## Execute

Execute is implemented in a manner very similar to quad input. The string to be executed is placed in the input buffer and TYPEIN is called, resulting in a block of code followed by a pointer to the symbol to the left of the execute symbol. The code is then executed and a return is made to that symbol. Note that both quad input and execute may be nested to any depth, as long as there is space to store the generated code.

## Saved Workspaces

Upon a ) save sommand, portions of tables and various pointers are written on a binary file. If a file by the specified name exists, it will be used. Otherwise, an attempt is made to create a two link permanent file. Two links equals 7680–36 bit words, enough to hold a full workspace. If unsuccessful, because the user has not been allocated sufficient file space, a temporary file is defined. Temporary files are released when the terminal session is ended. However, the user may transfer the data to a permanent file prior to logging off. Saved files may be less than two links by creating the files in advance of the APL session.

## An On-Line Proof Checker Operating under APL/360, with Educational Applications in Logic, Mathematics, and Computer Science.

P. D. Page
Campus Computing Network
UCLA

One of the long standing problems of Computer Science is the development of programs capable of checking, correcting, and possibly even prompting or suggesting logical reasoning in various areas of human activity. It is a problem with a good chance of solution. The obvious application for such a reasoning program is formal logic itself, but other application areas include reasoning about everyday real world problems, proving theorems of mathematics, and verifying correctness of procedures for solving problems, i.e., "programs" in one sense or another.

There are several valid starting points of attack on this problem. We have started by developing a proof checking program that will check the correctness of logical deductions written in a modern system of formal logic (D. Kalish and R. Montague, "Logic, Techniques of Formal Reasoning," Harcourt, Brace and World, Inc. 1964). This program has an additional theorem-proving capability and can construct formal proofs for all of the standard theorems of propositional and predicate logic. This facility allows the user to make routine "jumps" in his chain of reasoning without the program making trivial complaints about incorrectness.

At present, the facilities available with this program are as follows:
The proof checking program operates in an on-line interactive mode under the APL/360 System. A proof to be checked is entered line by line at the terminal. As each line is entered, the program either accepts the logical statement as a valid consequence of prior proof lines, or rejects the statement as invalid. When a statement is accepted, it is numbered, formatted and typed out by the program for later reference. When a statement is rejected, a message is typed stating the reason for rejection.

The proof checking program automatically determines when a proof is complete and the assertion has been proved. A message is typed as soon as proof completion has occurred, and checking terminates.

As indicated previously, this program checks proofs of predicate logic with equality formulated according to the system of Kalish and Montague. All of the necessary inference rules of the

Kalish-Montague system have been implemented. The basic mode of program operation requires an annotation, i.e., inference rule name and line references, to accompany each new logical statement. If the statement entered does not follow from the referenced lines by the given inference rule, the statement is not accpted.

In accordance with the formalization of the Kalish-Montague System, a new subproof or derivation may be started at any point in a proof. This subproof may make use of any logical statement already acepted as part of the larger proof. Subproofs may also contain subproofs; nesting of subproofs may be continued indefinitely. The proof checking program keeps track of these subproofs and provides an indication when each is completed. This subproof completion message identifies the line containing the assertion which has been proved.

The basic mode of operation is augmented by a number of special problem oriented features. One of the most convenient features is the ability to enter an annotation without an accompanying logical statement. In this case, the program automatically deduces and displays all logical inferences following from the given annotation.

The proof checking program also has the capability of maintaining a list of axioms and theorems for use in constructing proofs. An axiom or theorem from this list may be recalled and entered as a line of the proof by an appropriate annotation.

The final and most powerful feature is an integrated theorem proving mode. A logical statement entered without annotation causes the proof checking program to invoke the theorem proving mode. This theorem prover is currently powerful enough to generate derivations for all elementary theorems of the propositional and predicate logic. It can easily verify statements which follow intuitively from previous logical statements, and thus eliminates the effort involved in leading the checking program step by step to the same logical consequence.

Though a complete set of inference rules is currently implemented, it will in the future, be convenient to be able to expand this list to include more powerful derived rules of inference. At times it may also be convenient to have inference rules which are especially well suited to a particular theory. The current proof checking program provides a means by which such derived rules of inference may be easily added.

At present, the input function has been tailored to accept a natural input language, i.e., mathematically oriented notation, so as to provide a system for checking proofs in mathematics (algebra and number theory). Eventually the formal system will be extended so that proofs of correctness of fairly simple computer programs can be generated and checked.

The program as it now stands has some obvious educational applications. In particular, it is expected that the current program will be used as an instructional aid by students of symbolic logic for performing homework problems and other exercises.

**Example of Kalish-Montague Formal Logic Proof**

T19

1. ~~Show~~ $(\sim P \rightarrow P) \rightarrow P$
2. $\sim P \rightarrow P$
3. ~~Show~~ P
4. $\sim P$
5. P

2, 4, MP

$$\frac{\phi \rightarrow \psi \quad \phi}{\psi}$$

*Modus ponens* (MP)

**Proof Generated Interactively using the APL Proof Checker**

| Program Response | | User Input |
|---|---|---|
| *START* | | |
| *PROOF CHECKER --- VERSION 1.1A APRIL 1971* | | |
| | | *SHOW (~P→P)→P* |
| 1. *SHOW (~P→P)→P* | | |
| 2. *~((~P→P)→P)* | *ASSUMP* | |
| 3. *~P→P* | *ASSUMP* | |
| 4. *~P* | *ASSUMP* | |
| | | 3,4,*MP* |
| 5. *P* | 3,4,*MP* | |
| *PROOF COMPLETED, CHECKING TERMINATED.* | | |

**Proof Generated by the APL Theorem Proving Functions**

| | |
|---|---|
| *PROVE '(~P→P)→P'* | |
| 1. *SHOW (~P→P)→P* | |
| 2. *~((~P→P)→P)* | *ASSUMP* |
| 3. *~P→P* | *ASSUMP* |
| 4. *~P* | *ASSUMP* |
| 5. *(~P→P)∧~P* | 2,*NC* |
| 6. *P∨P* | 3,*CD* |
| 7. *P* | 4,3,*MP* |
| 8. *~~P* | 4,3,*MT* |
| *PROOF COMPLETED SUCCESSFULLY.* | |
| *PROOF USED 2.92 SECONDS OF CPU TIME.* | |

Proof Exhibiting Various Features of the Proof Checker

*START* ← APL function which invokes checking

*PROOF CHECKER --- VERSION 1.1A APRIL 1971*

*SHOW (P→Q)↔~(P∧~Q)*

1. *SHOW (P→Q)↔~(P∧~Q)* ← Assertion to be proved
2. *~((P→Q)↔~(P∧~Q))* *ASSUMP*

*SHOW*

3. *SHOW (P→Q)→~(P∧~Q)* ← An obvious assertion is supplied by the proof checker
4. *~((P→Q)→~(P∧~Q))* *ASSUMP*
5. *P→Q* *ASSUMP*
6. *~~(P∧~Q)* *ASSUMP*

6,*DN*

7. *P∧~Q* 6,*DN*

7,*S* ← Input in the abbreviated form

8. *P* 7,*S*
9. *~Q* 7,*S*

5,9,*MP*

*LINE DOES NOT FOLLOW.* ← An invalid deduction is detected

5,8,*MP*

10. *Q* 5,8,*MP*

*SUB-PROOF AT LINE 3 COMPLETED.* ← Subproof completion is detected

```
                                                    SHOW ~(P∧~Q→(P→Q)
SYNTAX ERROR IN INPUT, OR IMPROPER LINE.   ← Syntax error is detected
                                                    SHOW ~(P∧~Q)→(P→Q)
11.    SHOW ~(P∧~Q)→(P→Q)                  ← Beginning of subproof
12.     ~(~(P∧~Q)→(P→Q))          ASSUMP
13.     ~(P∧~Q)                   ASSUMP    Subproof is indented
14.     ~(P→Q)                    ASSUMP
                                                    P→Q
15.     SHOW P→Q                          ← Theorem prover is invoked to obtain
16.      ~(P∧~Q)∧~(P→Q)           12,NC       an obvious contradiction
17.      P∧~Q                     14,NC
        SUB-PROOF AT LINE 15 COMPLETED.
       SUB-PROOF AT LINE 11 COMPLETED.
                                                    3,1,
                                                      v   ← Correction of
                                                    1,CB      input (APL)
18.    (P→Q)↔~(P∧~Q)             3,11,CB
19.    ~(P∧~Q)↔(P→Q)             3,11,CB
PROOF COMPLETED, CHECKING TERMINATED.   ← Proof completion is detected
```

$$\frac{\sim\sim\phi}{\phi} \qquad \frac{\phi}{\sim\sim\phi}$$ Double negation (DN)

$$\frac{\phi\wedge\psi}{\phi} \qquad \frac{\phi\wedge\psi}{\psi}$$ Simplification (S)

$$\frac{\phi\rightarrow\psi \quad \phi}{\psi}$$ *Modus ponens* (MP)

$$\frac{\phi\rightarrow\psi \quad \psi\rightarrow\phi}{\phi\leftrightarrow\psi}$$ Conditional-biconditional (CB)

Example of a Predicate Logic Proof

Proof Generated Interactively using the APL Proof Checker

```
Program Response                                          User Input

      START
PROOF CHECKER --- VERSION 1.1A  APRIL 1971
                                                    SHOW Δ[X](F[X]→G[X])→
1.    SHOW Δ[X](F[X]→G[X])→(Δ[X]F[X]→Δ[X]G[X])          (Δ[X]F[
2.     ~(Δ[X](F[X]→G[X])→(Δ[X]F[X]→Δ[X]G[X])) ASSUMP
3.     Δ[X](F[X]→G[X])           ASSUMP
4.     ~(Δ[X]F[X]→Δ[X]G[X])      ASSUMP
                                                    SHOW Δ[X]F[X]→Δ[X]G[X]
5.     SHOW Δ[X]F[X]→Δ[X]G[X]
6.      Δ[X]F[X]                 ASSUMP
7.      ~Δ[X]G[X]                ASSUMP
```

```
8.      SHOW Δ[X]G[X]                                    SHOW Δ[X]G[X]
9.       ~G[X]                          ASSUMP
                                                         X 6,UI
10.      F[X]                           6,UI
                                                         X 3,UI
11.      F[X]→G[X]                      3,UI
                                                          10,11,MP
12.      G[X]                           10,11,MP
       SUB-PROOF AT LINE 8 COMPLETED.
     SUB-PROOF AT LINE 5 COMPLETED.
PROOF COMPLETED, CHECKING TERMINATED.
```

Notation:

Δ denotes Λ or ∀

∇ denotes V or ∃

Kalish-Montague Formal Proof

T201

1. ~~Show~~ $\Lambda x(Fx \rightarrow Gx) \rightarrow (\Lambda xFx \rightarrow \Lambda xGx)$
2. $\Lambda x(Fx \rightarrow Gx)$
3. ~~Show~~ $\Lambda xFx \rightarrow \Lambda xGx$
4. $\Lambda xFx$
5. ~~Show~~ $\Lambda xGx$
6. $Fx$ — 4, UI
7. $Fx \rightarrow Gx$ — 2, UI
8. $Gx$ — 6, 7, MP

$$\frac{\Lambda\alpha\phi}{\psi}\,,$$ Universal instantiation (UI)

$$\frac{\phi \rightarrow \psi \quad \phi}{\psi}$$ *Modus ponens* (MP)

## A LANGUAGE MACHINE

Rodnay Zaks
Center for Research in Management Science
University of California, Berkeley

## THE META–APL TIME-SHARING SYSTEM

META–APL is a multiprocessor time-sharing system developed at the University of California, Berkeley, for interactive real-time APL service. Its conceptual design has deliberately been kept simple and an expeptionally high performance to cost ratio has been obtained. Most of the functions traditionally performed by software have been pushed into the hardware-firmware execution unit. The core memory, although fast (750 nsec) is viewed by the processor strictly as an IO device.

The system was developed in two steps: completion of an APL processor and development of an operating-system processor, both processors being micro-programmed and