BLISS: A Language for Systems Programming

W.A. Wulf, D.B. Russell, A.N. Habermann

Carnegie-Mellon University

A language, BLISS, is described. This language is designed so as to be especially suitable for use in writing production software systems for a specific machine (the PDP-10): compilers, operating systems, etc. Prime design goals of the design are the ability to produce highly efficient object code, to allow access to all relevant hardware features of the host machine, and to provide a rational means by which to cope with the evolutionary nature of systems programs. A major feature which contributes to the realization of these goals is a mechanism permitting the definition of the representation of all data structures in terms of the access algorithm for elements of the structure.

Introduction

In the fall of 1969, Carnegie-Mellon University acquired a PDP-10 to support a research project on computer networks. This research involves the production of a substantial number of large systems programs of the type which have usually been written in assembly language. At an early stage of this design effort it was decided not to use assembly language, but rather some higher-level language. This decision immediately led to another question—which language? In turn this led to a consideration of the characteristics, if any, which are unique to, or at least exaggerated in, the production and maintenance of systems programs. One product of these deliberations was a new language, which we call BLISS. Clearly, a language is not the only tool needed; however, it is the one with which we deal in this paper.

We refer to BLISS as an "implementation language," although we admit that the term is somewhat ambiguous since, presumably, all computer languages are used to implement *something*. To us the phrase connotes a general purpose, higher-level language in which the primary emphasis has been placed upon a specific application, namely the writing of large, *production* software systems for a specific machine. Special purpose languages, such as compiler-compilers, do not fall into this category, nor do we necessarily assume that these languages need be machine-independent. We stress the word "implementation" in our definition and have not used words such as "design" and "documentation." We do not necessarily expect that an implementation language will be an appropriate vehicle for expressing the initial design of a large system nor for the exclusive documentation of that system. Concepts such as machine independence, expressing the design and implementation in the same notation, self-documentation, and others, are clearly desirable goals and are criteria by which we evaluated various languages.

However, they are not implicit in our definition. There are a few extant examples of languages which fit our definition: EPL (a PL/1 derivative which was initially used on MULTICS [1] but which has been replaced by full PL/1), B5500 Extended ALGOL (Burroughs Corporation [2]), PL360 [3], and BCPL [4].

The various arguments for and against the use of higher-level languages to write systems software have been discussed at length. We do not intend to reproduce them here in detail except to note that the skeptics argue primarily on two grounds: efficiency, and an assertion that the systems programmer must not allow anything to get between himself and the machine. The advocates argue on the grounds of production speed (and cost), maintainability, re-design and modification, and understandability and correctness. The report of the NATO Conference on Software Engineering held in Garmish (October 1968) [5] contains several discussions on these points.

It is our opinion that program efficiency, except possibly for a very small number of very small code segments, is determined by overall program design and not by locally tricky, "bit-picking" coding practices. Moreover, the critical code segments are frequently located only after the system is operational. This opinion is borne out by many systems which have experienced substantial performance improvements from redesign or restructuring resulting from understanding or insight after the system has been running for some time (see, for example, E.E. David's comments, [5, pp. 55–57]). One of the paramount design objectives of BLISS was to facilitate redesign and recoding. Since this redesign is frequently done by someone other than the program's original author, there is a corollary objective of readability. This argues for good documentation—but also for understandability of the code itself. Understandability is a function of many things, not all of which are inherent in the language in which a program is written—a programmer's individual style, for example. Nevertheless, the length of a program text and the structure imposed upon that text are important factors and argue strongly for the use of a higher-level language.

Presuming the decision to use an implementation language, which one should one choose? An argument might be made for choosing one of the existing languages, say FORTRAN, PL/1, or APL, and possibly extending it in some way. We have chosen to add to the tongues of Babel by defining yet another new language, and some justification is required. The only valid rationale for creating a new language is that the existing ones are inappropriate to the task. It was our judgement that no existing languages dealt with all of the proper issues and hence a new language was necessary. What then are the special characteristics of systems programs for which existing languages are inappropriate? (Later we shall discuss how these manifest themselves in BLISS.) The two special characteristics most frequently mentioned are efficiency and access to all hardware features of the machine. We add several things to these; the resulting list forms the design objectives of BLISS.

Requirements of Systems Programs

- space/time economy
- access to all relevant hardware features
- object code must not depend upon elaborate run-time support

Characteristics of Systems Programming Practice

- control over the representation of data structures
- flexible range of control structures (notably including recursion, coroutines, and asynchronous processes)
- modularization of a system into separately compilable submodules
- parameterization, especially conditional compilation

Overall Good Language Design

- encourages program structuring for understandability
- encourages program structuring for debugging and measurement
- economy of concepts (involution), generality, flexibility, etc.
- utility as a design tool
- machine independence

The order in the above list is not accidental. Those items found early in the list we consider to be absolute requirements, while those occurring later in the list may be thought of as criteria by which alternative designs are judged once the more demanding requirements are satisfied.

Not all of the goals mentioned above are compatible in practice. For example, efficiency, access to machine features, and machine independence are conflicting goals. In fact, the design of BLISS is not machine independent, although the underlying philosophy and much of the specific design are. The machine for which the language was being designed, the PDP-10, was constantly in the minds of the designers. The code to be generated for each proposed construct, or form of a construct, was considered before that construct was included in, or excluded from, the language. Thus the characteristics of the target machine pervade the language in both overt and subtle ways. This is not to say that BLISS could not be implemented for another machine; it could. It does say that BLISS is particularly well suited to implementation on the PDP-10 and that it could probably not be as efficiently implemented on most other machines. We think of BLISS as a member (the only one at present) of a class of languages that are similar in philosophy and that mirror a similar concern for the important aspects of systems programming, but where each is tailored to its host machine.

As another example of the incompatibility of these goals, consider the requirement for minimal run-time support and the use of the implementation language as a design tool. In some sense a design tool should be at a higher level than the object being designed—that is, the tool should relieve the designer from concern over whichever details the designer deems appropriate only for later consideration. Any language relieves its user from concern over certain details; even assembly language frees the coder from the need to make specific address assignments. Assembly language is not a good design tool, however, precisely because the class of such facilities is small; a higher-level language is better because the class is larger. There is a point, however, beyond which broadening the class of details which are handled automatically introduces substantial costs in run-time efficiency and requisite run-time support. The design of BLISS walks a very fine line between generality, efficiency, and minimal run-time support. In fact, BLISS programs require *no* run-time support routines.

Description of BLISS

BLISS may be characterized as an "ALGOL-PL/1" derivative in the sense that it has a similar expression format and operator hierarchy, a block structure with lexically and dynamically local variables, similar conditional and looping constructs, and (potentially) recursive procedures. As may be seen from the two simple examples shown below, the general format of BLISS code is quite ALGOL-like; however, the similarity stops shortly beyond this glib comparison.

function $factorial(n) =$
  if $.n \leq 1$ **then** 1 **else** $.n * factorial(.n - 1)$;
function $QQ\ search\ (K) =$
  begin register R,Q,A,E;
  $E \leftarrow R \leftarrow .K/.n$; $Q \leftarrow .K\ mod\ .n$;
  $A \leftarrow .const$;
  do if $.ST[.R] = .K$
    then return $.R$
    else $(R \leftarrow .R + .A;\ A \leftarrow .A \leftarrow .Q)$
  until $.R = .E$
  end;

The first of these examples is the familar recursive definition of factorial. The second example is the "quadratic quotient" hash search described by J. Bell [9].

We now describe the features of BLISS in terms of its major aspects—(1) the underlying storage, (2) control, and (3) data structures; finally, we mention some other miscellaneous features.

1. Storage

A BLISS program operates with and on a number of storage "segments." A storage segment consists of a fixed and finite number of "words," each of which is composed of a fixed and finite number of "bits" (36 for the PDP-10). Any contiguous set of bits within a word is called a "field." A field may be "named"; the value of a name is called a "pointer" to that field. In particular, an entire word is a field and may be named. In practice a segment generally contains either program or data, and if the latter, it is generally integer numbers, floating-point numbers, characters, or pointers to other data. To a BLISS program, however, a field merely contains a pattern of bits. Various operations may be applied to fields and bit patterns, such as fetching a bit pattern (value) from a field, storing a bit pattern into a field, integer arithmetic, comparison, Boolean operations, and so on. The interpretation placed upon a particular bit pattern and the consequent transformation performed by an operator is an intrinsic property of that operator and not of its operands. That is to say, there is no "type" differentiation as in ALGOL.

Segments are introduced into a BLISS program by declarations, called "allocation declarations," for example:

global g;
own x, $y[5]$, z;
local $p[100]$;
register $r1$, $r2[3]$;
function $f(a, b) = .a \uparrow .b$;

Each of these declarations introduces one or more segments and binds the identifiers mentioned (e.g. g, x, y) to the name of the first word of the associated segment. (The **function** declaration also initializes the segment named f to the appropriate machine code.)

The segments introduced by these declarations contain one or more words, where the size may be specified (as in **local** $p[100]$), defaulted to one (as in **global** g;), or defaulted to whatever length is necessary for initialization (as in the **function** declaration). The identifiers introduced by a declaration are lexically local to the block in which the declaration is made (that is, they obey the usual ALGOL scope rules), with one exception—namely, **global** identifiers are made available to other, separately compiled modules. Segments created by

own, **global**, and **function** declarations are created only once and are preserved for the duration of the execution of a program. Segments created by **local** and **register** declarations are created at the time of block entry and are preserved only for the duration of the execution of that block. The **register** segments differ from **local** segments only in that they are allocated from the machine's array of 16 general purpose (fast) registers. Reentry of a block before it is exited (by recursive function calls, for example) behaves as in ALGOL, that is, **local** and **register** segments are dynamically local to each incarnation of the block.

It is important to notice from the discussion above that identifiers are bound to names by these declarations, and that the value of a name is a pointer. Thus the value of an instance of an identifier, say x, is *not* the value of the field named by x, but rather is a pointer to x. This interpretation requires a "contents of" operator for which the symbol "." has been chosen. To a programmer who is used to the context-dependent interpretation of identifiers, it may seem that the notations x and $.x$ for a pointer to a field and value of that field, respectively, might better be replaced by $@x$ and x. However, a little comparison will soon show that the dot notation is to be preferred.

First, "." has a unique interpretation as a unary operator meaning: "take the contents of the field pointed at by x." $@x$ cannot be interpreted as the inverse of "$.x$", since the dot function does not have a unique inverse (there may be many locations with the same value as that of x). If the occurrence of x is interpreted as "compute the field pointed at by x and take its contents," one could attach to $@x$ the meaning: "perform the operation described above, but suppress the extraction of the contents." But "@" could still not be used as unary operator, since $@(x+y)$ or even $@@x$ would be meaningless, whereas $.(x+y)$ and $..x$ both make sense.

Second, one of the major objectives of BLISS is to permit the programmer to define arbitrary representations of data structures (as will be discussed later). In order to accomplish this it is necessary to not only allow operations on pointers, but also to allow the value of an arbitrary expression to be interpreted as a pointer, i.e., a name. Since the semantic interpretation of identifiers is independent of context, and consequently so is that of all expressions, it is possible to do this in a consistent manner. A consistent interpretation is much more difficult in the case of context-dependent interpretations (since static context is inadequate in expressions involving function calls, for example). The authors feel strongly that this context-independent interpretation of identifiers simplifies systems programming. Experience in using the language for nontrivial programming tasks supports this point of view.

There are two additional declarations whose effect is to bind identifiers to names, but which do not create segments; examples are:

external s;
bind $y2 = y + 2,\ pa = p + .a$;

An *external* declaration binds one or more identifiers to the names represented by the same identifier declared **global** in another, separately compiled module. The **bind** declaration binds one or more identifiers to the value of an expression at block entry time. At least potentially the value of this expression may not be calculable until run time—as in "$pa = p + .a$" above. Examples of the use of **bind** will be found in subsequent sections.

2. Control

BLISS is an "expression language," that is, every executable construct, including those which manifest control, is an expression and computes a value. There are no statements in the sense of ALGOL or PL/1. Expressions may be concatenated with semicolons to form compound expressions, where the value of a compound expression is that of its last component expression. Thus ";" may be thought of as a dyadic operator whose value is simply that of its right-hand operand. Compound expressions have a similar appearance and function as a list of statements in ALGOL. The pair of symbols **begin** and **end** or left and right parentheses may be used to embrace such a compound expression and convert it into a simple expression. A block is merely a special case of this construction which happens to contain declarations; thus the value of a block is defined to be the value of its constituent compound expression.

The operator "$\leftarrow$" is a dyadic operator which should be read as "store into." More precisely, the expression "$\epsilon_1 \leftarrow \epsilon_2$" means: the (uninterpreted) bit pattern resulting from the evaluation of the expression ϵ_2 is to be stored into the field named by the pointer resulting from the evaluation of ϵ_1. In ALGOL the statement $x := x + 1$ causes the value of x to be incremented by one. Coupling the definition of "$\leftarrow$" given above with the interpretation of identifiers and the dot operator, the equivalent BLISS would be $x \leftarrow .x + 1$. The value of the assignment operator is defined to be identical to that of its right-hand operand; thus, the value of $x \leftarrow .x + 1$ is the incremented value of the cell named by x. The compound expression "$(y \leftarrow x; z \leftarrow ..y+1)$" causes a pointer to x to be stored into y, then computes the value of the field named by x (accessed indirectly through y) plus one and stores this value in z; in this case this value is also that of the compound expression. The important thing to remember about the assignment operation, e.g. $\epsilon_1 \leftarrow \epsilon_2$, is that it assigns a value to the field *named* by ϵ_1, not ϵ_1 itself.

There is the usual complement of arithmetic, logical, and relational operators. Logical operators operate on all bits of a field; relational operators yield a value 1 if the relation is satisfied, and 0 otherwise.

We will describe six forms of explicit control expressions: conditional, looping, case-select, function call, coroutine call, and escape. For this discussion it will be convenient to use the symbols ϵ or e, possibly subscripted, to represent arbitrary expressions.

The conditional expression is of the form "**if** ϵ_1 **then** ϵ_2 **else** ϵ_3" and is defined to evaluate, and have the value of, ϵ_2 just in case the rightmost bit of ϵ_1 is a 1 and evaluates, and has the value of, ϵ_3 otherwise. The abbreviated form "**if** ϵ_1 **then** ϵ_2" is considered to be identical to "**if** ϵ_1 **then** ϵ_2 **else** 0".

There are six basic forms of looping expressions:

while ϵ_1 **do** ϵ
do ϵ **while** ϵ_1
until ϵ_1 **do** ϵ
do ϵ **until** ϵ_1
incr ⟨*name*⟩ **from** ϵ_1 **to** ϵ_2 **by** ϵ_3 **do** ϵ
decr ⟨*name*⟩ **from** ϵ_1 **to** ϵ_2 **by** ϵ_3 **do** ϵ

Each form of looping expression implies repeated execution (possibly zero times) of the expression denoted ϵ until a specific condition is satisfied. In the first form (**while...do**) the expression ϵ is repeated so long as the rightmost bit of ϵ_1 remains 1. The second form is similar to the first except that ϵ is evaluated before ϵ_1, thus guaranteeing at least one execution of ϵ. The **until** forms are similar to the **while** forms except that the condition is negated. The last two forms are similar to the familiar "**step...until**" construct of ALGOL, except: (1) the control variable is local to ϵ; (2) ϵ_1, ϵ_2, and ϵ_3 are computed only once (before entry to the loop); and (3) the direction of the step is explicitly indicated (**incr**ement or **decr**ement). Except for the possibility of an escape expression within ϵ (see below), the value of a loop expression is uniformly taken to be -1.

We shall treat somewhat simplified versions of the **case** and **select** expressions here; these forms are:

case e **of set** ϵ_0; ϵ_1 ; ... ; ϵ_{n-1} ; ϵ_n **tes**
select e **of**
  nset ϵ_0 : ϵ_1; ϵ_2 : ϵ_3 ; ... ; ϵ_{2n} : ϵ_{2n+1} **tesn**

The value of a **case** expression is ϵ_e ; that is, the expression e is evaluated, and this value is used to select one of the expressions, ϵ_i $(0 \le i \le n)$, to evaluate. The value of ϵ_i becomes the value of the entire **case** expression. The value of a **case** expression is undefined if e is not in the range $0 \le e \le n$. The **select** expression is somewhat similar to the **case** expression, with the distinction that the value of e is not restricted to the range $0 \le e \le n$. Execution of the **select** proceeds as follows: (1) the value of e is computed; (2) the values of the expressions ϵ_{2i} $(0 \le i \le n)$ are evaluated; (3) for each i such that $e = \epsilon_2$, the expression ϵ_{2i+1} is evaluated. If there is no i such that $e = \epsilon_{2i}$, the value of the **select** expression is defined to be -1. In the event that one or more values of i exist such that $e = \epsilon_{2i}$, each of these expressions is evaluated in ascending order of the values of i; in this case the final value of the **select** expression is taken to be that of the last of these expressions to be evaluated.

The particular choice of -1 as the value of loop expressions and select expressions is almost but not entirely arbitrary. It might have been preferable to have them return a unique "undefined" or "nil" value, but no such value was available for the particular machine for which BLISS was implemented. The value -1 was chosen principally because it is marginally cheaper (in code produced) to generate this value and test the sign of a value in the PDP-10. Also, zero-relative indexing is common, and a negative value is clearly illegal in such contexts. Beyond these minor justifications the only important property of this choice is its uniformity.

It should be noted that the set of control expressions presented thus far is not minimal. All of the loop expressions could be constructed from the "**while...do**" form, and **case** and **select** expressions could be constructed from conditional expressions, for example. The decision to include a fairly rich collection of control structures in part resulted from another decision, to be discussed shortly, *not* to include the familar **go to** statement form of control. This decision suggests that the designers must pay far more attention to the range of control forms included, since there is no way for the user to synthesize his own control from the more primitive (**go to**) control form. In the case of BLISS two criteria were applied to determine whether a proposed control form should be included:

1. Was there a reasonable application for which the mode of expression without the proposed construct was awkward and/or obscure?
2. Was it possible to compile better code utilizing the additional information provided by the new construct than would have been possible otherwise?

All of the control expressions discussed above satisfy at least one and usually both of these criteria. The **select** expression, for example, both produces more efficient code and is a more natural, obvious mode of expression than the equivalent **case** or **if** formulation when the selection criteria involves noncontiguous values.

A function call expression has the form "ϵ (ϵ_1, ϵ_2, ... , ϵ_n)". This expression causes activation of the segment named by ϵ as a subprogram with an initialization of the formal parameters named in the declaration of the function to the values of the actual parameters $\epsilon_1, \ldots, \epsilon_n$. Only call-by-value (in the ALGOL sense) parameters are allowed; however, call-by-reference is implicitly available since names, pointer values, may be passed. The value of a function call is that resulting from execution of the body of the function. Thus, for example, the value of the following block is 120.

> **begin**
> **function** $factorial(n) =$
>   **if** $.n \leq 1$ **then** 1 **else** $.n * factorial(.n - 1)$;
> $factorial(5)$
> **end**

Note that a function call need not explicitly name a function by its associated identifier; all that is required is that ϵ evaluate to the name of a segment. Thus expressions such as the following are valid and useful:

> (**case** $.x$ **of set** $P1$; $P2$; $P3$ **tes**)(**.z**)

Note in this example that the occurrence of a parameter list enclosed in brackets triggers a function call. An identifier by itself merely denotes a pointer to the named segment; thus $P1$, $P2$, and $P3$ are the names of functions (not the result of executing them) and the value of the case expression is the name of one of these functions. The value of the entire expression above is the result of executing one of the functions $P1$, $P2$, or $P3$ with actual parameter $.z$. Function calls with no parameters are written "ϵ ()".

The body of any function may be activated as a coroutine and/or asynchronous process. An arbitrary number of distinct incarnations of a single body are allowed; indeed, arbitrarily many incarnations of a function body as both coroutines and subroutines may exist simultaneously. In order to permit any of several useful styles of coroutine mechanism, only two primitive operations are provided directly in the language:

> **create** ϵ (ϵ^1, ϵ^2, ... , ϵ^n) **at** ϵ_2 **length** ϵ_3 **then** ϵ_4
> **exchj** (ϵ_5, ϵ_6)

More complex coroutine call conventions can easily be constructed from these primitives. (Note that any of the expressions represented by ϵ's above may evaluate at execution time.)

The effect of the **create** expression is to establish a new, independent context (that is a stack) for the function named by ϵ with actual parameter values ϵ^1, ... , ϵ^n. The stack is set up beginning at the word named by ϵ_2 and is of size ϵ_3 words (to provide overflow protection). The activation point for the newly created coroutine is set to the head of the function named by ϵ. The value of the **create** expression is a "process name" for the new coroutine. Control then passes on to the expression following the **create**—in particular, the expression ϵ_4 is not executed at this time and the body of ϵ is not activated. When two or more such contexts have been established, control may be passed from the currently executing one to any other by executing an "exchange jump," **exchj**, expression. An expression "**exchj**(ϵ_5, ϵ_6)" will cause control to pass to the coroutine named by ϵ_5 (the value of an earlier **create** expression). The value ϵ_6 becomes the value of the **exchj** operation which last caused control to pass out of the coroutine named by ϵ_5. Thus, effectively, ϵ_6 may be passed as a parameter to the coroutine being reactivated.

The expression ϵ_4, one will note, is not executed at the time at which a coroutine is created. Instead this expression is executed only when and if control passes out of the body of the coroutine by a normal subroutine-type return (e.g. "falling through the end of its body"). The normal minimal action to be expected of ϵ_4 is to return the stack space used by the coroutine and to **exchj** to some other, active, coroutine. In such a case, a subroutine-type return from a coroutine corresponds to the coroutine killing its own existence.

The coroutine mechanism described above is illustrated by the following skeletal example. (The

exit operations in this example are instances of an "escape expression" (which is explained in the subsequent material). In this case, if (when) executed they will cause control to pass to the end of the block.)

```
begin
  own pa, pb, s1[100], s2[100];
  function a =
    begin local la, x;
      ⋮
      x ← exchj(.pb, la);
      ⋮
    end;
  function b(z) =
    begin local lb, y;
      ⋮
      y ← exchj(.z, lb);
      ⋮
    end;
  pa ← create a() at s1 length 100 then exit;
  pb ← create b(.pa) at s2 length 100 then exit;
  exchj(.pa, 0);
end
```

Execution of the main body of this block creates two coroutine contexts, one for the **function** a and one for b, and stores process names for these in pa and pb, respectively. The **function** b has one formal parameter whose value is initialized to $.pa$, i.e. to the process name of an incarnation of a. The main body then causes control to pass, via the **exchj**, to the coroutine named by $.pa$—that is, an incarnation of a, in this case. The activation point of both coroutines at this time is at the head of their bodies. At some point in the execution of a the execution of "$x \leftarrow$ **exchj**$(.pb,\ la)$" will cause control to pass to the coroutine named by $.pb$, leaving the activation point of a at the store operation. Similarly, at some later time the execution of "$y \leftarrow$ **exchj**$(.z,\ lb)$" will cause control to return to a (since $.z \equiv .pa \equiv$ a process name of a) at its activation point and leave the activation point of b at its store-operation. The value of the **exchj** operation in a is defined to be that of the parameter in the **exchj** operation which caused control to return to a; hence in this case a pointer to the local variable lb will be stored in x.

The familiar "**go to**...*label*" form of control has not been included in BLISS. There are two reasons for this: (1) unrestricted **go to**'s require considerable run-time support (principally due to the possibility of jumping out of functions and/or blocks), and (2) the authors feel strongly that the general **go to**, because of the implied violation of program structure, is a major villain in programs which are difficult to understand, modify, and debug. There are "good" and "bad" ways to use a **go to**, and there are restrictions which could be imposed which eliminate the need for run-time support. Consideration of the nature of "good" ways, and of the restrictions necessary to eliminate run-time overhead, led us to eliminate the **go to** altogether, and to the inclusion of a rich collection of conditional, looping, and case-select expressions. These alone, however, are not sufficiently general, or convenient, and consequently the "escape" expressions were introduced. There are eight forms of the escape mechanism, one for each form of control environment:

exitblock ϵ	**exitcase** ϵ
exitcompound ϵ	**exitselect** ϵ
exitloop ϵ	**exit** ϵ
exitset ϵ	**return** ϵ

Each escape expression causes control to exit from a specified control environment (a block, a loop, or a conditional expression, for example) *and* defines a value (ϵ) for that control expression (**exit** exits from any form of control expression, **return** exits from a function). Essentially the escape mechanism provides a highly structured form of forward branch which is awkward to obtain with the other control expressions.

Consider a linked list of two word cells, the first of which contains a link (pointer) to the next cell (the last cell has link = 0) and the second of which contains data. The following expression illustrates one use of an escape expression; the expression has a value which is the pointer to the first negative data item, or a value of -1 if no such item is found. The address of the head of the list is contained in a field called *head*.

```
(register t; t ← head; while (t ← ..t) ≠ 0 do
    if .(.t + l) < 0 then exitloop .t);
```

Note that the initialization of ($t \leftarrow head$) sets the value of t to a pointer to *head*, not the contents of *head*.

It is interesting to note that the decision to remove the **go to** from BLISS and the decision to make BLISS an expression language are closely related. The presence of the **go to** presents some awkward situations in the implementation of an expression language—for example,

go to L; ... ; $x \leftarrow a + b * (L : C - d)$; ...

or

$x \leftarrow a*$(**if** b **then** c **else go to** L); ...

With the **go to** eliminated from the language it becomes desirable to implement an expression rather than a statement oriented language. Part of the burden carried by the **go to** in conventional languages shifts to numeric values which control conditional, loop, case, or select expressions.

3. Data Structures

One of the outstanding characteristics of systems programs is their concern with the wide variety of data structures and schemes for representing these structures. Observation of what systems programmers *do* reveals that a large fraction of their design effort is spent in constructing representations for efficiently encoding the information to be processed. It is frequently the case that the most difficult task in making a modification to an existing program is that of representing the additional information required (e.g. the infamous "find another bit" problem). Consequently the issue of representation was one of the central design considerations in BLISS.

Two principles were followed in the design of the data structure facility of BLISS:

- the user must be able to specify the *accessing algorithm* for elements of a structure,
- the structure definition and the algorithms which operate on the elements of a structure must be separated in such a way that either can be modified without affecting the other.

The first principle is in accordance with the flexibility and efficiency the BLISS programmer should be provided with in utilizing the machine features. It expresses our strong feelings that we cannot—and should not try to—predict which kind of structures a system programmer will need, so that a given set of primitive structures and other statically defined structures is inadequate. The feature of a structure declaration, on the other hand, in which the user himself specifies the way in which elements are accessed, provides the user with the full flexibility and efficiency he needs. This point is illustrated below by taking as an example the well-known array structure. The difference with the static array structure of ALGOL is demonstrated by discussing several varieties of accessing an array.

In order to achieve a language in terms of which it is possible to write large systems that may be easily modified, it is imperative that the specifications of the representation of a data structure be separated from the specification of algorithms which manipulate data in that structure. This principle is severely violated in assembly languages where, typically, the code to access an element of a structure, for example, simply a contiguous field of bits within a word, is coded "in line" at the point where the element is needed. A comparatively trivial change which alters the size or position of the field may require locating and modifying all references to the field. This simple problem could be solved by following good coding practice and, perhaps, by the use of macros; not all changes are of such a trivial nature, however.

The concept of a "pointer" to a field (of bits within a word) was mentioned earlier. Actually, in BLISS a pointer is a 5-tuple consisting of: (1) a word address, (2) a field position, (3) a field size, (4) an (index) register name, and (5) an "indirect address" bit. These five quantities are encoded in a single word and as such are a manipulatable item in the language (a prerequisite of algorithmic representational specification). For simplicity, we discuss only the first three of these quantities; the reader is referred to the BLISS reference manual [6] for more detail. The word address, wa, field of a pointer designates the physical machine address of the word; the position, p, and size, s, designate a field within a word in terms of the number of bits to the right of and within the field. The notation used in BLISS to specify a pointer (taking only the simple wa, p, s case) is "$wa\langle p, s\rangle$".

Assume that the declaration "**own** $x[100]$" has been made. The identifier x is bound by this particular declaration to a pointer to the 36-bit field which is the first word of this 100-word segment. That is, the word address of the pointer x is that of the location allocated to the segment, and the position and size fields have values of 0 and 36, respectively. If we denote the address of the segment by α_x, then an occurrence of x in a BLISS program is identical to an occurrence of "$\alpha_x\langle 0, 36\rangle$". If ϵ_0, ϵ_1, and ϵ_2 are expressions, then the syntactic form "ϵ_0 (ϵ_1, ϵ_2)" is by definition a pointer whose word address is the value of ϵ_0 (modulo 2^{18}) and whose position and size specifications are the values of ϵ_1 and ϵ_2 (modulo 2^6) respectively. Thus "$X\langle 3, 4\rangle$" is a pointer to a four-bit field three bits from the right end of a word named X. The word address, position, and size information are encoded within a pointer in such a way that adding small integers to a pointer increments the word address. Thus "$X + 1$" is a pointer to the word following X (unless the address field overflows).

In order to satisfy the objectives set out above for data structures, it is desirable to extend the allocation declarations (**global**, **own**, **local**, etc.) described above. However, for exposition we shall first describe the structure mechanism in terms of the allocation declarations already available, then describe the extensions when more motivation is possible.

The definition of a class of structures, that is, of an accessing algorithm to be associated with certain specific data structures, may be made by a declaration of the form:

structure $\langle$name$\rangle$[$\langle$formal parameter list$\rangle$] $= \epsilon$

Particular names may then be associated with a structure class, that is with an accessing algorithm, by another declaration

map $\langle$name$\rangle$ $\langle$name list$\rangle$,

where a $\langle$name list$\rangle$ is a sequence of names separated by colons.

Consider the following example:

```
begin
  structure ary2[i, j] = (.ary2 + .i * 10 * .j);
  own x[100], y[100], z[100];
  map ary2 x:y:z;
  ⋮
  x[.a, .b] ← .y[.b, .a];
  ⋮
end;
```

In this example we introduce a very simple structure, $ary2$, for two-dimensional (10×10) arrays, declare three segments with names x, y, and z bound to them, and associate the structure class $ary2$ with these names. The syntactic forms $x[\epsilon_1, \epsilon_2]$ and $y[\epsilon_3, \epsilon_4]$ are valid within this block and denote evaluation of the accessing algorithm defined by the $ary2$-**structure** declaration (with an appropriate substitution of actual for formal parameters).

Although they are not implemented in this way, for purposes of exposition one may think of the structure declaration as defining a function with one more formal parameter than is explicitly mentioned. For example, the structure declaration in the previous example,

structure $ary2[i,\ j] = (.ary2 + .i * 10 + .j)$;

conceptually is identical to a function declaration

function $ary2(f0,\ f1,\ f2) = (.f0 + .f1 * 10 + .f2)$;

The expressions $x[.a, .b]$ and $y[.b, .a]$ correspond to calls on this function, i.e. to $ary2(x,\ .a,\ .b)$ and $ary2(y,\ .b,\ .a)$.

A **function** declaration such as that shown above implicitly declares identifiers and allocates storage for its formal parameters. These are functionally equivalent to those declared **local** (in scope and extent), and are initialized to the positionally equivalent actual parameter when the function is invoked. Consistent with the interpretation of identifiers, the value of a formal parameter identifier, say $f0$, is a pointer to the location allocated for the formal (on this, possibly recursive, invocation), and $.f0$ denotes its value.

Since, in a structure declaration, there is an implicit, unnamed formal parameter, the name of the structure class itself is used to denote this "zero-th" parameter. This convention maintains the positional correspondence of actuals and formals. Thus, in the example above, $.ary2$ denotes the value of the zero-th parameter, that is the name of the particular segment being referenced, and $x[.a,\ .b]$ is equivalent to $(x + .a * 10 + .b)$. The value of this expression is a pointer to the designated element of the segment named by x.

In the following example the structure facility and bind declaration have been used to efficiently encode a matrix product ($z_{i,j} = \sum_{k=0}^{9} x_{ik} y_{kj}$). In the inner block the names xr and yc are bound to pointers to the base of a specified row of x and column of y respectively. These identifiers are then associated with structure classes which allow one-dimensional access.

```
begin
  structure ary2[i, j] = (.ary2 + .i * 10 + .j);
        row[i] = (.row + .i);
        col[j] = (.col + .j * 10);
  own   x[100], y[100], z[100];
  map   ary2 x:y:z;
  ⋮
  incr i from 0 to 9 do
     begin bind xr = x[.i, 0], zr = z[.1, 0];
       map row xr:zr;
       incr j from 0 to 9 do
         begin
           register t; bind yc = y[0, .j]; map col yc;
           t ← 0;
           incr k from 0 to 9 do
              t ← .t + .xr[.k] * .yc[.k];
           zr[.j] ← .t;
         end;
```

```
        end;
      ⋮
    end
```

Suppose now that one wishes to alter the representation of the structure *ary*2, and access to the array is to be made through a dope vector to define the relative base of each row. The major change required is to replace the current structure declaration for *ary*2 by

```
own i1[10]; map row i1;
structure ary2[i, j] = (.ary2 + .i1[, i] + .j);
```

With this representation, the use of a special accessing algorithm (structure) for accessing columns becomes

```
structure col[j] = (.col + .i1[.j]);
```

As can be seen, these fairly simple changes in the program completely change its representation of the data. No changes in the processing algorithm are required.

4. More on Data Structures

The example above has the disadvantage of using the size of the array explicitly in the access algorithm, so that separate structure declarations would be required for arrays of different size. It should be possible for obvious reasons to parameterize the dependency of the size of the information onto which the structure is going to be mapped. The required flexibility is achieved by observing that until now it only makes sense to use "dotted formals" in the access algorithm, because BLISS has a strict value substitution of parameters. Thus, if we wish (and we do so wish), another interpretation can be placed on the occurrence of "undotted formals" in the access algorithm. In particular, we shall use the undotted formal parameter names to denote the value of parameters associated with particular instances of a structure (as distinct from instances of accesses to that structure).

Using "{" and "}" to embrace optional syntax (i.e. "zero or one instance of"), the BLISS **structure** declaration is of the form

$$\textbf{structure}\ \langle\text{name}\rangle[\langle\text{formal parameter list}\rangle] = \{[\epsilon_1\{,\epsilon_2\}]\}\epsilon_3$$

where ϵ_3 is the accessing algorithm as before, ϵ_1 is an expression whose value determines the size (in words) of an instance of this structure, and ϵ_2 is an expression whose value is the name of a user defined dynamic allocation function. Any of the expressions ϵ_1, ϵ_2, or ϵ_3 (but especially ϵ_1 and ϵ_3) may involve undotted formals and thus be instance specific. Consider the following example, which also illustrates the extension to the allocation declarations:

```
begin
  structure ary2[i, j] = [i * j](.ary2 + .i * j + .j);
  own ary2 x:y:z[10, 10];
    ⋮
end;
```

This is essentially the same example as has already been presented of three, two-dimensional, 10×10 arrays. However, the information previously contained in a **map** declaration has now been included in the allocation declaration. More importantly, note that undotted formal names, which correspond to the "instance actuals: 10,10" are used to compute the size of the instances—as well as in the accessing algorithm itself. (As in ALGOL declarations, the instance-actuals distribute over the names to their left.) Thus the single structure declaration, *ary*2, may be used for other instances of similarly structured segments which happen to be of a size other than 10 × 10.

The form of allocation and structure declarations illustrated previously are instances of the extended syntax in which the obvious defaults are chosen.

Dynamic allocation, of an admittedly simple kind, is illustrated by the following example:

```
begin
  own space[10000]; own spaceptr; external error;
  function locspacemgr(tog, numb, base) =
    if .tog
    then
      begin
       if (spaceptr ← .spaceptr+.numb) > 10000 then
          return error();
       space + .spaceptr − .numb;
      end
    else (if (space + .spaceptr) > .base then
          spaceptr ← .base − space);
  structure lary[i, j] =
         [i * j, locspacemgr](.lary + .i * j + .j);
  local lary x[.n, .m + 1], y[3, 5];
    ⋮
end
```

The local allocation declaration in this example, **local** *lary x*..., contains expressions which must be evaluated at run-time as instance-actuals; hence dynamic allocation is required. Note that the structure declaration for *lary* contains the name of the function *locspacemgr* in the position denoted ϵ_2 in the extended structure declaration syntax. In fact the value of ϵ_2 is required to be that of a function with three formal parameters, say *pl*, $p2$, and $p3$. This function is automatically called (possibly several times) at entry and exit from a block containing an allocation declaration which specifies a structure declaration which mentions it. The interpretation of the parameters is:

$p1 = 1$ The function is to allocate a segment of size $p2$ (words), and return a pointer to this segment. $p2$ will be the value of the size expression in the structure declaration. $p3$ has no meaning.

$p1 = 0$ The function is to deallocate a segment of $p2$ words whose beginning is pointed to by $p3$ ($p3$ is the value returned by a previous call with $p1 = 1$). The value of the function in this case is immaterial.

Thus, in the example instances of structures of type *lary* are allocated from the segment named *space*. Examination of the function *locspacemgr* reveals that it allocates on a last-in-first-out basis, and hence this particular allocation function is only suitable for local (stack discipline) variables. It should also be noted that the example above is similar to the following block, which uses only the simpler declarations of the previous section. (Note that the example is not completely equivalent due to possible identifier conflicts.)

```
begin
  own space[10000]; own spaceptr;
  function locspacemgr(tog, numb, base) =
     % same as above %
  bind i1 = .n, j1 = .m + 1;
  structure lary1[i, j] = (.lary1 + .i * j1 + .j);
  bind i2 = 3, j2 = 5;
  structure lary2[i, j] = (.lary2 + .i * j2 + .j);
  bind x = locspacemgr(1, i1 * j1, 0); map lary1 x;
  bind y = locspacemgr(1, i2 * j2, 0); map lary2 y;
  ⋮
  locspacemgr(0, i1 * j1, x);
  locspacemgr(0, i2 * j2, y);
end
```

This example illustrates that the extended declarations introduce no additional power; however, the extensions do permit considerable simplification and clarity.

The requirement that the programmer provide his own dynamic allocation function was introduced principally to avoid prerequisite run-time support. However, the effect is that the user may define allocation policies particularly appropriate to his own application. Indeed, one might expect different allocation policies to be associated with different structures, or even different instances of the same structure, in a single program.

Conclusions

We have attempted to present above the main features of BLISS, a language we feel especially suited to that application area usually called "systems programming." At least one possible interpretation of this description is as an indirect definition of the system programming "problem area." In the simplest case, this manifests itself in our break with the traditional interpretation of identifiers in higher-level languages as the consequent demand on the programmer to be consciously aware of the distinction between name and value. Similarly, the structure mechanism may be interpreted as a statement of our judgement as to the extreme importance of the representation, modification, allocation issues in systems programming—and hence that these issues must be explicitly at the programmer's attention and control. The decision to exclude the **go to** statement is similarly a consequence of our judgement as to the importance of writing highly structured programs so that they may be read, understood, and modified.

Considerable experience has been gained in the use of BLISS for writing systems. The BLISS compiler itself, an APL system, a WATFOR-like fast FORTRAN compiler, SIMULA-like event based simulation system, chess playing programs, input-output routines, debugging aids, parts of an operating system, and so on, have been written in BLISS. This represents on the order of 100,000 lines of code, and forms a reasonable base for forming some conclusions about the language. By such measures as readability and modifiability, lines of (debugged) code produced per programmer per day, quality of code produced by the compiler, and user reaction, the language has been a success.

Of more interest, perhaps, are the things which we have learned which will cause us to extend the language or do things differently if we were to do them over again. We mention three, one of which is currently being implemented.

1. The implementation presumes a stack which is used for parameters to functions, return links, and local variables. The user does not have direct control over the implicit structure. Since the implementation is quite efficient, there is little reason for the user to override it so long as his entire

system is written in BLISS. However, if one wishes to use BLISS to rewrite parts of an existing system (for example, we are rewriting parts of the PDP-10 operating system), one finds that conflicts in parameter passing conventions, register conventions, etc., arise.

A partial solution to these problems is implemented in the current compiler in that the user may control the compiler's register allocation policy. Another partial solution (not yet implemented) would permit a *structure* to be associated with a function to specify how parameters are accessed. The most difficult aspect of the problem is to devise a means by which the user may specify the schema for generation of prologue and epilogue code.

2. The "escape" mechanism is essential in the context of a "**go to**-less" language. It is unfortunate that we have eight separate operations (**exitloop**, **exitblock**, etc.), all of which perform essentially the same function. Our mistake was in assuming that there is no need for a label once the **go to** is removed. It would have been better to permit a control environment (a block, a loop, or whatever) to be labeled, and to use a single operation, say "**leave** ⟨label⟩ ϵ", to cover all of these cases. A simple extension to this notion, "**leave** ⟨function name⟩ ϵ", could cause an exit from several (nested) calls.

3. An assumption made by advocates of implementation languages, including the authors, is that systems written in a higher-level language will be ultimately more efficient than those produced in assembly language. The reasoning behind this assumption is simply that the cost of redesigning and recoding "critical" portions of a system is smaller when it is written in a higher-level language than when it is written in assembly code, and hence it is more practical to polish the final product. The correctness of this assumption depends upon knowing which portions of the system are the "critical" ones. Experience indicates that our intuition about such things is poor.

It might be argued that programmers should build mechanisms into their system to measure its efficiency—indeed the same argument may be made for built-in debugging tools. In practice, however, given the decision of how to expend today's effort, a programmer will usually opt for pushing the main line of the project rather than building these support tools. On the other hand, he will use such tools if they already exist. One such tool installed in the BLISS compiler will, under control of a compile time toggle, cause a user-defined function to be called on each entry (and exit) from a control environment (block, conditional, loop, function, etc.). Parameters to this function specify the (source) line number, type of control environment, etc. The user may, of course, do whatever he pleases in this function; however, standard functions have been written which count the frequency of execution of the various expressions and accumulate the time spent in these expressions. Very simple analysis of the data collected by these routines can so helpful in determining when further effort will be fruitful.

Acknowledgments. We would like to express our deep gratitude to Messrs. Geschke, Wile, and Apperson (graduate students at Carnegie-Mellon University), each of whom has made valuable contributions to both the design and implementation of the language.

References

1. EPL reference manual, Project MAC, April 1966.
2. Burroughs B5500 Extended ALGOL reference manual. Burroughs Corp., Detroit, Mich.
3. Wirth, N. "PL/360, A programming language for the 360 computers." *J. ACM 15*, 1 (Jan. 1968), 37–74.
4. Richards, M. "BCPL: A tool for compiler writing and system programming." Proc. AFIPS 1969 SJCC, Vol. 34, AFIPS Press, Montvale, N.J., pp. 557–566.
5. Naur, P., and Randell, B. (Eds.) "Software engineering." Scientific Affairs Div., NATO, Brussels, Belgium (Conference held in Jan. 1969 in Garmish).
6. BLISS reference manual. Computer Science Dept. Rep., Carnegie-Mellon University, Pittsburgh, Pa., Jan. 15, 197O.
7. PDP-10 reference handbook. Digital Equipment Corporation, Maynard, MA, 197O.
8. Lang, Charles A. "SAL—Systems Assembly Language." Proc. AFIPS 1969 SJCC, Vol. 34, AFIPS Press, Montvale, N.J., pp. 543–555.
9. Bell, J. "The quadratic quotient method: A hash code eliminating secondary clustering." *Comm. ACM 13*, 2 (Feb. 1970), pp. 107–109.

■