SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2002; **32**:955–981 (DOI: 10.1002/spe.470)

The BLISS programming language: a history

Ronald F. Brender*,†

Hewlett-Packard Company, ZKO2-3/N30, 110 Spit Brook Road, Nashua, NH 03062, U.S.A.

SUMMARY

The BLISS programming language was invented by William A. Wulf and others at Carnegie-Mellon University in 1969, originally for the DEC PDP-10. BLISS-10 caught the interest of Ronald F. Brender of DEC (Digital Equipment Corporation). After several years of collaboration, including the creation of BLISS-11 for the PDP-11, BLISS was adopted as DEC's implementation language for use on its new line of VAX computers in 1975. DEC developed a completely new generation of BLISSs for the VAX, PDP-10 and PDP-11, which became widely used at DEC during the 1970s and 1980s. With the creation of the Alpha architecture in the early 1990s, BLISS was extended again, in both 32- and 64-bit flavors. BLISS support for the Intel IA-32 architecture was introduced in 1995 and IA-64 support is now in progress.

BLISS has a number of unusual characteristics: it is typeless, requires use of an explicit *contents of* operator (written as a period or 'dot'), takes an algorithmic approach to data structure definition, has no `goto`, is an expression language, and has an unusually rich compile-time language.

This paper reviews the evolution and use of BLISS over its three decade lifetime. Emphasis is on how the language evolved to facilitate portable programming while retaining its initial highly machine-specific character. Finally, the success of its characteristics are assessed. **Copyright © 2002 John Wiley & Sons, Ltd.**

KEY WORDS: BLISS; history; machine-oriented language; portable programming; system implementation language

INTRODUCTION

In 1968, Wulf joined the Computer Science Department at Carnegie-Mellon University (CMU), after earning a PhD degree at the University of Virginia. Wulf wanted to write an operating system, among other things, but first he wanted a high-level language in which to write it. When the

*Correspondence to: Ronald F. Brender, Hewlett-Packard Company, ZKO2-3/N30, 110 Spit Brook Road, Nashua, NH 03062, U.S.A.
†E-mail: ron.brender@hp.com

Copyright © 2002 John Wiley & Sons, Ltd.

Received 15 August 2000
Revised 15 March 2002
Accepted 16 March 2002

Department acquired a DEC PDP-10, Wulf and others set about designing and implementing a systems programming language for it, which came to be known as BLISS[‡].

In early 1970, Brender joined the newly formed Research Department within Software Engineering at Digital Equipment Corporation (DEC), after earning a PhD degree at the University of Michigan. His initial focus was evaluation of possible implementation languages for DEC's computer lines, certainly including the PDP-10 and hopefully also the 18-bit PDP-9 and PDP-15 series and the newly introduced 16-bit PDP-11.

Later in 1970, Brender traveled to CMU to meet Wulf and learn more about BLISS. Gradually their mutual interests turned into a multiple year collaboration and initiated a line of BLISS language and compiler development that has now spanned three decades.

This history is divided primarily into epochs which correspond to the organizational context in which BLISS language and compiler development took place, the technical challenges that motivated BLISS language evolution, or both.

THE CMU YEARS: 1969–1974

In the late 1960s and early 1970s, the use of high-level languages for systems programming was highly controversial. Most systems had very limited amounts of main memory; indeed, even so-called large systems generally had hardware maximum address space limitations of 16 or 18-bits of addressable memory—and no virtual memory support either. 'Size is the goal' for both code and data often summarized one side of the debate. Systems programmers wanted lots of control over every instruction and every cycle and every bit of memory used.

The advocates for using high-level languages cited the potential for improved productivity and reduced error rates through the use of structured programming and data abstraction. However, the languages that seemed expressive enough for the job at the time, languages such as PL/I or Algol 68, were not viewed as suitably efficient. Arguments raged over whether it was even conceivable, let alone practical, for compilers to produce code that was within some small incremental percentage of what could be achieved by good programmers. *No one* ever picked zero.

Lots of experimentation was in progress throughout the computer industry seeking languages with the right blend of high-level features, low-level control, ease of compiling to efficient code, and improved optimization technology. The few, then contemporary, languages that fit these goals according to the earliest paper on BLISS [1] were EPL [2] ('Early PL/I', a bootstrapping PL/I subset used for Multics [3]), Burroughs Extended Algol [4], PL360 [5] and BCPL [6,7].

In designing BLISS, Wulf [8] set forth these objectives and constraints for the language.

- It was targeted toward system programs, that is, programs where the application domain is the computer as such.
- It assumed knowledgeable users with respect to training and judgment.

[‡]Over the years, Wulf has steadfastly held that BLISS is not and never was an acronym. Undaunted, colleagues have offered suggestions to fill that gap. Perhaps most commonly reported is 'Basic Language for the Implementation of System Software'. Others have suggested 'Beautiful Language . . . ' or, perhaps inspired by the example of JOVIAL, 'Bill's Language . . . '.

Copyright © 2002 John Wiley & Sons, Ltd.

- It must provide high efficiency, with little or no required run-time support or distributed overhead, none of which is a substitute for programmer choice of good algorithms.
- It was expected that programs would be maintained, modified and enhanced over long periods of time by a succession of developers (the 'turn-over' problem as it was called).
- It must cater to and support good structured programming concepts and practices.

What makes BLISS interesting

BLISS has a number of characteristics that set it apart from most then contemporary and subsequent programming languages: it is typeless, requires use of an explicit *contents of* operator (written as a period or 'dot'), takes an algorithmic approach to data structure definition, has no `goto`, is an expression language, and has an unusually rich compile-time language.

In the sections that follow, examples are presented using the 'second generation' syntax that was developed at DEC during the mid-1970s, even when describing the original BLISS-10 and BLISS-11 languages.

BLISS is a *typeless* language; that is, all data is manipulated in terms of the underlying machine *word* sized units. The word size is usually the same as the integer or general purpose register size, which may be a multiple of the addressable unit size.

On the PDP-10, memory and registers consisted of 36-bit words.

A word can represent anything: an integer, an address, a floating point value, a character, a packed structure, whatever. The choice of interpretation is determined by the operation that is applied to that data and different operators can be applied to the same data at different times. In this regard, BLISS is much like BCPL [6], a contemporary of BLISS in origin.

In BLISS, the value of the name of a variable is the address of that variable—sometimes called its *lvalue*—regardless of whether the name occurs on the left- or right-hand side of an assignment. To fetch the contents of a variable, an explicit unary *contents of* operator must be applied. For example,

```
X = .Y + 2;
```

fetches the contents of variable Y, adds 2, and stores the result at address X. (Hereafter we will use the term *data segment* rather than variable for consistency with DEC's BLISS terminology.)

This 'context independent' evaluation of names makes it easy to interpret expressions that occur on either side of an assignment. For example, in

```
X+.I = .(X+.I) + 2
```

both occurrences of `X + .I` are interpreted identically: add the contents of `I` to the address of X, and use the resulting value as specified by the containing operators. Note, in particular, that the dot operator is applied again to the second occurrence of that expression, to fetch the contents at the computed address.

Of course, `X + .I` is easily recognized as an example of indexing, which leads naturally into the topic of structures. However, first we complicate matters to understand partial word accesses.

The DEC PDP-10 [9] was a 36-bit word addressable machine with an 18-bit address space. It included Load Byte (LDB) and Deposit Byte (DPB) instructions for accessing that part of a word (not crossing word boundaries) characterized by a position P, the number of bits from the low-order end of the word, and a size S, the number of bits.

Copyright © 2002 John Wiley & Sons, Ltd.

```
structure   VECTOR[I; N] = [N] VECTOR + I - 1;  ! 1-origin vector

global      J, A : VECTOR[10];

J = 2;
A[.J] = .A[.J+1] + 3;
!
! where the last assignment is equivalent to
!
! (A + (.J) - 1) = .(A + (.J+1) - 1) + 3;
```

Figure 1. BLISS structure example.

```
structure   UPPERDIAG[I, J; M] =       ! M x M upper diagonal matrix
                [M*(M-1)/2]            ! Store only upper diagonal
                UPPERDIAG + (J-2)*(J-1)/2 + I - 1;

own         B : UPPERDIAG[100];        ! Declare and allocate B

B[.K, 5] = .K[.K+1, .L];
!
! where the assignment is equivalent to
!
! T' = .K+1;
! ((B(.K-2)*(.K-1)/2+5 - 1) = .((B+(.T'-2)*(.T'-2)/2 + .L -1);
```

Figure 2. BLISS upper diagonal structure example.

To perform such accesses, the LDB and DPB instructions operated indirectly through an operand called a *byte pointer* which packed an 18-bit address (Y), a 4-bit index register name (X), a 1-bit indirection indicator (I), a 6-bit position value (P), and a 6-bit size value (S) into a 36-bit word size value.

BLISS-10 adopted the notation `Y<p, s, x, i>` to express a byte pointer. By default, the *undotted name* `Y` was equivalent to `Y<0, 36, 0, 0>`, meaning the 'byte' beginning at the lowest end of the word and extending for 36 bits without index modification of address Y (because register 0 did not do indexing) or indirection; that is, the entire word.

To understand the BLISS approach to data structures, consider Figure 1.

The structure declaration defines `VECTOR` to be a 1-origin vector (as in FORTRAN) rather than a 0-origin vector (as in C). This structure has two kinds of formal parameters: the *access formals*, which precede the semicolon, and the *allocation formals*, which follow the semicolon.

After the equals sign (here meaning 'is defined as') is an expression in square brackets called the *allocation expression*. This expression is used to compute the number of addressable storage units to allocate for a data item declared with that structure attribute. For example, the data segment A is declared with a `VECTOR` structure giving an *allocation actual* value of 10—with the result that 10 words of storage are allocated for A and the name A corresponds to the first of them.

Copyright © 2002 John Wiley & Sons, Ltd.

```
structure   REC[O, P, S; N] = [N] (REC + O)<P, S>;

macro
    Word1LowHalf = 0,  0, 18 %,
    Word1Flag1   = 0, 18,  1 %,
    Word2Full    = 1,  0, 36 %;

own         DATUM : REC[2];

DATUM[Word1LowHalf] = -1;
DATUM[Word1Flag1]   =  0;
DATUM[Word2Full]    =  1;
!
! where the assignments are equivalent to
!
!(DATUM + 0)<0, 18> = -1;
!(DATUM + 0)<18, 1> =  0;
!(DATUM + 1)<0, 36> =  1;
```

Figure 3. BLISS record-like structure example.

Following the allocation expression is the *access expression*, which is used when an element of the structure is accessed using the `[]` operator. The structure name itself acts as a kind of implicit formal that corresponds to the actual data segment. Thus the access `A[.J]` is semantically equivalent to `(A+.J-1)`—add the contents of J to the address of A and subtract one.

The expressive power of structures begins to emerge when considering an example such as shown in Figure 2, which declares and accesses an upper diagonal matrix. Note, in particular, that the matrix is both declared and accessed just like an ordinary square matrix would be, without having to expose its internal organization.

It is less obvious how to declare and access non-algorithmic, record-like data segments. This can be accomplished by exploiting the byte pointer notation as part of a structure declaration, as illustrated in Figure 3. Note that macro names are declared that define the word offset (`O`) and position and size within that word (`P` and `S`) for the component fields of a particular kind of record. These macros can then be used in structure accesses to provide a more mnemonic, record-like access notation.

The map and bind declarations provide the ability to exploit the untyped nature of memory.

The map declaration associates a new set of attributes with an already declared name. For example, suppose there were an inner scope where it was desirable to explicitly take advantage of the actual layout of the upper diagonal matrix B declared in the example of Figure 2. Such a scope might look like

```
begin
map B : VECTOR[100*99/2];       ! Same size as above
... B[.L] ...
end;
```

Within this block, the data segment B is accessed using the VECTOR structure, not the original UPPERDIAG structure.

Copyright © 2002 John Wiley & Sons, Ltd.

```
! Some data segment
!
own          RTN : vector[1000];

! Store instructions in data area
!
RTN[.I] = ...;

! Call as a routine to cause execution
!
RTN();
```

Figure 4. A data segment as both code and data.

The bind declaration allows associating a name and attributes with a dynamically computed address. For example,

```
bind     BD = MALLOC(100) : VECTOR[];
```

declares a new name BD and associates it with the value returned by calling the routine MALLOC. The value of the undotted name is defined to be the function result, which in this case is thereafter intended to be used as the address of a vector. Because the structure attribute is not used to allocate storage, it does not include allocation actual parameters.

Returning to Figure 2, an alternative to the above map declaration might be

```
bind     BV = B : VECTOR[...];
```

which allows the *same* storage to be viewed simultaneously as both an upper diagonal matrix and a vector. Such aliasing presents obvious challenges to code optimization; however, discussion of this is beyond the scope of this paper.

The same data segment can even sometimes be treated as data and sometimes executed as code, as illustrated in Figure 4.

Here the occurrence of the data segment name on the left-hand side of the assignment indicates that a storage assignment operation is to be performed, while later the `()` syntax indicates that a routine call operation is to be performed to that same data segment. This capability was later exploited to great advantage by DEC's APL-10 product which would create special case code on the fly for many APL operators and then call that code to evaluate the operation.

BLISS was the first commercial quality procedural language to heed Dijkstra's classic injunction about the harm of the `goto` [10] and not have one. Only recently has another major language, Java [11], followed suit.

BLISS did include an unusually rich set of control structures. These included simple loops with either pre- or post-loop tests (`while-do` and `do-while`) as well as complement tests (`until-do` and `do-until`). There were counted style loops that counted up (`incr-from-to-by-do`) or down (`decr-from-to-by-do`), and there were multiway branches using dispatch tables (`case-from-to-of-set-tes`) or repeated comparison (`select-of-set-tes`). Examples of these are shown in Figure 5.

Copyright © 2002 John Wiley & Sons, Ltd.

```
! Counted loop (control variable is local to loop body)
!
decr J from .I to 0 by 2 do
    begin
    local   ABS;
    if .ARR[.J] gtr 0 then ABS = .ARR[.J] else ABS = -.ARR[.J];
    ABSSUM = .ABSSUM + .ABS;
    end;

! Multiway decision using a dispatch table
!
case F(.PTR) from 1 to 100 of
    set
    [1] :         ... ! F(.PTR) equals 1
    [3, 5, 7] :   ... ! F(.PTR) equals 3 or 5 or 7
    [inrange] :   ... ! from 1 to 100 not otherwise handled
    [outrange] : ... ! outside range 1 to 100
    tes;

! Assign last expression of block (.Y + .Z) to X
!
X = begin Y = ...; Z = ...; .Y + .Z end;

! Assign expression of executed alternative to M
!
MAX = (if .A gtr .B then .A else .B);

! Assign value of exitloop if executed, or -1 if
! loop terminates, to CNT
!
P = .HEAD;
CNT = while .P neq 0 do
    begin
    if .(.P+4) eql 5 then exitloop .P+10;
    P = ..P;
    end;
```

Figure 5. Control flow and expression value examples.

A set of exit constructs allowed premature termination of the innermost containing blocks or control constructs (which we discuss later).

BLISS is an *expression language*; that is, all of the control constructs are defined to yield usable values. The examples in Figure 5 are a bit contrived, but they illustrate the idea.

In the last example, `exitloop` (which exits the innermost containing loop) supplies the value of the `while` loop expression if it executes and the value is otherwise -1 if the loop runs to completion. (Minus one is an arbitrary but usually acceptable default value.)

The PDP-10 instruction set encoding was so simple and regular that nearly any instruction could be invoked using a special function call; all that was required was to declare a suitable name and the value

Copyright © 2002 John Wiley & Sons, Ltd.

of the opcode to use in a machop declaration. By this means, the entire PDP-10 instruction set became directly usable without needing to resort to an assembler tool[§].

BLISS-10 implementation and use

At the time Brender first met Wulf in 1970, the BLISS compiler for the PDP-10 was well along, though hardly complete. It had been successfully bootstrapped into itself. While structures were used in its writing, these were not yet implemented; rather, structure references were implemented using a separate preprocessor.

The compiler was (mostly) single pass, used recursive descent parsing and generated object code 'on the fly'. The code was not immediately emitted, however; rather it was buffered and subjected to various post-passes. The resulting code quality was reasonably good, but still allowed many opportunities for improvement. This compiler structure reflected the view that global optimization was part of the programmer's job of algorithm selection while best possible local code quality was the province of the compiler.

BLISS easily attracted a following at CMU. It was used for SCRIBE (an early document formatting utility), PQCC (in the Production Quality Compiler Compiler research project), a WATFOR-like compiler, as well as a variety of other departmental and personal programming efforts.

As the BLISS compiler stabilized, Brender turned his attention toward introducing its use at DEC. Fortunately, a new optimizing FORTRAN compiler development project (known as FORTRAN-10) was starting at a good time to use BLISS-10. As compiler and language oriented people themselves, the development team was eager to use a high-level language for their work, as well as being perhaps less risk averse than others regarding the strange new language from CMU. The FORTRAN-10 group plunged into using BLISS—and never looked back.

However, the PDP-10 was a popular and highly-successful system used by timesharing service bureaus of that period; many of those customers very much wanted the new higher performance FORTRAN and did not want to risk it being implemented using some experimental University language. Before FORTRAN-10 was completed, the twice annual DECUS (Digital Equipment Corporation User's Society) meetings regularly sparkled with debate over the use of BLISS. At one especially memorable session, many of the critics showed up wearing large lapel buttons that read 'BLISS is ignorance'!

Fortunately, FORTRAN-10 did meet its goals and BLISS-10 was deemed to have contributed to that success. BLISS became regularly used for new projects in the Large Computer Group and other parts of DEC.

BLISS for the PDP-11

Even as BLISS for the PDP-10 was becoming established, attention began to focus on creating a version of BLISS for the PDP-11, which became known as BLISS-11.

[§]Such a general, low-level mechanism could hardly go unabused. Several colleagues [12,13] have reported knowing of or being themselves involved in attempts to use `machop` together with the PDP-10 `JUMPA` or `JRST` instructions to create a 'fake' `goto` in BLISS—'just for fun', of course.

Copyright © 2002 John Wiley & Sons, Ltd.

Table I. Summary of key hardware characteristics.

Property	PDP-10	PDP-11	VAX-11	Alpha
Word size	36 bits	16 bits	32 bits	64 bits
Addressable unit	36-bit word	8-bit byte	8-bit byte	8-bit byte
Byte pointer	Address<p, s, x, i>	Address only	Address only	Address only
Byte access size	Up to 36 bits	8-bit only	Up to 32 bits	8/16/32/64-bit only
Byte access sign	Zero extend	Sign extend	Sign or zero extend	Zero extend
General registers	16	8	16	32
Floating point	General registers	Not available	General registers	Separate registers

The differences between BLISS-11 and BLISS-10 are generally of two kinds: those that result from differences in the underlying hardware and general language improvements.

The most important hardware differences that influenced BLISS between the PDP-11 and the PDP-10 are summarized in Table I.

The word size (36 versus 16), addressable unit size (36 versus 8) and byte pointer (arbitrary versus 8-bit only) differences together meant that the BLISS-10 notion of a byte pointer could not be retained in BLISS-11: a 16-bit word could hold an address but no more. To accommodate this, the BLISS-10 `<>` operation was reformulated to become an optional part of the fetch and assignment syntax.

For example, in BLISS-11

```
Y<1, 1> = .X<0, 1>;
```

means the same as in BLISS-10: fetch the low-order bit from location X and store it in the next to low bit of location Y. However,

```
Y = X<3, 4>;
```

is not valid in BLISS-11.

To take advantage of byte addressability on the PDP-11 and allow convenient use of small data segments, BLISS-11 added the `BYTE` and `WORD` attributes which could be given on a data segment declaration; the default is `WORD`. These *allocation attributes* implied a default position and size of `<0, 8>` and `<0, 16>`, respectively, if no explicit specification was given. Thus, given

```
global      P : BYTE,       Q : WORD,       R;
```

the following pairs are equivalent

```
Q = .P;   and   Q = .P<0, 8>;
P = .Q;   and   P<0, 8> = .Q;
R = .Q;   and   R<0, 16> = .Q<0, 16>;
```

Copyright © 2002 John Wiley & Sons, Ltd.

A more subtle difference arose because the PDP-11 `MOVB` (move byte) instruction performed a sign extension when copying an 8-bit byte from memory into one of the 16-bit registers. It cost an extra two-word instruction to clear the high-order bits if zero extension was wanted. We were very concerned that defining the BLISS-11 fetch operation to always zero extend to match BLISS-10 would generally be too expensive, especially for cases where it was known *a priori* that possible values were limited in range to less than 128. However, neither could we justify sign extending subword sized fields of other than 8 bits. Ultimately, we retained zero-extension for all subword fetches and left it as a challenge to the BLISS-11 compiler to minimize the cost.

Address comparisons on the PDP-10 were always performed using signed relational instructions—the limited (18-bit) range of address values made signed and unsigned comparison of addresses equivalent in practice. This was not the case on the PDP-11, where the word size and address size were both 16-bits.

For BLISS-11, we added unsigned versions of the relational operators, naming them after their BLISS-10 counterparts but with a suffix '`U`' appended (`EQLU`, `LEQU`, `LSSU`, and so on).

The PDP-10 supported single precision floating point with values held in the general registers; BLISS-10 provided binary infix operators FADR (add), FSBR (subtract), FMPR (multiply) and FDIV (divide), as well as the unary prefix operator FNEG (negate) to allow use of that capability.

The early PDP-11s provided no floating point hardware support. Even when floating point hardware options were later introduced, the floating point (32- and 64-bit) values did not fit in the 16-bit BLISS model and used a set of registers separate from the general purpose set. Because they did not fit the BLISS model, they were left out of BLISS-11.

Significant language improvements were made in a number of areas, including most importantly: macros, labels and exit forms.

The macro facility of BLISS-10 was very simple and very limited. It allowed substitution of actual arguments for formal parameters during macro expansion, and little else. Because it was character stream oriented, various kinds of lexical oddities tended to interfere with routine use.

For BLISS-11, a completely new and much more functional lexeme-based macro mechanism was introduced. There were three kinds of macros: simple macros, which provided straightforward expansion; conditional macros, which allowed a limited kind of conditional expansion based on whether an argument was present; and iterative macros, which expanded repeatedly based on the arguments present in a call. Simple macros require no further description. Conditional macros proved too limited and were used very little. Iterative macros proved enormously useful and are illustrated in Figure 6.

Along with the macro mechanisms came a new notion, *lexical functions*, which are completely evaluated during compilation. In the case of those illustrated in Figure 6, these functions provide access to characteristics of the macro call and/or its expansion. Other lexical functions such as `%name` and `%string`, which formed a name or a string, respectively, out of concatenation of their arguments, were also introduced. (This foreshadowed an explosion of compile-time lexical mechanisms in later BLISSs).

Because BLISS-10 omitted the `goto` statement, it seemed sensible then that it should also omit labels. It did and that was later seen as a mistake.

The utility of an early escape from a loop is widely appreciated—witness the loop exiting constructs of C, C++ (for example, `break`) and numerous other languages. BLISS-10 carried this to the extreme. The general form was:

escape-keyword [*level*] *expression*

Copyright © 2002 John Wiley & Sons, Ltd.

An iterated macro expands repeatedly, using the same actuals each time for the fixed formal parameters (declared within parentheses) and successive actual parameters corresponding to the iterated formals (declared in square brackets) until there are too few unused actuals to match the list of iterated formals. Recursion is allowed.

Example Declarations:

```
macro   LOAD_VEC(NAME)[VALUE] = NAME[%count] = VALUE %, ! no final ;
        INIT_LOCAL_VEC(NAME)[] =
            local NAME : VECTOR[%length-1];
            LOAD_VEC(NAME, %remaining); %;
```

where

- `%count` is a lexical function whose expansion is the number of expansions of the containing macro that have so far completed.
- `%length` is a lexical function whose expansion is the number of actual arguments in the call.

Example Call:

```
INIT_LOCAL_VEC(TABLE2, 1, 5, 9-5);
```

which expands to

```
local   TABLE2 : VECTOR[3];
TABLE2[0] = 1;
TABLE2[1] = 5;
TABLE2[2] = 9-5;
```

Figure 6. BLISS-11 iterated macros.

The escape-keywords included: `exitblock`, `exitcompound`, `exitloop`, `exitcond` (exit one arm of a conditional, e.g., `if-then-else`), `exit` (any control scope), `exitcase`, `exitset` (exit one arm of a case for one value of the selector), and `exitselect`—literally one for every possible kind of enclosing control construct in the language! The optional level, whose enclosing brackets distinguished it from the optional expression, allowed multiple levels of the given construct to be exited; the default was `[1]`. The optional *expression* specified the value to use for the exited construct, if one were needed because of its usage context. `return` was also regarded as an escape, although it did not allow a level specification.

The methodological difficulties of characterizing the 'target' of an escape in terms of the kind of syntactic container and a level count are obvious: any change in the intermediate source may easily require a coordinated change in the escape, but the need for that change is easy to overlook.

In BLISS-11, labels were introduced into the syntax—virtually any construct could have a label. To avoid parsing ambiguities in dealing with undeclared names, labels had to be declared. The set of escape constructs was collapsed into a single `leave` construct written as:

`leave` *label-name* [`with` *expression*]

As a pragmatic concession to popular usage, the single level `exitloop` was also retained.

Copyright © 2002 John Wiley & Sons, Ltd.

BLISS-11 implementation and use

Brender drafted the first version of a proposal for BLISS-11 in early 1972. Brender, Wulf and others at CMU debated and evolved the language proposal over the several months following.

Rather than adapt the BLISS-10 compiler framework, Wulf undertook a completely new design intended to allow more focus on optimization. The small memory size available in the PDP-11's 16-bit address space made code quality especially critical. While there were some difficult periods, the BLISS-11 compiler worked out so well that a book [14] and several PhD theses [15–18] in part either inspired or were inspired by that development. Indeed, this design served as the basis for the next generation of BLISS compilers.

At CMU, BLISS-11 became the implementation language for the Hydra operating system on the C.mmp multi-processor system and other Departmental projects.

At DEC, Brender led the project that made the first use of BLISS-11 in a DEC product, PDP-11 FORTRAN IV-PLUS [19]. This was a 'state of the cookbook art' optimizing FORTRAN compiler and run-time library system for the PDP-11/45 with floating point processor. BLISS-11 was used as the predominant implementation language for the compiler but not the run-time software (which was often modified by customers).

There were but a few other groups within PDP-11 Software Engineering that were willing to tolerate the use of a cross-compiler at a time when networks were just emerging—BLISS-11 was hosted on the PDP-10 and compiled code had to be hand carried from the compiling to the debugging systems. Nonetheless, Diagnostics Engineering did become a significant and enthusiastic user of BLISS-11.

Once again BLISS had been successfully used at DEC to produce important products that fulfilled their performance, quality and schedule goals.

THE SECOND GENERATION OF BLISSs: 1975–1978

During 1973 and 1974, DEC made several attempts to design extensions to the PDP-11 architecture that would eliminate the critical 16-bit limit on addressing—none of them were deemed attractive enough to proceed past the paper proposal stage. Finally, it was decided to initiate a new design, which became known as VAX-11, whose most fundamental characteristics are: 32-bit address and word size, 8-bit addressable bytes, with PDP-11 compatible data types. The summary in Table I compares key characteristics with the earlier hardware architectures.

With a new hardware design starting up, there was considerable agreement (but not universal to be sure) that a high-level language was needed to write the new software that would go with it. But which language?

The great language debate raged for close to a year. There were evaluation committees and study reports, and enormous emotional energy spent supporting favorite candidates and criticizing others. Pascal and BLISS were the leading candidates, while PL/I, RTL-2 and a smattering of others also had advocates. C had yet to emerge from Bell Laboratories, although we knew of it.

The decision to use BLISS as the implementation language came in the early Fall of 1975, BLISS-VAX (as it was initially called) would be based on BLISS-11. By October, a BLISS development team of five programmers headed by Brender was formed to begin the language revision and compiler development efforts. The host Development Methods Department was expected to grow rapidly to

Copyright © 2002 John Wiley & Sons, Ltd.

16 people and the total cost was projected at nearly $1.7 million. This was an extraordinary investment in *software tools* for DEC; at that time DEC's revenue was $736 million for the fiscal year ending June 1976 [20] and its entire software development organization was just 328 programmers in February 1976 [21].

The original vision for DEC's new generation of BLISSs seemed simple enough at first: extend BLISS-11 in as straightforward a manner as possible to make it 'fit' the emerging new VAX design, clean up a few known language problem areas along the way, and eventually adapt BLISS-10 to be more like BLISS-VAX. However, three forces complicated the vision and our aspirations:

- Trying to converge the BLISS-10 and BLISS-11 variants of the language at the same time that we tried to extend it for VAX proved technically much harder than originally envisioned.
- Management concluded that the new BLISS must be a vehicle for portable programming. Having chosen to start with a language foundation that emphasized machine access and machine dependence, the portability goal forced even more radical change than expected.
- Once we had broken the bonds of close compatibility with BLISS-11, it was very hard to limit the reach of our language redesign.

Language evolution

Most of the changes motivated directly by the VAX hardware were well served by the BLISS-11 example. For compatibility with the PDP-11, VAX retained the term *word* to mean two bytes; it coined the term *longword* to mean four bytes. The longword sized value became the size of all values in BLISS-VAX. The keyword `LONG` was adopted to indicate a value of this size.

During the design of BLISS-11, we had waffled over whether subfields of a word should be signed and eventually stayed with unsigned, for compatibility with BLISS-10. However, as experience continued to accumulate, it seemed clearer and clearer that there was a valid need for both.

Thus, the `SIGNED` and `UNSIGNED` *sign extension attributes* were introduced. The `<>` operation was extended to have an optional third parameter that specified whether to perform sign extension (1) or zero extension (0) on a fetch; it was allowed, but ignored for an assignment which always did simple unchecked truncation. Thus, given

```
global  A : BYTE,   B : WORD SIGNED,    C : LONG SIGNED,    D;
```

the following pairs are equivalent:

```
D = .A;   and   D<0, 32, 0> = .A<0,  8, 0>;
D = .B;   and   D<0, 32, 0> = .B<0, 16, 1>;
D = .C;   and   D<0, 32, 0> = .C<0, 32, 1>;
```

Notice that for 32-bit sized accesses, the signed versus unsigned distinction makes no difference given the VAX two's complement binary representation of integers.

We could have formulated at least the single precision floating point operations as infix and unary operators on VAX, but we did not. For consistency, completeness and uniformity across all three platforms, these operations were formulated as special functions built into the compiler that took one or two address operands to specify the inputs and an additional address operand that specified where

Copyright © 2002 John Wiley & Sons, Ltd.

Table II. Predefined literals for portability.

Characteristic	Name	PDP-10 value	PDP-11 value	VAX value
Bits per BLISS value (word)	%BPVAL	36	16	32
Bits per addressable unit	%BPUNIT	36	8	8
Addressable units per BLISS value (%BPVAL/%UPVAL)	%UPVAL	1	2	4
Bits per address	%BPADDR	18	16	32

to store the result. Unknowingly, this approach neatly solved the problem for several future BLISSs as well.

Portability is often viewed as a property of a programming language. That is, it is expected that (almost) all programs written in that language will behave the same when compiled for and executed on diverse machine architectures. If the machine architectures are not too different, and more importantly if the main data types such as integer and floating point values have similar properties, this view is often true enough for many purposes. However, when the language is untyped, so that the fundamental (and only) data type is the machine sized word, then portability becomes much more a property of a program rather than the programming language as such. Portability needs to be planned in advance rather than merely tacked on as part of a porting exercise.

Our challenge was to provide the language mechanisms that would make it as straightforward as possible to write portable programs using BLISS.

Predefined literals were introduced whose values described characteristics of the target architecture of a compilation. These are shown in Table II.

The predefined literals made it possible to declare a set of standard structures: `VECTOR` for a one-dimensional array of scalar elements, `BITVECTOR` for packed vectors of one-bit elements, `BLOCK` for record-like structures, and `BLOCKVECTOR` for a vector of block elements. Figures 7 and 8 illustrate how `BITVECTOR` and `BLOCK` were declared in a portable manner.

In principle, it was not necessary to predefine these structures—they could have been defined by any user that needed them. However, we built them into the language definition and implementation for several reasons.

First, we knew from several years experience using BLISS that these were the dominant structures that would be needed by most users most of the time.

Second, and actually more importantly, we were not prepared to build a debugger with support general enough to handle the full power of BLISS expressions. We took advantage of the fact that this particular set of just four structures was general enough for most purposes and built support for them directly into the debug symbol table representation and the debugger.

Third, and perhaps something of a rationalization, we believed that it was far more valuable for a program to include within itself ‘dumper’ routines that could be called from a debugger to provide

Copyright © 2002 John Wiley & Sons, Ltd.

```
! Bitvector: a vector of 1-bit components
!
structure   BITVECTOR[I; N] = [(N + (%BPUNIT-1))/%BPUNIT]
                (BITVECTOR + I/%BPUNIT)<I mod %BPUNIT, 1, 0>;
```

where the formal parameters of the structure have the following interpretations:

Formal Name	Meaning
I	The number of the element to be referenced (0 origin)
N	The number of elements in the vector

Figure 7. Predefined structures for portability: BITVECTOR.

```
! Block
!
structure   BLOCK[O, P, S, E; BS, UNIT=%UPVAL] = [BS*UNIT]
                (BLOCK + O*UNIT)<P, S, E>;
```

where the formal parameters of the structure have the following interpretations:

Formal Name	Meaning
I	The number of the element to be referenced (0 origin)
O	The offset from the base of each block of a field
S	The size (in bits) of a field
E	The sign-extension rule (0 means zero-extend, 1 means sign-extend)
BS	The size (in UNITs) of each block
UNIT	The number of addressable units in each element (defaults to %UPVAL)

Figure 8. Predefined structures for portability: BLOCK.

an interpretation of a data structure. Such an interpretation can be made much closer to the problem domain being addressed than any mere field by field data structure dump.

When we added the unsigned relational operators in BLISS-11, we were not thinking about transportability—and we got it wrong. What was needed was a set of operators specifically for comparing addresses that could be implemented on the PDP-10 using signed comparison instructions (because 18-bit addresses compare the same whether viewed as signed or unsigned integers) and implemented on the PDP-11 and VAX using unsigned comparison instructions. We added address versions of the relational operators, naming them after their BLISS-10 counterparts, but with a suffix 'A' appended (EQLA, LEQA, LSSA, and so on). The result was three sets of relational operators for signed and unsigned integer and address comparisons, respectively.

Character (string) data on the PDP-10 was typically stored five 7-bit characters or six 6-bit characters in a 36-bit word. On the PDP-11 and VAX it was stored one 8-bit character in an 8-bit byte. To make portable character handling feasible across these domains, we developed a set of *character handling*

Copyright © 2002 John Wiley & Sons, Ltd.

functions to handle the allocation of character string buffers, the creation of a *character pointer*, and the reading and writing of characters indirectly using character pointers. In most respects, the character handling functions are straightforward and not particularly surprising. The trick, if you will, was to formulate them to be abstract enough to be compiled to multiple machine architectures generally as well as be efficient on the machines we particularly cared about.

One aspect of the PDP-10 does show through in a way that might not be expected. We defined the sequencing and access functions to advance the character pointer *first* and then to read/write the character at the updated position, rather than the opposite order. This choice directly reflects the PDP-10 hardware `ILDB` (increment and load byte) and `IDPB` (increment and deposit byte) instructions, which performed the increment and access in that order.

Conditional compilation, the conditional inclusion or exclusion of program text based on a compile-time test, was a well-established feature of many assembly languages, but is rarely available in high-level languages, even today. We debated over many months whether to add a form of conditional processing that, like macros, would take place as part of lexical processing and prior to syntactic parsing. Wulf recommended against it. There was concern that interactions between the conditional mechanism and macros generally would be complicated and hard to understand.

Eventually, a conditional compilation mechanism was adopted which took the form:

`%if` *texical-test* `%then` *lexemes...* `%else` *lexemes...* `%fi`

The lexical test could be any expression whose value was known at compile-time. If the value was `true`, then the lexemes following `%then` up to the following `%else` if present, or `%fi` otherwise, are included as part of the program, and those following the `%else` were ignored; otherwise, conversely. `%if` required an explicit closing `%fi`, which was not analogous with the normal executable `if-then-else`—we simply could not find an attractive way to do without it!

We limited the interactions between this conditional mechanism and macros with well-formedness rules regarding nesting. Basically, if a macro body, macro actual argument or then-part or else-part of a lexical conditional contains the lexeme `%if`, then it must also contain the matching `%then`, `%else` (if present) and `%fi`. Further, this rule must be satisfied by the source file before *any* lexical processing has been performed.

In retrospect, the lexical conditional mechanism accomplished everything that we wanted of it and has not been a source of abuse or confusion; in particular, the well-formedness restrictions have proved just right in this regard. Moreover, I am confident that had we also included `%elsif` and some form of `%case`—purely syntactic sugar, really—these would have proven convenient and safe as well.

As we evolved and expanded the language, we became concerned that the growing number of names would become too large to be intellectually manageable. We ended up with over 500 defined names in the BLISS language manual!

We certainly did not want to reserve all defined names across all variants of BLISS as portability goals might suggest. We also wanted some flexibility to introduce additional new capabilities over time without risking conflicts with existing programs if at all possible.

We divided the set of names into several categories as follows (where the number of names in each category is shown in parenthesis).

- A *reserved keyword* (such as `if` or `local`) has a fixed meaning and use; it cannot be declared as a name under any circumstances (107). The lexical operation names (such as `%if` or `%count`)

Copyright © 2002 John Wiley & Sons, Ltd.

technically all count as reserved keywords. However, since no user names beginning with `%` could be declared in any case, they are not a potential source of conflict (70).

- An *unreserved keyword* (such as the names of module or program section options) can be declared as a user name. Unreserved names occur in the language definition in very specific contexts and typically select among a fixed set of alternatives of some kind; that is, where a declaration or lack thereof of the name has no relevance (102).
- A *predeclared* name (such as the absolute value function `ABS`) can be used as an explicitly declared name. Such a declaration makes the predeclared meaning unavailable within the scope of the explicit declaration. Conceptually, predeclarations exist in an imaginary scope that contains the program being compiled and such names are subject to nested redeclaration in the normal way (55).
- A *built-in name* must always appear in an explicit declaration. If it is declared by a built-in declaration, as in

```
builtin R0, ARGPTR; ! R0 a register, ARGPTR an argument block pointer
```

  then it has its predefined meaning; otherwise, it has the meaning given by its explicit declaration, just as if it were not a predefined name. Most of the predefined names in this category are functions that correspond to particular machine instructions of the target architecture whose effect cannot otherwise be achieved in a natural or direct manner (19 in common, 36 in BLISS-16, 103 in BLISS-32 and 45 in BLISS-36).

While 107 reserved keywords is rather large for a high-level language, it is a whole lot better than over 500!

To aid in making a program transportable, a `LANGUAGE` switch option was provided which expressed the intention to compile the program with a particular set of BLISS compilers. The parameter could be either `COMMON`, which specified all BLISS dialects (which theoretically could—and later did!—change over time), or a list of specific dialects from among `BLISS16`, `BLISS32`, and `BLISS36`. For example, if you intended your program to be used on either the PDP-11 or VAX, you would specify `LANGUAGE(BLISS16, BLISS32)`. In this context, the BLISS-32 compiler would give a warning if the attribute `LONG` was used, because that attribute was not available in BLISS-16.

Many of the general language improvements were motivated by weaknesses and limitations experienced while using BLISS-10 and BLISS-11, while others were motivated by a desire to actually increase the degree of hardware integration for each respective architecture. (Our management occasionally needed to be reminded that the latter was an important part of our goal set.)

We introduced a considerable number of lexical functions. Most of these allowed conversion between different kinds of arguments (such as forming a string, name, or numeric value from concatenation of the arguments), determined properties of an argument (number of characters in a name or string, if name is declared, the storage allocated for a data segment name), or performed side-effects outside of the language (such as annotating a listing or providing diagnostic reports).

Of particular interest, however, is use of the `%assign` lexical function in combination with compile-time variables. The declaration

```
compiletime Y = 0;
```

Copyright © 2002 John Wiley & Sons, Ltd.

```
macro   FANCY = CALL_FOO(2) %;
!
! At this point FANCY expands to CALL_FOO(2)

undeclare %quote FANCY;   ! End the current declaration of FANCY
macro   FANCY   = CALL_BAR(3) %;
!
! At this point FANCY expands to CALL_BAR(3)
```

Figure 9. BLISS macro re-declaration example.

declares a 'literal' (which must be given an initial value) whose value can be changed at compile time using `%assign`. The value of the literal is the value most recently assigned as in

```
%assign(Y, Y+1)
```

which increments the value of Y. Lexically the `%assign` function disappears; it is used solely for its side effect of changing the value of a compile-time variable.

The `%quote` lexical function is also of special interest. BLISS-11 had introduced the undeclare declaration, which logically terminated the scope of a prior declaration. It was originally intended as a way to prevent accidental access to data segment names defined in outer blocks, for example, when that data segment was being manipulated using an alias with a different structure. However, it also made the name available to be declared again.

In BLISS-11 there was no way to redeclare a macro name because the macro call was expanded before the undeclare declaration was syntactically parsed. This was surmounted by the new `%quote` lexical function which prevented macro expansion. This is illustrated in Figure 9.

The combination of macro and undeclare declarations facilitated by `%quote` meant that macros could be used at compile time to hold varying lexeme streams.

The combination of a highly-flexible macro mechanism generally (which, in particular, provides certain forms of iteration), a flexible lexical condition facility (`%if`) to provide compile-time decision making, compile-time declarations and `%assign` to provide compile-time numeric manipulation, macro and undeclare declarations together with `%quote` to provide compile-time lexeme stream manipulation, and a rich set of lexical functions more generally, together provide truly extraordinary compile-time capability. Some of the more elaborate techniques that evolved to exploit this capability are described in [22].

One of the first applications of the BLISS compile-time mechanisms was to establish a set of macros to aid in defining portable record-like data structures. There have been a number of variations in this set over the years, but this sketch will illustrate the basic idea. To begin, consider Figure 10 and compare it with Figure 3.

The `$structure` macro takes one argument, an identifier that later serves as the name of the 'record type' being declared. This macro call initializes various compile-time names that are used to keep track of the allocation as it progresses, remembers the identifier for later use in the body of a macro it declares, and emits (expands to) the beginning of a field declaration. (A field declaration declares a

Copyright © 2002 John Wiley & Sons, Ltd.

```
$structure(FORM1)
    Word1LowHalf = [$address],
    Word1Flag1   = [$bit],
    Word2Full    = [$int]
$end_structure

! which expands (in BLISS-36) to
!
!field
!   FORM1__FIELDS =
!       set
!       FORM1__BASE  = [0, 0,  0, 0],
!       Word1LowHalf = [0, 0, 18, 0],
!       Word1Flag1   = [0, 18, 1, 0],
!       Word2Full    = [1, 0, 36, 1]
!       tes;
!
!literal FORM1__SIZE = 2;
!
!macro   FORM1 = BLOCK[FORM1__SIZE] field(FORM1__FIELDS) align(0) %;

own DATUM : FORM1;

DATUM[Word1LowHalf] = -1;
DATUM[Word1Flag1]   =  0;
DATUM[Word2Full]    =  1;
```

Figure 10. BLISS portable data structure example.

name for a set of field names, which can be specified using a data segment attribute as the only set of field names that are valid in structure accesses for that data segment.)

Following the `$structure` macro call are a series of normal field name definitions, except that instead of a series of numbers there is a macro call that mimics the name of a type. Typical macros include `$address` for an address, `$int` for a signed integer (default size), `$uint` for an unsigned integer (default size), `$bit` for a bitfield or flag, and `$bits`(*n*) for a field of *n* bits. Each of these macros performs several computations to make an appropriate allocation: the allocation (number of addressable units and bit position within unit) is rounded up as appropriate to the alignment of the new component, the appropriate offset, position, size and sign extension values are emitted for the component, the largest alignment encountered is computed for later use as the alignment of the record as a whole, and the allocation is again increased by the size of the component.

The `$end_structure` macro call completes the field declaration, after adjusting the size for any applicable alignment requirements. It then emits a literal declaration for the size of the record. Finally it emits a macro declaration that defines the original parameter name to expand to a `BLOCK` built-in structure attribute of the appropriate size followed by appropriate field and alignment attributes.

As a result of all this, the original parameter name becomes suitable to use as a sort of pseudo-type name in any kind of data segment declaration.

Copyright © 2002 John Wiley & Sons, Ltd.

In more sophisticated versions of these macros, there are means for handling unions and nested structures, reverse (negative offset) allocation from the base of the structure, vectors of and pointers to other pseudo-types, and so on.

In many cases, these macros could share implementation details based on the built-in literals such as `%BPUNIT`; in other cases, they were different for each respective target architecture. What mattered most, of course, was that the code that used these macros was target independent and adapted automatically to the target for which it was being compiled.

Building the compilers

The extensive evolution of the language definition relative to BLISS-10 and BLISS-11 greatly complicated the task of first implementing compatible compilers, and then bootstrapping them into the revised language and onto the new VAX machines. A complete account of that process is found in [23].

By the end of 1979, the new generation of self-hosted and highly-language compatible BLISS compilers was firmly established at DEC. We had BLISS-36 targetting and hosted on the PDP-10, BLISS-32 targetting the VAX-11 and hosted on both the PDP-10 and VAX-11, and BLISS-16 targetting the PDP-11 and hosted on both the PDP-10 and VAX-11.

In 1981 an attempt was made to shoe-horn a variant of BLISS-16 onto the PDP-11 using a combination of language subsetting, somewhat simplified code generation, and heavy overlaying. A constraint was that we were not willing to compromise much on code quality compared to that available from the BLISS-16 cross-compilers. While the resulting compiler generally worked, it was quite limited in the size of module it could compile, because the 65 Kbyte PDP-11 address space was just too small. The compromises were deemed too severe and the experiment was abandoned.

THE GOLDEN YEARS AT DEC: 1978–1990

With a common BLISS language, with BLISS compilers available for all three of DEC's major architectures, and with the backing of Corporate policy, BLISS became established as the language of choice for new software development.

For the PDP-10 family, most software that had been written in BLISS-10 was eventually converted to use BLISS-36. A number of new products were initiated which were intended for both PDP-10 and VAX systems.

Similarly for the PDP-11 family, software that had been written in BLISS-11 was eventually converted to use BLISS-16. But the lack of a native BLISS-16 compiler proved a strong disincentive to expanded usage. Further, the PDP-11 line was rapidly overshadowed by the VAX family.

BLISS-32 and the VAX architecture were created and grew up together, so it is natural that BLISS usage flourished on VAX/VMS. Large parts of VMS itself and almost all layered products on VMS were written in BLISS for many years.

THE ALPHA YEARS: 1990–TO DATE

The Alpha 64-bit architecture was introduced in late 1991. It is a RISC, load/store, base-register oriented addressing architecture in contrast with the CISC, multiple operands with complex addressing

Copyright © 2002 John Wiley & Sons, Ltd.

mode characteristics of the VAX architecture. Table I summarizes the key Alpha hardware properties. Despite these differences, the impact on the BLISS language was quite small and straightforward—in most respects.

The most interesting aspect of BLISS for Alpha is that there is not just a new 64-bit variant of the language, BLISS-64, as might be expected. There was a new 32-bit variant as well. More interesting yet is the mixed 32-bit/64-bit hybrid that eventually emerged.

For compatibility with the VAX, Alpha retained the terms *word*, *longword* and *quadword* to mean two, four and eight bytes, respectively. The quadword sized value became the size of all values in BLISS-64, of course. The keyword `QUAD` was adopted to indicate a value of this size.

When the Alpha architecture was first being designed, it was expected to support two operating system environments: a 64-bit version of UNIX and a 32-bit version of OpenVMS. Because OpenVMS/VAX was such a huge part of DEC's installed base, it was felt that everything possible must be done to make porting from OpenVMS/VAX to OpenVMS/Alpha as straightforward as possible while retaining the ability to extend OpenVMS with full 64-bit support later. Thus, a 32-bit variant of BLISS (BLISS-32E as it came to be called) was needed for Alpha as well as a 64-bit variant (BLISS-64E).

BLISS-32E also came to be used on UNIX in conjunction with the `taso` option, in which a program can be compiled and linked for execution in only the lowest 32-bit segment of the Alpha address space. This was strictly a software convention—no special hardware or operating system modes were involved. For a number of years, most of DEC's compilers on UNIX themselves operated in the `taso` environment, were built using BLISS-32E, but produced code that operated in the full 64-bit environment.

The problem with producing a 64-bit application using a compiler operating in the 32-bit `taso` domain is that BLISS-32E is truly 32-bit, which is obvious on the one hand and has fundamental implications on the other. At the simplest level, it means that internal compiler data structures are limited to sizes representable in 32-bits. This limits even the *description* of entities that made use of more than 32-bits of storage, such as FORTRAN COMMON blocks or C global variables. Further, to perform compile-time constant arithmetic, even input conversion of literals, required the same sort of awkward techniques that are required to cross-compile from a 32-bit to a 64-bit environment. Finally, even though the registers were 64-bit wide, algorithms that should benefit from that width (small sets, bitmask techniques for memory management, searching, or the like) do not benefit from the available width in the real environment.

The UNIX compilers were originally developed and cross-compiled on 32-bit VAX/VMS, which meant that these limitations were viewed as a necessary concession. However, full 64-bit clean support was wanted for the subsequent natively hosted compilers. Was there a way to have 32-bit pointers using BLISS-64?

To overcome these limitations but keep the space advantage of 32-bit pointers, a new command option `/ASSUME=LONG_DEFAULT` and the module switch `LONG_DEFAULT` were introduced. Basically, these cause the 64-bit BLISS to perform default 32-bit allocation for scalars, and assume fetches and stores are 32-bits unless the data segment is explicitly declared larger. The result was to create a compilation mode in which addresses are stored and used as 32-bit values, but otherwise the full 64-bit word size is usable.

Through use of these switches in combination with some adjustments in the GEM macros that declare (pseudo-)types, it was possible to compile the same sources on VAX/VMS using BLISS-32

Copyright © 2002 John Wiley & Sons, Ltd.

to create a BLISS-32E cross-compiler (VAX/VMS to Alpha Tru64 UNIX) with 32-bit limitations, on VAX/VMS using the BLISS-32E cross-compiler to create a BLISS-64E native compiler with 32-bit limitations, or on Alpha Tru64 UNIX using BLISS-64E to create a clean native-compiler without 32-bit limitations.

Other environments

BLISS has also been ported and used with non-DEC/non-Compaq hardware and software environments.

For a brief period from about 1989 through 1993, DEC offered a DECstation line of workstation and server products based on the MIPS R3000 architecture [24] and Ultrix, which was DEC's version of the UNIX operating system. The characteristics that matter most to BLISS, namely word size and addressable unit size, are the same as for VAX. While this was the first RISC architecture supported by BLISS (unless you count the PDP-10), in most respects the machine architecture had little impact on the language. In fact, many of the changes were influenced by the operating system environment more than hardware considerations.

In the mid-1990s, GEM was retargetted to support the Intel IA-32 architecture in order to offer a DIGITAL Fortran 90 product under Windows 95/98/NT. That meant porting BLISS as well, of course. The original DIGITAL Visual Fortran (DVF), now known as Compaq Visual Fortran (CVF), was a joint DEC and Microsoft venture that replaced Microsoft's Powerstation Fortran in early 1997.

Recently (1999), BLISS was ported to the Alpha Linux environment. This was done as part of making the GEM-based FORTRAN 90, C and C++ compilers from Compaq Tru64 UNIX also available on Alpha Linux. The most significant work from the GEM perspective was adding ELF object file and DWARF2 debugging symbol table support. No language changes were needed as a result of this port.

As a result of Compaq's June 2001 announcement that it will be phasing out its Alpha architecture and porting its operating system products to Intel's IA-64 architecture, BLISS has been retargetted to support the VMS operating system port. About the only changes relate to the set of built-in functions and the naming and use of registers.

RETROSPECTIVE ASSESSMENT

With some 30 years of usage experience to draw on, it is appropriate to attempt an assessment of those characteristics that make BLISS unusual and interesting in the family of program languages, to suggest which have been positive and deserve consideration by future languages versus which are of little consequence or even harmful.

Typelessness

Of all the things that we set out to do to and with BLISS in 1976, the one thing that never happened was to introduce data typing. We initially thought that data typing was a prerequisite for solving many portability problems. However, along the way, as we gained experience porting the BLISS compilers themselves, we learned alternatives.

Copyright © 2002 John Wiley & Sons, Ltd.

We also learned some hard lessons in how difficult it can be to add features to a programming language. However much certain characteristics of BLISS-10 seemed undesirable, a simple fact was that it had a coherence and internal consistency that sometimes made apparently simple changes more difficult or far reaching than expected. One example is seen in the evolution of the `<p, s>` field access notation from a 'normal operator' that produced one value from another in BLISS-10, to optional operands to the fetch and assignment operators in BLISS-11. It sounds simple and we covered it earlier in just a few words—but it fact it had subtle and far reaching effects on the meaning of structure declarations, how they were compiled and how they were used, that were still being worked out as we implemented the second generation of BLISSs.

Still, we thought that data typing could be introduced in BLISS. Both Brender and Wulf (separately) claimed that they thought it could be done 'in a reasonable and consistent manner' [25]—but no proposals ever got written down. In the end, there was just too much else to do (all of which did get done) and we learned how to do without it.

This is not to say that data typing, of itself, is a panacea. Many programmers have learned the hard way, for example, that without careful forethought the C types `int`, `long` and `void *` provide enormous opportunity to compromise the portability of a program.

The effect of typelessness has been most strongly felt when trying to deal with scalar quantities that are more than the machine word size. Thus, on the PDP-11 it was necessary to adopt an awkward coding style to deal with 32-bit integers or floating point values. This was also true when dealing with 64-bit entities using the 32-bit variants of BLISS on the Alpha architecture. This property more-or-less completely prevented BLISS from gaining any significant usage in mathematical application domains. Ever since the PDP-11, floating point has been the sore point that most severely challenged the BLISS data model. Note that it is not typelessness as such that leads to this difficulty, but the size limitation resulting from the single size of all data.

If you reflect on the way that the scalar allocation attributes (`BYTE`, `WORD` and so on) and sign extension (`SIGNED` and `UNSIGNED`) attributes interact with the fetch and assignment operators, it is tempting to view them as a kind of very weak data typing. In fact they do just the opposite: these attributes really serve to wash out the size and sign differences and normalize values to the one and only word size.

It is interesting to speculate on how differently BLISS might have evolved if *size*, and only size, had been formalized as an intrinsic part of each BLISS value. That perhaps weakest of all possible type properties, all by itself, would have eliminated many of the limitations of BLISS in dealing with 'large' values, compared to the fixed size nature it has today.

At the time BLISS was first designed, the typeless model simplified the language by eliminating a large and complicated aspect of most high-level languages and allowed more focus on low-level performance and hardware integration. While that tradeoff was reasonable and productive at the time, it is no longer so.

Structures

BLISS structures actually work quite well for aggregates that really are algorithmic in nature (arrays, matrices, even many sparse variants). However, they do not really work quite right for simple static arrangements—it just feels wrong to need to compose simple records (`structs` if you will) out of a combination of predefined structures and a bunch of macro and/or field definitions.

Copyright © 2002 John Wiley & Sons, Ltd.

Even as we evolved and introduced Common BLISS at DEC, we reserved the identifier `record` as a keyword in the hope that we would be able to later devise and add a new declaration that would provide a more intuitive and direct means to specify simple records. That never happened—`record` remains reserved and unused to this day.

More importantly, the structure and supporting mechanisms, fundamental and flexible though they are, do not go far enough to provide full separation between the access notation and the actions to be performed. For example, there is no way to define a field composed of several disjoint parts yet access it, both for fetching and assignment, using the same uniform notation. Here it would seem that at least fetch versus assignment usage context must be made available in the definitional mechanism—it is not.

The dot operator

The dot operator usually throws new users for a loop for a while, but this gradually subsides. 'Missing dot' bugs make good coffee stand conversation, but in fact do not happen all that often and when they do they tend to have dramatic, easy to find effects rather than subtle ones (not always, but usually).

Overly enthusiastic new users sometimes try to code things with two or even three dots (fetches) in a row[¶]. While not necessarily wrong, experience teaches that this is most often a stylistic error that can and should be avoided. After these lessons are learned, using the dot becomes a routine part of one's coding practice—often like indentation—that one does rather automatically while thinking about larger issues.

This surely sounds like a 'damned by faint criticism' sort of defense—and in a way it is. On the other hand, the dot operator interacts with other aspects of BLISS, notably structures, typelessness, pervasive expressions, even map and bind declarations as well as parameter passing, in ways that would make it very hard to eliminate. Eliminating the dot operator would definitely require compensating changes in many of these other parts of the language.

No `goto`

The elimination of `goto` was a wonderful idea!

BLISS-10 eliminated the `goto` *and* labels, and compensated by including way too many forms of exit constructs. BLISS-11 corrected this mistake, and the resulting formulation of `leave` has been carried forward without any perceived need for reconsideration.

The experience of this author has been typical: whenever a piece of code gets to the point that it is tempting to use a `goto`, there is virtually guaranteed to be a reformulation without using `goto` that is cleaner, clearer and more elegant.

Expression language

The expression language characteristic was often highly touted in the early years of BLISS. While there is a certain conceptual elegance that results, in practice this characteristic is not exploited much.

[¶]One colleague reports that there was a case in a certain piece of VMS software (prior to V5.5), where five consecutive dots were used; further, not only was it correct, but he asserts that it was the clearest way to document exactly what was being done!

Copyright © 2002 John Wiley & Sons, Ltd.

The most common applications use the `if-then-else` expression, for example, in something like the maximum calculation illustrated in Figure 5. Very occasionally there is some analogous use of a `case` expression. Examples using loops (taking advantage of the value of `leave`), however, tend not to work well on human factors grounds: the value computed tends to be visually lost in the surrounding control constructs and too far removed from where it will be used; an explicit assignment to a temporary variable often seems to work better.

It is interesting to compare `leave` with `return` in this respect. Perhaps it is the prominence of the containing construct, so that it is quite clear where the value is used, that makes `return` so widely accepted.

On balance, the expression characteristic of BLISS was not terribly important.

Compile-time language

The creation of a comprehensive compile-time facility was not, of course, one of the original BLISS goals; rather, that capability began to emerge in the form of more flexible macros in BLISS-11 and really came to fruition in the second generation of BLISSs created at DEC.

It is not possible to stress enough just how highly the compile-time facility is regarded here at Compaq. It has been used perhaps most importantly in any number of schemes for table creation, dumpers, and other tools that, because they are an integrated part of the language and the code being created, contribute to uniformity, consistency and ultimately, reliability.

No other commercial quality language comes close to matching this capability.

C

BLISS and C both emerged about the same time from the same historical context and they shared many points of view. Thus, it is inevitable that some kind of 'BLISS versus C' comparison or evaluation should be attempted[‖].

When all is said and done, perhaps the most important difference is that the designers of C chose to keep a mostly traditional data type model that was both simple and 'good enough'. In particular, scalars that happen to be larger that the machine word size (not the least unusual for floating point some 30 years ago) and records (`structs`) could be declared simply and used in straightforward ways. The BLISS typeless data model, based on all data being of the underlying machine word size, definitely made some of these simple things not so simple. BLISS gradually evolved a whole lot of machinery to help mitigate those problems; the result is effective, but far from simple.

On a technical level, I will share one blatant opinion: C would benefit greatly from more flexible and comprehensive compile-time capabilities.

One other consideration is probably crucial as well: C originally came complete with an operating system whose vitality (and economics) strongly aided its spread (propelled by what might be considered the original 'open source' movement). The C language was smaller than BLISS and the early compilers less concerned with code quality, making C compilers easier to port and retarget.

[‖]However, quite honestly, I am probably one of the worst candidates to offer an objective perspective. After all, C 'won', right?

Copyright © 2002 John Wiley & Sons, Ltd.

C compilers became available just about everywhere and just about everyone who did any system programming became exposed if not expert in its use.

BLISS had no such vehicle. While DEC did make its BLISS compilers available for sale, it did nothing to promote them and gave no thought to retargetting, or even helping others to retarget them, to architectures outside of its own product set. Over time management became painfully aware that most new employees already knew C but few knew anything about BLISS, which meant more lengthy training and integration periods.

To participate as an equal in outside technical exchanges, as a contributor as well as a consumer, DEC itself began to allow and then prefer using C over BLISS for new project starts or where BLISS was not already well established.

On at least two occasions over the years, as the emergence and popularity of C became apparent, there were serious efforts to explore whether and how the BLISS code base could be converted to C. In each case, it was the lexical and compile-time features of BLISS that proved an insurmountable barrier. Since there were no comparable features in C, translation from BLISS to C was much more than just a mechanical process—it required a wholesale change in coding style and code base management.

SUMMARY

From a highly machine-specific implementation language on the PDP-10, BLISS evolved into a family of highly-compatible and portable languages and compilers for a variety of hardware (PDP-10, PDP-11, VAX, MIPS, Alpha, IA-32, soon IA-64) and operating system (TOPS-10/-20, RSX-11M/D, VMS, Ultrix, Tru64 UNIX, Linux, and Windows 95/98/NT) environments. Each BLISS remains true to an initial goal to allow and support low-level machine-specific and idiosyncratic access and control *when wanted and needed*, while providing mechanisms for portability.

There is no doubt that the amount of language evolution that took place during the creation of DEC's second generation of BLISSs was far greater than anyone (including those of us doing it) had anticipated. About the only way to assess whether we 'got it right' is to look at the degree of evolution that took place as BLISS was extended to later machine architectures that had not been considered as part of the original design. Since that later evolution was both small and almost entirely within the framework established to allow extension, it appears fair to conclude that by and large we did get it right—at least within the constraint of being typeless.

During the Golden Years, BLISS enjoyed widespread support and use within DEC. While there were a modest number of enthusiastic users and supporters outside of DEC, DEC never promoted BLISS beyond making it available as a product and never sought to port it beyond its own product lines. As a result, BLISS never became very well known in the general computing community. Ironically, as DEC itself faltered in the 1990s, the divestiture of software products written in BLISS did more to spread knowledge and use of BLISS outside DEC (now Compaq) than ever before.

While the language is clearly near the end of its life, it continues to be an effective implementation language in those niches were it is well established, though only as a captive tool.

ACKNOWLEDGEMENTS

It is truly impossible to credit everyone who has made a significant contribution to the development and evolution of BLISS over the last 30 years. However, some contributions really stand out and deserve special mention.

Copyright © 2002 John Wiley & Sons, Ltd.

At CMU, Bill Wulf was the creator of the language, whose inspiration, coding and leadership brought the ideas to reality. Chuck Geschke, Steve Hobbs, Rich Johnsson, Bruce Leverett, and Dave Wile contributed code, generated language ideas and infused new technology into BLISS while earning their PhDs during the CMU years.

At DEC, Gordon Bell had the courage to point DEC, and invest heavily, toward the use of a high-level systems implementation language at a time when many thought it risky or foolish or both. Marty Jack was a member of the team that created the second generation of BLISSs, whose superb coding skills and outstanding productivity got the new implementations solidly under way quickly. Ike Nassi provided lots of language ideas for making BLISS as portable as possible. Rich Grove was the first major BLISS-11 user at DEC when he led the compiler part of the PDP-11 FORTRAN IV-PLUS product.

REFERENCES

1. Wulf WA, Russell DB, Habermann AN. BLISS: A language for systems programming. *Communications of the ACM* 1971; **14**(12):780–790.
2. MIT Project MAC. *EPL Reference Manual*, April 1966.
3. Graham RM, Daley RC. Epl subset for systems programming, 1969. Section BB.2.01 in Multics system-programmers' manual. http://multicians.org/mspm-bb-2-01.html.
4. Burroughs Corporation, Detroit, MI. *Burroughs B5500 Extended Algol*.
5. Wirth N. PL/360, a programming language for the 360 computers. *Journal of the ACM* 1968; **15**(1):37–74.
6. Richards M. Memorandum M-352. *BCPL Reference Manual*. MIT Project MAC, 1967.
7. Spring Joint Computer Conference. *BCPL: A tool for compiler writing and systems programming*, 1969. AFIPS SJCC 34 (1969), 557–566.
8. Wulf W. Bliss/11 dec mini course [sic]. *Unpublished Lecture Notes*, 1972.
9. Digital Equipment Corporation, Maynard, MA. *DECsystem-10/DECSYSTEM-20 Processor Reference Manual* (5th edn). Updated, AA-H391A-T1, 1982. http://www.3bbit.org/dec/manual/ad-h391a-t1.pdf.
10. Dijkstra EW. Go to statement considered harmful. *Communications of the ACM* 1968; **11**(3):147–148.
11. Arnold K, Gosling J. *The Java Programming Language* (*The Java Series*). Addison-Wesley: Reading, MA, 1996.
12. Hobbs S. Personal communication, 2000.
13. Parke B. Personal communication, 2000.
14. Wulf WA, Johnsson R, Weinstock CB, Hobbs SO, Geschke CM. *The Design of an Optimizing Compiler* (*Programming Languages Series*, vol. 2), Cheatham TE (ed.). American Elsevier: New York, 1975.
15. Geschke CM. Global program optimizations. *PhD Thesis*, Carnegie-Mellon University, 1972.
16. Johnsson RK. An approach to global register allocation. *PhD Thesis*, Carnegie-Mellon University, 1976.
17. Wile DS. A generative, nested-sequential basis for general purpose programming languages. *PhD Thesis*, Carnegie-Mellon University, 1973.
18. Weinstock CB. Dynamic storage allocation techniques. *PhD Thesis*, Carnegie-Mellon University, 1976.
19. Brender RF. BLISS-11 and FORTRAN IV-PLUS: A case study in the application of a high level cross-compiler to product development. *Minicomputer Software*, Bell JR, Bell CG (eds.). North Holland, 1976; 53–69.
20. Digital Equipment Corporation. *Annual Report*, 1976.
21. Segal B. BLISS Program Plan, Internal Document, April 1976. Development Methods Department, Digital Equipment Corporation.
22. Brender RF, Brett BR, Mitchell CZ. Pragmatics in the development of VAX Ada. *Digital Technical Journal* 1988; **1**(6):91–100.
23. Brender RF. Generation of BLISSes. *IEEE Transactions on Software Engineering* 1980; **6**(6):553–562. (Also as *Technical Report CMU-CS-79-125*, May 1979, Carnegie-Mellon University).
24. Kane G, Heinrich J. *MIPS RISC Architecture*. Prentice-Hall: New York, 1992.
25. Wulf WA. *Letter to Steve Gutz (Development Methods Manager)*, September 16, 1976.

Copyright © 2002 John Wiley & Sons, Ltd.