972

Dor 72 Doran RW and Navankasattusas T
Designing High-Level/Low-Level Computer Languages
Massey University Computer Unit
TR-MUCUP 7. May 1972

DuB 71 Du Bois PJ and Martin JT
The LRLRAN Language as Used in the FROST and FLOE Time-Sharing Operating Systems
Sigplan Notices 6,9 pp92-104. Oct 1971

Eng 61 Englund D and Clark E
The CLIP-translator
CACM 4,1 pp19-22. Jan 1961

Far 71 Farber DJ
A Survey of the Systematic Use of Macros in Systems Building
Sigplan Notices 6,9 pp29-36. Oct 1971

Fel 66 Feldman JA
A Formal Semantics for Computer Languages and its application in a Compiler-Compiler
CACM 9,1 pp3-9. Jan 1966

Fel 68 Feldman JA and Gries D
Translator Writing Systems
CACM 11,2 p77. Feb 1968

Fel 69 Feldman JA and Rovner PD
An Algol-Based Associative Language
CACM 12,8 pp439-449. Aug 1969

Fle 72 Fletcher JG et al
On the Appropriate Language for System Programming
Sigplan Notices 7,7 p29. July 1972

Fre 69 Freiburghouse RA
The Multics PL/I Compiler
Proc FJCC 35 pp187=199. Nov 1969

Gal 67 Galler B and Perlis AJ
A proposal for Definitions in ALGOL
CACM 10,4 pp204-219. April 1967

Gar 64 Garwick JV
GARGOYLE, A Language for Compiler Writing
CACM 7,1 pp16-20. June 1964

Gar 68 Garwick JV
GPL, A Truly General Purpose Language
CACM 11,9 pp634-638. Sept 1968

Gea 65 Gear CW
High Speed Compilation of Efficient Object Code
CACM 8,8 p483. Aug 65

GMC 70 General Motors Corporation
MALUS
General Motors Corporation. 1970

Goo 72 Goos G
On System Programming Languages
IFIP WG2.1 Fonteinbleu 1972

Gor 72 Gordon NG
PL 503 Users Manual
TR-ISM 62 Information Science Dept., Victoria University, Wellington, N.Z. Nov 72

Gra 70 Graham R
The Use of High Level Languages for Systems Programming
Proc Inv Workshop on Network of Computers (NOC-69)
National Security Agency, Fort George Nedde, Maryland. 20755
Oct 1970

Gri [illegible] Griswold RE et al
The SNOBOL 4 Programming Language
Prentice Hall. 1968

Hai 73 Haines EC
AL : A Structured Assembly Language
Sigplan Notices 8,1 p15. Jan 1973

Hay 68 Hay RE and Rulifson JF
MOL-940, Preliminary Specification for an ALGOL-like machine-orientated language for the SDS940
SRI Project 5890 Stanford, California
Interim Technical Report 2. March 1968

Hei 70 Heidt JS and Fricks CL
SL/8, A Synthesis Language gor the PDP-8/I
TR-GITIS-70-02, Georgia Institute of Technology, Atlanta, Georgia. 1970

Hoa 73 Hoare CA
Recursive Data Structures
TR CS-73-400 Stanford Artificial Intelligence Lab. Oct 1973

Hop 68 Hopkins M
SABRE PL/I
Datamation 14,12 p35. Dec 1968

Hop 71 Hopkins M
Problems of PL/I for Systems Programming
Sigplan Notices 6,9 p89. Oct 1971

Hus 62	Huskey HD A Language for Aiding Compiler Writing Proc Symbolic Language in Data Processing p187 Gordon and Breach. New York. 1962
Iro 70	Irons ET Experience with an Extensible Language CACM 13,1 pp31-40. Jan 1970
Ive 62	Iverson KE A Programming Language Wiley. 1962
Jen 70	Jenks RD META/LISP, An Interactive Translator Writing System TR-RC 2968 IBM, TJ Watson Research Centre, Yorktown Heights New York 10598
Joh 73	Johnson SC and Kernighan BW The Programming Language B TR-CS-8 Bell Labs, Murray Hill, N.J. 07974. Jan 73
Knu 71	Knuth DE and Floyd RW Notes on Avoiding "GO TO" Statements Information Proc Letters pp23-31. 1971
Lan 66	Landin PJ The Next 700 Programming Languages CACM 9,3 pp157-166. March 1966
Lan 69a	Lang CA SAL, System Assembly Language Proc SJCC 34 pp543-555. Nov 1969
Lan 69b	Lang CA Languages for Writing Systems Programs NATO Conference on Software Engineering Techniques p101. Oct 1969
Low 69	Lowry ES and Medlock CW Object Code Optimisation CACM 12,1 pp13-21. Jan 1969
Lyl 71	Lyle DM A Hierachy of High Order Languages for Systems Programming Sigplan Notices 6,9 pp73-78. Oct 1971
Mar 73	Martin CW Assemblers : Ancient & Modern Proc DATAFAIR 73 Conf. Vol 2 pp443-449. 1973
Mas 60	Masterton KS Compilation for Two Computers with NELIAC CACM 3,11 p607. Nov 1960
McI 60	McIlroy MD Macro Instruction Extensions of Compiler Languages CACM 3,4. April 1960
McK 70	McKeeman WM et al A Compiler Generator Prentice Hall. 1970
McL 65	McLure TMG, A Syntax Directed Compiler Proc ACM 20th National Conference p262. 1965
Men 68	Mendicino SF et al The LRLTRAN Compiler CACM 11,11 pp747-755. Nov 1968

MIT 60	Massachussetts Institute of Technology EPL Reference Manual Project MAC
Mop 65	Mopers C and Deutsch LP TRAC, A Text Handling Language Proc ACM 20th National Conference p229. 1965
Nau 63	Naur P Revised Report on Algorithmic Language ALGOL60 CACM 6,1 p1. Jan 1963
Opp 66	Oppenheim DK and Haggerty DP META 5: A Tool to Manipulate Strings of Data Proc ACM 21st Nat'l Conf. 1966
Pes 71	Peschke JV PL/I Subsets for Software Writing Sigplan Notices 6,4. May 1971
Rai 72	Rain M Some Formal Language Aspects of MARY ALGOL BULLETIN AB34.4.2. p45. June 1972
Rey 65	Reynolds JC An Introduction to the COGENT Programming System Proc ACM 20th National Conference p422. 1965
Ric 69	Richards M BCPL, A Tool for Compiler Writing and Systems Programming Proc SJCC 34 pp557-566. May 1969
Ros 69	Ross DT Introduction to Software Engineering with the AED-0 Language MIT Cambridge, Massachussetts. 1969
Ros 70	Rossiensky JP and Tixier VT LP70: A Systems Programming Language with Parallel Processes Proc ACM International Computer Symposium, Bonn p492. 1970
Rus 71	Russell RD Preliminary Specifications of PL 11: a programming language for the DEC PDP11 Computer. TR-SW-29 CERN DD/OM Development Note. 1971
Sam 69	Sammet JE Programming Languages, History and Fundamentals Prentice-Hall, Englewood Cliffs NJ 07632. 1969
Sam 71	Sammet JE A Brief Survey of Languages Used in Systems Implementation Sigplan Notices 6,9 p1. Oct 1971
Sam 74	Sammet JE Roster of Programming Languages for 1973 Sigplan Notices 9,11 pp18-31. Nov 1974
Sap 71	Sapper GR The Programming Language PS440 as a Tool for Implementing a Time-Sharing System Sigplan Notices 6,9 p37. Oct 1971
Sau 67	Sauter J GOGOL III, An Algol-like Language for the PDP-6 TR-CS 239 Stanford University. 1967

Sch 69 Schneider V
Some Syntactic Methods for Specifying Extendible Programming Languages
Proc FJCC p145-156. 1969

Sha 63 Shaw CJ
A Specification of JOVIAL
CACM 6,12 p721. Dec 1963

Sib 61 Sibley RA
The SLANG-System
CACM 4,1 p75. Jan 1961

Sli 71 Slimick J
Current Systems Implementation Languages: One User's View
Sigplan Notices 6,9 p20-28. Oct 71

Sol 72 Solntseff N
A Classification of Extensible Programming Languages
Information Proc Letters 1 pp91-96. 1972

Ste 74 Stephens PD
The IMP Language and Compiler
Computer Journal 17,3. 1974

Swi 68 Swinehart D
GOGOL III
ON 48 Stanford Artificial Intelligence Laboratory
Stanford University. Dec 1968

Ter 74 Terashima N
SYSL - System Description Language
Sigplan Notices 9,12 p35. Dec 1974

Tix 69 Tixier V
O.S. Writing Systems
Informal Working Paper for NATO Conference on Software Engineering. Oct 1969

Van 74 Van Wijngnaarden A (Ed)
Revised Report on the Algorithmic Language ALGOL68
TR-74-3 Computer Science Dept., University of Alberta

Wie 71 Wiederhold G and Ehrman J
Inferred Syntax and Semantics of Pl/S
Sigplan Notices 6,9 p111-121. Oct 1971

Wir 63 Wirth N
A generalisation of ALGOL
CACM 6 pp547-554. 1963

Wir 66 Wirth N and Hoare CA
A Contribution to the Development of ALGOL
CACM 9,6 pp413-431. June 1966

Wir 68 Wirth N
PL360, A Programming Language for the 360 Computers
JACM 15,1 pp37-74. Jan 1968

Wir 69 Wirth N
The Programming Language PASCAL and its Design Criteria
NATO Conference on Software Engineering. Rome. Oct 1969

Wir 71a Wirth N
The Programming Language PASCAL
Acta Informatica 1 pp35-63
Springler-Verlag. 1971

Wir 71b	Wirth N Program Development by Stepwise Refinement CACM 14,4 p221-227. August 1971
Whi 73	White JR and Presser L A Structured Language for Translator Construction to be published in the Computer Journal. 1973
Wul 71a	Wulf WA et al Reflections on a Systems Programming Language Sigplan Notices 6,9 p42-49. Oct 1971
Wul 71b	Wulf WA et al BLISS, A Language for Systems Programming CACM 14,12 pp780-790. Dec 1971
Wul 72	Wulf WA Systems for Systems Implementations - Some Experiences from BLISS Proc FJCC pp943-948. 1972
Yer 66	Yershov AP ALPHA, An Automatic Programming System of High Efficiency JACM 13,1 p17. Jan 1966
Zah 73	Zahn CT A Control Statement for Natural Top-down Structured Programming TR DD-CTZ-1g Cern Switzerland. Oct 1973