LFCS

Laboratory for Foundations of Computer Science
Department of Computer Science - University of Edinburgh

DIALOG: A Theorem-proving Environment designed to unify Functional and Logic programming

by

Mark Tarver

ECS-LFCS-89-80
(also published as CSR-298-89)

LFCS Report Series

May 1989

LFCS
Department of Computer Science
University of Edinburgh
The King's Buildings
Edinburgh EH9 3JZ

Copyright © 1989, LFCS

DIALOG: A Theorem-Proving Environment Designed To Unify Functional And Logic Programming

Mark Tarver,
mt@uk.ac.ed.lfcs,
LFCS, Department Of Computer Science, University Of Edinburgh.

April 10, 1989

DIALOG is a prototype logic programming language designed to facilitate the construction of interactive theorem-proving systems in the medium of LISP. DIALOG has two principal parts:-

1. A functional programming component that allows the user to enter functional definitions in a high-level pattern-matching notation reminiscent of HOPE and ML.
2. A logic programming component that runs on many-sorted logic and which will be capable of coping with queries up to many-sorted first-order logic.

Functional Programming In DIALOG

The functional component of DIALOG allows the user to construct pattern-directed invocations of function calls. The form of DIALOG function definitions is:-

```
(defML <identifier>
  <pattern_1> → <action_1>
  .....
  <pattern_n> → <action_n>)
```

The DIALOG interpreter tracks through each <pattern$_i$ > until it finds a match and then it executes the corresponding action. A typical DIALOG functional definition is the *occurs check* function occcheck which checks for the occurrence of an element in a given list, and returns nil if it finds it and t if it does not.

```
(defML occcheck
  _ atom? → t
  x [x : y] → [ ]
  x [atom? : y] → (@ occcheck x y)
  x [y : z] → (@ and (@ occcheck x y) (@ occcheck x z)))
```

The colon is the list seperator in DIALOG which takes the place of | in PROLOG; atom? is a sorted variable that matches only atoms. @ indicates function application while _ is the wild card symbol, as in PROLOG and ML, that matches any term. In LISP, the occcheck function would appear as:-

```
(defun occcheck (X Y)
  (cond ((atom Y) t)
        ((equal X (car Y)) nil)
        ((atom (car Y)) (occcheck X Y))
        (t (and (occcheck X (car Y)) (occcheck X (cdr Y))))))
```

The DIALOG compiler receives DIALOG function definitions and compiles them down into low-level LISP code.

DIALOG recognises as a variable either x y z or any atomic symbol ending in ?. Variables themselves are either *unsorted* or *sorted.* A sorted variable is a variable which is lambda-bound as the identifier of a LISP predicate - which itself can be either hand-written as part of a possibly foreign LISP program or DIALOG-generated from a defML definition. Sorted variables only match to their respective arguments if these arguments satisfy the relevant predicates. Sorted variables need not be declared as such to the DIALOG compiler since the compiler will check to see if variables are lambda-bound during compilation. In certain cases, it is necessary to *declare* variables as sorted to the compiler because there is no ordering of definitions which will issue in the appropriate code.

```
(defML propwff1?
  x → (@ propwff? x))

(defML propwff2?
  x → (@ propwff? x))

(defML propwff?
  atom? → t
  [~ propwff?] → t
  [propwff1? => propwff2?] → t
  [propwff1? & propwff2?] → t
  [propwff1? V propwff2?] → t
  [propwff1? <=> propwff2?] → t
  _ → [ ])
```

In the example above propwff? is a defML definition which uses its own identifier as a sorted variable. *Per impossible* propwff? cannot be defined before its own definition so there is no way of registering propwff? as sorted by lambda-binding. Instead propwff? can be declared as sorted by entering it as such in the environment header of a DIALOG file.

(environment (sorted propwff? ...))

Context Sensitive Pattern Recognition

DIALOG supports *context-sensitive* pattern recognition. Context sensitive pattern-recognition is required when pattern-matching cannot be decided on the basis of applying one-place predicates to elements of the input, but requires the use of at least two-place predicates in the pattern matching stage. This requirement often arises when the issue of whether a term fits the pattern is determined by terms that have preceded it. DIALOG permits the embedding of the appropriate function call in the pattern itself to decide context-sensitive issues. The top-level of the unify function illustrates use of embedded function-calls to extend DIALOG to context-sensitive pattern-recognition.

```
(defML unify
  x x → t
  var? x (@ occcheck var? x) → (@ setq frame [[var? x] : frame])
  x var? (@ occcheck var? x) → (@ setq frame [[var? x] : frame])
  list1? list2? (@ = (@ length list1?) (@ length list2?))
                              → (@ unifyl list1? list2?))
```

Here the requirement that unification satisfy the occur-check and that unification between two lists can only succeed if the two lists are of equal length are encoded as context-sensitive features in the pattern-matching process. The rule on introducing function calls on the LHS of a defML pattern-action pair is that any variables in the function call must precede the function-call itself. Thus

```
(@ occcheck var? x) var? x → (@ setq frame [[var? x] : frame])
```

is not well-formed, since var? and x are used without being introduced prior to their use within the function-call.

The Logic Programming Component of DIALOG

DIALOG supports logic programming through its use of defLOG definitions. These definitions are used to enter Horn-clauses to the DIALOG interpreter. The form of a defLOG definition is:-

```
(defLOG <identifier>
  <head_1 > ← <body_1 >
  .....
  <head_n > ← <body_n >)
```

For instance the definition of APPEND in the defLOG syntax would be:-

```
(defLOG APPEND
  [ ] x x ←
  [x : xs?] y [x : z] ← (APPEND xs? y z))
```

In the Edinburgh standard syntax this would be written as:-

```
append([ ],X,X).
append([X|XS],Y,[X|Z]) :- append(XS,Y,Z).
```

Two reasons determined the departure of DIALOG from the Edinburgh standard syntax. First, it was thought desirable that all DIALOG definitions should have a similar syntax. This syntax is LISPlike in character and reflects the origens of the DIALOG interpreter. Secondly DIALOG syntax is clearer and shorter than the Edinburgh standard syntax for PROLOG. The relevant predicate is cited only once instead of being repeated. This requires the user to group clauses belonging to the same logical procedure together which is good programming practice. The prevalence of commas and stops in PROLOG, which was shed at an early stage in the development of LISP, is likewise shed in DIALOG.

Unlike PROLOG, function symbols in DIALOG defLOG definitions can be genuine carriers of function-calls. The following program, one of the first written for the DIALOG interpreter, defines a semantic tableau theorem-prover PROP for the propositional calculus. The interplay between DIALOG's functional and logic programming components is reflected clearly in this simple program which takes only 67 lines of DIALOG.

```
(defML propwff?
  atom? → t
  [∼ propwff?] → t
  [propwff1? & propwff2?] → t
  [propwff1? V propwff2?] → t
  [propwff1? => propwff2?] → t
  [propwff1? <=> propwff2?] → t
  [[All var?] propwff?] → t
  [[Some var?] propwff?] → t
  - → [ ])

(defML propwff1?
  propwff? → t
  - → [ ])

(defML propwff2?
  propwff? → t
  - → [ ])
```

```
(defML propwffs?
  [ ] → t
  [propwff? : y] → (@ propwffs? y)
  _ → [ ]

(defML contradictory?
  [ ] → [ ]
  [x : y] → (@ or (@ contradictory1? x y) (@ contradictory? y)))

(defML contradictory1?
  _ [ ] → [ ]
  x [[~ x] : _] → t
  [~ x] [x : _] → t
  x [_ : y] → (@ contradictory1? x y))

(defML literal?
  atom? → t
  [~ atom?] → t
  _ → [ ])

(defML consistent?
  [ ] → t
  [literal? : y] → (@ consistent? y)
  _ → [ ])

(defML hasdisjunct?
  [ ] → [ ]
  [[_ V _] : _] → t
  [_ : y] → (@ hasdisjunct? y))

(defML splitdisjunct1
  [ ] → [ ]
  [[x V y] : z] → [x : z]
  [x : y] → [x : (@ splitdisjunct1 y)])

(defML splitdisjunct2
  [ ] → [ ]
  [[x V y] : z] → [y : z]
  [x : y] → [x : (@ splitdisjunct2 y)])
```

```
(defML mk-canonical
  [ ] → [ ]
  [[x & y] : z] → [x y : (@ mk-canonical z)]
  [[x => y] : z] → [[[~ x] V y] : (@ mk-canonical z)]
  [[x <=> y] : z] → [[x => y] [y => x] : (@ mk-canonical z)]
  [[~ [~ x]] : z] → [x : (@ mk-canonical z)]
  [[~ [x & y]] : z] → [[[~ x] V [~ y]] : (@ mk-canonical z)]
  [[~ [x V y]] : z] → [[~ x] [~ y] : (@ mk-canonical z)]
  [[~ [x => y]] : z] → [x [~ y] : (@ mk-canonical z)]
  [[~ [x <=> y]] : z] → [[[x & [~ y]] V [y & [~ x]]] : (@ mk-canonical z)]
  [x : y] → [x : (@ mk-canonical y)])

(defLOG SEMANTIC_TABLEAU
  propwff? propwffs? ← (BUILDTREE [[~ propwff?] : propwffs?]))

(defLOG BUILDTREE
  contradictory? ←
  consistent? ← (OUTPUT invalid)
  hasdisjunct? ← (BUILDTREE (@ splitdisjunct1 hasdisjunct?))
                 (BUILDTREE (@ splitdisjunct2 hasdisjunct?))
  x ← (BUILDTREE (@ mk-canonical x)))
```

The logic programming component of PROP is expressed in the final two defLOG definitions. Free use is made of sorted variables to shorten the program and help make it clearer. Thus SEMANTIC_TABLEAU receives as terms a propwff? and a list of propwffs?, and passes control to BUILDTREE having negated propwff? and added it to propwffs?.

BUILDTREE solves goals by showing that nodes of the tableau tree are contradictory?. contradictory? is of course a sorted variable which is defML-defined near the top of the program. If a node is found to be consistent? then control is passed to OUTPUT which always returns to the top-level after printing its argument. If the node contains a disjunct then splitdisjunct1 and splitdisjunct2 split the node around the disjunct to generate two daughter nodes which are in turn passed to BUILDTREE. Finally mk-canonical is a function that puts each propwff? into canonical form by applying the usual well-known laws - *deMorgan, double negation* etc.

Loop Detection In DIALOG

One outstanding deficiency of PROLOG as a theorem-prover is that it will easily enter loops, if given an unsuitable set or ordering of clauses. Sometimes PROLOG will behave in this manner even if an answer to the user's query is deducible from the clauses held in memory. The simplest way to produce this behaviour is to enter a symmetry axiom.

```
brother(X,Y) :- brother(Y,X).
```

The query brother(dick,tom)? will cause the following (endless) series of goals:-

```
brother(dick,tom),
brother(tom,dick),
brother(dick,tom),...
```

This form of looping is by far the most common in PROLOG programs, Sterling and Schapiro [35] refer to such rules as *left-recursive.* A rather rarer variety is exemplified in the following example:-

```
jewish(bar-hillel)?
jewish(X) :- jewish([mother,X]).
```

Here the query causes a different, but also endless, series of goals to be generated:-

```
jewish(bar-hillel),
jewish([mother,bar-hillel]),
jewish([mother,[mother,bar-hillel]]),...
```

In the symmetry case, the path of the computation repeats itself in a cyclic fashion, whereas the second case there is no such repetition. It is useful and important to distinguish between these two kinds of looping so we can call the first *cyclic looping* and the second *non-cyclic looping.*

There can be no effective procedure for coping with all looping: i.e. no decision procedure that would tell us that a Horn-clause program was going to loop endlessly. Given that pure PROLOG is Turing-equivalent[39] this would entail the solvability of the Halting Problem. However it is possible to detect and allow for cyclic looping. Loop detection is either *static*, if it is performed at assertion time by the analysis of the logic program, or *dynamic*, if it is performed at run-time by the interpreter.

DIALOG uses dynamic loop detection. The DIALOG interpreter maintains a history of the search process which is lambda-bound throughout the computation. The history is stored in an extremely compacted form and does not, as may be thought, increase in size with each new lambda-binding. DIALOG checks this history with after each successful unification to determine if a goal has called itself. If so, the continuation is to fail the next call and to backtrack. Loop detection functions in linear time relative to the length of the branch of the search-tree currently being extended.

Unlike PROLOG, DIALOG is forgiving of the way in which clauses are ordered and written. This makes possible a more declarative approach to logic programming and to experiment with programming techniques that the PROLOG programmer would not touch. For instance, suppose that we wished to implement a theorem-prover for modal logic in which procedures MODTAC and FOLTAC are used. MODTAC applies tactics appropriate to modal logic problems, and FOLTAC tactics appropriate to FOL

problems. We want MODTAC to be able to give and receive control from FOLTAC. The natural way to implement this is to have MODTAC call FOLTAC, and then have FOLTAC call MODTAC after it has finished breaking down the problem to a more manageable form. In a given proof, control may pass between MODTAC and FOLTAC several times before a fixpoint relative to both is generated. The natural way to encode this is:-

<MODTAC clauses>
((MODTAC problem?) ← (FOLTAC problem?))
<FOLTAC clauses>
((FOLTAC problem?) ← (MODTAC problem?))

However in PROLOG, this form of encoding is an invitation to disaster since there is an indirect left-recursion in the relations betwen FOLTAC and MODTAC. The consequence is that a great deal of thought has to be expended on seeing that MODTAC and FOLTAC interrelate properly. In DIALOG , this "ball-passing" form of programming is perfectly safe since THORN will fail a call when one procedure fails to effect a change in the problem. DIALOG allows *tactical composition*; the layering of tactical procedures upon one another in a way that PROLOG does not support.

User Interaction In DIALOG

One of the main goals of DIALOG was to slash the time taken to develop theorem-provers. Since interactive capability is considered a prerequisite in modern theorem-proving implementations; some care was given to providing DIALOG with an *inbuilt* interactive capability so that any DIALOG implementation would *automatically* be provided with an interactive front end to the users own creation.

This was done through recognising that the concept of an *advanced interface* to a post-PROLOG logic programming language could be identified with a *front end* to a theorem-prover. The principle difference between a trace package in orthodox languages and a front end to a theorem-prover, is that a trace package lets you view a process without giving you the capability to interfere with it. The DIALOG trace package gives this capability and so the difference disappears.

If the user types in the command (interact <identifier> ... <identifier>), where each <identifier> is the name of a defLOG definition, then the DIALOG interpreter will pause after each successful unification involving one of the identifiers, print the goalstack and the rule being used and query the user as to what action is desired. A typical step in the user interaction might look like this:-

Step 1
Rule Applied: (SEMANTIC_TABLEAU propwff? propwffs?)
← (BUILDTREE ((~ propwff?) propwffs?))
Problem Becomes: (BUILDTREE ((~ (p V q)) p q))
What Now?

A request for help will cause the following display to appear.

```
f/F ..... continue forwards
b/B ..... backtrack (fail the current goal)
ff/FF <number> .....fast forward <number> inferences
fb/FB <number> ..... fast back to step <number>
a/A ..... abort (return to DIALOG top level)
see/SEE ..... see the proof so far
ev/EV <expr> ..... evaluate <expr>
```

The DIALOG interpreter will respond appropriately.

The command (uninteract <identifier> ... <identifier>) will make each <identifier> function non-interactive.

SLAM

The heart of the defLOG interpreter is a virtual machine for efficiently processing defLOG definitions. This machine is SLAM (the *S*orted *L*ogic *A*bstract *M*achine). An exact description of SLAM is beyond the scope of this paper, but is included in Tarver [40]. At present SLAM processes many-sorted Horn-clause logic, but when complete SLAM will take DIALOG as far as many-sorted first-order logic. As with all virtual machines, SLAM has a small set of recognised virtual machine instructions which in DIALOG are wired to low-level LISP code.

In Tarver [40], SLAM is described in terms of a hierarchy of abstract machines, PAM (the *P*attern Matching *A*bstract *M*achine), a machine called p-WAM (short for *pseudo*-WAM) and SLAM itself. Each machine is more powerful expressively than its predecessor. PAM uses pattern-matching and backtracking. p-WAM can tackle unification and is functionally as powerful as the PROLOG interpreter. SLAM in its entirety will encompass many-sorted FOL with embedded function calls. In order of efficiency, PAM is faster than p-WAM which is faster than SLAM.

In DIALOG, there is only one machine; SLAM. By issuing the appropriate compiler directives the user can tell the DIALOG compiler that he wishes the code to have the functionality of a given machine. Thus the command PAM in the environment header will have the effect of making SLAM behave as if it were PAM. In effect what the compiler does is wire SLAM instructions to a different set of LISP instructions and mask, or ignore, certain SLAM directives.

Performance, Compactness And Portability

DIALOG-generated LISP code from defML definitions is about 2-3 times slower than the equivalent hand-written LISP code. Programs written in the functional subset of DIALOG are however nearly twice as compact as the corresponding LISP program;

showing that writing in DIALOG is programming at a higher level than can be obtained with LISP. DIALOG is bootstrapped; the functional subset of the language occupies only 300 lines of DIALOG code.

Because DIALOG is bootstrapped and the DIALOG interpreter generates very low-level LISP, implementations written in DIALOG are extremely portable. At present DIALOG runs only in Franz Lisp. But an experienced programmer who understands the object code of the DIALOG could port DIALOG to any Lisp system (DEC-10 Lisp, Common Lisp, Zeta Lisp, Scheme etc) within a few days. Disk space allocation is surprisingly small for so powerful a system; 100K will suffice to store the entire LISP source code and a current micro could probably run DIALOG without too much trouble. In Edinburgh we are looking forward to running DIALOG on an Apple Macintosh.

As regards the logic programming component of DIALOG, comparisons with PROLOG are misleading. PROLOG is in many ways a lower-level programming language than DIALOG; automatic loop-detection and proof-maintenance are not free features of the PROLOG interpreter as they are of DIALOG. The fairest test of relative performance was gained by generating LISP code from DIALOG defLOG definitions and bowdlerising it by removing all the extra machinery that PROLOG does not enjoy. The resulting code was 15 times faster than the standard code. The following defLOG definition of MEMBER was tested in this way with MEMBER declared under PAM.

```
(defLOG MEMBER
  x [x : _] ←
  x [_ : y] ← (MEMBER x y))
```

MEMBER ran at 3300 LIPS on a MICROVAX when the object LISP code was run through the Franz LISP compiler. This performance was only 3 times slower than the that of the native LISP member function. It compares well to C-PROLOG which ran at 1200 LIPS on the same problem - nearly three times slower than the DIALOG MEMBER. More recent releases of LISP would certainly improve performance, but so far no comparative tests have been carried out.

Bibliography

[1] Bledsoe W. & Bruell P. — **A Man Machine Theorem-Proving System** Artificial Intelligence 51-72 (1974).

[2] Brough D. R. & Hogger C.J. — *The Treatment Of Loops In Logic Programming* unpublished paper.

[3] Bundy A. — **The Computer Modelling Of Mathematical Reasoning** Academic Press (1983).

[4] Campbell J. A. — **Implementations Of Prolog**, Ellis Horwood 1984

[5] Carlsson M. & Kahn K.M. — *How to Implement PROLOG On A LISP Machine* in Campbell [4]

[6] Chang C. L. & Lee R.C.T. — **Symbolic Logic And Mechanical Theorem-Proving** Pergamon Press, (1973).

[7] Darlington J., Field, & Pull — *The Unification Of Functional And Logic Languages.* in DeGroot & Lindstrom [8] 37-70.

[8] DeGroot D. & Lindstrom G. — **Logic Programming: Functions, Relations, And Equations**, Prentice-Hall (1986).

[9] Elfrink B. & Reichgelt H. — *Assertion-time Inference For Logic-based Knowledge Bases* in Jackson, Reichgelt,

[10] Ernst C.W. & Hookway R.J. — *Proving In The Case Verifier* Machine Intelligence 10, (1982).

[11] Goldberg A. — *Computer Assisted Instruction: the Application of Theorem-proving to Adaptive Response Analysis* IMSS Stanford Technical Report, (1973).

[12] Good D.I. — *Mechanical Proofs About Computer Programs* in **Computing and Mathematical Logic** ed. Hoare & Shepherson, Prentice-Hall (1985).

[13] Hogger C. J. — **Introduction To Logic Programming,** Academic Press (1984).

[14] Hunter R. — **Compiler Design: an introduction in Pascal** Academic Press (1981).

[15] Jackson P. Reichgelt H. & van Harmden F. **Logic-based Knowledge Representation,** MIT Press, Cambridge, Massachusetts, (1988).

[16] Johnson S.J. *Yacc: Yet Another Compiler-Compiler* Bell Laboratories, New Jersey, UNIX Programmer's Manual, Supplementary Documents.

[17] Kaplan S. *A Compiler For Conditional Term Rewriting Systems* in **Rewriting Techniques and Applications** Lecture Notes In Computer Science, Springer-Verlag.

[18] Kluzniak F. & Szpakowicz S. *Prolog - a Panacea?* in Campbell [4]

[19] Knuth D. & Bendix P. *Simple Word Problems in Universal Algebra,* in **Computational Problems In Abstract Algebras** Pergamon Press (1970).

[20] Kornfeld W. *Equality For PROLOG* in DeGroot & Lindstrom [8].

[21] Kowalski R. *Linear Resolution with Selection Function* Artificial Intelligence, 227 - 260 (1971).

[22] Kowalski R. **Logic For Problem-solving**, North Holland (1979).

[23] Lesk M.E. *Lex - a Lexical Analyser Generator* Computing Science Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J., (1975).

[24] Loveland D.W. **Automated Theorem-Proving: A Logical Basis** North-Holland, Amsterdam, The Netherlands, (1978).

[25] Maier D. & Warren S. **Computing With Logic**, Benjamin Cummings (1988).

[26] Malachi Y. *Tablog: A New Approach To Logic Programming* in DeGroot & Lindstrom [8].

[27] McDermott D. V. & Doyle J. *Non-monotonic Logic* Artificial Intelligence 13, 41-72 (1980).

[28] Nakashima J. & Ueda S. *What Is A Variable In PROLOG ?* Proceedings Of The International Conference On Fifth Generation Computer Systems, ICOT, Tokyo, (1984).

[29] Nilsson N.J. **Principles Of Artificial Intelligence** Tioga Publishing Company, Palo Alto, California, (1980).

[30] Poole D.L. & Gibson R.G. *Gracefully Adding Negation And Disjunction To PROLOG* International Symposium On Logic Programming (1985).

[31] Ramsay A. **Formal Methods In Artificial Intelligence** Cambridge Tracts In Theoretical Computer Science, C.U.P., 1988.

[32] Reddy U.S. *On The Relationship Between Logic And Functional Languages* in DeGroot & Lindstrom [8] 3-36.

[33] Robinson J. A. *A Machine-Oriented Logic Based On The Resolution Principle* Journal of the ACM, Vol 12., 23-41, (1965).

[34] Robinson J. A. *LOGLISP: An Alternative To PROLOG* in Machine Intelligence 10, ed. Hayes, Michie, Pao. (1982).

[35] Sterling L. & Schapiro E. **The Art Of PROLOG: advanced programming techniques** MIT Press (1986).

[36] Stickel M. E. *A PROLOG Technology Theorem-Prover* New Generation Computing 2 (1984).

[37] Stickel M.E. *A Prolog Technology Theorem Prover Implementation by an Extended Prolog Compiler* in 8th International Conference on Automated Deduction, Springer-Verlag, Lecture Notes In Computer Science 230.

[38] Subrahmanyam P. A. & Jia-Huai You *Funlog: a computational model integrating Logic Programming And Functional Programming* in DeGroot & Lindstrom [8] 157-197.

[39] Tarnlund S. *Horn Clause Computability*, BIT 17 (1977).

[40] Tarver M. *THORN: A Theorem-Prover That Compiles FOL into LISP* draft thesis.

[41] Thayse A. **From Standard Logic To Logic Programming** Wiley, 1988.

[42] Tomura S. *TDProlog: An Extended Prolog with Term Description* Proceedings Of The 4th Logic Programming Conference 1985 Springer-Verlag (vol 221).

[43] Walther C. *A Mechanical Solution of Schubert's Steamroller by Many-Sorted Resolution* Artificial Intelligence 26 (2).

[44] Warren D. *Logic Programming And Compiler Writing* Software - Practice and Experience 10, (1980).

[45] Wikstrom A. **Functional Programming Using Standard ML** Prentice Hall International Series In Computer Science, ed. C.A.R. Hoare.

[46] Wos L., Overbeek R. Lusk E. & Boyle J. **Automated Reasoning: introduction and applications** Argonne National Laboratory, Argonne, Illinois, Prentice-Hall, (1984).

Copyright © 1989, Laboratory for Foundations of Computer Science,
University of Edinburgh. All rights reserved.

Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.