TINY HI

by Martin Buchanan

Dear *DDJ*, Sept. 2, 1976

For the past eight months, I've worked on designing my own programming language. I've now conceived several clean and fairly powerful procedural languages. The simplest is described in the attached article. I also expect to specify TINY LISP, and TINY SNOBOL in coming months. Core is cheap compared to people, and the major economy I want is in syntax, number of data types, and number of data structures seen by the user. One key to such economy is the elimination of the *multilevel approach* and the *specialization* that characterizes much existing system software. Software that is "OS dependent" is even less portable than machine code on many systems. I don't want the user to have to learn assembler, editor, JCL, file handling packages, etc. I want him to see conceptually, and syntactically integrated systems.

During the latter part of my spinning of imaginary languages, I've become dissatisfied with my previous definition of HI, as a procedural language which is both cleaner and more useful than PL/1, and incorporating associative retrieval, pattern matching, and list processing facilities. Several ideas have changed my direction:

1. A tiny language (or any language) should be *extensible.*
2. Most existing extensible languages (LISP, TRAC, FORTH) are limited by primitive syntax.
3. The concept of "set" or "relation" is powerful enough to include all the diverse data structures found in computer applications, and to manipulate them with common tools.
4. A further equivalence can be established between relations and functions; one is defined extensionally; the other is defined operationally.
5. The relational calculus is a powerful form for manipulating such structures. In the past, it has been restricted to data base operations (Codd's DSL ALPHA).
6. Declarations make it feasible for the user to see only a single data structure, ignoring the different *internal* representations of structures.

I'm attempting to design an extensible relational language with a context-sensitive grammar. Work is presently stalled by a 50-hour work week, 17 semester-hours of school, my home computer, and a promised article.

FORTH is a fascinating new language ($1000 for 8080 systems). I've ordered manuals, and will report on what I find out.

DDJ is great; things like your floating point routines for the 6502 (which MOS Technology did not have available) really help.

Design for the future! In 5 years your home computer will be a 64K memory, 16 bit dynamically microprogrammable processor, four billion bytes of fast mass storage, and goodies I can't imagine.

Keep up the good work.

Martin Buchanan
2040 Lord Fairfax Rd.
Vienna, VA 22180
(703) 893-7978

HI is a family of general purpose programming languages designed in 1976 by Martin Buchanan. TINY HI (TINY) is the simplest version of HI designed, and will be implemented on the author's IMSAI 8080 system when I find the time.

TINY is a hybrid interpreter-compiler running on a dedicated virtual machine with virtual memory. TINY is completely structured. There are no GOTO statements or labels. There is no COMMON or RETURN; procedures have one entrance and one exit. TINY is a debugging translator, insuring that variables are defined, monitoring subscript range, etc.

TINY supports two data types, INTEGER, and STRING, and one data structure, the vector. Memory is allocated dynamically, and data type is determined at execution time, or by context, eliminating declarations.

The simplicity of TINY syntax, the small number of well defined operators, the ability to manipulate vectors, convenient input-output, and flexible commenting and mnemonic naming facilities, all combine to make TINY a very clean, easy to learn and use, language.

In the examples below, both interpretive sessions and programs are excerpted. TINY outputs are underscored. ↓ is used to indicate a carriage return.

Arithmetic:

Numbers in TINY are integers i: $-2^{15} \leq i \leq 2^{15}-1$

4 infix arithmetic operators are defined:

+ addition
- subtraction.
* multiplication
/ division

Division by zero or overflow will produce an error message.

1 prefix arithmetic operator is defined:

- negation

Negation cannot produce an error.

The hierarchy of operators in TINY is:

highest	()	[]	nesting or subscripting
	ƀ		(Blank) concatenation
	#	(number)	- (negation)
	*	/	
	+	–	
lowest	=	< >	<= >= <>

Expressions are evaluated from left to right.

Vectors:

Numbers may be concatenated to form vectors:

```
A ← 1 2
A ← A 3 4
A
    1   2   3   4
```

The left pointing arrow, ←, is assignment. Any expression standing alone on a line is output. Vectors of equal length may be combined in expressions:

```
1 2 3 + 4 5 6
     5   7   9
```

Vectors of length one may be added, subtracted, etc. to another vector and be "distributed":

```
1 + 4 5 6
     5   6   7
```

The prefix operator # (number) returns the number of elements in a vector or string:

```
#5 73 -1
     3
```

Both integer vectors and character vectors (strings) are limited to 255 elements.

Strings:
A string is a sequence of characters, other than ", enclosed in quotes:

"THIS IS A STRING"

A string may contain no characters:

" "   /* the null string */

Strings may be concatenated:

```
A ← "JOHN"
B ← "AND"
C ← "MARY"
A  B  C
JOHNANDMARY
```

Subscripting:
Vectors and strings can be subscripted by integer expressions:

```
A ← "ALGEBRA"
A[5 1 3 2]
BAGL
```

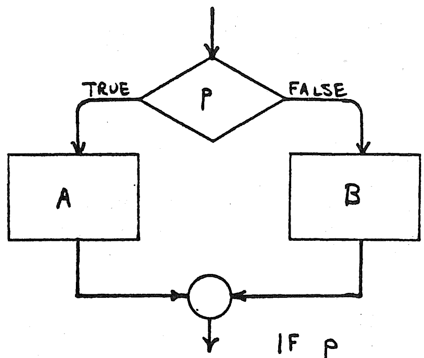
Note that: x[i j k] is equivalent to:
              x[i] x[j] x[k]

Global variables:
Globals are available to all TINY programs, and normally exist before and after program execution. I/O devices, data files, programs, and convenient system constants (.TRUE) are all good globals. All variables defined in the interpreter are globals. Globals are distinguished by use of the period prefix when referenced:

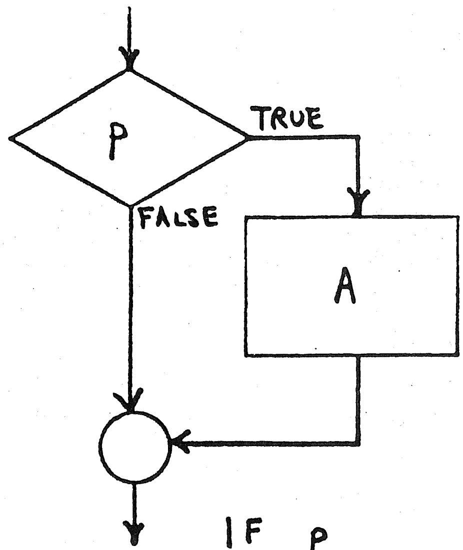
.SIN
.EMPLOYEE-FILE
.PRINTER

Programs:
TINY programs have only one entrance and one exit. Parameters are passed by location. Assignment to parameters is prohibited within a program. Programs can return vector or string values, and are invoked by use in

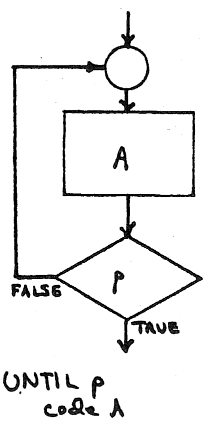
TINY CONTROL STRUCTURES



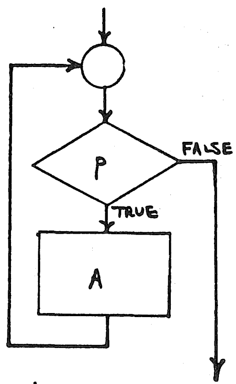

```
IF p
   code A
ELSE
   code B
END
```




```
IF p
   code A
END
```




```
UNTIL p
   code A
END
```




```
WHILE p
   code A
END
```

an expression. Sample program:

```
BEGIN MOD (A,B)
/*
  RETURN THE REMAINDER OF A/B
*/
  MOD ← A—B*(A/B)
  IF MOD<0
    MOD ← —MOD
  END
END

BEGIN GCD              /*
    ?X ?Y                USE EUCLID'S ALGOR-
  IF X <Y                ITHM TO FIND GREAT-
    T ← X                EST COMMON FACTOR
    X ← Y                OF X AND Y.
    Y ← T              */
    T ←
  END IF
  R ← Y
  WHILE R > 0
    R ← .MOD (X,Y)
    X ← Y
    Y ← R
  END WHILE
  "GCD=" Y
END GCD
```

The program GCD has several unusual features. ?X ?Y forces the evaluation of the expression ?X ?Y, which inputs X and Y, without making any assignment. T ← deletes T, allowing the compiler to reuse the allocated space. As shown by END IF, END WHILE, and END GCD, "noise" may be added to END statements; normally this is used to indicate the structure ended. /* begins the reservation of all columns to the right of, and including /* for comments, until */ is encountered. In TINY it is easy to implement Gerard Weinberg's concept of placing comments on the right hand side of the page, to be covered during debugging.

The program as shown is unusual; except in introductory programming classes, main programs do not calculate greatest common factors; also the mnemonic name is inconsistent with the use of "greatest common *factor*," rather than "divisor." To rewrite the program as a function, it should also be changed so that the values of the parameters are not changed*:

*This change is now *required* for program correctness reasons.

```
BEGIN .GCF (XD,YD)     /*
  X ← XD                 USE EUCLID'S ALGOR-
  Y ← YD                 ITHM TO FIND GREAT-
  IF X <Y                EST COMMON FACTOR OF
    T ← X                XD AND YD.
    X ← T              */
    Y ← T
    T ←
  END IF
  R ← Y
  WHILE R > 0
    R ← .MOD(X,Y)
    X ← Y
    Y ← R
  END WHILE
END .GCF
```

Flow of control:

TINY control is always based on an expression which is true or false, called a *predicate.* TINY does not include logical variables or operators. All TINY predicates are of the form:

<exp > <relational > <exp>

The two expressions must agree in type. Vectors or strings must be of equal length to be equal. The first element of a string or vector is most significant when evaluating a predicate; the last element is least significant. TINY has two basic control structures with a total of four variants:

Do one thing or another depending on a predicate;
Repeat something until a predicate changes.

These structures are illustrated in the accompanying figures.

Input-output:

To output an expression, simply place it on a separate line. If a string expression, the string is output. Integers are converted to a 6 character string with leading zeroes replaced by blanks and any minus sign to the immediate left of the most significant digit. In output, concatenation between integer and string expressions is allowed:

```
"A+B=" (A+B)
```

To input a variable, *anywhere,* place a question mark before the variable. Input is free format. Input and output may be combined in one statement, like the INPUT statement of some BASICs:

```
"A=" ?A
A=37
```

Operating System (OS):

TINY is its own operating system. Users run on a dedicated machine (their own micro) or a virtual dedicated machine. Programs are automatically compiled when defined and recompiled whenever source code is changed. TINY will include access to a virtual machine level where relocatable and reentrant machine code may be written using an assembler. This level cannot be defined until subroutine linkages and data formats are defined in greater detail. TINY has a virtual memory; the user sees a single large homogeneous memory space. Files, programs, utilities, and system variables are all represented as global variables in this memory.

Conclusion:

I'd appreciate feedback about TINY design and implementation. My address is at the end of this article.

Program correctness:

It is straightforward to construct correctness proofs for TINY programs. Relevant features include:

Restricted control structures;
All programs have only a single entrance and exit;
Programs return only a single structure;
Parameters may not be altered by programs;
No floating point arithmetic (10.0*.1≠ 1.0);
External variables share a common name table, and must be explicitly indicated with the . prefix;
The deletion feature explicitly delimits the range of

temporary variables;

The simplicity of the language.

These features do not follow from the desire for a small language. GOTOs would be easy to add, and checking for assignment to parameters will actually cost core. I maintain that one should not add features to a language indiscriminantly, considering only available core or CPU time. Psychological factors can also make a language good or bad. PL/1, APL, "SUPER FORTRAN", "BASIC PLUS" are all baroque languages, whose descriptions occupy sizeable books and whose complete features are rarely mastered.

"I see a great future for very systematic and very modest programming languages." — *The Humble Programmer* by E. W. Dijkstra

TINY LANGUAGE SUMMARY

Vocabulary: BEGIN END IF ELSE WHILE UNTIL
Comments: /* */
Arithmetic infix operators: + — * /
Arithmetic prefix operators: -
Concatenation: ƀ (blank)
Length operator: # (number)
Relational operators:
Assignment:
Input: ?

Global: .
Data types: INTEGER, STRING
Data structures: the vector
Maximum number of elements in a vector: 255
Formats: Integers convert to a six character string

TINY ERROR MESSAGES

compile:
- assignment to parameter
- mixed mode
- value undefined
- global undefined
- unmatched parenthesis
- missing end statement
- syntax error

execution:
- overflow
- division by zero
- vector too long
- string too long
- subscript out of range

Direct letters about TINY HI to: Martin Buchanan
2040 Lord Fairfax Rd.
Vienna, VA 22180

TINY HI SYNTAX

Top Down

```
<PROGRAM>   ::= BEGIN <NAME> { ( <NAME> { , <NAME> }_0^∞ ) }_0^1 ↵
                    { <PROGRAM> }_0^∞
                    <XCODE>
                <END>
<XCODE>     ::= { <XSTMT> }_0^∞
<XSTMT>     ::= <IF> | <LOOP> | <ASNMT> | <IO>
<END>       ::= END ƀ <ANYTHING> ↵
<IF>        ::= IF ƀ <PREDICATE> ↵
                    <XCODE>
                { ELSE ↵
                    <XCODE> }_0^1
                <END>
<LOOP>      ::= <LOOP TYPE> ƀ <PREDICATE> ↵
                    <XCODE>
                <END>
<LOOP TYPE> ::= WHILE | UNTIL
<ASNMT>     ::= <VARIABLE> ← { <EXPR> }_0^1 ↵
<IO>        ::= <EXPR> { ƀ <EXPR> }_0^∞ ↵
```

TINY HI SYNTAX

Bottom Up

```
<LETTER>      ::= A | B ··· | Z
<DIGIT>       ::= 0 | 1 ··· | 9
<ALPHA>       ::= <LETTER> | <DIGIT> | _
<INTEGER>     ::= { - }_0^1 { <DIGIT> }_1^5
<OPERATOR>    ::= + | - | * | /
<RELATIONAL>  ::= > | < | = | >= | <= | <>
<NAME>        ::= { ? }_0^1 <LETTER> { <ALPHA> }_0^254 |
                  { . }_0^1 <LETTER> { <ALPHA> }_0^254
<VARIABLE>    ::= <NAME> { [ <INTEGER EXP> ] }_0^1
<INTEGER EXP> ::= <INTEGER> | <VARIABLE> | <FUNCTION> |
                  # <EXPR> | - <INTEGER EXP> |
                  <INTEGER EXP> <OPERATOR> <INTEGER EXP> |
                  <INTEGER EXP> ƀ <INTEGER EXP>
<FUNCTION>    ::= <NAME> ( <EXPR> { , <EXPR> }_0^∞ )
<EXPR>        ::= <INTEGER EXP> | <STRING EXP>
<STRING EXP>  ::= <STRING> | <VARIABLE> | <FUNCTION> |
                  <STRING EXP> ƀ <STRING EXP>
<PREDICATE>   ::= <INTEGER EXP> <RELATIONAL> <INTEGER EXP> |
                  <STRING EXP> <RELATIONAL> <STRING EXP>
```