The Programmer's Apprentice:

Knowledge Based Program Editing

Richard C. Waters

The Artificial Intelligence Laboratory
Massachusetts Institute of Technology
545 Technology Square
Cambridge MA 02139

The goal of the Programmer's Apprentice project at the MIT Artificial Intelligence Laboratory is to develop a theory of how expert programmers analyze, synthesize, modify, explain, verify, and document programs. Recognizing that the long-term goal of totally automatic programming is very far off, we are applying our research towards the immediate goal of developing an intelligent computer assistant for programmers, called the *Programmer's Apprentice*. As a first demonstration of how the Programmer's Apprentice can help a programmer, we have implemented a new kind of program editor which understands how a program is composed out of common algorithmic fragments.

This *knowledge based* program editor makes it possible to modify a program by issuing commands which refer directly to the logical structure of the algorithms being used, rather than commands which refer to the textual or syntactic structure of the program. For example, with existing program editors, a single logical change to a program must often be achieved by many separate changes to different parts of the program text. With the knowledge based editor, such a change can often be achieved by using a single command.

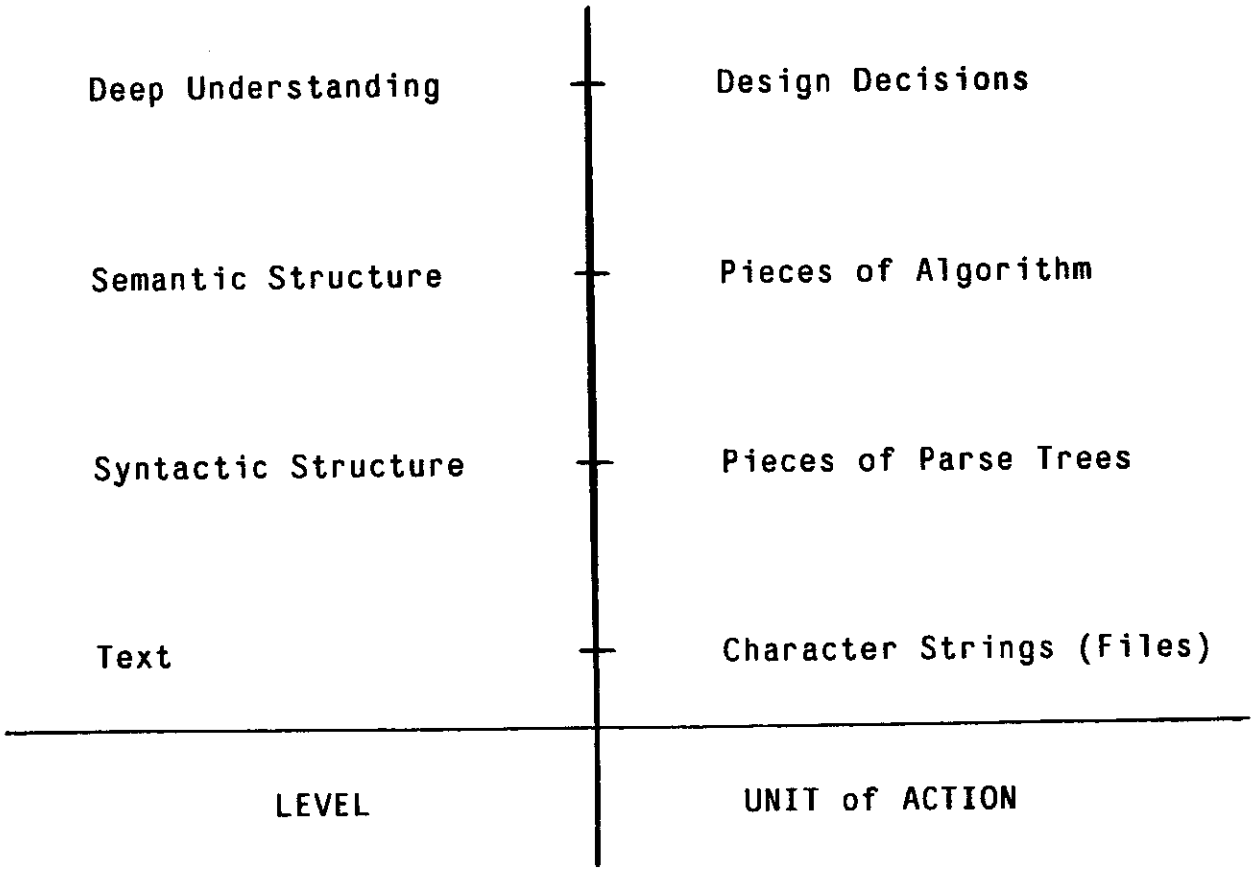
In order to support this mode of interaction, the editor maintains a *plan* which represents the logical structure of the program. The user requests specify changes in this plan. In order to be able to communicate with other parts of the programming environment, the system maintains a corresponding textual representation for the program. When the user specifies a change to be made to the plan, the editor determines what changes this causes in the text. If the user directly changes the text, the new text is analyzed in order to determine what the plan should be.

As our underlying theory develops further, we expect the capabilities of the Programmer's Apprentice to increase, with a corresponding reduction in the amount of work the programmer is required to do.

Increasing Programmer Productivity and Program Reliability

* Programming consists of many phases: design, implementation, testing, debugging, documentation, maintenance, and modification.
* In order to have an order of magnitude effect, you must impact all the phases.
* The introduction of high level languages may be the only example of this to date.
* Knowledge is the key to making another dramatic productivity increase.
* AI technologies can make this possible.
* The ultimate goal of AI in programming is *Automatic Programming* but this is decades away.
* Fortunately, there are many intermediate tools possible.

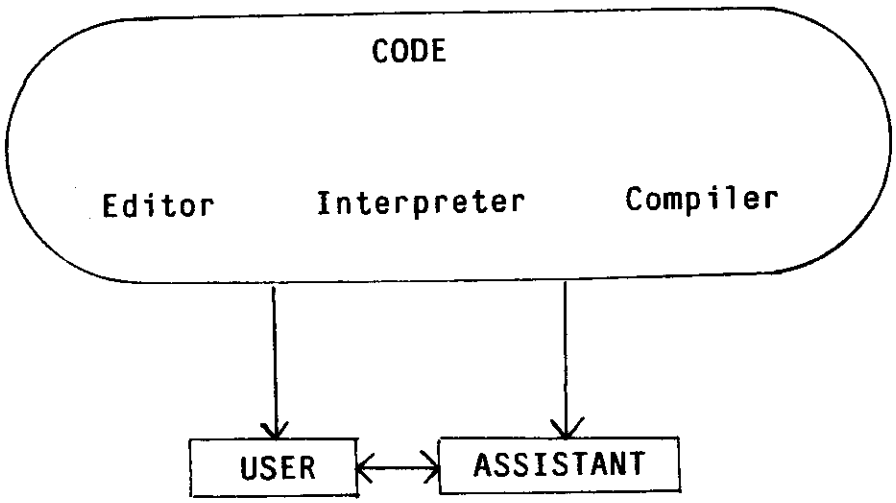
Levels of Knowledge



* No current programming aid exhibits intelligence in any significant way.

The Assistant Metaphor

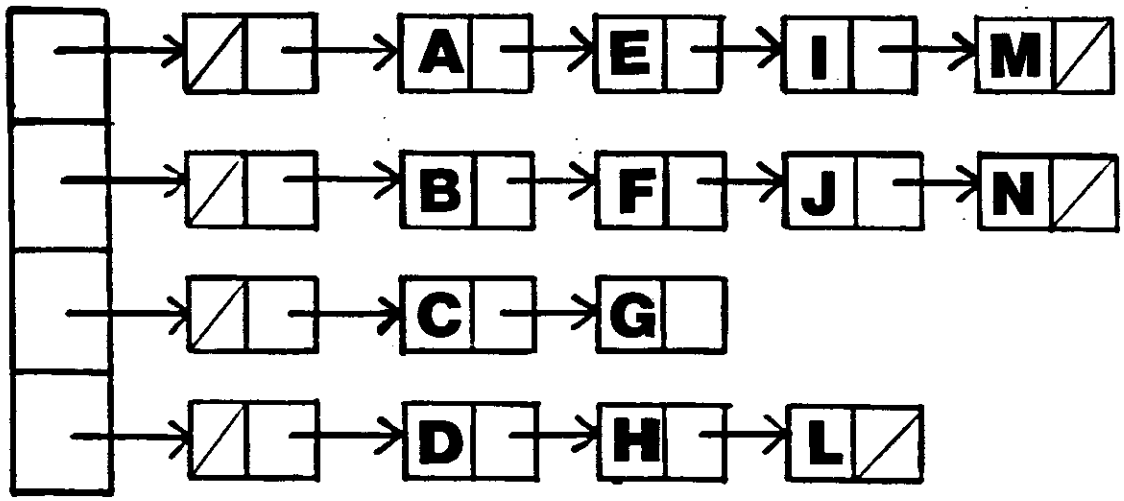
[State of the Art]
[Programming Environment]



An Example

Implementing a set as a hash table:

The set {A B C D E F G H I J L M N} could be stored in a hash table using buckets with header cells of NIL as:



The hashing operation selects the bucket corresponding to a symbol.

Various other operations such as adding into and deleting from a set then act on this bucket.

Implementing the DELETE Operation

Define a program DELETE with a parameter SYMBOL.

```
/* The function HASH maps a symbol into the corresponding
   bucket in TBL.                                          */

HASH: PROCEDURE(SYMBOL) RETURNS (POINTER);
        DCL SYMBOL POINTER,
            SXHASH EXTERNAL ENTRY (POINTER) RETURNS (FIXED),
            TBL (*) POINTER EXTERNAL,
            TBLSIZE FIXED EXTERNAL;
        RETURN (TBL(ABS(MOD(SXHASH(SYMBOL), TBLSIZE))));
      END HASH;

/* The function DELETE removes a symbol from TBL.          */

DELETE PROCEDURE(SYMBOL) RETURNS (...);
         DCL SYMBOL ...,
       END DELETE;
```

Implement the program as a linear search.

```
/* The function DELETE removes a symbol from TBL.          */

DELETE PROCEDURE(SYMBOL) RETURNS (...);
          DCL SYMBOL ...,
              AGGREGATE ...;
          AGGREGATE = arg-of-enumerator;
      LP: IF empty-test-of-enumerator(AGGREGATE)
             THEN RETURN (NULL());
          IF test() THEN RETURN (action());
          AGGREGATE = step-of-enumerator(AGGREGATE);
          GOTO LP;
        END DELETE;
```

ZMACS (LISP) AI: PA; HASH > (1)

Implement the enumerator as a list enumeration of "HASH(SYMBOL)->NEXT".

```
/* The function DELETE removes a symbol from TBL.          */

DELETE: PROCEDURE(SYMBOL) RETURNS (...);
          DCL SYMBOL POINTER,
              LIST POINTER,
              HASH EXTERNAL ENTRY (POINTER) RETURNS (POINTER),
              NODE_PTR POINTER,
              1 NODE BASED (NODE_PTR),
                2 VALUE ...,
                2 NEXT POINTER;
          LIST = HASH(SYMBOL)->NEXT;
       LP: IF LIST=NULL() THEN RETURN (NULL());
          item = LIST->VALUE;
          IF test() THEN RETURN (action());
          LIST = LIST->NEXT;
          GOTO LP;
        END DELETE;
```

Implement the test as "ITEM=SYMBOL".

```
/* The function DELETE removes a symbol from TBL.          */

DELETE: PROCEDURE(SYMBOL) RETURNS (...);
          DCL SYMBOL POINTER,
              LIST POINTER,
              HASH EXTERNAL ENTRY (POINTER) RETURNS (POINTER),
              NODE_PTR POINTER,
              1 NODE BASED (NODE_PTR),
                2 VALUE ...,
                2 NEXT POINTER;
          LIST = HASH(SYMBOL)->NEXT;
        LP: IF LIST=NULL() THEN RETURN (NULL());
          IF LIST->VALUE=SYMBOL THEN RETURN (action());
          LIST = LIST->NEXT;
          GOTO LP;
        END DELETE;
```

ZMACS (LISP) AI: PA; HASH > (1)

Implement the action as a splice out of the previous value of 'LIST.

```
/* The function DELETE removes a symbol from TBL.          */

DELETE: PROCEDURE(SYMBOL);
          DCL SYMBOL POINTER,
              LIST POINTER,
              PREV POINTER,
              HASH EXTERNAL ENTRY (POINTER) RETURNS (POINTER),
              NODE_PTR POINTER,
              1 NODE BASED (NODE_PTR),
                2 VALUE ...,
                2 NEXT POINTER;
          LIST = HASH(SYMBOL)->NEXT;
          PREV = initial-value;
      LP: IF LIST=NULL() THEN RETURN;
          IF LIST->VALUE=SYMBOL
             THEN DO; PREV->NEXT = PREV->NEXT->NEXT;
                      RETURN; END;
          PREV = LIST;
          LIST = LIST->NEXT;
          GOTO LP;
        END DELETE;
```

ZMACS (LISP) AI: PA; HASH > (1)

Implement the initial value as "HASH(SYMBOL)".

```
/* The function DELETE removes a symbol from TBL.          */

DELETE: PROCEDURE(SYMBOL);
          DCL SYMBOL POINTER,
              LIST POINTER,
              PREV POINTER,
              HASH EXTERNAL ENTRY (POINTER) RETURNS (POINTER),
              NODE_PTR POINTER,
              1 NODE BASED (NODE_PTR),
                2 VALUE ...,
                2 NEXT POINTER;
          LIST = HASH(SYMBOL)->NEXT;
          PREV = HASH(SYMBOL);
       LP: IF LIST=NULL() THEN RETURN;
          IF LIST->VALUE=SYMBOL
             THEN DO; PREV->NEXT = PREV->NEXT->NEXT;
                      RETURN; END;
          PREV = LIST;
          LIST = LIST->NEXT;
          GOTO LP;
        END DELETE;
```

ZMACS (LISP) AI: PA; HASH > (1)

Share the first HASH and the second HASH.

```
/* The function DELETE removes a symbol from TBL.          */

DELETE: PROCEDURE(SYMBOL);
          DCL SYMBOL POINTER,
              LIST POINTER,
              PREV POINTER,
              HASH EXTERNAL ENTRY (POINTER) RETURNS (POINTER),
              NODE_PTR POINTER,
              1 NODE BASED (NODE_PTR),
                2 VALUE ...,
                2 NEXT POINTER;
          PREV = HASH(SYMBOL);
          LIST = PREV->NEXT;
       LP: IF LIST=NULL() THEN RETURN;
          IF LIST->VALUE=SYMBOL
             THEN DO; PREV->NEXT = PREV->NEXT->NEXT;
                      RETURN; END;
          PREV = LIST;
          LIST = LIST->NEXT;
          GOTO LP;
        END DELETE;
```

ZMACS (LISP) AI: PA; HASH > (1)

Describe the ports of the program.
Display the source of the TBL output.

```
/* The function DELETE removes a symbol from TBL.          */

DELETE: PROCEDURE(SYMBOL);
          DCL SYMBOL POINTER,
              LIST POINTER,
              PREV POINTER,
              HASH EXTERNAL ENTRY (POINTER) RETURNS (POINTER),
              NODE_PTR POINTER,
              1 NODE BASED (NODE_PTR),
                2 VALUE ...,
                2 NEXT POINTER;
          PREV = HASH(SYMBOL);
          LIST = PREV->NEXT;
      LP: IF LIST=NULL() THEN RETURN;
          IF LIST->VALUE=SYMBOL
             THEN DO; PREV->NEXT = PREV->NEXT->NEXT;
                      RETURN; END;
          PREV = LIST;
          LIST = LIST->NEXT;
          GOTO LP;
        END DELETE;
```

The ports of the program DELETE:
  The SYMBOL parameter
  The TBL free variable input
  The TBLSIZE free variable input
  The return value
  The TBL side-effect output

ZMACS (LISP) AI: PA; HASH > (1)

Create a comment for the program.

```
/* The function DELETE removes a symbol from TBL.          */

/* Searches a list enumeration of the NEXT of the HASH of
   SYMBOL for an element which satisfies the = comparison of
   the VALUE of the sublist under examination and SYMBOL.
   IF found, splices out the successor of the previous value
   of the sublist under examination.                       */

DELETE: PROCEDURE(SYMBOL);
          DCL SYMBOL POINTER,
              LIST POINTER,
              PREV POINTER,
              HASH EXTERNAL ENTRY (POINTER) RETURNS (POINTER),
              NODE_PTR POINTER,
              1 NODE BASED (NODE_PTR),
                2 VALUE ...,
                2 NEXT POINTER;
          PREV = HASH(SYMBOL);
          LIST = PREV->NEXT;
      LP: IF LIST=NULL() THEN RETURN;
          IF LIST->VALUE=SYMBOL
             THEN DO; PREV->NEXT = PREV->NEXT->NEXT;
                      RETURN; END;
          PREV = LIST;
          LIST = LIST->NEXT;
          GOTO LP;
        END DELETE;
```

ZMACS (LISP) AI: PA; HASH > (1)

```
Use language LISP.
```

```
;;; The function DELETE removes a symbol from TBL.

;;; Searches a list enumeration of the CDR of the HASH of
;;; SYMBOL for an element which satisfies the = comparison of
;;; the CAR of the sublist under examination and SYMBOL.
;;; IF found, splices out the successor of the previous value
;;; of the sublist under examination.

(DEFUN DELETE (SYMBOL &AUX LIST PREV)
  (PROG ()
        (SETQ PREV (HASH SYMBOL))
        (SETQ LIST (CDR PREV))
     LP (COND ((NULL LIST) (RETURN NIL)))
        (COND ((EQ (CAR LIST) SYMBOL)
               (RPLACD PREV (CDDR PREV))
               (RETURN NIL)))
        (PSETQ PREV LIST
               LIST (CDR LIST))
        (GO LP)))
```

ZMACS (LISP) AI: PA; HASH > (1)

The Key Features of the System

* Supports Editing of algorithmic structures.

  - Has a library of algorithmic fragments.

* Records and can display a variety of information about a program.

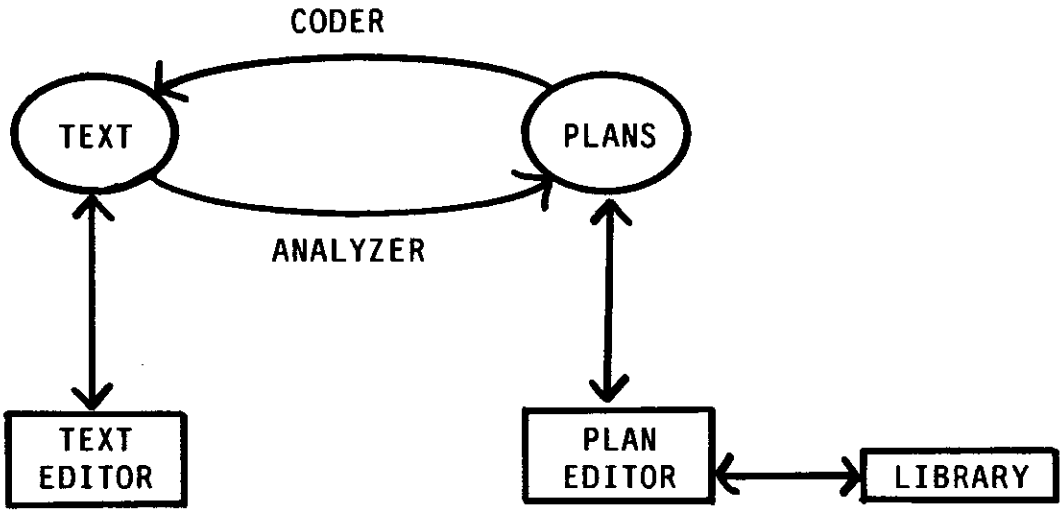
* Supports implementation and modification.

* Supports the intermixing of text editing and knowledge based editing.

  - Can operate on programs not created by the system.

* Language independence.

Architecture of the Knowledge Based Editor



CODER - Works for LISP, PLI, and HIBOL.

ANALYZER - Works for LISP, FORTRAN, and COBOL.
Does not yet recognize library fragments.

TEXT EDITOR - The standard Lisp Machine editor.

PLAN EDITOR - Supports direct modification of plans.

LIBRARY - Plans for common algorithmic fragments.
Currently, only a few are defined.

PLANS: The Underlying Representation Used by the Editor

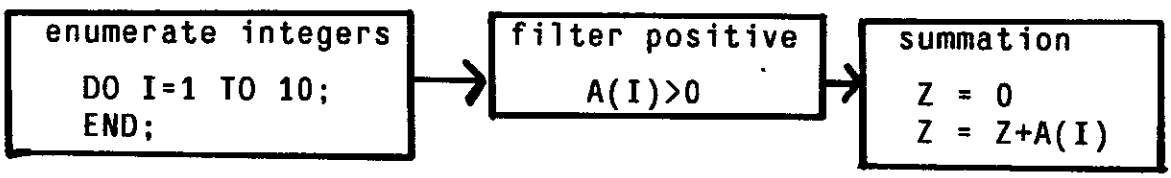
Enable the editor's actions to be small and local.
Explicit data flow and control flow simplify deduction.
Does not need to be expressible on paper.

Provides programming language independence.
Focuses on features of the algorithm rather than the language.

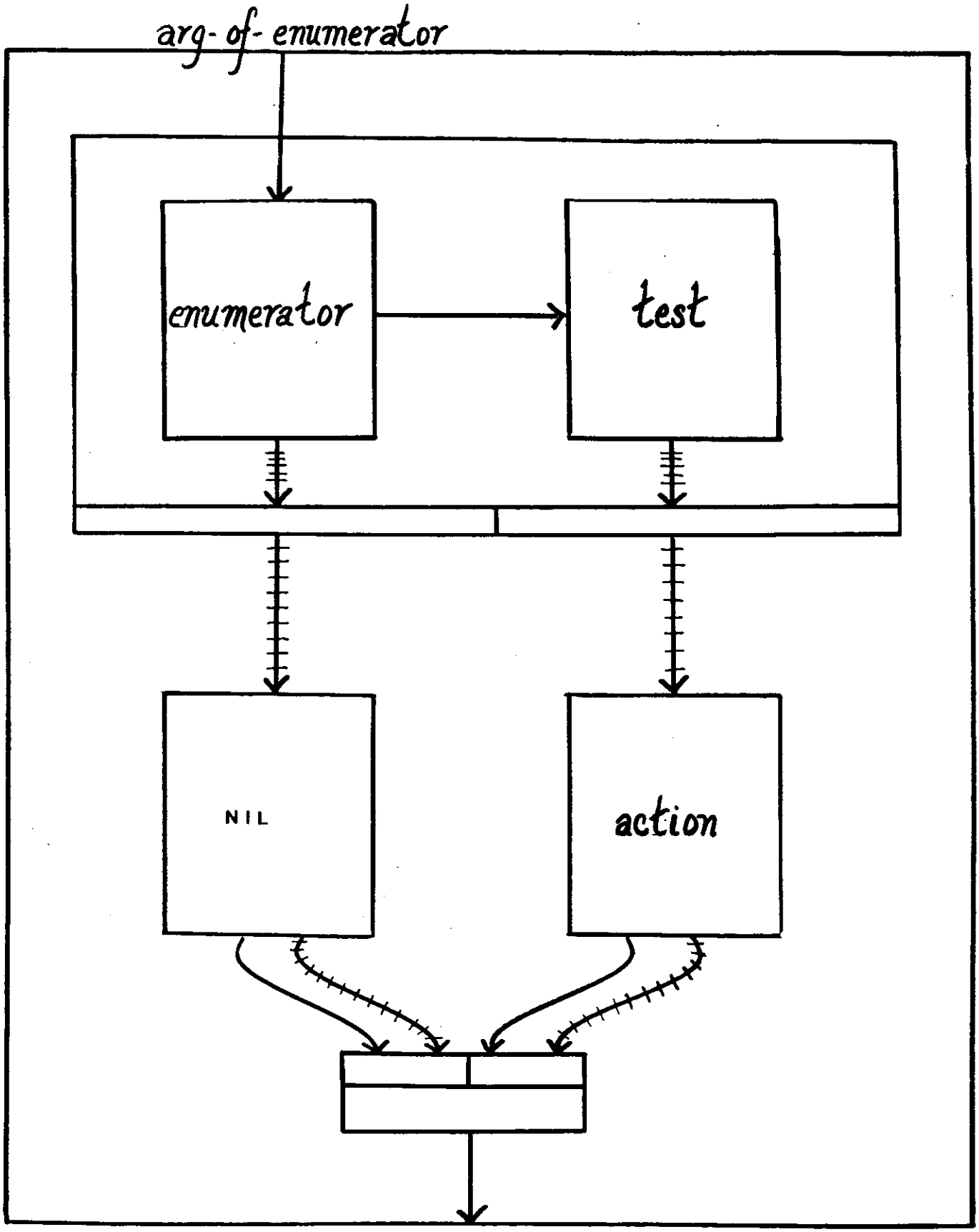
Used to represent the algorithmic fragments in the library.

Loops are represented as compositions of loop fragments.

```
Z = 0;
DO I=1 TO 10;
   IF A(I)>0 THEN
      Z = Z+A(I);
END;
```



The PLAN For Linear Search



The Depth of Understanding of the Editor

In order to construct DELETE the user had to say:

```
Define a program DELETE with a parameter SYMBOL.
Implement the program as a linear search.
Implement the enumerator as a list enumeration of
"HASH(SYMBOL)->NEXT".
Implement the test as "ITEM=SYMBOL"
Implement the action as a splice out of the previous value
of 'LIST.
Implement the initial value as the value of the HASH.
```

* It does not understand data structures.

* It does not understand specifications.

* It does not understand any interrelationships between fragments.

* It uses the fragments only in very simple ways.

The Second Version of the PA: User Guided Synthesis

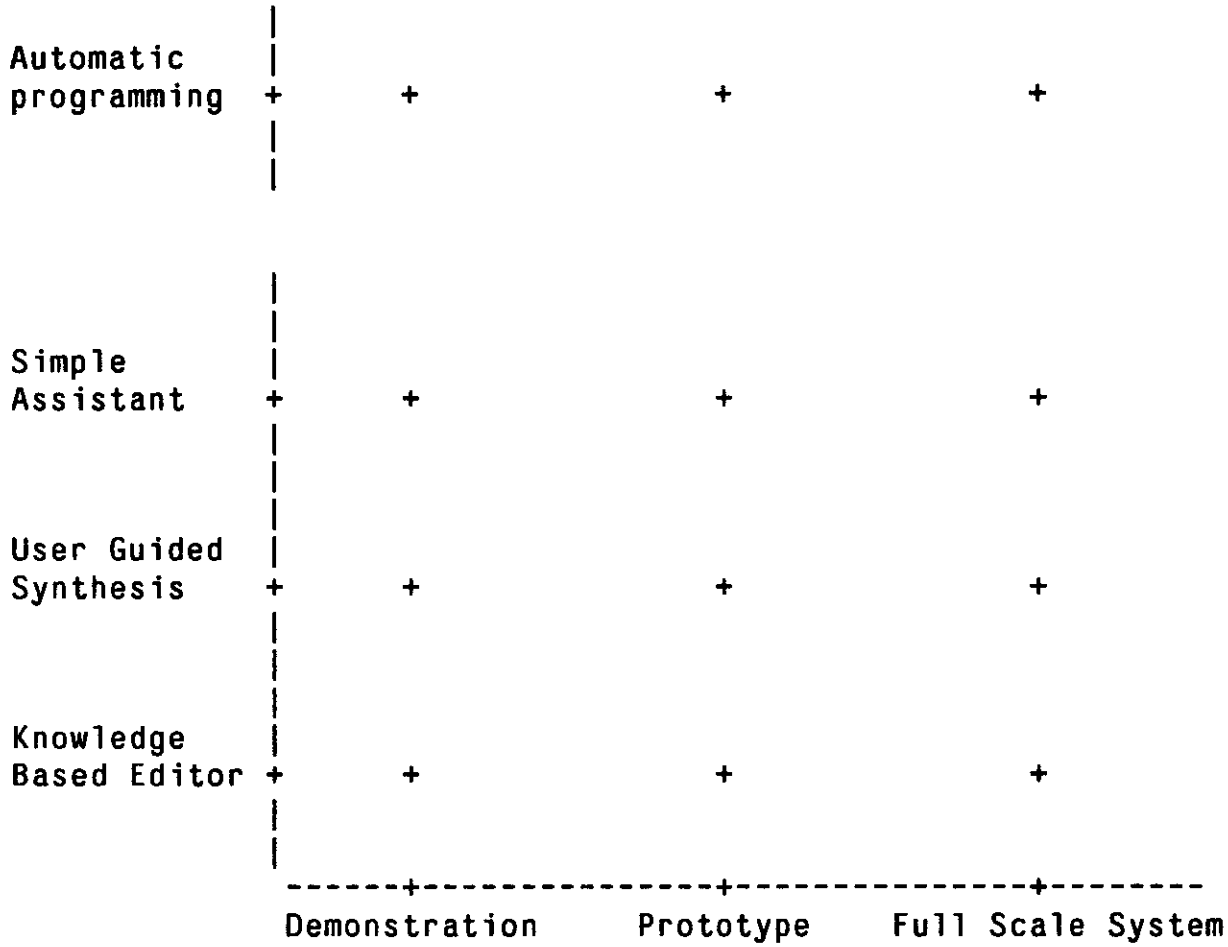
Before creating the function HASH, the user would say:

```
TBL is a global vector of size TBLSIZE implementing a set as a
hash table whose buckets are lists with header cells of NIL.
Define a program HASH which maps a given SYMBOL into the
corresponding bucket in TBL.
```

In order to construct DELETE he would then say.

```
Define a program DELETE which removes a given SYMBOL from TBL.
Implement the program as a search for an occurrence of SYMBOL
in TBL.
Implement the action as a splice out of this occurrence.
```

Future Goals of the Programmer's Apprentice Project



Now

1 year

3 years