SEL—A self-extensible programming language

G. Molnar
Istituto di Elaborazione della Informazione, CNR, Pisa, Italy

In this paper a self-extensible programming language, SEL or, rather, a general method for a system building tool, is described. The designing criteria are given, the terminology which will be used is specified, the basic concepts of SEL are presented, some typical SEL instructions are illustrated, efficiency and protection problems are discussed, and an implemented version of SEL is outlined. For illustrative purposes, two self-extension programs are given. A short conclusion closes the paper.

(Received May 1970)

1. Introduction

One of the classic ideas for resolving the problem due to programming language proliferation, is to design a universal language. However, until now, although the new 'universal' languages have contributed to clarify many problems, they have complicated many others. We don't expect these difficulties to be completely resolved by the previously conceived self-extensibility as, e.g. ALGOL 68, SIMULA 67, etc. for the following reasons:

1. These universal languages are high level languages. They require a rather complex compiling system which excludes them from the field of small computers.
2. They have an initially defined syntax which, although extensible, binds the system to remain syntax directed. It is not certain that this is the most natural way of describing programming systems, e.g. an operating system.
3. There is no possibility to introduce new basic concepts into the language (e.g. in case of hardware modification) without a drastic modification of the compiling system.
4. There is no possibility to extend the language to make it formally equivalent to other, already existing, languages, e.g. FORTRAN, ALGOL 60, etc.
5. While there is, in these languages, a general method of compiling, this method cannot be as efficient as the particular one, made for one language.

We consider the most serious restrictions to be points (3) and (4).

In designing SEL, we have taken account of the above criteria. Therefore, SEL is not a high level language, it doesn't use a syntax recompiling mechanism as a built-in facility. Self-extension is procedural, not declarative, so, SEL is also a metalanguage for its own extension. A 'piece of program' can be not only a building element for a further extension but also an operative tool for doing it. The freedom of choosing symbols for denoting new concepts allows the system to describe conventional programming languages in their own syntactic form. Particular care has been taken on efficiency.

2. Some remarks on terms used

2.1. The term *language* is used for denoting a tool for man-machine communication which doesn't imply previous grammar definition.

2.2. By *extensibility* we denote the capacity to alter the meaning of a message expressed in the language, or make pragmatically interpretable other messages which were not such before. The term *self-extensibility* will be used to indicate the possibility of extending a language by means of a program written in the same language. Self-extensibility we shall call also *vertical extensibility* in order to distinguish it from the possibility of introducing new basic concepts in the language (by modifying its basic implementation) which we shall call *horizontal extensibility*.

2.3. The basic elements of the external representation of a SEL program, i.e. those of its text, are *names* (words), in agreement with the preference that human beings show towards a text taken as a sequence of words rather than as a sequence of characters. Normally, a man is in fact able to single out a word without necessarily knowing its meaning.

2.4. In a text a name has two attributes: (i) the *occurrence* which is its position in the text and (ii) its *meaning* which is a reference to some physical objects (values) denoted by it.

2.5. A text of a program—although it may have a meaning—does nothing: it is just a *declaration*. For associating an action with a text we have to define what we want the program to do. The *definition* of a program is a mapping of the text into actions. Before executing it, we have to *generate* the program according to its definition. The generated program can then be executed or, generally speaking, *applied*. Finally, a program can be *deleted*.

3. Basic concepts of SEL

SEL is self-extensible vertically but, as its structure also facilitates horizontal extensibility, it is to be considered as a general method for building programming systems rather than building a well defined language. The basic instructions presented in this paper are just some examples, and are by no means suggested as an optimal choice.

3.1. A SEL program is an ordered set of *objects*. An object is composed of two parts: an external object which is the name of the object, and an internal one which is its value. By convention, a SEL object always has two values: an executable one, i.e. a *procedure*, and a *static one* of some type (e.g. integer, real, etc.).

One of the basic ideas of SEL is just this active feature of its 'words', i.e. the fact that each object (its procedure) can be executed. To extend this ability to all names, we introduce the convention that to each object not otherwise defined 'belongs' an error message procedure. The internal part of an object can be redefined by applying (executing) other suitable objects (i.e. their procedure). The result of the application of an object is completely defined by:

1. The name of the object
   (*a*) its occurrence by which the object is actually entered — external object
   (*b*) the meaning of the object (the reference to the value of the object).
2. The value of the object
   (*a*) procedure — internal object
   (*b*) static value.

3.2. It is a consequence of what was said before that the definition of a SEL program, i.e. the mapping of words into actions is the simplest one-to-one correspondence between external and internal objects. It follows that any arbitrary sequence of SEL words is a formally correct SEL program. Pragmatic errors must be controlled by the program itself, so any 'error message' is to be considered as a result of an execution, not just a message from a supervisor program.

3.3. The name of an object is singled out from the input character string without taking into consideration its meaning. There is a fixed convention for the inner syntax of words, which cannot be altered by any self-extension. (Although it could be modified by a horizontal extension.) In our implemented version the SEL system recognises identifiers, integer constants, real constants, string constants (all of which are defined as in ALGOL) as well as an additional class of special identifiers and one of special characters.

3.4. SEL has a basic object set (basic vocabulary). The procedures of the basic objects represent the basic actions of SEL. Moreover, each basic object can generate a new internal object (see Section 3.8) which (or a combination of some new internal objects) can be assigned to an external object. Object generation is the main facility of self-extensibility in SEL.

3.5. An object (i.e. its procedure) can operate both on its own value and on the ones belonging to other objects, i.e. its parameter can be either itself or another object. As for the basic vocabulary, for the sake of simplicity, we assume that if a procedure has one parameter other than itself, this parameter is the object whose occurrence is successive to the occurrence by which the procedure was entered. For example, if the symbol $PSB has the procedure which prints out the name of a word, the execution of

$PSB APPLE

will cause the print out of the symbol APPLE. Similarly, if a basic procedure has more than one parameter, these parameters are the objects whose occurrences are successive to the ones by which the procedure was entered. For example the execution of

$SUM A B C

assigns the sum of the static values of A and B to C if A, B and C are defined as possessing integer values (see Sections 4.1 and 4.5.).

Parameter evaluation takes part of the procedures of the basic SEL objects by using a system vector (called the occurrence vector) which contains the external representation of the SEL program in form of pointers to object names. Names contain (point to) values.

3.6. Macros can be easily defined in SEL: an ordered set of objects can be considered as one internal object and can be assigned to another external object. One can do this by applying some basic objects (i.e. executing their procedures). E.g.

<* $PSB APPLE *> → Y

assigns the sequence of objects '$PSB APPLE' to the external object having the name Y. The procedure of the object denoted by the name '<*' defines a macro, i.e. a compound object consists of sequence of objects whose names follow the '<*' until '*>'. The next object to be executed is that following '*>', in this case '→' whose procedure assigns the macro defined to that external object whose name follows it, i.e. to 'Y'. Formally, a symbol which denotes a macro is intended to be substituted statically by the sequence of symbols closed between '<*' and '*>'. When the macro is defined as in the above example, any execution of the procedure of the object 'Y' produces the print out of 'APPLE'. In SEL a defined macro can be executed immediately like a subroutine (as in the above example) or can be generated as a real macro (see Section 3.8).

3.7. Macros may have formal parameters. One way of introducing this is by using an object which is also a macro. Another way of introducing this is by using *references*, which are a type of value. A particular object, $ADR has the procedure which, executed, defines its successor as possessing a reference. Another procedure (that of $ASR) can assign a value to it. E.g. executing

$ADR R $ASR Q R

assigns a value to R which is a reference to Q. A particular object denoted by '$REF' is a formal parameter object. Its internal object, the value of the parameter, is the one to which the successor refers. With the above reference definition the execution of the SEL program

$PSB $REF R

will cause the print out of 'Q'.

There are two system references accessible to the user: $THS and $SUC. The value of $THS is the reference to the symbol to which belongs the macro under execution. E.g. if after the definition

<* $PSB $REF $THS *> → T

one executes T, it produces the print-out of its own symbol, i.e. T. Similarly $SUC has the value which is a reference to the symbol whose occurrence is successive to the occurrence of the symbol referred by $THS, by which the macro was called. E.g. after the definition

<* $PSB $REF $SUC *> → T ,

if one calls the macro by the occurrence of T in the sequence . . . T W . . . , it produces the print-out of W. Two particular words $NXT and $PRV can cause to move forewards or backwards respectively the system pointer which indicates the occurrence of the symbol pointed out by $SUC (altering, in such a way also the value of $SUC). This makes possible a look-up from a macro to the 'defining' level.

3.8. The SEL system is always in one of two states: interpretation or generation. If the state is interpretation when, by its occurrence, a basic object is entered, the procedure is executed; while, if the state is generation, a new similar procedure is generated. Particular objects having the name '<:' and ':>' can switch the state of SEL from interpretation to generation or vice versa respectively. Therefore, executing

<: $PSB APPLE :> → X

doesn't cause any print out, but entering an occurrence of X after this definition, will result in the print-out of APPLE. (The same symbol '→' could be used for assigning either a macro definition or a macro generation.) A generated macro can have formal parameters as in the case of macro definition. The main difference between the execution of a macro definition and that of a generated macro is that the latter is an optimised procedure while the former is a sequence of calls of distinct procedures under the supervisor SEL system. Note that a macro generated has no internal definition; the entering of the occurrence of the symbol which denotes it will cause the execution of the procedure independently from the state of SEL.

4. Some typical instructions of SEL

4.1. Any symbol can be defined as possessing a value of some type (either variable or constant, see also Section 4.5). The words $ADR, $BOL, $INT, $REL, and $CHR define their successor as possessing reference, Boolean, integer, real and character values. Vectors and structured values can be introduced by field generation and references. On each type of value there are defined some elementary operations: as arithmetic, Boolean operations, assignments of various types, etc.

4.2. The whole program written in SEL is intended to be included between two conventional words: $PRP and $SYS. $PRP initially has no procedure, but the user can generate and assign one to it. So each, execution may begin with a user defined preprocessing. $SYS ends the program execution and, when a conversational version of SEL is available, returns the control to the user. The procedure of the word $PGO assigns the Boolean value **true** to its successor if $SUC has no meaning (it points out of the program). Obviously, it has a practical meaning only internally to a generated procedure.

4.3. The procedure of $SKP causes its successor to skip. If the successive symbol denotes a macro of symbols, then the execution of $SKP causes the whole macro to skip. The words $SKT and $SKF cause their second successor to skip if the value of their first successor is Boolean, respectively **true** or **false.**

4.4. The word $RET causes a return from the execution either of a macro of words or of a procedure. $REP causes the repetition of the execution of either a group or a procedure.

4.5. A word can represent either a variable or a constant. By means of the execution of explicit instructions $VAR or $CST one can change the state of a symbol into a variable or into a constant respectively. Executing $CCB assigns the Boolean value **true** to its second successor if its first successor is a constant, **false** otherwise. E.g. '→' has the procedure to assign the last defined group or generated procedure to its successor. After the execution of

<* $PSB 'CONSTANT ERROR' $SYS *> → ERC \EXECUTING ERC IT GIVES AN ERROR MESSAGE AND ENDS THE PROGRAM\
<: $CCB $REF $SUC B $SKF B ERC → $REF $SUC $NXT :> →→
\IF THE SUCCESSOR OF THE CALLING SYMBOL (OF THE SAME '→') IS NOT A CONSTANT, IT ASSIGNS THE LAST GENERATED PROCEDURE OR GROUP TO THIS SUCCESSOR I.E. IT DOES THE SAME AS BEFORE SAVE THE CONTROL ON CONSTANTS. B IS A BOOLEAN VARIABLE\

no new symbol group or procedure can be ssigned to a constant.

4.6. The word $LAB defines its successor as a label pointing to the present occurrence of the symbol referred to by $SUC. The execution of a label consists in the transfer of the control to the occurrence denoted by the label. E.g. if we wish to define as label any word having an occurrence followed by ':'; pointing to the successor of the same occurrence, we can define the following preprocessor:

$ADR SYM \DEFINES SYM AS POSSESSING A REFERENCE. AS IT IS NOT DEFINED AS A CONSTANT, IT IS A VARIABLE\
<* $NXT $LAB $REF SYM *> → LB \THE MACRO DENOTED BY LB ADVANCES THE SUCCESSOR WHICH THE PROCEDURE BELONGS TO AND DEFINES THE SYMBOL REFERRED BY SYM AS A LABEL DENOTING THE PRESENT OCCURRENCE OF THE SUCCESSIVE SYMBOL\
<: $ASA $SUC SYM $NXT \ASSIGNS THE REFERENCE VALUE OF $SUC TO SYM AND ADVANCES $SUC\ $CPS $REF $SUC :
B \COMPARES THE SYMBOL REFERRED BY $SUC WHETHER IT IS A ':'. IF IT IS SO, IT ASSIGNS THE BOOLEAN VALUE TRUE TO B, FALSE OTHERWISE\
$SKF B LB \SKIP IF THE VALUE OF B IS FALSE, OTHERWISE DEFINE THE LABEL.\
$SPGO B $SKT B $REP :> → $PRP \CONTROL WHETHER THE WHOLE PROGRAM IS SEARCHED, REPEAT THE PROCEDURE OTHERWISE. ASSIGN THE PROCEDURE TO $PRP\ $CST $PRP \PROTECT $PRP AGAINST CHANGING PROCEDURE\

5. Efficiency and protection

5.1. SEL is not designed so that its basic vocabulary can be programmed directly. Since its basic instructions are sometimes more elementary than those of a computer, this could result in being too cumbersome. However, an extension of SEL can be tailored to offer a suitable programming language to the user. One can also 'protect' the actual extension, e.g. a compiler against posterior extension by switching out of the system all basic SEL objects. Using the self-extensible facility of SEL, the volume and complexity of a SEL program (the text of the program) doesn't increase linearly with that of the problem to be described, but less, presumably logarithmically.

5.2. The interpretation of a program using only the basic instructions of SEL is far from being efficient. However, by generating the same program, an optimised routine results. The basic SEL instructions are chosen to facilitate this built-in optimisation.

We feel that SEL can produce a program almost as efficient as a program written in machine language for the following reasons:

5.2.1. Basic SEL is an elementary language. Its basic procedures are easily transformed into a machine language without a complicated compile system.

5.2.2. From the practical point of view one of the main reasons for considering elements of a language as expressions, instead of elementary operations to be executed, is the problem of auxiliary storage necessary for the evaluation of an expression. The expression view permits an elaborate built in optimisation using fast registers, while in the case of the procedural view it is difficult to distinguish between temporary storage and variables. The solution offered by SEL is the possibility to declare objects as temporary variables. As for macro generation, an auxiliary variable is intended as destructive. Its value can be referred to only once. E.g. having declared X as auxiliary variable, by executing

$AUX X

the internal object generated by

: $SUM A B X $DIV X C D :

can be easily optimised by a built in mechanism resulting the object code

```
LOAD A
SUM  B
DIV  C
STO  D
```

Due to the low level of basic SEL, it can be easily translated by exploiting the existing machine instructions, e.g. by translating a skip and a jump as a conditional jump; a compare operation and a skip as a compare-skip instruction, etc.

5.2.3. The possibility of defining an object as a constant has a fundamental role in optimising macro generation. An operation on a constant object is executed during procedure generation. E.g. the object identified by

$REF Y

if Y is a constant, is identified during generation. Otherwise it is identified during execution.

5.2.4. An external SEL object is composed of a procedure entry and of a store for its static value. We consider the store required for its name (character string) as not strictly necessary for the execution of a program. Obviously, the space requirement is not minimal if the program 'uses' only the static value of an object. It is true for the basic SEL system. An extension of it can allocate all static values (also types) in a unique vector field, yielding in such a way, an optimum allocation.

5.3. The basic SEL instructions are protected against a few types of error. Any extension implies new conventions introduced into the system, in other words, possibility of new types of error. The strategy of SEL is to offer the means for program error checking. The various levels of boot-strapping allow one to check for any error at the lowest level in which it is detectable; in other words, at compile time or when compiling the compiler, etc. On the other hand, the program itself, which defines a self-extension of SEL, shows clearly the kind of possible errors and how they are checked.

6. Implementation

A first version of SEL has been implemented on the Hewlett Packard 2116/B computer of 8k configuration and has been running since February 1970. Implementation has required about 3,000 machine instructions and six man-months. The present version is not highly optimised, because of its experimental character, and, as we had no backing store, because we wanted to reserve more core memory for programs. Based on our experience we believe that a more sophisticated implementation of SEL would require about 4,500 to 5,000 machine instructions (computers with more sophisticated instructions require more sophisticated optimisation; so, the number of instructions required should be more or less the same) and about 6 to 12 man-months.

The present version is conversational. Some instructions such as delete a line, etc., are included in the basic vocabulary of SEL and therefore can also be generated.

In a future more sophisticated version, we should like to include also instructions for file management. In such a way, operating systems could also be described in SEL.

7. An example of a SEL program

The program in Appendix A defines a self-extension of SEL. After its execution one can declare nouns, adjectives, and other words as pairs of the English and Italian representation. E.g.

NOUN CAT GATTO, DOG CANE;
ADJECTIVE BLACK NERO, BIG GRANDE;
WORDS IS E, THE IL;

Now any program composed of declared English words produces its print-out in Italian and vice versa, taking into account that in English the attribute precedes the subject, while in Italian it follows. If the program were

THE BIG DOG IS BLACK

the print-out would be

IL CANE GRANDE E NERO

For simplicity sake, parameters will be indicated as $p1$, $p2$ and $p3$ and they are meant as the first, second and third object successive to the one referred to.

Name	No. of parameters	Brief description of its procedure
$ASA	2	Assigns the static address value of $p1$ to $p2$
$ASP	2	Assigns the procedure of $p1$ to $p2$
$ASI	2	Assigns the static integer value of $p1$ to $p2$
$CPS	3	Compares the name of $p1$ with that of $p2$, assigns the Boolean result to $p3$
$CPP	3	Compares the procedure of $p1$ with that of $p2$, assigns the Boolean result to $p3$
$IEQ	3	Assigns the Boolean value (int. stat. of $p1$ = int. stat. of $p2$) to $p3$
$IGT	3	Assigns the Boolean value (int. stat. of $p1$ > int. stat. of $p2$) to $p3$
$ILT	3	Assigns the Boolean value (int. stat. of $p1$< int. stat. of $p2$) to $p3$
$VEC	3	Generates a vector field of $p3$ units of type $p2$ and makes $p1$ point to this field
$INC	3	Increases the static reference value of $p1$ by $p3$ units of type $p2$
$DEC	3	Decreases the static reference value of $p1$ by $p3$ units of type $p2$
$NEG	2	Assigns the static integer value of $p1$ negated to $p2$
$RDI	1	Reads an integer value and assigns it to $p1$
$PVI	1	Prints the static integer value of $p1$
$PTS	—	Prints a blank
$PTI	—	Prints a carriage return and linefeed
$TXT	—	Executes the program in the text buffer
$DTX	—	Erases the contents of the text buffer

Fig. 1. List of some basic SEL objects used in the examples

Or, if the program were

IL GATTO NERO E GRANDE

The output of the program would be

THE BLACK CAT IS BIG

Note: (a) Any text printed by the user is just a program not data.
(b) This example of SEL program is not intended to solve any problem of linguistics.

The program in Appendix B defines and applies an interpreter for a simple subset of ALGOL 60. The interpreter is designed for a 'small' computer.

To help with understanding these programs **Fig. 1** gives a list of some of the basic SEL objects used in the examples.

8. Conclusion

Since the basic vocabulary of SEL is not rigorously defined, SEL is a general method of system building rather than a programming language in the conventional sense. However, it approximates the design criteria in Section 1. As for point (3) in the Introduction, we remark that SEL can be extended also horizontally simply by adding a new object to the basic set. However, the degree of altering the system in this way is not independent of the optimisation technique used.

A criticism of SEL is that, because of its self-extensible facility, it requires rather a specialist as a user. In the present stage, an erroneous program can also break down the system. This is the reason why we consider SEL as essentially a tool for system designers.

9. Acknowledgements

Thanks are gladly due to Prof. A. Caracciolo, to Mr. A. Andronico for stimulating discussions, to Mr. Sprugnoli for supervising and Mrs. Carobbi and Mr. Ricciardi for executing the implementation.

Appendix A

```
 *%10  <* $PSB "SYNTAX ERROR" $SYS *> -> ERR  \ERROR ROUTINE\
 %20   $ADR Q $BOL B $AUX B   \Q AND B ARE AUXILIARY VARIABLES\
 %30   <: $ADR $REF $SUC $ASA $SUC Q $NXT $ADR $REF $SUC
 %40   \DEFINES THE TWO NEXT SYMBOLS AS ADDRESS VARIABLES\ $ASA $SUC
 %50   $REF Q $ASA Q $REF $SUC \ASSIGNES TO EACH OF THE.TWO SYMBOLS THE
 %60   REFERENCE TO THE OTHER ONE\  $ASP V $REF Q $ASP W $REF $SUC
 %70   \ASSIGNES THE PROCEDURES OF V AND OF W TO THE TWO SYMBOLS\
 %80   $NXT $CPS $REF $SUC ., B $NXT $SKF B $REP \REPEAT THE PROCEDURE
 %90   IF THE SYMBOL ARE FOLLOWED BY ","\  $PRV $CPS $REF $SUC ; B
 %100 $SKT B ERR $NXT :> -> VOC \TERMINATE THE PROCEDURE IF THE SYMBOLS
 %110 ARE FOLLOWED BY ";", GIVE ERROR MESSAGE OTHERWISE\
 %120 <* $PSB $REF $REF $SUC $PTS $NXT *> -> FL \ROUTINE FOR PRINTING
 %130 THE SUCCESSOR WORD IN THE OTHER LANGUAGE, ADVANCE SUCCESSOR\
 %140 <* $PSB $REF $REF $THS $PTS *> ->  PRINT \ROUTINE FOR PRINTING
 %150 THE WORD IN THE OTHER LANGUAGE\
 %160 <: $CPP $REF $SUC Y B $SKF B FL PRINT :> -> X  \IF THE SUC-
 %170 CESSOR HAS THE PROCEDURE Y, BEFORE PRINTING THIS ONE, PRINT THE
 %180 SUCCESSOR WORD IN THE OTHER LANGUAGE\ <:PRINT:>->Y <:PRINT:>->Z
 %190 \TWO DIFFERENT PROCEDURE IS GENERATED FOR PRINTING\
 %200 <: $ASP Y V $ASP X W $REF VOC :> -> NOUN
 %210 <: $ASP X V $ASP Y W $REF VOC :> -> ADJECTIVE.
 %220 <: $ASP Z V $ASP Z W $REF VOC :> -> WORD
 %230 \ASIGN TO V AND W THE REQUIRED PROCEDURES, CALLS VOC\

*$TXT

*NOUN CAT GATTO, DOG CANE;

*ADJECTIVE BLACK NERO, BIG GRANDE;

*WORD IS E, THE IL;

*THE BIG DOG IS BLACK
IL CANE GRANDE E NERO
*


*IL GATTO NERO E GRANDE
THE BLACK CAT IS BIG
*
```

Appendix B

```
*%10 $ADR $SYM $BOL $B $ASP $NOP :
%20 <* $LAB $REF $SYM $NXT *> -> LB
%30 <: $ASA $SUC $SYM $NXT $CPS $REF $SUC : $B
%40 $SKF $B LB $PGO $B $SKF $B $REP :> -> $PRP
%50  $INT $V $AUX $V $ADR $R $AUX $R
%60  <: $ASA $SUC $R $NXT $ASI $REF $SUC $V $NXT $MUL $V $REF $SUC
%70  $REF $R $NXT $CST $REF $R :> -> STYP
%80  $ADR $ND $ADR $N1 $ADR $PR $ADR $VL $ADR $VK $BOL $B $AUX $B
%90  $TYP $ST $ADR 2 $SUM $ST $INT $ST $VEC $S $ST 20
%100 <: $ASA $S $ND $ASA $S $PR $INC $PR $ADR 1 $ASA $S $VL
%110 $INC $VL $ADR 2 $ASR ; $REF $PR $ASR $SER $REF $ND :> -> $I
%120 $INT ; $ASI 0 ; $I
%130 <: $ASA $ND $N1 $ASA $VL $VK $INC $ND $ST 1 $INC $PR $ST 1 $INC
%140 $VL $ST 1 $ASR $SER $REF $ND :> -> $IS
%150 <: $ASA $N1 $ND $ASA $VK $VL $DEC $N1 $ST 1 $DEC $PR $ST 1
%160 $DEC $VK $ST 1 :> -> $OS
%170 <:$PSB $REF $THS $PTS $PSB "SYNTAX ERROR" $PNT $I $SYS:> ->$SER
%180 <* $PNT $SER *> -> $E $ADR $OP
%190 <: $REF $V $NXT $CPS $REF $SUC , $B $SKF $B K $CPS $REF $SUC ; $B
%200 $SKT $B $E $NXT $RET K:$NXT $REP :> -> $DIC
%210 <: $CPP $REF $REF $ND $RND $B $SKF $B $E $ASA $THS $REF $ND :>
%220 -> $RND -> 0 -> 1 -> 2 -> 20 -> $IRC
%230 <: $CCB $REF $SUC $B $SKF $B $E $INT $REF $SUC $ASP $RND $REF
%240 $SUC :> -> $VI <: $ASP $VI $V $PNT $DIC :> -> INTEGER
%250 <: $CPP $REF $SUC $RND $B $SKT $B $E $RDI $REF $SUC :> -> $V2
%260 <: $ASP $V2 $V $PNT $DIC :> -> READ $ASP $NOP $GOTO
%270 <* $PNT $SER *> -> $E1  $ADR $MON <: $NOP :> -> $SGC
%280 <: $CPS $REF $REF $ND $SER $B $SKF $B $E1 :> -> $C1
%290 <: $PNT $C1 $CPS $REF $REF $N1 $SER $B $SKF $B $E1 :> -> $C2
%300 <: $PNT $C2 $CPP $REF $REF $N1 $RND $B $SKT $B $E1 :> -> $C3
%310 <: $PRV $PRV $CPP $SGC $REF $SUC $B $SKF $B K $PNT $C1 $PVI
%320 $REF $REF $ND $PTS $ASR $SER $REF $ND $NXT $NXT $RET K: $PSB
%330 $REF $SUC $NXT $NXT :> -> $SMP $INT $SMP $ASI 1 $SMP
%340 <: $PNT $IS $ASR $SMP $REF $PR $PTI :> -> PRINT
%350 <: $IGT $REF $THS $REF $REF $PR $B $SKF $B L $REF $REF $REF $PR
%360 $PNT $OS $REP L: $REF $J :> -> $O
%370 <: $PNT $IS $ASA $OP $REF $PR :> -> $SC
%380 <: $ASP $SC $J $PNT $O :> -> $OPR
%390 <: $IGT $REF $REF $PR $REF $THS $B $SKT $B Q $REF $REF $REF $PR
%400 $PNT $OS $REP Q: $REF $J :> -> $OO
%410 <: $ASP $NOP $J $PNT $OO $CPA $ND $S $B $SKT $B $E1 $ASR $SER
%420 $REF $ND :> -> ; $INT ; $ASI 0 ;
%430 <:$ASP $NOP $J $PNT $OO $IEQ $REF $THS $REF $REF $PR $B
%440 $SKT $B $RET $REF $REF $REF $PR :> ->, $INT, $ASI 1,
%450 <:$NOP:>->$MP $INT $MP $ASI 10 $MP
%460 <* $REF $REF $N1 $REF $REF $ND $REF $VK $ASA $VK $REF $N1
%470 *> -> $X  $INT $MN $ASI 10 $MN
%480 <: $PNT $C2 $SUM $X :> -> $BP  $INT $BP $ASI 10 $BP
%490 <: $CPP $REF $REF $ND $SER $B $SKF $B K $PNT $OPR $RET
%500 K: $ASA $MON $OP $PNT $OPR :> -> $PN
%510 <: $PNT $C1 $NEG $REF $REF $ND $REF $VK $ASA $VK $REF $N1 :>->$MN
%520 <: $PNT $C1 $NEG $REF $REF $ND $REF $VL $ASA $VL $REF $ND
%530 $PNT $BP :> -> $BN $INT $BN $ASI 10 $BN
%540 <: $PNT $C2 $MUL $X :> -> $M $INT $M $ASI 11 $M
%550 <: $PNT $C2 $DIV $X :> -> $DL $INT $DL $ASI 11 $DL
%560 <: $ASR $MP $MON $ASR $BP $OP $PNT $PN :> -> + $INT + $ASI 10 +
%570 <: $ASR $MN $MON $ASR $BN $OP $PNT $PN :> -> - $INT - $ASI 10 -
%580 <: $ASR $M $OP $PNT $OPR :> -> * $INT *  $ASI 11 *
%590 <: $ASR $DL $OP $PNT $OPR:> -> / $INT /  $ASI 11 /
%600 <: $PNT $IS $ASR $LP $REF $PR :> -> ( $INT ( $ASI 4 (
%610 <: $CPS $REF $THS ) $B $SKT $B $E1 $ASA $REF $ND $REF $N1
%620 $PNT $OS :>-> $LP $INT $LP $ASI 4 $LP
%630 <: $CPS $REF $REF $PR $LP $B $SKT $B $E1 $REF $REF $REF $PR
%640 :> -> $JX <: $ASP $JX $J $PNT $OO :> -> )  $INT ) $ASI 4 )
%650 <: $CCB $REF $REF $N1 $B $SKF $B $E1 $PNT $C3 $ASI $REF $REF $NB
%660 $REF $REF $N1 :> -> $AS $INT $AS $ASI 2 $AS
%670 <:$ASR $AS $OP $ASP $SC $J $PNT $OO :> -> := $INT := $ASI 2 :=
%680 $INT $T $INT $K $AUX $T $AUX $K
%690 <: $ASI $REF $REF $N1 $T $ASI 1 $K L:$ILT $K $REF $REF $ND $B
%700 $SKT $B $RET $MUL $REF $REF $N1 $T $REF $VK $ASA $VK $REF $N1
%710 $SUM 1 $K $K L :> -> $EX $INT $EX $ASI 12 $EX
%720 <: $ASR $EX $OP $PNT $OPR :> -> ↑ $INT ↑ $ASI 12 ↑
%730 $BOL $IB $AUX $IB $ASP $NOP GOTO
%740 <* $REF $REF $N1 $REF $REF $ND $IB $ASR $IB $REF $N1 *> -> $X1
%750 <: $IGT $X1 :> -> $GT $NT $GT $ASI 9 $GT
%760 <: $ILT $X1 :> -> $LT $INT $LT $ASI 9 $LT
%770 <: $IEQ $X1 :> -> $EQ $INT $EQ $ASI 9 $EQ
%780 <: $IEQ $X1 $NOT $IB $IB :> -> $NE $INT $NE $ASI 9 $NE
%790 <: $ASR $GT $OP $PNT $OPR :> -> >  $INT >  $ASI 9 >
%800 <: $ASR $LT $OP $PNT $OPR :> -> <  $INT < $ASI 9 <
%810 <: $ASR $EQ $OP $PNT $OPR :> -> =  $INT =  $ASI 9 =
%820 <: $ASR $NE $OP $PNT $OPR :> -> # $INT  #  $ASI 9 #
%830 <: $PNT $IS $ASR $SER $REF $PR :> -> IF $INT $SER $ASI 1 $SER
%840 <: $ASP $NOP $J $PNT $OO $CPS $SER $REF $REF $PR $B $SKT $B $E1
%850 $CPS $IB $REF $REF $ND $B $SKT $B $E1 $PNT $OS $SKF $IB $RET
%860 R:$CPS $REF $SUC ELSE $B $SKF $B J $CPS $REF $SUC ; $B
%870 $SKF $B $RET $NXT R J:$NXT :> -> THEN $INT THEN $ASI 1 THEN
%880 <: $CPS $REF $SUC ; $B $SKF $B $RET $CPS $REF $SUC , $B
%890 $SKF $B $RET $NXT $REP :> -> ELSE  $ASP $SYS STOP
$TXT

*$DTX


*%10 INTEGER I,J,N,K,L;
%20  PRINT "UPPER LIMIT "; READ N;; I:=5;
%30  PRINT 1," ST ","PRIME NUMBER = ", 2;
%40  PRINT 2," ND ","PRIME NUMBER = ", 3;
%50  PRINT 3," RD ","PRIME NUMBER = ", 5; K:=3;
%60  ALFA:I:=I+2; IF I>N THEN STOP; J:=2;
%70  BETA: IF I/J*J=I THEN GOTO ALFA ELSE J:=J+1;
%90  IF (J-1)↑2<I THEN GOTO BETA; K:=K+1; L:=K-K/10*10;
%100 PRINT IF L=0 THEN K ELSE L," TH ","PRIME NUMBER = ",I;
%110 GOTO ALFA;
*$TXT

UPPER LIMIT :100

    1  ST PRIME NUMBER =     2
    2  ND PRIME NUMBER =     3
    3  RD PRIME NUMBER =     5
    4  TH PRIME NUMBER =     7
    5  TH PRIME NUMBER =    11
    6  TH PRIME NUMBER =    13
    7  TH PRIME NUMBER =    17
    8  TH PRIME NUMBER =    19
    9  TH PRIME NUMBER =    23
   10  TH PRIME NUMBER =    29
    1  TH PRIME NUMBER =    31
    2  TH PRIME NUMBER =    37
    3  TH PRIME NUMBER =    41
    4  TH PRIME NUMBER =    43
    5  TH PRIME NUMBER =    47
    6  TH PRIME NUMBER =    53
    7  TH PRIME NUMBER =    59
    8  TH PRIME NUMBER =    61
    9  TH PRIME NUMBER =    67
   20  TH PRIME NUMBER =    71
    1  TH PRIME NUMBER =    73
    2  TH PRIME NUMBER =    79
    3  TH PRIME NUMBER =    83
    4  TH PRIME NUMBER =    89
    5  TH PRIME NUMBER =    97
```