A LISP MACHINE WITH VERY COMPACT PROGRAMS

L. Peter Deutsch
Xerox corporation, Palo Alto Research center (PARC)
Palo Alto, California 94304

Abstract

This paper presents a machine designed for compact representation and rapid execution of LISP programs. The machine language is a factor of 2 to 5 more compact than S-expressions or conventional compiled code, and the.compiler is extremely simple. The encoding scheme is potentially applicable to data as well as program. The machine also provides for user-defined data structures.

Introduction

Pew existing computers permit convenient or efficient implementation of dynamic storage allocation, recursive procedures, or operations on data whose type is represented explicitly at run time rather than determined at compile time. This mismatch between machine and language design plagues every implementor of languages designed for manipulation of structured information. Neither of the usual software solutions to this problem is entirely satisfactory. Interpretive systems are easy to build and flexible, but intrinsically inefficient; compilers which approach the efficiency of those for conventional languages are hard to write and often force the implementor (and user) to sacrifice valuable but expensive language features for the sake of efficiency. On many machines compiled code also occupies at least as much space as a structured representation of the source program.

An alternative approach to this problem is to design machines whose code structure more closely resembles that of their major programming language(s). This approach of tailoring the machine to the language was first used in the Burroughs B5000 and successor machines, which were designed to execute ALGOL 60 programs.' In recent years, the availability of microprogrammed processors and the continuing decline in the cost of processor hardware and design have prompted several experiments of this sort at universities* and at least one successful experiment by a large company* and one unsuccessful new commercial venture.[3]

The present paper describes a machine design for efficient representation and execution of BBN-LISP programs. BBN-LISP is an interactive system developed from the Lisp language.* 10 Readers unfamiliar with LISP should consult weissman's excellent primer[5]; some details particular to BBN-LISP appear in the next sections of this paper. A complete and well-maintained but voluminous reference manual for BBN-LISP is also available." The machine design presented here will be referred to as MicroLISP, a name intended to connote both code compactness and possible microprogrammed implementation.

Data Types

LISP has many data types (e.g. list, symbolic atom, integer) but no declarations. The usual implementation of languages with this property affixes a tag to each datum to indicate its type, in LISP, however, the , vast majority of data are pointers to lists or atoms, and it would be wasteful to leave room for a full word plus tag (the space needed for an integer datum, for example) in every place where a datum can appear such as the CAR and CDR of list cells. Consequently, in BBN-LISP every datum is a pointer; integers, strings, etc. are all referenced indirectly. Storage is allocated in quanta, and each quantum holds data of only one type, so what type of object a given pointer references is just a function of the object's address, i.e. the pointer itself.

The chief drawback of this scheme is that every built-in function which produces a number as a result (such as PLUS, the addition function) must allocate a word to hold the result. This leads to frequent, time-consuming garbage collections. BBN-LISP circumvents this problem for the most part by permanently storing all the integers from -1536 to +1535 in consecutive cells and just returning a pointer to one of these cells if a numerical result is in this range, rather than allocating a new cell. In MicroLISP, which is intended as a reasonably efficient numerical language, data on the stack (temporary results and variable bindings) carry a type tag identifying them as integers, floating point numbers, or pointers. In this way, long numerical calculations can take place without any consumption of allocated space.

No existing LISP system permits the user to define his own packed data structures. MicroLISP includes such a facility, since it can be made inexpensive when implemented in the machine language and since its absence from LISP is one of the reasons most frequently cited for choosing other languages for complex symbolic computation. The details are presented in Appendix A, since they are somewhat peripheral to the rest of this paper. It is worth noting that the scheme could be implemented within BBN-LISP and even lends itself to efficient compilation in the usual case.

Control and Binding Structure

MicroLISP uses a single stack structure for both control and variable bindings, essentially as described in a recent paper.[1] A function call allocates a "basic frame" for the arguments and a "frame extension" for control information and temporary values* The basic frame contains the function name, the argument values (bindings), and a pointer to the argument names. The frame extension holds a pointer to the caller's frame extension and a variety of other bookkeeping information. The FUNARG capability of LISP 1.5, i.e. the ability to construct a data object comprising a function and a binding environment, is provided through a primitive function which creates an "environment descriptor" pointing to a specified frame. As long as there are accessible references to this descriptor, the frame continues to exist. Environment descriptors also allow the user to construct cooperating sequential processes (coroutines); the stack becomes tree-structured rather than linear, as in the Burroughs B6500.[1]*

BBN-LISP, like most programming languages, recognizes two kinds of accesses to variables: "load" and "store". This duality actually exists for data structures as well (CAR-RPLACA, GET-POT, etc.) but is not treated systematically. MicroLISP systematizes this concept by allowing a function to have, in effect, two definitions, one for the (normal) "load" context, one for the "store" context. The SET function is extended so that if the first argument is a list

    (fn argl ... argn)

rather than a variable, the function fn is called in "store" mode with arguments argl ... argn and newvalue (the second argument of SET). SETQ is alsO extended in the obvious way, but is not particularly useful. A more useful function is

    (SETFQ (f n argl ... argn) newvalue)

which quotes the function name and evaluates everything else. This allows RPLACA, for example, to be defined as

    (LAMBDA (X Y) (SETFQ (CAR X) Y>) -

The semantics of variables are simple in principle: search the current basic frame, then the caller's frame, etc. for a binding of a variable with the desired name; if none is found, consult the "value cell" of the variable; if this contains the special value NOBIND, the variable is unbound. (In fact, the search follows a chain through *an* "access link" pointer in the frame extension rather than the caller pointer or "control link", to cover application of FUNARGs.) MicroLISP (and compiled BBN LISP) actually use three variations of this searching strategy depending on the situation. Searching for the arguments of the current function is pointless: their relative locations in the basic frame are known to the compiler and they can be accessed by indexing. Searching for variables which are set at the top level and never rebound is time-consuming: there is a compiler declaration to force references to specific variables to bypass the search and go directly to the value cell. Repeated searches for a variable referenced more than once in a given function are wasteful; in MicroLISP the search always occurs at the time of the first reference and is not repeated thereafter.

In both BBN-LISP and MicroLISP, all variable bindings appear in the basic frame. In BBN-LISP half of each word in the basic frame is reserved for the name, In MicroLISP, the basic frame contains a single pointer to a table of names (the LNT; see below). Either scheme requires that any PROG or open LAMBDA which does not constitute the entire body of a function be made a separate subfunction, since PROG variables are bound at the time the frame is created, i.e. when the function is entered. The MicroLISP scheme may slow down free variable searches, since a name table may not be in core any longer when the search wants to scan it. Its advantages are that it is not necessary to insert the name of each variable at function entry time, and that the entire word is available for holding the binding, which (with the help of a few type bits elsewhere in the frame) may thus be a full-word integer or real number.

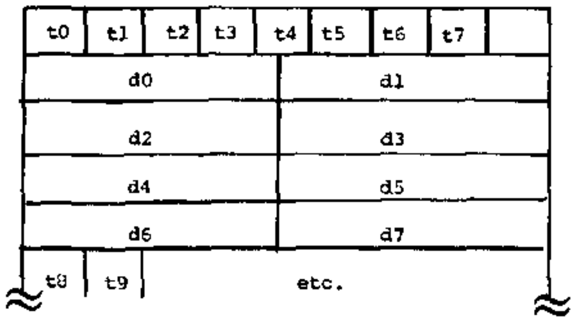
Code Design

Conventional machines generally take the attitude that it must be convenient for any instruction to reference any word in the overall address space. This approach tends to produce instruction formats in which a large fraction (half or more) of the bits are devoted to a memory address. MicroLISP takes advantage of the observed fact that a given LISP function references rather few functions and variables and therefore can make do with very short addresses which just index a global table (of commonly used functions) or a function-local table (of local variables and less common functions). Furthermore, a given name is usually only used as either a function or a variable, not both. MicroLISP tags each name in the tables with a function/variable flag, which eliminates the need for levels of list structure as a syntactic device, and tags functions with an argument count, which eliminates the need for sublists as scope delimiters. Thus MicroLISP code is essentially a string of byte-sized instructions, representing the original s-expression in postfix form, where most bytes reference either a "global name table" (GNT) or a "local name table" (LNT) as just described.

The LNT actually has additional internal structure: argument names come first, then PROG and free variables, then everything else. The "binding" of a free variable is a pointer to the true binding, and the variable searching algorithm uses this knowledge: since all the bindings in a given frame are identified by a single pointer from the basic frame to the associated LNT, the searching process can tell from the tag if the match was on a free variable, and if so, follow the pointer one more step to obtain the value if desired.

MicroLISP programs, like LISP programs, are structured into functions. Each function haB a header which gives the expected number of arguments and the length of the LNT. The former determines the size of the basic frame. The latter determines the function's entry point, since the LNT immediately follows the header and precedes the code, and also fixes the range of byte values that addresses the LNT: larger byte values address the GNT, after being adjusted downward by the size of the LNT.

Each GNT or LNT entry consists of a it-bit tag and a datum (pointer) whose interpretation depends on the value of the tag. To accommodate the usual organization of memories into words, each NT is organized into blocks of entries: the arrangement for a 36-bit memory, for example, appears below.



The algorithm for computing the location of the i'th name in a N is atually quite simple and only involves addition and shifting. The possible tag values are presented immediately below and discussed in the following paragraphs.

CONST GVAR IVAR FVAR
FNO FN1 FN2 FN3 FN4 FN5 FN6
FN*

Function tag values must include the number of supplied arguments; the datum holds the function name. The tags FNO ... FN6 represent function calls with the most common argument counts. FN* represents a function call with more than 6 arguments: the actual argument count is supplied as the last argument, and the machine removes it before constructing the new frame. The primitive functions APPLY and APPLY* provide the ability to call a function whose name is computed: this ability is not represented directly by a tag value.

The four variable tags represent different strategies for obtaining the value of the variable. All variable references eventually result in pushing the value of the variable onto the end of the current frame extension; a function call severs the appropriate number of arguments from the end of the old frame extension for incorporation in the new basic frame. CONST (CONstant) simply pushes the datum itself. GVAR (Global VARiable) pushes the contents of the value cell of the variable whose name is the datum, or traps if the value cell contains NOBIND. IVAR (Indexed VARiable) does not use the datum: it just pushes the N'th value from the basic frame, where N is the actual byte value. FVAR (Free VARiable) works similarly, but takes the value as a pointer to the true binding; if the pointer has not been set up, a stack search occurs first to find the nearest binding and set the pointer to it.

A few primitive operations, such as returning from a function, cannot be represented by function calls, so a few byte values are reserved for them. These are the only real "opcodes" in MicroLISP. Some of them are followed by displacements or other parametric information in the next byte or bytes; a few (STORE, DSTORE) are followed by an ordinary variable reference which is interpreted specially. The convention followed in the description of the opcodes, and also in the examples in Appendix B, is that upper-case words like STORE represent opcodes; lower-case words represent parameter bytes; upper-case words in [brackets] represent references to functions; lower-case words in brackets represent references to variables.

Data Movement

STORE, [v]
This causes the top value on the stack, A, to be stored. The interpretation depends on the tag of v:

IVAR, GVAR:
The value in the binding is replaced by Z.

FVAR:
The value in the addressed binding is replaced by Z.

CONST:
Error (trap).

FNO ... FN6:
The function is called at its "store" entry point with one more argument than its tag specifies.

FN*:
The function is called at its "store" entry point with one more argument than the count (immediately below *Z* on the stack) specifies.

DSTORE, [v]
Performs the same action as STORE followed by POP.

ADDRX, n1, n2; ADDRXX, n1, n2 n3
These serve to increase the range of addresses. The 2-byte or 3-byte parameter is interpreted as an address in the LNT or GNT as appropriate.

POP
Removes the top item from the stack.

COPY
Pushes the top value on the stack onto the stack, only apparent use is for SELECTQ.

ARG
If N is the top value on the stack (an integer), replaces N by the N'th argument of the function.

SETARG
If Z is the top item and N is the next item (an integer), sets the N'th argument of the function to *Z* and removes N from the stack (but retains z, squeezing N out).

Control

The jump opcodes are followed by a parameter byte, d, which is interprete as a 2's complement address displacement relative to the opcode itself. If positive, d is adjusted by +3 to eliminate meaningless small values. A few values of *6* are reserved to indicate extension into a second byte to provide a larger range of displacements.

JUMP, d
: Always jumps d bytes relative to the instruction.

TJUMP, d
: Tests the top value on the stack and pops it; then jumps if the datum was true (not NIL).

FJUMP, d
: The inverse of TJUMP (jumps if NIL).

NTJOMP, d
: Like TJUMP, but pops the value only if the jump fails (value is NIL). This is for COND's with clauses lacking a consequent, where the value of the test becomes the value of the COND if true.

TYPEJUMP, t, d
: The bottom bits of t give a type number; the top bit of t selects jumping on true or false. The top value on the stack is removed, then jump or no jump depending on its type.

GOTOSELF
: Calls the current function recursively by jumping to its entry point after replacing the arguments, i.e. a PROGITER-type call.

RETURN
: Returns the top value on the stack as the value of the current function.

Conclusions and Comments

MicroLISP programs are consistently one-third to one-fourth the size of BBN-LISP compiled programs, and the NicroLISP compiler is about one-third the size of the corresponding part of the BBN-LISP compiler. Some of the former advantage is due to design decisions in BBN-LISP which result in bulky code: ITS LISP[11], for example, is remored to produce code one-third the size of BBN-LISP or only one-third larger than MicroLISP. However, this compactness is achieved at the expense of many of the attractive features of BBN-LISP: recall the observations about compilers in the introduction, since no MicroLISP machine exists, there are no comparable timing data. However, a microprogrammed implementation and a software interpreter are in preparation.

MicroLISP has been presented as a machine language, but slight additions would permit unambiguous decompilation into the original S-expression for editing. This approach is only feasible in general when the machine language closely resembles the source code: compilers for conventional machines must rearrange and suppress the original program structure extensively to achieve efficient execution. Interpretive systems, of course, generally do reconstruct the source text from an intermediate representation, often using their knowledge of the program structure to advantage (e.g. indenting to indicate depth of logical nesting).

Several factors prompted the author to investigate the type of design just presented. One was the feeling that the constant demands from the Artificial Intelligence community for larger primary memories were based as much on disinclination to spend time contemplating alternatives to traditional machine and program organization as on a real need to deal with larger amounts of information. Another was the hope, based on an earlier experience with a small computer[7], that a LISP minicomputer could provide, at a fraction of the cost, the kind of facilities now available only through large, expensive time-shared installations, A recent product announcement for a desktop BASIC machine is encouraging in this regard.

Realizing this hope for less expensive LISP systems requires compressing the data as well as the program. One approach is to provide facilities for the user to define his own packed data structures; a simple proposal along this line is described in an appendix. Another is to consider "compiling" data in a manner similar to programs. A careful reading of the MicroLISP design reveals that the encoding scheme works on arbitrary lists, not just programs. The essential ideas are:

- Eliminating CDR pointers by forcing logically successive data to be physically consecutive;
- Eliminating non-atomic CAR pointers by associating an operand count with each operator, so the end of a sublist (subexpression) is defined implicitly;
- Compressing atoms by use of tables, on the assumption that some few atoms (different for different contexts) will account for most of the references.

These ideas are applicable, separately or together, to data as well as programs, and offer a partial solution to the "address explosion" problem; the tendency for addresses to become longer and longer as virtual memories become larger, so that one winds up paying for many bits of memory used to hold largely uninteresting links.

Acknowledgements

The idea of using very short instructions and accessing the entire environment through a table originated in the design of the Burroughs B5000. The author was inspired to contemplate the present design by some suggestions of Butler Lampson and Charles Thacker of xerox PARC. The idea of a STORE entry to a function is due to Alan Kay, also of PARC.

Appendix A: Deer-Defined Data structures

The data structure definition facility allows the user to define classes of objects which are essentially generalizations of list cells. List cells have two components, which are pointers; user-defined structures may have any (fixed) number of pointers, integers, and reals (floating point numbers). CONS called with fewer than two arguments fills in the missing components with NIL: the user may specify the default values for his own structures. There are corresponding generalizations of CAR and CDR for extracting components from user structures, and of RPLACA and RPLACD for replacing components.

The user defines a new class of structures by calling

    (STRUCTURE number-of-pointers
    number-of-integers number-of-reals
    initial-value-list).

STRUCTURE returns (a pointer to) a "template" for objects of the new class. The template serves three purposes. First,

    (STRUCPARS template)

returns a list of the arguments to the call of STRUCTURE which created the template. Second, applying the template as a function to a list of component values creates a new object of the class, e.g. if complex numbers are defined by

    (PUTD (QUOTE COMPLEX)
        (STRUCTURE 0 0 2)),

then (COMPLEX 1 -1) would create the complex number 1-i. Third, there is a function

    (STRUCP any-datum)

which returns the template if any-datum is an object from a user-defined class and NIL otherwise.

The generalized extraction function

    (ELTR object component-number first-bit
number-of-bits)

returns a component selected by positions components are numbered from 0, first pointers, then integers, then reals. First-bit and number-of-bits are only legal if the component is an integer; if omitted, a full word is fetched. The corresponding replacement function is

    (SETFQ (ELTR ...) value),

consistent with the MicroLISP notion of "load" and "store" entries to a function- For efficiency,

    (ELTFN template component- first-bit
    number-of-bits)

returns a function f such that

    (f object)

is equivalent to

    (ELTR object component-number first-bit
    number-of-bits)

provided that the object is of the class given by the template, or at least of a class whose components up to and including the specified one all have the same types as the corresponding components of the class given. In MicroLISP, the function f is of a special data type called "selector" which works as efficiently as CAR and CDR; CAR is actually implemented as
ELTFN (STRUCTURE 2) 1) and CDR as
(ELTFN (STRUCTURE 2) 0).

Appendix ,B; 16- AND 32-Bit Words

It is always awkward to implement systems involving pointers on machines with 16- or 32-bit words, since 16 bits is not quite enough for a pointer but 32 is too many. However, a slightly different application of the basic idea of MicroLISP (the use of statistical knowledge about the topology of data structures to reduce the number of bite required to represent them) can produce a useful 256K address space on a 32-bit machine. The idea is to make *u* subspaces, each of 64K (requiring 16-bit pointers), and using global conventions to supply the subspace number when following any given pointer.

The software MicroLISP implementation currently under construction uses the following subspaces: (A) stack; (B) strings, atom print-names, and the atom hash table; (C) arrays and compiled code; (D) lists, atom heads, and other descriptors. The subspace number for pointers from each of these areas is supplied as follows:

Stack
: The "control link" and "access link" refer to space (A); the "resumption point" carries an explicit subspace designator, since it may refer to an S-expression (space (D)), compiled code (space (C)), or machine code; all other pointers are to space (D).

Strings, print-names
: There are no pointers in these spaces.

Atom hash table
: All pointers are to atoms, in space (D).

Arrays, compiled code
: All pointers are to space (D).

Lists
: All pointers are to space (D).

Atom heads
: CAR (value cell) and CDR (property list) are to space (D); the definition carries an explicit subspace designator, for the same reason as the resumption point on the stack; the print-name pointer is to space (6).

String descriptors
: These point to space (B).

Environment descriptors
: These point to space (A).

Array descriptors
: These point to space (c).

This scheme works as long as the number of different arrays, environments with descriptors, and strings is not too large. When these numbers become large, a great deal of space (D) becomes devoted to uninteresting descriptors.

These examples compare the s-expression, the MIcroLISP code, and the PDP-10 code produced by the present BBN-LISP compiler. The MicroLISP code assumes that a pointer occupies 2 bytes and that *H* bytes fill a word. The size figures for the compiled codes do not include 1 word of header for MicroLISP and 2 words for BBN-LISP respectively.

```
****** REVERSE ******

S-expression: 36 LISP cells
LAMBDA (X)
    (PROG (Y)
    LP (COND ((NLISTP X) (RETURN Y)) )
          (SETQ Y (CONS (CAR X) Y))
          (SETQ X (CDR X))
          (GO LP)
    ))

MicroLISP:  26 bytes
Tags:
    IVAR IVAR [+ padding: 4 bytes]
Names:
    X Y [total 4 bytes]
Code:
    (lp)
    [x]: TYPEJUMP, listp, a; [y]: RETURN
    (a)
    [x]; [CAR]; [y]; [CONS]; STORE, [y]
    [x]; [CDR]; STORE, [x]; JUMP, lp

PDP-10 compiled code: 22 words
    REV1
         (JSP 7 , ENTERF)
         (262144 0)
         (0 PLITORG)
    t1   (PUSH PP , KNIL)
    LP   (HRRZ 1 , X)
         (PSTN1 LISTT)
         (JRST t4)
    t5   (HRRZ 1 , Y)
         (JRST t3)
    t4   (HRRZ 1 , X)
         (HRRZ 1 , 0 (1))
         (HRRZ 2 , Y)
         (PUSHJ CP , CONS)
         (HRRM 1, Y)
         (HRRZ 1 , X)
         (HLRZ 1 , 0 (1))
         (HRRM 1 , X)
         (JRST LP))
    t3   (SUB PP , BHC 1)
    t2   (POPJ CP ,)
    LITORG
    PLITORG
         X
         Y

****** SUBST ******

S-expression: 41 cells
(LAMBDA (X Y Z)
    (COND
    ((NLISTP Z)
         (COND ((EQ Z Y) X) (T Z)) )
    (T (CONS (SUBST X Y (CAR Z)) (SUBST X Y
    (CDR Z)))) )
    ))
```

```
Byte LISP: 37 bytes
Tags:
    IVAR IVAR IVAR FN3 [+ padding: 4 bytes]
Names:
X Y Z SUBST [total 8 bytes]
Code:
    [z]; TYPEJUMP, listp, a
    [y]; [z]; [EQ]; FJUMP, b; [x]; RETURN
    (a)
    [z]; RETURN
    (b)
    [x]; [y]; [z]; [CAR]; [SUBST]
    [x]; [y]; [z]; [CDR]; [SUBST]; [CONS];
    RETURN

PDP-10 compiled code: 39 cells
    SUBST
         (JSP 7 , ENTERF)
         (786432 0)
         (0 PLITORG)
    t1   (HRRZ 1 , Z)
         (PSTN1 LISTT)
         (JRST t3)
         (HRRZ 1 , Z)
         (HRRZ 2 , Y)
         (CAME 1 , 2)
         (JRST t4))
         (HRRZ 1 , X)
         (JRST t5)
    t4   HRRZ 1 , Z)
    t5   (JRST t6)
    t6   (HRRZ 1 , X)
         (PUSH PP , 1)
         (HRRZ 1 , Y)
         (PUSH PP , 1)
         (HRRZ 1 , Z)
         (HRRZ 1 , 0 (1))
         (PUSH PP , 1)
         (CCALL 3 , ' SUBST)
         (PUSH PP , 1)
         (HRRZ 1 , X)
         (PUSH PP , 1)
         (HRRZ 1 , Y)
         (PUSH PP , 1)
         (HRRZ 1 , Z)
         (HLRZ 1 , 0 (1))
         (PUSH PP , 1)
         (CCALL 3 , ' SUBST)
         (MOVE 2 , 1)
         (POP PP , 1)
         (PUSHJ CP , CONS)
    t6
    t2   (POPJ CP ,)
    LITORG
    PLITORG
         X
         Y
         Z
         SUBST
```

eferences

I Daniel G. Bobrow, Ben Wegbreit
A Model and Stack Implementation of Multiple Environments
BBN Report #2334
March 1972

' Burroughs B5500 Information Processing System Reference Manual
Burroughs Corporation
196*

3 computer Operations Incorporated
GEMINI Reference Manual
(distributed internally)

4 J. Moore, D. Steingart, H.R. zaks
A Firmware APL Time-Sharing System
1971 SJCC

5 Clark Weissman
LISP 1.5 Primer
Dickenson Publishing company
1967

• R. Rice et al.
SYMBOL - A Major Departure from Classic Software-Dominated Von Neumann Computing Systems
1971 SJCC

7 E.C. Berkeley, L.P. Deutsch
The LISP Implementation for the PDP-1 Computer in the Programming Language
LISP: Its Operation and Applications
M.I.T. Press
1966

• Hewlitt-Packard Corporation
advertisement in Scientific American
Feb. 1973

9 John McCarthy
Recursive Functions of Symbolic Expressions and Their Computation by Machine
Communications of the ACM
April 1960

10 John McCarthy et al.
LISP 1.5 Programmer's Manual
M.I.T. Press
1962

11 M.I.T. Project MAC Artificial Intelligence Laboratory
no available reference

12 E.A. Hauck, B.A. Dent
Burroughs B6500/B7500 Stack Mechanism
1968 SJCC

13 W. Teitelman et al.
BBN-LISP TENEX Reference Manual
Bolt Beranek and Newman Inc.
Cambridge, Mass.
latest revision: Feb. 1972

14 J.F. Rulifson, J.A. Derkson, R.A. Waldinger
QA4: A Procedural Calculus for Intuitive Reasoning
Artificial Intelligence center Technical Note 73
Stanford Research Institute
Menlo Park, California
Nov. 1972