WHAT'S IN THE MANUAL

Notation

The following syntax notation is used in this manual:

{ *item* }	means any number of repetitions of the *item* (or none at all)
[*Hem*]	means the *item* is optional

Keywords, operators and delimiters, such as get, procedure, := and) are written in boldface. Comments in example Turing programs are written in *italics* in a smaller font than the rest of the program. Identifiers are written in *italics.* Explicit constants such as 27 or "Hello" are written normally.

Items are presented by giving their *syntax,* a general *description of* their meaning, *examples* of their use, and then *details* about the item. The description is intended to satisfy the reader who mainly wants to know basic information about the item, for example, that the string type represents character strings. The examples illustrate important patterns of usage of the item, and should in many cases answer the question in mind. The detailed information that follows gives a full technical description of the item as well as references to related items. Virtually all the information in the Turing Report appears in this Manual, although not in its original form. This Manual takes liberties with the original syntax of the language to make explanations easier to understand. For example, the Manual describes *declarations* as a single item, and then explains restrictions that disallow certain declarations from appearing in particular contexts. By contrast, the Report uses the form of the context free syntax to imply, in a less obvious way, these same restrictions.

The Turing language was designed so it can be easily implemented in a compatible way on many types of hardware. This *portability* of the language is important because it allows us to write our programs without concern for the technical details of the underlying hardware. But sometimes the user wants the opposite of portability and wishes to use the particular aspects of the hardware at hand. For example, graphics devices remain highly dependent on individual hardware. To adapt to this reality, a number of extensions have been added to the implementations of Turing. These extensions, such as the drawbox procedure, are described in this Manual, along with the inherent features of the language. The reader should be aware that these extensions may not be available in some implementations of Turing.

pesign of the Turing Language

The Turing language has been designed to be a general purpose ge, meaning that it is potentially the language of choice for a wide 1 siof applicat'ons- Because of its combination of convenience and °aSfessive power it is particularly attractive for learning and teaching. Because of its clean syntax, Turing programs are relatively easy to read and write.

The language helps in the writing of reliable programs by eliminating or constraining error-prone language features; for example, Turing eliminates the goto statement and constrains pointers to locate elements of collections. It provides many compile-time and run-time checks to catch bugs before they lead to disaster. These checks guarantee that a Turing program behaves according to this reference manual, or else a warning message is given.

There are production versions of Turing that provide maximal efficiency by allowing you to remove run-time checking. Using these versions, Turing programs are as efficient as programs written in machine-oriented languages such as C. This efficiency arises from the fact that each construct in Turing has an obvious, efficient implementation on existing computer hardware.

Turing has been designed to eliminate the security problems of languages such as Pascal. For example, Turing's variant records (unions) have an explicit tag field that determines the active variant of the record; run-time checks guarantee that the program can never access fields that have been assigned in a different variant. In principle, a Turing compiler prevents function *side effects* (changing values outside of the function) and *aliasing* (having more than one way to modify a given value). However, existing Turing implementations do not necessarily enforce these restrictions.

Turing has modules, which are information hiding units. These modules are *objects* in the sense of "object-oriented programming"- They allow the programmer to divided the program into units that have Precisely controlled interfaces.

fr m the ei ghth grade and to university graduate courses. Turing has been extensively used to teach programming concepts,

Turing has been designed so it can be supported by either compilers IBM pIP"*61,,8- At present (Fall 1989) the Turing system used for teaching on sPeak COmPatibles is called an interpreterfalthough technically intern "f " 'S * compiler that generates pseudo-code). This same P eter runs on Macintoshes, Icons, Suns, Vaxes and IBM mainframes.

The Turing compilers and interpreters have all been written in Turing or in its extension called Turing Plus. Turing Plus includes systems programming features such as concurrency, exception handling, explicit type cheats and separate compilation. The Turing Plus compiler optionally generates C code instead of machine language.

Basic Concepts in Turing

Like most programming languages, Turing has variables whose values are changed by assignment statements. Example:

```
var i: int          % This declaration creates variable i
i := 10             % This assignment sets the value ofi to 10
```

This example uses comments that begin with a percent sign (%) and which end with the end of the line. The names of items, such as)', are *identifiers.*

A *named constant* is a named item whose value cannot change. In the following, *c* and *x* are named constants:

```
const c r= 25
const x := sin ( y ) ** 2
```

In this example, c's value is known at compile-time, so it is a *compile-time* (or *manifest)* value. In the Pascal language, all named constants are compile-time values, but in Turing these values may be computed at run-time, as is the case for *x* in this example.

An *explicit constant* (or *literal constant)* is a constant that denotes its own value; for example, 27 is an explicit constant and so is "Hello".

In a Turing program, each named item such as a variable has a lifetime, which is called its *scope.* The lifetime of the item begins with its declaration and lasts to the end of the construct (such as a procedure or a loop) in which it is declared. More detail can be found under *declaration* in the main text. Turing's *scope rules* tell you where an item can be used. These rules are very similar to rules in Pascal.

Turing, like Pascal, has two kinds of subprograms, called *procedures* and *functions.* Procedures can be thought of as named sequences of statements and functions as operators that map values to values.

Turing allows you to declare an array whose size is not known until run time; these are called *dynamic arrays.* This is in contrast to languages such as Pascal in which the sizes of arrays must be known at compile time.

Turing can be thought of either as an *algorithmic language,* in which you write algorithms that will not necessarily be run on a computer,

vramming language whose purpose is to be used on a computer. In or as a F ^cgse^ me language is called *Ideal Turing* and is a mathematical the forrne mucn 'as algebra and arithmetic are mathematical notations. In n0ta/r''ri'n2, numbers have perfect accuracy (this is the nature of pure *Ideal* X'^ and there js no Jimit on tne size of a program or its data. By ""ntrast, *Implemented Turing* is a language that you can use on actual computers.

Implemented Turing (which we usually call simply Turing) pproximates Ideal Turing as closely as is practical. However, its integers have a fixed maximum range (31 bits of precision) and its real numbers have limited precision (about 16 decimal digits) and limited size of exponent. Whenever Implemented Turing cannot carry out your program in the same way as Ideal Turing, it gives you a warning message, although it does not warn you about the limited precision of real numbers. For example, if your program tries to create an array with a million elements in it, but there is not enough space, you will be warned.

Implemented Turing checks to make sure that your running program is meaningful; for example, it checks that variables are initialized before being used, that array subscripts are in bounds, and that pointers locate actual values. This checking guarantees *faithful execution,* which means that your program behaves as it would in *Ideal Turing* or else you get a warning message. In production versions of Turing, this checking can be turned off, for maximal efficiency of production programs.

Compile-time expressions

In certain situations, the Turing language requires you to use values that are known at compile time. Such values are called *compile-time values.* For example, the maximum length of a string type, such as the N in string(N), must be known at compile time; you are not allowed to read a value from the input and use it as the maximum length of a string. Here is a list of the places where *compile-time values* are required:

(a) The maximum size N of a string, as in string(N).
(b) The value of a label in a case statement or in a union type.
(c) Each value used in init to initialise an array, record or union.
(d) The lower bound L and upper bound U of a subrange, as in L.. *U* , ith one exception. The exception, called a *dynamic array,* is an array eclared using var or const that is not part of a record, uniort, or another array; in a dynamic array, the upper bound *U* (but not the lower bound L) is allowed to be read or computed at run time.

Case (d) implies that the size of set types is known at compile time.

The technical definition of a *compile-time* value is any expression th at consists of only:

(1) explicit integer, real, boolean and string constants, such as 25, false, and *"Charles DeGaulle"*, as well as enumerated values such as *color.red*.

(2) Set constructors containing only compile-time values or all.
(3) Named constants that name compile-time values.
(4) Results of the integer operators +, -, *, div and mod.
(5) Results of the *chr* and *ord* functions.
(6) Results of the catenation operator +.
(7) Parenthesised versions of any of these expressions.

You are not allowed to use *run-time values* (any values not covered by items 1-7) in the places listed above (a-d).

Technical Problems

Any technical problems or questions should be directed to our technical support line at Holt Software Associates. The number for this support line (416)978-8363.

Acknowledgements

J.R.Cordy, the co-designer of the Turing Language, is acknowledged as the source of much of the material in this manual. S.G. Perelgut has rewritten the Introductory section of this manual and T.L.West has done the final work - correcting this manual, preparing appendices and preparing it for publication.

LIST OF TECHNICAL TERMS

absolute value function

SYNTAX:

abs(*expn*)

DESCRIPTION: The abs function is used to find the absolute value of a number (the *expn).* For example, abs (-23) is 23.

EXAMPLE: This program outputs 9.83.

```
varx : real := -9.83
put abS (X )        % Outputs 9.83
```

DETAILS: The abs function accepts numbers that are either int's or real's; the type of the result is the same type as the accepted number. The abs function is often used to see if one number is within a given distance *d* of another number y; for example:

```
if abs (x - y) <= d then ...
```

all all members of a set type

SYNTAX:

setTypeName (all)

DESCRIPTION: Given a set type named *S*, the set of all of the possible elements of *S* is written *S* (all).

EXAMPLE:

```
type smallSet : set of 0 .. 2
var x : smallSet := smallSet (all)
                    % set x contains elements 0, 1 and 2
```

DETAILS: See set type for details about sets.

(boolean) operator

SYNTAX:

A and *B*

DESCRIPTION: The and (boolean) operator yields a result of true if and only if both operands are true, and is a short circuit operator; for example, if *A* is false in *A* and B then B is not evaluated.

EXAMPLE:

```
var success : boolean := false
var continuing := true   % the type is boolean

continuing := continuing and success
```

DETAILS: *continuing* is set to true if and only if both *continuing* and *success* are true. Since Turing uses short circuit operators, once *continuing* is false, *success* will not be looked at.

See also *boolean* (which discusses true/false values), *explidtTrueFalseConstant* (which discusses the values true and false), *precedence* and *expn* (expression).

LO.

arctail arctangent function (radians)

SYNTAX:

```
arctan ( r : real): real
```

DESCRIPTION: The arctan function is used to find the arc tangent of an value. The result is given in radians. For example, arctan (1) is pi/4.

EXAMPLE: This program prints out the arctangent of 0 through 3 in radians.

```
for/:0..12
    const arg := i 1 4
    put "Arc tangent of", arg," is",
        arctan ( a r g ) , " radians"
end for
```

DETAILS: See also the arctand function which finds the arc tangent of an value with the result given in degrees. (2 * pi radians are the same as 360 degrees.)

NOTE: The formulae for arcsin and arccos are:

arcsin *(x)* = arctan (sqrt *((x*x)* / (1 - (x*x))))

arccos *(x)* = arctan (sqrt ((1 - *(x *x)*) / *(x *x)*))

arctangent function (degrees)

SYNTAX:

```
arctand ( r : real) : real
```

DESCRIPTION: The arctand function is used to find the arc tangent of an angle given in degrees. For example, arctand (0) is 0.

EXAMPLE: This program prints out the arctangent of values from 0 to 3 in degrees.

```
for/:0..12
    const arg:= i / 4
    put "Arc tangent of ", arg, " is ",
        arctand ( arg ) , " degrees"
end for
```

DETAILS: See also the arctan function which finds the arc tangent of an value with the result given in radians. (2 * pi radians are the same as 360 degrees.)

array type

SYNTAX: An array Type is:

array *indexType { , indexType}* of *typeSpec*

DESCRIPTION: An array consists of a number of elements. The *typeSpec* gives the type of these elements. There is one element for each item in the (combinations of) range(s) of the *indexType(s).* In the following example, the array called *marks* consists of 100 elements, each of which is an integer.

EXAMPLE:

```
var marks : array 1 .. 100 of int

var sum : int := 0
for / :1 ..100          % Add up the elements of marks
    sum \- sum + marks (/)
end for
```

DETAILS: In the above example, *rmrks(i)* is the i-th element of the *marks* array. We call *i the index or subscript of marks.* In Turing, a subscript is surrounded by parentheses, not by square brackets as is the case in the Pascal language.

EXAMPLE: The *prices* array shows how an array can have more than one dimension. This array has one dimension for the year (1988,1989 or 1990) and another for the month (1 .. 12). There are 36 elements of the array, one for each month of each year.

```
var price : array 1988 .. 1990,1 .. 12 of int

var sum : int, := 0
for year : 1988 .. 1990          % For each year
    for month : 1 .. 12          % For each month
        sum := sum + price (year, month)
    end for
end for
```

DETAILS: Each *indexType* must contain at least one item, for example, the range 1 .. 0 would not be allowed. Each index type must be a subrange

of the integers or of an enumerated type, an (entire) enumerated type, or a named type which is one of these.

Arrays can be assigned as a whole (to arrays of an equivalent type), but they cannot be compared.

An array can be initialized in its declaration using init; for details, see var and const declarations.

EXAMPLE: In this example, the size of the array is not known until run time.

```
var howMany : int
get howMany
var height : array 1 .. howMany of real
    ...read in all the elements of this array...
function total (a: array 1 .. * of real) : real
    var sum : int := 0
    for / : 1 .. upper ( a)
        sum := sum + a (/ )
    end for
    result sum
end total
put "Sum of the heights is ", total ( height )
```

DETAILS: The ends of the range of a subscript are called the *bounds* of the array. If these values are not known until run time, the array is said to be *dynamic.* In the above example, *height* is a dynamic array. Dynamic arrays can be declared as variables or constants, as in the case for *height.* However, dynamic arrays cannot appear inside other types such as records and cannot be named types. Dynamic arrays cannot be assigned and cannot be initialized using init.

In the above example, *upper(a)* returns the size of *a.* See also upper and lower.

In the declaration of an array parameter, the upper bound can be given as a star (*), as is done in the above example. This means that the upper bound is taken from that of the corresponding actual parameter (from *height* in this example).

You can have arrays of other types, for example arrays of record. If R is an array of records, then *R(i).f* is the way to access the / field of the f-th element of array R.

assert statement

SYNTAX: An *assertStatement* is:

assert *trueFalseExpn*

DESCRIPTION: An assert statement is used to make sure that a certain requirement is met; this requirement is given by the *trueFalseExpn*. The *trueFalseExpn* is evaluated. If it is true, all is well and execution continues. If it is false, execution is terminated with an appropriate message.

EXAMPLE: Make sure that *n* is positive.

```
assert n >= 0
```

EXAMPLE: This program assumes that the *textFile* exists and can be opened, in other words, that the open will set the *fileNumber* to a non-zero stream number. If this is not true, the programmer wants the program halted immediately.

```
var fileNumber : int
open : fileNumber, "textFile", read
assert fileNumber not= 0
```

DETAILS: In some Turing systems, checking can be turned off. If checking is turned off, assert statements may be ignored and as a result never cause termination.

assignability
of an expression to a variable

DESCRIPTION: A value, such as 24, is assignable to a variable, such as *i*, if certain rules are followed. These rules, given in detail below, are called the *assignability* rules. They must be followed in assignment statements as well as when passing values to non-var parameters.

EXAMPLES:

```
var i : int
i := 24                 % 24 is assignable to i

var width : 0 .. 319
width := 3 * i          % 3 * i is assignable to width

var a : array 1 .. 25 of string
a (i) := "Ralph"        % "Ralph" is assignable to a(i)

var name : string (20)
name := a (i)           % a(i) is assignable to name


var b : array 1 .. 25 of string
b := a                  % Array a is assignable to b

type personType :
    record
        age : int
        name: string (20)
    end record
var r, s: personType

s := r                  % Record r is assignable to s
```

: The expression on the right of := must be *assignable* to the variable on the left. An expression passed to a non-var parameter must be assignable to the corresponding parameter.

An expression is defined to be *assignable* to a variable if the their two *root* types are equivalent or if an integer value is being assigned to a real variable (in which case the integer value is

automatically converted to real). Two types are considered to be equivalent if they are essentially the same type (see *equivalence* for the detailed definition of this term).

We now define *root* type. In most cases a root type is simply the type itself. The exceptions are subranges and strings. The root type of a subrange, such as 0 .. 319, is the type of its bounds, int type in this example. The root type of a string, such as the type string(9), is the most general string type, namely string.

When a subrange variable, such as *width*, is used as an expression, for example on the right side of an assignment statement, its type is considered to be the *root* type, integer in this case, rather than the subrange. When an expression is assigned to a subrange variable such as *width*, the value, 3*i in this example, must lie in the subrange. Analogously, any string variable used in an expression is considered to be of the most general type of string. When a string value is assigned to a string variable its length must not exceed the variable's maximum length.

statement

SYNTAX: A *assignmentStatement* is;

variableReference := expn

DESCRIPTION: An assignment statement calculates the value of the expression *(expn)* and assigns that value to the variable *(variableReference).*

EXAMPLES:

```
var i : int
i := 24                    % Variable i becomes 24
var a : array 1 .. 25 of string
a(i) := "Ralph"            % The i-th element of a becomes "Ralph"

var b : array 1 .. 25 of string
b := a                     % Array b becomes (is assigned) array a
```

DETAILS: The expression on the right of := must be *assignable* to the variable on the left. For example, in the above, any integer value, such as 24, is assignable to *i*, but a real value such as 3.14 would not be not assignable to i. Entire arrays, records and unions can be assigned. For example, in the above, array *a* is assigned to array *b*. See *assignability* for the exact rules of allowed assignments.

You cannot assign a new value to a constant (const).

begin statement

SYNTAX: A *beginStatement* is:

```
begin
    statementsAndDeclarations
end
```

DESCRIPTION: A begin statement is used to limit the scope of declarations within it. In Turing, begin is rarely used, because declarations can appear wherever statements can appear and because every structured statement such as if ends with an explicit end.

E>(AMPLE:

```
begin
    var bigArray : array 1 .. 2000 of real
    ... bigArray will exist only inside this begin statement-
end
```

"-JET AILS: In Pascal programs, begin statements are quite common because they are required for grouping two of more statements, for example, to group the statements that follow then. In Turing this is not necessary, because each place where you can write a single statement, you can as well write several statements.

declaration

SYNTAX: A *bindDedaration* is: -

```
bind [var] id to variableReference
    { , [var] id to variableReference }
```

DESCRIPTION: The bind declaration creates a new name (or names) for a variable reference (or references). You are allowed to change the named item only if you specify var.

EXAMPLE: Rename the n-th element of array A so it is called *item* and then change this element to 15.

```
bind var item to A ( n )
item := 15
```

DETAILS: The scope of the identifier *(item* above) begins with the bind declaration and lasts to the end of the surrounding program or statement (or to the end of the surrounding part of a case or if statement). During this scope, the original name of the variable *(A* above) reference cannot be used. During this scope, a change to a subscript *(i* above) that occurred in the variable reference does not change the element that the identifier refers to.

You are not allowed to use bind at the outermost level of the main program (except nested inside statements such as if) or at the outermost level in a module.

body declaration

SYNTAX: A *bodyDeclaration* is one of:

(a) **body procedure** *procedureId*
statementsAndDeclarations
end *procedureId*

(a) **body function** *functionId*
statementsAndDeclarations
end *functionId*

DESCRIPTION: A procedure or function is declared to be forward when you want to define its header but not its body. This is the case when one procedure or function calls another which in turn calls the first; this situation is called *mutual recursion.* The use of forward is necessary in this case because every item must be declared before it can be used. Following the forward declaration must come a body declaration for the same procedure or function. For details, see forward declarations.

EXAMPLES: The example given here is part of a complete Turing program that is given with the explanation of forward declarations.

```
var token: string

forward procedure expn (var eValue: real)
    import (forward term, var token)
... other declarations appear here ...
body procedure expn
    var nextValue: real
    term (eValue)                % Evaluate t
    loop                         % Evaluate { + 1}
        exit when token not= "+"
        get token
        term (nextValue)
        eValue := eValue + nextValue
    end loop
end expn
```


pcrRjPTION: The syntax of a *bodyDeclaration* presented above has been simplified by omitting the optional result identifier, import list, re and post condition and init clause. See procedure and function declarations for descriptions of these omissions.

boolean type
(the true-false type)

SYNTAX: **boolean**

DESCRIPTION: The boolean type is used for values are are either true or false. These true-false values can be combined by various operators such as or and and.

EXAMPLE:

```
var success : boolean := false
var continuing := true          % The type is boolean

success := mark >= 60
continuing := success and continuing
if continuing then ...
```

DETAILS: This type is named after the British mathematician, George Boole, who formulated laws of logic.

The operators for true and false are and, or, =>, and not. For two true/false values *A* and B, these operators are defined as follows:

A and B is true when both are true
A or B is true when either or both are true
A => *B* (A implies B) is true when both are true or when A is false
not *A* is true when A is false

The and operator has higher precedence than or, so *A* or B and C means *A 01 (B* and C).

The operators or, and and => are short circuit operators; for example, if *A* is true in *A* or B, B is not evaluated.

See also *explicitTrueFalseConstant* (which discusses the values true and false), *precedence* and *expn* (expression).

statement

SYNTAX: A *caseStatement* is:

```
case expn of
    { label expn {, expn } :
        statementsAndDeclarations }
    [ label :
        statementsAndDeclarations ]
end case
```

DESCRIPTION: A case statement is used to choose among a set of Statements (and declarations). One set is chosen and executed and then execution continues just beyond end case.

The expression *(expn)* following the keyword case is evaluated and used to select one of the alternatives (sets of declarations and statements) for execution. The selected alternative is the one having a label value equalling the case expression. If none are equal and there is a final label with no expression, that alternative is selected.

EXAMPLE: Output a message based on value of mark.

```
case mark of
    label 9, 10    put "Excellent"
    label 7, 8 :   put "Good"
    label 6 :      put "Fair"
    label:         put "Poor"
end case
```

DETAILS: The case expression is required to match one of the labels or else there must be a final label with no expression. Label expressions must have values known at compile time. All labels values must be distinct. The case expression and the label values must have the same type, which must be integer or an enum type.

catenation (+)

joining together strings

SYNTAX: A *catenation* is:

stringExpn + stringExpn

DESCRIPTION: Two strings *(stringExpns)* can be joined together (catenated) using the + operator.

EXAMPLES:

```
var lastName, wholeName: string
lastName := "Austere"
wholeName := "Nancy" +" " + lastName
                % The three strings Nancy, a blank and Austere
                % catenated together to make the string
    ;           % "Nancy Austere". This string becomes the
                % value of wholeName
```

DETAILS: The length of a catenation is limited to at most 255 characters.

See also *substrings* (for separating a strings into parts), repeat (for making repeated catenations), string type, length, and index (to determine where one string is located inside another).

Catenation is sometimes call *concatenation.*

real-to-integer function

SYNTAX:

ceil(*r* : real): int

DESCRIPTION: Returns the smallest integer greater than or equal to *r*.

DETAILS: The ceil (ceiling) function is used to convert a real number to an integer. The result is the smallest integer that is greater than or equal to *r*. In other words, the ceil function rounds up to the nearest integer. For example, ceil (3) is 3, ceil (2.25) is 3 and ceil(-8.43) is -8.

See also the floor and round functions.

chr integer-to-character function

SYNTAX:

chr(/: int): string (1)

DESCRIPTION: The chr function is used to convert an integer to a character, that is, to a string of length 1. The character is the /'-th character of the ASCII sequence of characters (except on the IBM mainframe, which uses the EBCDIC sequence.) For example, chr (65) is "A".

The selected character must not be number 0 (a reserved character used to mark the end of a string) or 128 (a reserved character used to mark uninitialised strings). The ord function is the inverse of chr, so for any character *c*, chr (ord (c)) = *c*.

See also the ord, intstr and strint functions.

millisecs used procedure [PC, Mac and Unix only]

SYNTAX:

clock (var *c* : int)

DESCRIPTION: The clock statement is used to determine the amount of time since this program (process) started running. Variable *c* is assigned the number of milliseconds since the program started running.

EXAMPLE: This program tells you how much time it has used.

```
var timeRunning : int
clock ( timeRunning )
put "This program has run ", timeRunning,
        "milliseconds"
```

DETAILS: See also the delay, time, sysclock, wallclock and date statements.

On IBM PC compatibles, this is the total time since the Turing system was started up and the hardware resolution of duration is in units of 55 milliseconds. For example, clock(i) may be off by as much as 55 milliseconds.

On Apple Macintoshes, this is the total time since the machine was turned on and the hardware resolution of duration is in units of 17 milliseconds (1/60-th of a second).

close file statement

SYNTAX: A closeStatement is:

close : *fileNumber*

DESCRIPTION: In Turing, files are read and written *using a fileNumber*. In most cases, this number is gotten using the open statement, which translates a file name, such as "Master", to a file number, such as 5. When the program is finished using the file, it disconnects from the file using the close statement.

EXAMPLE: This programs illustrates how to open, read and then close a file.

```
var fileName: string := "Master"   % Name of me
var fileNo : int                   % Number of me
var inputVariable: string (100)
open : fileNo, fileName, read

read : fileNo, inputVariable

close : fileNo
```

DETAILS: In a Turing implementation, there will generally be a limit on the number of currently open files; this limit will typically be around 10. To avoid exceeding this limit, a program that uses many files one after another should close files that are no longer in use.

If a program does not close a file, the file will be automatically closed when the program finishes.

See also the open, get, put, read, write, seek and tell statements.

There is an older and still acceptable version of close that has this syntax:

close (*fileNumber : int*)

clear screen graphics procedure

SYNTAX:

els

DESCRIPTION: The els (clear. screen) procedure is used to blank the screen. The cursor is set to the top left (to row 1, column 1).

DETAILS: In *"graphics"* mode all pixels are set to color number 0, so the screen is displayed in background color.

The screen should be in a *"screen"* or *"graphics"* mode; if not, it will automatically be set to *"screen"* mode. Seesetscreen for details.

Collection declaration

SYNTAX: A *collectionDedaration* is one of:

(a) **var *id { , id } :* collection of** *typeSpec*
(t>) **var** *id { , id } :* **collection of forward** *typeId*

DESCRIPTION: A collection declaration creates a new collection (or collections). A collection can be thought of as an array whose elements are dynamically created (by new) and deleted (by free). Elements of a collection are referred to by the collection's name subscripted by a pointer. See also new, free and pointer.

EXAMPLE: Create a collection that will represent a binary tree.

```
var tree : collection of
    record
        name : string ( 1 0 )
        left, right: pointer to tree
    end record

var root: pointer to tree
new tree, root
tree (root).name := "Adam"
```

DETAILS: The statement "new *C,p"* creates a new element in collection C and set *p* to point at it; however, if there is no more memory space for the element, *p* is set to *nil (C*), which is the null pointer for collection C. The statement "free *C,p"* deletes the element of C pointed to by *p* and sets *p* to *nil (C)* . In each case, *p* is passed as a var parameter and must be a variable of the pointer type of C.

The keyword forward (form b above) is used to specify that the *typeId* of the collection elements will be given later in the collection's scope. The later declaration must appear at the same level (in the same list of declarations and statements) as the original declaration. This allows cyclic collections, for example, when a collection contains pointers to another collection which in turn contains pointers to the first collection. In this case, the *typeId* is the name of the type that has not yet been declared; *typeId* cannot be used until its declaration appears. A collection whose element type is forward can be used only to declare pointers to it until the type's declaration is given.

Suppose pointer *q* is equal to pointer *p* and the element they noint to is deleted by "free *C,p".* We say <j is a *dangling pointer* because it seems to locate an element, but the element no longer exists. A dangling pointer is considered to be an uninitialised value; it cannot be assigned, compared, used as a collection subscript, or passed to free.

Collections cannot be assigned, compared, passed as parameters, bound to, or named by a const declaration. Collections must not be declared in procedures, functions, records or unions.

Color text color graphics procedure

SYNTAX:

color (*Color*: int)

DESCRIPTION: The color procedure is used to change the currently active color. This is the color of characters that are to be put on the screen. The alternate spelling is colour .

EXAMPLE: This program prints out the message "Bravo" three times, each in a different color.

```
setscreen ("graphics")
for /: 1 .. 3
    color (/')
    put "Bravo"
end for
```

EXAMPLE: This program prints out a message. The color of each letter is different from the preceding letter. For letter number *i* the color number is /' mod maxcolor + 1. This cycles repeatedly through all the available colors.

```
setscreen ("screen")
const message := "Happy New Year!!"
for /: 1 .. length (message)
    color (/ mod maxcolor + 1)
    put message (/)..
end for
```

DETAILS: See setscreen for the number of colors available in the various *"graphics"* modes. On Unix systems, color may have no action.

The screen should be in a *"screen"* or *"graphics"* mode; if not, it will automatically beset to *"screen"* mode. See setscreen for details.

See also colorback, whatcolor, whattextcolor and maxcolor.

Colorback background color graphics procedure [PC only]

SYNTAX:

colorback (*Color:* **int)**

DESCRIPTION: The colorback procedure is used to change the current background color. The alternate spelling is colourback .

In *"screen"* mode on IBM PC compatibles, this sets the background color to one of the colors numbered 0 to 7. This is the color that surrounds characters when they are put onto the screen. On Unix dumb terminals, colorback(l) turns on highlighting and colorback(O) turns it off. On other systems, this procedure may have no effect.

In *"graphics"* mode on IBM PC compatibles, this is used to associate a color with pixel color number 0, which is considered to be the background color. Using colorback immediately changes the color being displayed for all pixels with color number 0.

EXAMPLE: Since this program is in *"screen"* mode, changing the background color has no immediately observable effect. When the message "Greetings" is output, the background surrounding each letter will be in color number 2.

```
setscreen ("screen")

colorback (2)
put "Greetings"
```

kXAMPLE: Since this program is in *"graphics"* mode, changing the background color immediately changes the colors of all pixels whose color number is 0.

```
setscreen ("graphics")

colorback(2)
```

' AILS: The screen should be in a *"screen"* or *"graphics"* mode; if not, it will automatically be set to *"screen"* mode. See setscreen for details. See also color and whatcolorback.

comment

DESCRIPTION: A *comment* is a remark to the reader of the program, which is ignored by the computer. The most common form of comment in Turing starts with a percent sign (%) and continues to the end of the current line; this is called an *end-of-line* comment. There is also the *bracketed* comment, which begins with the /* and ends with */ and which can continue across line boundaries.

EXAMPLES:

```
% This is an end-of-line comment
var x : real          % Here is another end-of-line comment
const s := "Hello"
/* Here is a bracketed comment that
     lasts for two lines */
const pi := 3.14159
```

DETAILS: In the Basic language, comments are called *remarks* and start with the keyword REM. In Pascal, comments are bracketed by (* and *).

comparisonOperator

: A *comparisonOperator* is one of:

```
(a)          % Less than
(b)          % Greater than
(c)  =       % Equal
(d)  <=      % Less than or equal; subset
(e)  >=      % Greater than or equal; superset
(f)  not=    % Not equal
```

DESCRIPTION: A comparison operator is placed between two values to determine their equality or ordering. For example, 7 > 2 is true and so is "Adam" < "Cathy". The comparison operators can be applied to numbers as well as to enumerated types. They can also be applied to strings to see determine the *ordering* between strings (see the string type for details). Arrays, records, unions and collections cannot be compared. Boolean values (true and false) can be compared only for equality (= and not=); the same applies to pointer values. Set values can be compared using <= and >=, which are the subset and superset operators. The not= operator can be written as ~=.

See also *infix* operators and *precedence* of operators. See also the int, real, string, set, boolean and enum types. See also string comparison.

const

constant declaration

SYNTAX: A *constantDeclaration* is:

const *id [: typeSpec] := initializingValue*

DESCRIPTION: A const declaration creates a name *id* for a value.

EXAMPLES:

```
const c := 3
const s := "Hello"          % The type ofs is string
const x := sin(y) ** 2
const a : array 1..3 of int := init(1, 2, 3)
const b: array 1..3 of int := a
const c: array 1..2,1..2 of int := init(1, 2, 3 , 4)
                % So c(1,1)=1, c(1,2)=2, C(2,1)=3, c(2,2)=4
```

DETAILS: The initialising value can be an arbitrary value or else a list of items separated by commas inside init (…). The syntax of *initializingValue is:*

a. *expn*
b. init (*initializingValue, initializingValue)*

Each init (…) corresponds to an array, record or union value that is being initialized; these must be nested for initialisation of nested types. In the Pascal language, constants must have values known at compile time; Turing has no such restriction.

When the *typeSpec* is omitted, the variable's type is taken to be the (root) type of the initialising expression, for example, int or string. The *typeSpec* cannot be omitted for dynamic arrays or when the initialising value is of the form init (…). The values inside init (…) must be known at compile time.

The keyword pervasive can be inserted just after const. When this is done, the constant is visible inside all subconstructs of the constant's scope. Without pervasive the constant is not visible inside modules unless explicitly imported. Pervasive constants need not be imported. You can abbreviate pervasive as a star (*).

constantReference

use of a named constant

SYNTAX: A *constantReference* is:

constantId {componentSelector }

DESCRIPTION: In a Turing program, a constant is declared and given a name *(constantId)* and then used. Each use is called a *constant reference.*

If the constant is an array, record or union, its parts *(components)* can be selected using subscripts and field names (using *componentSelectors).* The form of a *componentSelector* is one of:

(a) (*expn {, expn]*)
(b) *.fieldId*

Form (a) is used for subscripting (indexing) arrays. The number of array subscripts must be the same as in the array's declaration. Form (b) is used for selecting a field of a record or union. The use of component selectors is the same as for variable references; see *variableReference* for details. See also const declaration and *explicitConstant.*

EXAMPLES:

```
var radius : real
const pi := 3.14159     % Constant declaration

put "Area is:", pi * radius **2
                        % pi is a constant reference
```

COS cosine function (radians)

SYNTAX:

cos (*r :* real): real

DESCRIPTION: The cos function is used to find the cosine of an angle given in radians. For example, cos (0) is 1.

EXAMPLE: This program prints out the cosine of pi/6,2*pi/6,3*pi/6, up to 12*pi/6 radians.

```
constp/:=3.14159
for/:1..12
    const angle := /* pi / 6
    put "Cos of", angle," is ", cos ( angle)
end for
```

DETAILS: See also the cosd function which finds the cosine of an angle given in degrees. (2 * pi radians are the same as 360 degrees.)

cosine function (degrees)

SYNTAX:

cosd (*r :* real): real

DESCRIPTION: The cosd function is used to find the cosine of an angle given in degrees. For example, cosd (0) is 1.

EXAMPLE: This program prints out the cosine of 30,60,90, up to 360 degrees.

```
for/:1 .. 12
    const angle := / * 30
    put "Cos of", angle," is", cosd (angle)
end for
```

DETAILS: See also the cos function which finds the cosine of an angle given in radians. (2 * pi radians are the same as 360 degrees.)

SYNTAX:

date (var *d* : string)

DESCRIPTION: The date statement is used to determine the current date. Variable *d* is assigned a string in the format *"dd mmm yy"*, where *mmm* is the first 3 characters of the month, e.g., *"Apr"*. For example, if the date is Christmas 1989, *d* will be set to *"25 Dec 89"*.

EXAMPLE: This program greets you and tells you the date.

```
var today : string
date (today)
put "Greetings!! The date today is ", today
```

DETAILS: See also the delay, clock, sysclock, wallclock and time statements.

Be warned that on some computers such as IBM PC compatibles or Apple Macintoshes, the date may not be set correctly in the operating system; in that case, the date procedure will give incorrect results.

declaration

SYNTAX: A *declaration* is one of:

(a) *variableDeclaration*
(b) *constantDeclaration*
(0 *typeDeclaration*
(d) *bindDeclaration*
(e) *procedureDeclaration*
(f) *functionDeclaration*
(9) *moduleDeclaration*

DESCRIPTION: A *declaration* creates a new name (or names) for a variable, constant, type, procedure, function or module. These names are called *identifiers*, where *id* is the abbreviation for *identifier*.

EXAMPLES:

```
var width : int                 % Variable declaration
const pi := 3.14159             % Constant declaration
type range : 0 .. 150           % Type declaration

procedure greet                 % Procedure declaration
    put "Hello world"
end greet
```

DETAILS: Ordinarily, each new name must be distinct from names that are already visible; that is, redeclaration is not allowed. There are certain exceptions to this rule, for example, names of parameters and fields of records can be the same as existing visible variables. It is also allowed for variables declared inside a subprogram (a procedure and function) to be the same as variables global to (outside of) the subprogram.

The effect of a declaration (its *scope)* lasts to the end of the construct in which the declaration occurs; this will be the end of the program, the end of the surrounding procedure, function or module, the end of a loop, for, case or begin statement, or the end of the then, elsif, or else clause of an if statement, or the end of the case statement alternative.

A name must be declared before it can be used; this is called the *DBU (Declaration Before Use)* rule. The exception to this rule is the form forward *id*, occurring in import lists and in collection declarations.

A *declaration* can appear any place a *statement* can appear; this is different from the Pascal language, in which declarations are allowed only at the beginning of the program or at the beginning of a procedure or function. Each declaration can optionally be followed by a semicolon (;).

There are certain restrictions on the placement of declarations. Procedures and functions cannot be declared inside other procedures and functions nor inside statements (for example, not inside an if statement). A bind declaration cannot appear inside a procedure or functions nor at the outer level of either the main program or a module.

procedure

[PC, Mac and Unix only]

SYNTAX:

delay (*duration* : int)

DESCRIPTION: The delay statement is used to cause the program to pause for a given time. The time duration is in milliseconds.

EXAMPLE: This program prints the integers 1 to 10 with a second delay between each.

```
for/: 1 .. 10
    put /
    delay (1000)        % Pause for 1 second
end for
```

DETAILS: See also the sound, clock, sysclock, wallclock, time and date statements.

On IBM PC compatibles, the hardware resolution of duration is in units of 55 milliseconds. For example, delay(SOO) will delay the program by about half a second, but may be off by as much as 55 milliseconds.

On Apple Macintoshes, the hardware resolution of duration is in units of 17 milliseconds (l/60th of a second). For example, delay(500) will delay the program by about half a second, but may be off by as much as 17 milliseconds.

SYNTAX:

div

DESCRIPTION: Th$_{ed}$

DETAILS: ₒV6r

operators and the mod

SYNTAX:

drawarc (*x, y, xRadius, yRadius :* int,
initialAngle, finalAngle, Color : int)

pESCRIPTION: The drawarc procedure is used to draw an arc whose center is at *(x y).* This is just like drawoval, except that you must also give two angles, *initial Angle and final Angle,* which determine where Sylovl^ draWing> Zerode8reesis ''three o'dock'" 9° d=ff«s is twelve o clock ', etc. The horizontal and vertical distances from the center to the arc are given by *xRadius* and *yRadius.*

yRadius
AnnalAngleX
.4—\ tlnltialAngle
' y) xRadius 7

idrde(actually,an roxm
ine ml ₒenter 'S *(midx*0),* the bottom center of the
determine th^l°rnumber L ^e maxx and maxy functions are used to
^ me maximum x and y values on the screen.

11 ("graphics")
midx := **maxx div 2**
Qrawarc (m'^– **0, maxy, maxy,** 0, 180,1)

palette- Z^^3TM''8 ₒf the Co/or number depends on the current
F tue, see the palette statement.

automatically bo ybesett»' - thc screen is not in a * ''mode; ^ thesetscreen graphics" mode. V«P^s'' mode, it will

and drawer SetSCreen, maxx, max y- drawdot, drawline, drawbox,

drawbox graphics procedure [pcan$_{dM}$

SYNTAX:

drawbox (*x1, y1, x2, y2, Color* : int) -

DESCRIPTION: The drawbox procedure is used to draw a box on the screen with bottom left and top right comers of *(xl, yl)* to *(x2,* y2) using the specified *Color.*

(x2,y2)

EXAMPLE: This program draws a large box, reaching to each corner of the screen using color number 1. The maxx and maxy functions are used to determine the maximum x and y values on the screen. The point (0,0) is the left bottom of the screen and (maxx, maxy) is the right top.

```
setscreen ("graphics")
drawbox (0, 0, maxx, maxy, 1 )
```

DETAILS: The meaning of the *Color* number depends on the current palette; see the palette statement.

The screen should be in a *"graphics"* mode; see the setscreen procedure for details. If the screen is not in a *"graphics"* mode, it will automatically be set to *"graphics"* mode.

See also setscreen, maxx, maxy, drawdot and drawline.

graphics procedure [PC and Mac only]

drawdot (x, y, *Color:* int)

egCRlPTlON: The drawdot procedure is used to color the dot (pixel) at location (x, y) using the specified *Color.*

maxy

0 1 2 3 maxx

Origin

EXAMPLE: This program randomly draws dots with random colors. The maxx, maxy and maxcolor functions give the maximum x, y and color values.

```
setscreen ("graphics")
var x, y, c: int
loop
    randint (x, 0, maxx)          % Random x
    randint (y, 0, maxy)          % Random y
    randint (c, 0, maxcolor)      % Random color
    drawdot (x, y, c)
end loop
```

DETAILS: The meaning of the Color number depends on the current palette; see the palette statement.

The screen should be in a *"graphics"* mode; if not, it will automatically be set to *"graphics"* mode. See setscreen for details

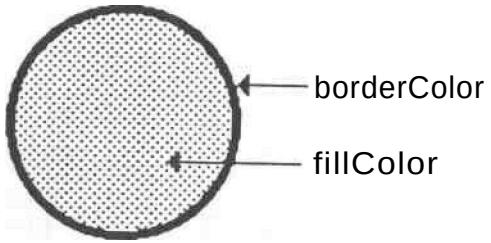
See also maxx, maxy, maxcolor and drawline.

drawfill graphics procedure

SYNTAX:

drawfill (*x, y :* **int,** *fillColor, borderColor:* int)

DESCRIPTION: The drawfill procedure is used to color in a figure that is on the screen. Starting at *(x, y)*, the figure is filled with/i7/Co7or to a surrounding border whose color is *borderColor*.



EXAMPLE: This program draws on oval with x and y radius of 10 in the center of the screen using color 1. Then the oval is filled with color 2. The maxx and maxy functions are used to determine the maximum x and y values on the screen.

```
setscreen ("graphics")
const midx := maxx div 2
const midy := maxy div 2
drawoval (midx, midy, 10,10,1)
drawfill ( midx, midy, 2,1)
```

DETAILS: The meaning of the *Color* number depends on the current palette; see the palette statement.

The screen should be in a *"graphics "* mode; see the setscreen procedure for details. If the screen is not in a *"graphics"* mode, it will automatically be set to *"graphics"* mode.

See also setscreen, maxx, maxy, drawdot, drawline, drawbox, and drawoval.

Warning: In Version 4.2 of Turing for IBM PC compatibles, drawfill fails to completely fill in some complicated figures that contain "islands" within them surrounded by the *borderColor*.

graphics procedure [PC and Mac only]

SYNTAX:

drawline (*x 1 , y1, x2, y2, Color* : int)

DESCRIPTION: The drawline procedure is used to draw a line on the screen from *(xl, yl)* to *(x2, y2)* using the specified *Color*.

EXAMPLE: This program draws a large X, reaching to each corner of the screen using color number 1. The maxx and maxy functions are used to determine the maximum x and y values on the screen. The point (0,0) is the left bottom of the screen, (maxx, maxy) is the right top, etc.

```
setscreen ("graphics")
% First draw a line from the left bottom to right top
drawline (0,0, maxx, maxy, 1)
% Now draw a line from the left top to right bottom
drawline (0, maxy, maxx, 0,1)
```

DETAILS: The meaning of the *Color* number depends on the current palette; see the palette statement.

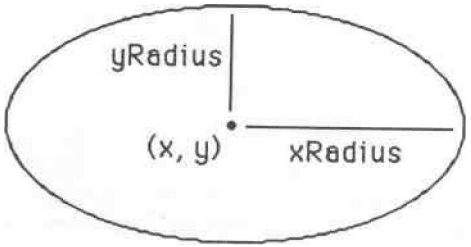
The screen should be in a *"graphics"* mode; see the setscreen procedure for details. If the screen is not in a *"graphics"* mode, it will automatically be set to *"graphics"* mode.

See also setscreen, maxx, maxy, drawdot and drawbox.

SYNTAX:

drawoval (*x, y, xRadius, yRadius, Color :* int)

DESCRIPTION: The drawoval procedure is used to draw an oval whose center is at *(x,* y). The horizontal and vertical distances from the center to the oval are given by *xRadius* and *yRadius.*



EXAMPLE: This program draws a large oval that just touches each edge of the screen using color number 1. The maxx and maxy functions are used to determine the maximum x and y values on the screen. The center of the oval is at (midx, midy), which is the middle of the screen.

```
setscreen ("graphics")
const midx := maxx div 2
const midy := maxy div 2
drawoval (midx, midy, midx, midy, 1)
```

DETAILS: Ideally, a circle is drawn when *xRadius = yRadius.* In fact, the aspect ratio (the ratio of height to width of pixels displayed on the screen) of the IBM PC compatibles is not 1.0, so this does not draw a true circle. In CGA graphics mode this ratio is 5 to 4.

The meaning of the *Color* number depends on the current palette; see the palette statement.

The screen should be in a *"graphks"* mode; see the setscreen procedure for details. If the screen is not in a *"graphks"* mode, it will automatically be set to *"graphics"* mode.

See also setscreen, maxx, maxy, drawdot, drawline, and drawbox.

SYNTAX:

```
drawpic ( x, y : int,
    buffer : array 1 .. * of int,
    picmode : int )                    \
```

DESCRIPTION: The drawpic procedure is used to copy of a rectangular picture onto the screen. The left bottom of the picture is placed at *(x, y).* In the common case, the buffer was initialized by calling takepic. The values of *picmode* are:

0: Copy actual picture on screen.
1: Copy picture by XORing it onto the screen.

XORing a picture onto the screen twice leaves the screen as it was; this is a convenient way to move images for animation. XORing a picture onto a background effectively superimposes the picture onto the background.

DETAILS: See takepic for an example of the use of drawpic and for further information about buffers for drawing pictures.
The screen should be in a *"graphics"* mode; see the setscreen procedure for details. If the screen is not in a *"graphics"* mode, it will automatically be set to *"graphics"* mode.

See also takepic and sizepic.

See also setscreen, maxx, maxy, drawdot, drawline, drawbox, and drawoval.

enum type
enumerated type

SYNTAX: An *enumemtedType* is:

enum (*id { , id })*

DESCRIPTION: The values of an enumerated type are distinct and increasing. They can be thought of as the values 0,1,2 and so on, but arithmetic is not allowed with these values.

EXAMPLE:

```
type color : enum ( red, green, blue)
var c : color := color.red
var d : COlor := SUCC( C)      %d becomes green
```

DETAILS: Each value of an enumerated type is the name of the type followed by a dot followed the the element's name, for example, *color.red.* Enumerated values can be compared for equality and for ordering. The *succ* and *pred* functions can be used to find the value following or preceding a given enumerated value. The *ord* function can be used to find the enumeration position of a value, for example, *ord(color.red)* is 2.

Enumerated types cannot be combined with integers or with other enumerated types.

eflumeratedValue

SYNTAX: An *enumeratedValue* is:

enumeratedTypeId . enumeratedld

DESCRIPTION: The values of an enumerated type are written as the type name *(enumeratedTypeId)* followed by a dot followed by one of the enumerated values of the type *(enumeratedld).*

EXAMPLE: In this example, *color.red* is an *enumeratedValue.*

```
type color : enum ( red, green, blue)
var c : color := color.red
    d : COlor \=SUCC(C)      % d becomes green
```

DETAILS: The above description has been simplified by ignoring the possibility that the enum type can be exported from a module. If this is the case, each use of one of the enumerated values outside of module M must be preceded by the module name and a dot, as in *Mxolor.red.*

See also the enum type and *explicit constants.*

end-of-file function

SYNTAX:

eof (*streamNumber:* int)

DESCRIPTION: The eof (end of file) function is used to see if there is any more input. It returns true when there are no more characters to be read. The parameter and its parentheses are omitted when referring to the standard input (usually this is the keyboard); otherwise the parameter specifies the number of a stream; this number has been determined (in most cases) by an open statement.

EXAMPLE: This program reads and outputs all the lines in the file called "info".

```
var line : string
var fileNumber : int
open : fileNumber, "info", get
loop
    exit when eof (fileNumber)
    get: fileNumber, line :*
    put line
end loop
```

DETAILS: See also the description of the get statement, which gives more examples of the use of eof. See also the open and read statements.

When the input is from the keyboard, the user can signal end-of-file by typing controI-Z on a PC (or control-D on Unix). If a program tests for eof on the keyboard and the user has not typed control-Z (or control-D) and the user has typed no characters beyond those that have been read, the program must wait until the next character is typed. Once this character is typed, the program knows whether it is at the end of the input, and returns the corresponding true or false value for eof.

equivalence of types

C R l P T I : Two types are *equivalent* to each other if they are essentially the same types (the exact rules are given below). When a variable is passed to a var formal parameter, the types of the variable and the formal parameter must be equivalent because they are effectively the same variable. When an expression is assigned to a variable, their root types must be equivalent, except for the special case that it is allowed to assign an integer expression to a real variable (see *assignability* for details).

EXAMPLES:

```
vary : int

var b : array 1 .. 25 of string

type personType :
    record
        age : int
        name : string ( 20 )
    end record

procedure p ( var i : int,
              var a : array 1 .. 25 of string,
              var r : personType )
    ... body of procedure p, which modifies
    each of i, aandr...
end p

var s : personType
P ( i, b, S )    % Procedure call to p
                 % i and j have the equivalent type int
                 % Arrays a and b have equivalent types
                 % Records r and s have equivalent types
```

Two types are defined to be *equivalent* if they are:

(a) the same standard type (int, real, boolean or string [(...)],
(b) subranges with equal first and last values,
(c) arrays with equivalent index types and equivalent component types,
(d) strings with equal maximum lengths,

(e) sets with equivalent base types, or
(0 pointers to the same collection; in addition,
(g) a declared type identifier is equivalent to the type it names (and to the type named by that type, if that type is a named type, etc.)

Each separate instance of a record, union or enumerated type (written out using one of the keywords record, union or enum) creates a distinct type, equivalent to no other type. By contrast, separate instances of arrays, strings, subranges and sets are considered equivalent if their parts are equal and equivalent.

Opaque type T, exported from a module M as opaque, is a special case of equivalence. Outside of M this type is written M.T, and is considered to be distinct from all other types. By contrast, if type LI is exported non-opaque, the usual rules of equivalence apply. The parameter or result type of an exported procedure or function or an exported constant is considered to have type M.T outside of M if the item is declared using the type identifier T. Outside of M, the opaque type can be assigned, but not compared.

£ealstr real-to-string function

SYNTAX:

erealstr (*r* : real,
width, fractionWidth, exponentWidth : int) : **string**

DESCRIPTION: The erealstr function is used to convert a real number to a string; for example, erealstr (2.5el, 10, 3, 2)="b2.500e+01 " where *b* represents a blank. The string (including exponent) is an approximation to *r*, padded on the left with blanks as necessary to a length of *width*.

The *vridth* must be non-negative int value. If the *width* parameter is not large enough to represent the value of *r*, it is implicitly increased as needed.

The fractionWidth parameter is the non-negative number of fractional digits to be displayed. The displayed value is rounded to the nearest decimal equivalent with this accuracy, with ties rounded to the next larger value.

The *exponentWidth* parameter must be non-negative and gives the number of exponent digits to be displayed. *If exponentWidth* is not large enough to represent the exponent, more space is used as needed. The string returned by *erealstr* is of the form:

{blank)[-]digit.{digit)e sign digit (digit)

where *sign* is a plus or minus sign. The leftmost digit is non-zero, unless all the digits are zeros.

The erealstr function approximates the inverse of strreal, although round-off errors keep these from being exact inverses.

See also the frealstr, realstr, strreal, intstr and strint functions.

exit statement

SYNTAX: An *exitStatement* is one of:

(a) **exit when** *trueFalseExpn*
(b> **exit**

DESCRIPTION: An exit statement is used to stop the execution of a loop or for statement. Form (a) is the most common. In it the true/false expression is evaluated. If it is true, the loop is terminated and execution jumps down and continues just beyond end loop or end for. If it is false, the loop keeps on repeating. Form (b) always causes the loop to terminate; this form is almost always used inside another conditional statement such as if.

EXAMPLE: Input names until finding Jones.

```
var name : string
loop
    get name
    exit when name = "Jones"
end loop
```

DETAILS: Exit statements must occur only inside loop or for statements. An exit takes you out of the mostly closely surrounding loop or for. The only other ways to terminate a loop or for is by return (in a procedure or in the main program, in which case the entire procedure or main program is terminated) or by result (in a function, in which case the entire function is terminated and a result value must be supplied).

The form "exit when *trueFalseExpn"* is equivalent to "if *trueFalseExpn* then exit end if".

exponentiation function

SYNTAX:

exp (*r* : real): real

DESCRIPTION: The exp function is used to find e to the power r, where e is the natural base and r is the parameter to exp. For example, exp(0) returns 1 and exp(1) returns the value of e.

EXAMPLE: This program prints out the exponential values of 1,2,3,... up to 100.

```
for /: 1 .. 100
    put "Exponential of", /," is", exp (/)
end for
```

DETAILS: See also the In (natural logarithm) function.

explicitConstant

SYNTAX: An *explicitConstant* is one of:

(a) *explicitStringConstant* *%e.g.:* "Hello world"
(b) *explicitIntegerConstant* *%e.g.:* 25
(c) *explicitRealConstant* *%e.g.:* 51.8
(d) *explicitTrueFalseConstant* %e.g.: **true**

DESCRIPTION: An *explicitConstant* gives its value directly, for example, the value of the explicit constant 25 is twenty-five.

EXAMPLES: In the following, the explicit constants are "Hello world", 3.14159 and 2. Note that *pi* is a *named* constant rather than an explicit constant.

```
put "Hello world"
var diameter : real
const pi := 3.14159
diameter := pi* r ** 2
var x := diameter
```

DETAILS: In some programming languages, *explicit constants* are called *literals* or *literal values*, because they literally (explicitly) give their values.

For further details about explicit constants, see explicitStringConstant, explicitIntegerConstant, explicitRealConstant and explicitBooleanConstant. See also enumeratedValue.

explicitIntegerConstant

SYNTAX: An *explicitIntegerConstant* is a sequence of one or more decimal digits (0 to 9) optionally preceded by a plus or minus sign.

EXAMPLES: In the following, the explicit integer constants are 0,115 and 5.

```
var count : int := 0
const height := 115

count := height - 5
```

DETAILS: In Turing, the range of integers is from -2147483647 to 2147483647. In other words, the maximum size of integer is 2**31 - 1.

explicitRealConstant

SYNTAX: An *explicitRealConstant* consists of an optional plus or mfnus sign, a *significant digits part,* and an *exponent part.*

EXAMPLES: In the following, the explicit real constants are 0.0 and 2.93e3.

```
var temperature : real := 0.0
const speed := 2.93e3          % vaiueis 2,930.0
```

DETAILS: The significant digits part (or *fractional part)* of an explicit real constant consists of a sequence of one or more digits (0 to 9) optionally containing a decimal point (a period). The decimal point is allowed to follow the last digit as in 16. or to precede the first digit, as in .25

The exponent part consists of the letter *e* or E followed optionally by a plus or minus sign followed by one or more digits. For example in -9.837e-3 the exponent part is e-3. The value of -9.837e-3 is -9.837 times 0.001.

If the significant figures part contains a decimal point then the exponent part is not required.

eXplicitStringConstant

SYNTAX: An *explicitStringConstant* is a sequence of characters surrounded by quotation marks.

EXAMPLES: In the following, the explicit string constants are "Hello world","" and "273 O'Reilly Ave.".

```
var name : string := "Hello world"
name := ""        %Null string, containing zero characters
var address : string := "273 O'Reilly Ave."
```

DETAILS: Within an explicit string constant, the back slash character (\) is used to represent certain other characters as follows:

\"	quotation mark character
\n or \N	end of line character
\t or \T	tab character
\f or \F	form feed character
\r or \R	return character
\b or \B	backspace character
\e or \E	escape character
\d or \D	delete character
\\	backslash character

For example, put "OneXnTwo" will output One on one line and Two on the next.

Explicit string constants cannot cross line boundaries. To represent a string that is longer than a line, break it into two or more strings on separate lines and use + (catenation) to join the individual strings.

An explicit string constant can contain at most 255 characters (this is in implementation constraint).

An explicit string is not allowed to contain characters with the code values of 0 or 128; these character values are called *eos* (end of string) and *uninitchar* (uninitialised character). These are reserved by the implementation to mark the end of a string value and to see if a string variable has a value.

explicitTrueFalseConstant

SYNTAX: An *explicitTrueFalseConstant is* one of:

(a) **true**
(b) **false**

EXAMPLE: The following determines if string *s* contains a period. After the for statement, *found* will be true if there is a period in *s*.

```
var found : boolean := false
for i: 1 .. length (s)
    If s = "." then
        found := true
    end if
end for
```

DETAILS: True/false values are called *boolean* values. A boolean variable, such *as found* in the above example, can have a value of either true or false. See also boolean type.

expression

: An *expn* is one of:

	explicitConstant	*%e.g.:* 25
(a)	*variableReference*	*% e.g.: width*
(b)	*constantReference*	*%e.g.: pi*
(0	*expn infixOperator expn*	*% e.g.: 3 + width*
(d)	*prefixOperator expn*	*% e.g.: • width*
(e)	*(expn)*	*% e.g.: (width - 7)*
(*)	*substring*	*%e.g.: s* (3 ..5)
(9)	*functionCall*	*%e.g.: sqrt* (25)
(h)	*setConstructor*	*% e.g.: modes* (4, 3)
(0	*enumeratedValue*	*%e.g.: color, red*

DESCRIPTION: An expression *(expn)* returns a value; in the general case, this may involve a calculation, such as addition, as in the expression 3 + *width.*

EXAMPLES:

```
put "Hello world"                  % "Hello world" is an expn
var diameter : real
const pi := 3.14159                % 3.14159 is an expn
diameter := pi * r ** 2            % pi * r ** 2 is an expn
var x := diameter                  % diameter is an expn
```

DETAILS: In the simplest case, an expression *(expn)* is simply an explicit constant such as 25 or *"Hello world".* A variable by itself is considered to be an expression when its value is used, as is the case above when the value of *diameter* is used to initialise *x.* More generally, an expression contains an operator such as + and carries out an actual calculation. An expression may also be a substring, function call, set constructor or enumerated value; for details, see the descriptions of these items.

The infix operators are: +, -, *, /, div, mod, **, <, >, =, <=, >=, not=, not, and, or, =>, in, and not in. For details, see *infixOperator.* The prefix operators are +, - and not. For details see *prefix* operator.

See also *precedence* of operators, as well as the int, real, string and boolean types.

export list

SYNTAX: *AnexportList* is:

export([opaque]/c/ {,[opaque] *id})*

DESCRIPTION: An export list is used to specify those items declared in a module that can be used outside of it. Items that are declared inside a module but not exported cannot be accessed outside of the module.

EXAMPLE: In this example, the procedures names *pop* and *push* are exported from the *stack* module. These two procedures are called from outside the module on the last and third from last lines of the example; notice that the word *stack* and a dot must precede the use of these names. Since *top* and *contents* were not exported, they can be accessed only from inside the module.

```
module stack
    export (push, pop)
    var top: int := 0
    var contents: array 1..100 of string
    procedure push... end push
    procedure pop... end pop
end stack

stack.push ("Harvey" )
var name : string
stack.pop (name)          % This sets name to Harvey
```

DETAILS: Only procedures, functions, constants and types can be exported. It is not allowed to export variables or modules.

In the most common case, the optional keyword opaque is omitted. The keyword is allowed only in front of exported types names. When it is used, it specifies that outside the module, the type is considered to be distinct from all other types; this means, for example, that if the type is an array, it cannot be subscripted outside of the module. See module declaration for details about opaque types.

Note: The parentheses in the export statement are optional. The export statement in the example could have been:

export pop, push

[Compiler only]

SYNTAX: An *externalSubprogram* is: • ^

external [*overrideName] subprogramHeader*

DESCRIPTION: This syntax provides an extension to the Turing language to allow the Turing program to call programs written in other languages such as the C language. This extension is *not* supported in the current Turing interpeter.

The optional *overrideName* must be an explicit string constant, such as "print/". When it is omitted, the name used for external linking is the name of the procedure or function, as given in the *subprogramHeader.*

The subprogramHeader is one of:

(a) procedure *id [(paramDeclaration IparamDecIaration]*)]
(b) function *id [(paramDeclaration \jparamDeclaration]*)]
[id]: *typeSpec*

false

boolean value (as opposed to true) -

SYNTAX:

```
false
```

DESCRIPTION: A boolean (true/false) variable can be either **true** or false (see boolean type).

EXAMPLE:

```
var found : boolean := false
var word: int
for/ : 1 .. 10
    get word
    found := found or word- "gold"
end for
if found = true then
    put "Found 'gold' in the ten words"
end if
```

DETAILS: The line if/owmf=true then can be simplified to *if found* then with no change to the meaning of the program.

(fetch argument) function [pc,MacandUnixoni$_y$i

SYNTAX:

```
fetcharg (/: int): string
```

DESCRIPTION: The fetcharg function is used to access the i-th argument that has been passed to a program from the command line. For example, if the program is run from the Turing environment using

```
:rfilelfile2
```

then fetcharg(2) will return "file2". If a program called *prog.x is* run under Unix using this command

```
prog.x filel file2
```

the value of fetcharg(2) will similarly be "filc2".

The nargs function, which gives the number of arguments passed to the program, is usually used together with the fetcharg function. Parameter j passed to fetcharg must be in the range 0.. nargs . See also nargs.

The 0-th argument is the name of the running program.

EXAMPLE: This program lists its own name and its arguments.

```
put "The name of this program is :", fetcharg(O)
for i: 1 .. nargs
    put "Argument", /," is ", fetcharg(/)
end for
```

floor real-to-integer function

SYNTAX:

```
floor( r : real): int
```

DESCRIPTION: Returns the largest integer that is less than or equal to *r*.

DETAILS: The floor function is used to convert a real number to an integer. The result is the largest integer that is less than or equal to *r*. In other words, the floor function rounds down to the nearest integer. For example, floor (3) is 3, floor (2.75) is 2 and floor<-8.43) is -9.

See also the ceil and round functions.

statement

SYNTAX: A *forStatement* is:

```
for [decreasing] [id ] : first... last
    statementsAndDeclarations
end for
```

DESCRIPTION: The statements and declarations in a for statement are repeatedly executed with the identifier increasing by 1 (it decreases by 1 if you specify decreasing) *(mm first* to *lost*, which are integer values (or else enumerated values).

EXAMPLE: Output 1, 2, 3, up to 10.

```
for /:1 ..10
    put /
end for
```

EXAMPLE: Output 10,9,8, down to 1.

```
for decreasing j: 10 .. 1
    put/
end for
```

DETAILS: The for statement declares the counting identifier (a separate declaration should not be be given for i or /). The scope of this identifier is restricted to the for statement.

If *first* is a value beyond *last*, there will be no repetitions (and no error message). The counting identifier is always increased (or decreased) by 1; in some languages such as Basic, you can specify a step size other than 1, but this is not possible in Turing. Executing an exit statement inside a for statement causes a jump to just beyond end for. You are not allowed to change the counting variable (for example, you are not allowed to write *i :=* 10).

The counting identifier can be omitted; the statement is just as before, except the value of the identifier cannot be used by the program.

If decreasing is not present, *first.. last* can be replaced by the name of a subrange type, for example by *dozen*, declared by:
*type dozen : I..*12

Procedures, functions and modules cannot be declared inside a for statement. Just preceding the statements and declarations, you are allowed to write an "invariant clause" of the form:

invariant *trueFalseExpn*

This clause is equivalent to: assert *trueFalseExpn.*

forward declaration

SYNTAX: A *forwardDedamtion* is one of:

(a) **forward procedure** *procedureId*
*[(paramDeclaration {, paramDeclaration }***)**]
import (*importItem {, importItem}*)

(a) **forward function** *functionId*
[(paramDeclaration {, paramDeclaration })]
[resultId]: *resultType*
import (*importItem {, importItem}*)

DESCRIPTION: A procedure or funcdon is declared to be forward when you want to define its header but not its body. This is the case when one procedure or function calls another which in turn calls the first; this situation is called *mutual recursion.* The use of forward is necessary in this case because every item must be declared before it can be used.

EXAMPLES: This example program evaluates an input expression *e* of the form / { + *t)* where t is of the form *p {* p }* and *p* is of the form *(e*) or an explicit real expression. For example, the value of 1.5 + 3.0 * (0.5 + 1.5) halt is 7.5.

```
var token: string

forward procedure expn (var eValue: real)
    import (forward term, var token)

forward procedure term (var tValue : real)
    import (forward primary, var token)

forward procedure primary (var pValue: real)
    import (expn, var token)

body procedure expn
    var nextValue: real
    term (eValue)                % Evaluate t
    loop                         % Evaluate { +1}
```

ft.

```
            exit when token not= "+"
            get token
            term (nextValue)
            eValue := eValue + nextValue
        end loop
    end expn

    body procedure term
        var nextValue: real
        primary (tValue)          % Evaluate p
        loop                      % Evaluate { * p}
            exit when token not= "*"
            get token
            primary (nextToken)
            tValue := tValue + nextValue
        end loop
    end term

    body procedure primary
        if token = "(" then
            get token
            expn(pValue)          % Evaluate (e)
            assert token =")"
        else                      % Evaluate "explicit real"
            pValue := strreal (token)
        end if
        get token
    end primary

    get token                     % Start by reading first token
    var answer: real
    expn (answer)                 % Scan and evaluate input expression
    put "Answer is ", answer
```

DETAILS: Following a forward procedure or function declaration, the body of the procedure must be given at the same level (in the same sequence of statements and declarations as the forward declaration). This is the only use of the keyword body; see also body.

Any procedure or function that is declared using forward is required to have an import list. In this list, imported procedures or functions that have not yet appeared must be listed as forward; for example, the import list for *expn* is import (forward *term* ...). Before a procedure or function can be called and before its body appears and before it can be passed as a parameter, its header as well as headers of procedures or functions imported directly or indirectly by it must have appeared.

The keyword forward is also used in collection declarations; see also collections.

frealstr real-to-string function

SYNTAX:

frealstr (*r* : real, *width, fractionWidth,:* int) : **string**

DESCRIPTION: The frealstr function is used to convert a real number to a string; for example, frealstr (2.5el, 5,1)="&25.0" where *b* represents a blank. The string is an approximation to *r,* padded on the left with blanks as necessary to a length of *width.*

The number of digits of the fraction to be displayed is given by *fractionWidth.*

The *width* must be non-negative. If the *width* parameter is not large enough to represent the value of *r,* it is implicitly increased as needed.

The fractionWidth must be non-negative. The displayed value is rounded to the nearest decimal equivalent with this accuracy, with ties rounded to the next larger value. The result string is of the form:

(blank) [-HdigitJ. {digit}

If the leftmost digit is zero, then it is the only digit to the left of the decimal point.

The frealstr function approximates the inverse of strreal, although round-off errors keep these from being exact inverses.

See also the erealstr, realstr, strreal, intstr and strint functions.

free statement

SYNTAX: A *freeStatement* is:

free *collectionid, pointerVariableReference*

DESCRIPTION: A free statement destroys (deallocates) an element of a collection.

EXAMPLE: Declare a list. Allocate and then later deallocate a node.

```
var list : collection of
    record
        contents : string (10)
        next : pointer to list
    end record
var first: pointer to list
new list, first         % Allocate an element of list;
                        % place its location in first

free list, first        % Deallocate the element of list
                        % located by first.
```

DETAILS: The free statement sets the pointer variable to the nil value, in this example, to *nil (list).*

See also the collection declaration, the pointer type, and the new statement.

function declaration

SYNTAX: A *functionDeclaration* is:

```
function id
    [ (paramDeclaration {, paramDeclaration }) ]
    : typeSpec
    statementsAndDeclarations
end id
```

DESCRIPTION: A function declaration creates (but does not run) a new function. The name of the function *(id)* is given in two places, just after function and just after end.

EXAMPLES:

```
function doublelt (var x: real): real
    result 2.0 * x
end doublelt

put doublelt (5.3 )          % This outputs 10.6
```

DETAILS: The set of parameters declared with the function are called *formal* parameters; for example, in the *doublelt* function, x is a formal parameter. A function is called (invoked) *by a function call* which consists of the function's name followed by the parenthesised list of *actual* parameters (if any); for example, *doublelt* (5.3) is a call having 5.3 as an actual parameter. If there are no parameters, the call does not have parentheses. The keyword function can be abbreviated to fen. See also *functionCall* and *procedureDeclaration.*

Each actual parameter must be assignable to the type of its corresponding formal parameter; see also *assignability.*

A function must finish by executing a result statement, which produces the function's value. In the above example, the result statement computes and returns the value 2.0 * *x.*

In principle, a function should not change any variables outside of itself (global variables); in other words, it should have no *side effects.* However, this restriction is not necessarily enforced by the implementation. A function should not have var parameters, as these would allow the function to change values outside of itself.

The upper bounds of arrays and strings that are parameters may declared to be star (*), meaning the bound is that of the actual oararneter. See *paramDeclaration* for details about parameters.

Procedures and functions cannot be declared inside other procedure and functions.

The syntax of *a functionDeclaration* presented above has been SJmplified by leaving out the result identifier, import list, pre and post condition and init clause; the full syntax is

```
function id
    [ (paramDeclaration {, paramDeclaration }) ]
    [ resultld ] : typeSpec
    ( import ([ [var] id {, [var] i d } ] ) ]
    [ pre trueFalseExpn ]
    [ init/'d := expn {, id := expn } ]
    [ post trueFalseExpn ]
    statementsAndDeclarations
end id
```

See import list, pre condition, init clause and post condition for explanations of these additional features. The *resultld* is the name of the result of the function and can be used only in the post condition.

A function must be declared before being called; to allow for mutually recursive procedures and functions, there are forward declarations with later declaration of the procedure or function body. See forward and body declarations for explanations.

functionCall

SYNTAX: *A functionCall* is;

functionId [(expn { , expn})]

DESCRIPTION: A function call is an expression that calls (invokes or activates) a function. If the function has parameters, a parenthesised list of expressions *(expns)* must follow the function's naming *(functionId)*

EXAMPLES: This function takes a string containing a blank and returns the first word in the string (all the characters up to the first blank)

```
function firstWord ( str: string ): string
    for i : 1 .. length (str)
        if str (i) = "" then
            result str (1 .. i - 1)
        end if
    end for
end firstWord

put "The first word is:", firstWord ("Henry Hudson")
        % The function call is firstWord (sample)
        % The output is Henry.
```

DETAILS: The parameter declared in the header of a function is a (formal) parameter, for example, *str* above is a formal parameter. Any expression in the call is an *actual* parameter, for example, *sample* above is an actual parameter. In a function, a formal parameter should not be declared using var.

Each actual parameter passed to its non-var formal parameter must be assignable to that parameter (see *assignability* for See also *functionDeclaration* and *procedureDeclaration.*

In this explanation *of functionCall,* we have up to this point ignored the possibility of procedures exported from modules. A function is being called from outside of a module from which it has been exported, the syntax of *the functionCall* is:

moduleId. functionId [(expn {, expn})]

get statement

cYNTAX: A *getStatement* is:

get [: *streamNumber*,] *getItem { , getItem }*

DESCRIPTION: The get statement inputs each of the *getItems.* Ordinarily, the output comes from the keyboard. However, if the *streamNumber* is present, the input comes from the file specified by the stream number (see the open statement for details). Also, input can be redirected so it is taken from a file rather than the keyboard by a command such as a: < fileName done in the Turing environment.

The syntax of a *getItem* is one of:

(a) *variableReference*
(b) **skip**
(c) *variableReference: **
(d) *variableReference: widthExpn*

These items are used to support three kinds of input:
(1) token-oriented input: supported by forms (a) and (b),
(2) line-oriented input: supported by form (c), and
(3) character-oriented input: supported by form (d).
Examples of these will be given, followed by a detailed explanation of the kinds on input.

EXAMPLES: Token-oriented input.

```
var name, title: string
var weight : real
get name        % If input is Alice, it is input into name
get title       % If input is "A lady", A lady is input
Var Weight      % If input is 9.62, it is input into weight
```

EXAMPLE: Line-oriented input.

```
var query: string
get query :*    % Entire line is input into query
```

EXAMPLE: Character-oriented input.

```
var code: string
get code : 2          % Next 2 characters are input into code.
```

DETAILS: A *token is* defined as a sequence of characters surrounded by *white space,* where *white space* is defined as the characters blank, tab, form feed, new line, and carriage return as well as end-of-file. The sequence of tokens making up the token are either all non white space or else the token must be a quoted string (an explicit string constant). Form (a) of *getItem* skips white space and then reads a token into the *variableReference,* which must be a string, integer or real. If the *variableReference* is a string, the token is assigned to the variable (if the token is quoted, the quotation marks are first removed); see the examples involving *name* and *title* above. If the *variableReference* is an integer or a real, the token is converted to be numeric before being assigned to the variable; see the example involving *weight* above. When the input is coming from the keyboard, no input is done until Return is typed. The line that is input may contain more than one token; any tokens that are not input by one get statement will remain to be input by the next get statement.

In form (b) *of getItem,* skip causes white space in the input to be skipped until non white space (a token) or the end-of-file is reached. This is used when the program needs to determine if there are more tokens to be input. To determine if there are more tokens to be read, the program should first skip over any possible white space (such as a final new line character) and then test to see if eof (end-of-file) is true. This is illustrated in this example:

EXAMPLE: Using token-oriented input, input all tokens and list them.

```
var word: string
loop
    get skip          % Skip over any white space
    exit when eof     % Are there more characters?
    get word          % Input next token
    put word          % Output the token
end loop
```

In the above and the next example, if the input has been redirected so that it is from a file, eof becomes true exactly when there are no more characters to be read. If the input is coming from the keyboard, you can signal eof by typing control-Z (on a PC) or control-D (on Unix).

r-TAILS In form (c) of *getItem,* the *variableReference* is followed by :» which implies line-oriented input. This form causes the entire line (or !Ie remainder of the current line) to be read; in this case the variable Inust be a string (not an integer or real). The new line character at the ~nd of the line is discarded. It is an error to try to read another line then vou are already at the end of the file. The following example shows how to use line-oriented input to read all lines in the input.

Using line-oriented input, input all lines and list them.

```
var line: string
loop
    exit when eof     % Are there more characters?
    get line: *       % Read entire line
    put line
end loop
```

DETAILS: In form (d) of *getItem,* the *variableReference* is follows by : *widthExpn* which specifies character-oriented input. This form causes the specified number *(widthExpn)* of characters to be input (or all of the remaining characters if not enough are left); if no characters remain, the null string is read and no warning is given. In this form, the new line character is actually input into the *variableReference* (this differs from line-oriented input which discards new line characters). The following example shows how to use character-oriented input to read each character of the input.

EXAMPLE: Using character-oriented input, input all characters and list them.

```
var ch: string(1)
loop
    exit when eof     % Are there more characters?
    get ch: 1         % Read one character
    put ch ..         % Output the character, which
                      % may be a new line character
end loop
```

DETAILS: See also the read statement, which provides binary file input.

getCrl (get character) procedure

SYNTAX:

```
getch (var ch : string (1))
```

DESCRIPTION: The getch procedure is used to input a single character without waiting for the end of a line. The parameter *ch* is set to the most recently type character.

EXAMPLE: This program contains a procedure called *pause* which causes the program to wait until a key is pressed.

```
setscreen ("graphics")

procedure pause
    var ch : string (1)
    getch (ch)
end pause

for /: 1 .. 1000
    put /: 4," Pause till a key is pressed"
    pause
end for
```

DETAILS: The screen should be in a *"screen"* or *"graphics"* mode; see the setscreen procedure for details. If the screen is not in one of these modes, it will automatically be set to *"screen"* mode.

See also the hasch (has character) procedure which is used to see if a character has been typed but not yet read.

On the IBM PC keystrokes which do not provide an ASCII value (left arrow key, insert key, delete key, function keys and so on) return the scan code of the keystroke with 128 added to it, unless the scan code already has a value of 128 or greater. This provides a unique value for every key on the keyboard. Use a reference to the IBM PC to find out the scan codes produced by the keyboard.

(get environment) function [PC a

SYNTAX:

```
getenv ( symbol: string ): string
```

pESCRIPTION: The getenv function is used to access the environment string whose name is *symbol*. These strings are determined by the shell (command processor) or the program that caused your program to run. See also the nargs and fetcharg functions.

EXAMPLE: Retrieves the environment variable USERLEVEL and prints extra instructions if USERLEVEL had been set to NOVICE. On an IBM PC, this could be set with the command SET USERLEVEL = NOVICE in the autoexec.bat file or in any batch file.

```
const userLevel: string
userLevel := getenv ("USERLEVEL")
if userLevel = "NOVICE" then
                        % put a set of instructions
end if
```

getpid (get process id) function

[PCand%

SYNTAX:

```
getpid : int
```

DESCRIPTION: The getpid function is determine the process id of the current process. On a personal computer, this number is of little use. Under Unix, the number is used, for example, for creating a unique name of a file. See also nargs, fetcharg and getenv.

(has character) function

```
hasch : boolean
```

DESCRIPTION' The hasch procedure is used to determine if there is a character that has been typed but not yet been read.

EXAMPLE: The *flush* procedure gets rid of any characters that have been typed but not yet read.

```
procedure flush
    var ch : string(1)
    loop
        exit when not hasch
        getCh (ch)      % Discard this character
    end loop
end flush
```

DETAILS: The screen should be in a *"screen"* or *"graphics"* mode; see the setscreen procedure for details. If the screen is not in one of these modes, it will automatically be set to *"screen"* mode.

id (identifier)

name of an item in a Turing program

DESCRIPTION: Variables, constants, types, procedures, etc. in Turing programs are given names such as *incomeTax*, *x*, and *height.* These names are called identifiers *(ids).*

An identifier must start with a letter (large or small) and can contain up to 50 characters, each of which must be a letter, a digit (0 to 9) or an underscore (_)• Large and small letters are considered distinct, so that *A* and *a* are different names; this is different from Pascal in which large and small letters in names are equivalent.

Every character in a name is significant in distinguishing one name from another.

By convention, words that make up an identifier are capitalised (except the first one), as in *incomeTax* and *justlnTime.*

An item in a Turing program cannot be given the same name as a keyword such as get nor as a reserved word such as index. See appendix A for a list of keywords and reserved words.

statement

cYNTAX: An *ifStatement* is:

```
if trueFalseExpn then
    statementsAndDeclarations
{ elsif trueFalseExpn then
    statementsAndDeclarations }
[ else
    statementsAndDeclarations ]
end if
```

DESCRIPTION: An if statement is used to choose among a set of statements (and declarations). One set (at most) is chosen and executed and then execution continues just beyond end if.

The expressions (the *trueFdseExpressions)* following the keyword if and each elsif are checked one after the other until one of them is found to be true, in which case the statements (and declarations) following the corresponding then are executed. If none of these expressions evaluates to true, the statements following else are executed. If no else is present and none of the expressions are true, no statements are executed and execution continues following the end if.

EXAMPLE: Output a message based on value of mark.

```
if mark >= 50 then
    put "You pass"
else
    put "You fail"
end if
```

EXAMPLE: Output A, B, C, D or F depending on mark.

```
if mark >= 80 then
    put "A"
elsif mark >= 70 then
    put "B"
elsif mark >= 60 then
    put "C"
```

```
elsif mark >= 50 then
    put "D"
else
    put "F"
end if
```

EXAMPLE: If *x* is negative, change its sign.

```
if x < 0 then
    x := -x
end if
```

DETAILS: Several statements and declarations can appear after a particular then.

See also case statements for another way to select among statements.

import list

SYNTAX: *AnimportList* is:

import(*[howlmport] id {, [howlmport] id]*)

DESCRIPTION: An import list is used to specify those items that a procedure, function or module uses from outside of itself. Commonly, procedures and functions are written without import lists, which means that the list is determined automatically by the compiler by looking to see what items are actually used.

EXAMPLE: In this example, the type *T* is imported into the *stock* module and used as the type that can be pushed onto or popped off the stack. Since no other items are imported, the only identifiers from outside of *stock* that can be used in it must be predefined, such as sqrt, or declared to be pervasive.

```
type T : string

module stock
    import ( T )
    export (push, pop)      % alternate: export push, pop
    var top : int := 0
    var contents: array 1..100 of 7"
    procedure push... end push
    procedure pop...end pop
end stock
```

DETAILS: There are various ways to import items, as determined by *howlmport*. The form of *howlmport* is one of:

(a) var
(b) forward

In the most common case, the *howlmport* is omitted, which means the item cannot be changed within the body of the importing procedure, function or module. If the *howlmport* is var, the item is necessarily a variable (or a module), and the importing body is then allowed to change the variable (or call a procedure in the module).

If the *importltem* is forward, the import list is necessarily part of a forward procedure or function declaration and the imported item is itself necessarily a procedure or function; see forward declarations for details and an example.

Can * frCely omitted for Procedures and functions
T T5510,, Simp,y means ^ *e implementation
t if f determ T S the list ^ l°°king at acLl use of items. By
assumes h^h'TT * ofa module is omitted, the implementation
In other wnrH £ T*™* },meanin S that no items «r^ imported
id^

member of a set

SYNTAX:

in

DESCRIPTION: The in operator determines if an element is in a set.

EXAMPLES:

```
type rankSet: set of 0 .. 10
var rankings: rankSet := rankSet(0) % The set {0}

if 5 in rankings then ...          % Is 5 in the ranking set?
```

DESCRIPTION: The not in operator is exactly the opposite of in. For example, 7 not in *rankings* means the same as not (7 in *rankings).*

It is required that the element be in the set's index type; in the above example this is satisfied because element 5 is in the index type 0.. 10.

See also the set type, *infix operators*, and *precedence* of operators.

include source files

SYNTAX: An *includeConstruct* is:

include *fileName*

DESCRIPTION: An include is used to copy parts files so that they become part of the Turing program. This copying is temporary in the since that is does not change any files. The file name must be an explicit string constant such as "stdstuff'.

EXAMPLE: On IBM PC compatible computers, there are arrow keys that produce character values such as 200 and 208. Let us suppose that a file called *arrows* contains definitions of these values:

```
const upArrow := 200
const dpwnArrow := 208
const rightArrow := 205
const leftArrow := 203
```

These definitions can be included in any program in the following manner:

```
include "arrows"

var ch : string (1)
getch (ch)                          % Read one character
case ord (ch) of
    label upArrow:
        ...handle up arrow...
    label downArrow:
        ...handle down arrow..
    label rightArrow:
        ...handle right arrow...
    label leftArrow:
        ...handle left arrow...
    label:
        ...handle any other key...
end case
```


DETAILS: An include file can itself contain include constructs. This can continue to any level, although a circular pattern of includes would be a mistake as it would lead to an infinitely long program.

It is common to save procedures, functions and modules in separate files, that are collected together using include.

index (find pattern in string) function

SYNTAX:

index (s, *patt* **: string): int**

DESCRIPTION: The index function is used to find the position *ofpatt* within string s. For example, index ("chair", "air") is 3.

EXAMPLE: This program outputs 2, because "ill" is a substring of "willing", starting at the second character of "willing".

```
var word : string := "willing"
put index (word, "ill" )
```

DETAILS: If the pattern *(patt)* does not appear in the string (s), index returns 0 (zero); for example, here is an if statement that checks to see if string s contains a blank:

```
if index (s, " " ) not= 0 then ...
```

The index is sometimes used to efficiently determine if a character is one of a given set of characters; for example, here is an if statement that checks to see if *ch,* which is declared using var *ch:* string (1), is a digit:

```
if index ( "0123456789", M not= 0 then ...
```

If a string contains more that one occurrence of the pattern, the leftmost location is returned; for example, index ("pingpong", "ng") returns 3.

If *patt* is the null string, the result is 1.

indexType

An *indexType* is one of:

```
(B>  subrangeType
 n   enumeratedType
     namodTypO          % Which is a subrange or enumerated type
(c)
```

DESCRIPTION: An index type defines a range of values that can be used as an array subscript, as a selector (tag) for a union type, or as the base type of a set type.

EXAMPLE:

```
varz : array 1 .. 9 of real      % 0..9 is an index type
type smallSet : set of 0 .. 2    %0..2 is an index type
```

infix operator

SYNTAX: *AninftxOperator* is one of:

(a)	+	*o Integer and real addition; set union; string catenation*
(b)	-	*'<, Integer and real subtraction; set difference*
(c)	*	*3 Integer and real multiplication; set intersection*
(d)	/	*> Real division*
(e)	dl'V	*i Truncating integer division*
(f)	mod	*. Remainder*
(g)	**	*• Integer and real exponentiation*
(h)	<	*Less than*
(0	>	*Greater than*
G)	=	*Equal*
(k)	<=	*% Less than or equal; subset*
(I)	>=	*% Greater than or equal; superset*
(m)	not=	*% Not equal*
(n)	and	*And (boolean conjunction)*
(o)	or	*Or (boolean disjunction)*
(P)	=>	*Boolean implication*
(q)	in	*Member of set*
W	not in	*Not member of set*

DESCRIPTION: An *infix operator is* placed between two values or *operands* to produce a third value, for example, the result of 5 + 7 is 12. In some cases the meaning of the operator is determined by its operands; for example, in "pine" + "apple", the + operator means string catenation while in 5 + 7 it means integer addition. There are also *prefix operators* (-, + and not), which are placed in front of a single value; see *prefix operator.*

In expressions with several operators, such as 3 + 4 * 5, the order of doing the operator is determined by *precedence* rules (see *precedence* for a listing of these rules); in this example, the multiplication is done before the addition, so the expression is equivalent to 3 + (4 * 5).

The numerical (integer or real) operators are +, -, *, /, div, mod, and **. AH of these except div produce a real result when at least one of their operands is real; if both operands are integers, the result is an integer except for real division (/) which always produces a real result regardless of the operands.

The div operator is like real division (/), except that it always produces an integer result, truncating any fraction to produce the nearest integer in the direction of zero.

The mod operator produces the remainder, which is the between real division (/) and integer division (div). When both operands are positive, this is the *modulo,* for example, 14 mod 10 is 4. If one of the operands is negative, a negative answer may result, for example, -7 mod 2 is -1. See also the int and real types.

The comparison operators (<, >, =, <=, >=, not=) can be applied to numbers as well as to enumerated types. They can also be applied to strings to see determine the *ordering* between strings (see the string type for details). Arrays, records, unions and collections cannot be compared. Boolean values (true and false) can be compared only for equality (= and not=); the same applies to pointer values. Set values can be compared using <= and >=, which are the subset and superset operators. The not= operator can be written as ~=.

Strings are manipulated using catenation (+) as well as substring expressions (see *substring)* and the *index* function (see *index).* See also the string type.

The operators to combine true/false values are and, or, and => (implication), as well as equality (= and not=). See also the boolean type-

The set operators are union (+), intersection (*), set difference (-), subset (<=), superset (>=), and membership (in and not in). See also the set type.

init initialisation

SYNTAX:

init

DESCRIPTION: The init (initialisation) keyword is used for two different purposes in Turing. The most common is for initialising arrays, records and unions. The less common is for recording parameter values in procedures for later use in post conditions.

EXAMPLE:

```
var mensNames: array 1 .. 3 of string :=
    init ("Tom", "Dick", "Harry" )
put mensNames ( 2)    % This outputs Dick
var names : array 1 .. 2,1 .. 3 of string :=
    init ("Tom", "Dick", "Harry",
          "Alice", "Barbara", "Cathy")
put names ( 2,1 )     % 7771/5 outputs Alice
```

DETAILS: The order of initialising values for multi-dimensional arrays is based on varying the right subscripts (indexes) most rapidly. This is called *row major order.* Initialisation of records and unions is analogous to initialising arrays; values are listed in the order in which they appear in the type. See array, record, and union types.

EXAMPLE: This procedure is supposed to set integer variable *i* to an integer approximation of its square root. The init clause records the initial value of *i* as/ so it can be used in the post condition to make sure that the approximation is sufficiently accurate. The name; can be used only in the post condition and nowhere else in the procedure.

```
procedure intSqrt (var i: int)
    pre i >= 0
    inity := i
    post abs (i - sqrt (j)) <= 1
    ... statements to approximate square root-
end intSqrt
```

DETAILS: See also pre and post assertions and procedure declarations.

type

SYNTAX: -

int

DESCRIPTION: The **int** (integer) type has the values ... -2, -1, 0, 1, 2 ... Integers can be combined by various operators such as addition (+) and multiplication (*). Integers can also be combined with real numbers, in which case the result is generally a real number. An integer can always be assigned to a real variable, with implicit conversion to real.

EXAMPLE:

```
var counter, i   int
vary: int := 9
var tax := 0        % The type is implicitly int because
                    %0isan integer
```

DETAILS' See also *explidtlntegerConstant.* The real type is used instead of int when values have fractional parts as in 16.837; see the real type for details^

The Turing operators on integers are +, -, * (multiply), div (truncating integer division), mod (integer remainder), ** (exponentiation), as well as comparisons (+, not=, >, >=, <, <=).

Real numbers can be converted to integers using *ceil* (ceiling), *floor,* and *round* (see descriptions of these functions). Integers can be converted to real numbers using *intreal,* but in practice this is rarely used, because an integer value used in place of a real value will be automatically converted to real.

Integers can be converted to strings and back using *intstr* and *strint.* Integers can be converted to corresponding ASCII (or EBCDIC) characters using *chr* and *ord.* See the descriptions of these functions.

Pseudo random sequences of integers can be generated using *randint;* see *randint.*

In Turing, the range of integers is from -2147483647 to 2147483647. In other words, the maximum size of integer is 2**31 -1.

integer-to-real function

SYNTAX:

intreal (/ : int) : real

DESCRIPTION: The intreal function is used to convert an integer to a real number. This function is rarely used, because in Turing, an integer value can be used anyplace a real value is required; when this is done, the intreal function is implicitly called to do the conversion from int to real. See also the floor, ceil and round functions.

integer-to-string function

SYNTAX:

intstr (/ , *width* : int): string

DESCRIPTION: The intstr function is used to convert an integer to a string. The string is equivalent to *i*, padded on the left with blanks as necessary to a length of *width*, for example, intstr (14,4)="bbl4" where *b* represents a blank. The *width* parameter is optional; if omitted, the string is made just long enough to hold the value. For example, intstr (-23) = "-23".

The *width* parameter must be non-negative. If *width* is not large enough to represent the value of i, the length is automatically increased as needed.

The string returned by intstr is of the form:

{blank)[-]digit{digits)

where (blank) means zero or more blanks, [-] means an optional minus sign, and digit(digit) means one or more digits. The leftmost digit is non-zero, or else there is a single zero digit; in other words, leading zeros are suppressed.

The intstr function is the inverse of strint, so for any integer i, strint (intstr (;')) = *i*.

See also the chr, ord and strint functions.

invariant assertion

SYNTAX: An *invariantAssertion* is:

invariant *trueFalseExpn*

DESCRIPTION: An invariant assertion is a special form of an assert statement that is used only in loop and for statements and in modules. It is used to make sure that a certain requirement is met; this requirement is given by the *trueFalseExpn*. The *trueFalseExpn* is evaluated. If it is true, all is well and execution continues. If it is false, execution is terminated with an appropriate message. See assert, loop and for statements and the module declarations for more details.

EXAMPLE: This program uses an invariant in a for loop. The invariant uses the function *namelnList* to specify that a key has not yet been found in an array of names.

```
var name: array 1 .. 100 of string
var key: string
... input name and key...

function namelnList (n: int): boolean
    for l: 1 .. n
        if key = name (l) then
            result true
        end if
    end for
    result false
end namelnList

for l: 1..100
    invariant not namelnList (j - 1)
    if key - name (j) then
        put "Found name at", j
        exit
    end if
end loop
```


length of a string function

SYNTAX:

length (s : string): int

DESCRIPTION: The length function returns the number of characters in the string. The string must be initialised. For example, lengthC'table") is 5.

EXAMPLE: This program inputs three words and outputs their lengths.

```
var word : string
for l: 1 .. 3
    get word
    put length (word)
end for
```

If the words are "cat", "robin" and "crow", the program will output 3, 5 and 4.

DETAILS: The length function gives the current length of the string. To find the maximum length of a string, use upper; for example, given the declaration var s: string (10), upper (s) returns 10. See also upper.

In natural logarithm function

SYNTAX:

```
ln ( r : real): real
```

DESCRIPTION: The In function is used to find the natural logarithm (base e)ofanumber. For example, In (1) isO. ;

EXAMPLE: This program prints out the logarithms of 1,2,3,... up to 100.

```
for /: 1 .. 100
    put "Logarithm of", /," is ", ln (/)
end for
```

DETAILS: See also the exp (exponential) function. It is illegal to try to take the logarithm of zero or a negative number.

NOTE: $\log_n$ (/) = ln (/) / ln *(n)*

locate procedure

SYNTAX:

```
locate ( row, column : ml)
```

DESCRIPTION: The locate procedure is used to move the cursor so that the next output from put will be at the given row and column. Row 1 is the top of the screen and column 1 is the left side of the screen.

EXAMPLE: This program outputs stars of random colors to random locations on the screen. The variable *coir* is purposely spelled differently from the word *color* to avoid the procedure of that name which is used to set the color of output. The row number is purposely chosen so that it is one less that maxrow to avoid the scrolling of the screen which occurs when a character is placed in the last column of the last row.

```
setscreen ("screen")
var row, column, coir: int
loop
    randint (row, 1 , maxrow - 1 )
    randint ( column, 1 , maxcol)
    randint (coir, 0, maxcolor)
    color (coir)
    locate (row, column)
    put "*" .. % Use dot-dot to avoid clearing end of line
end loop
```

DETAILS: The locate procedure is used to locate the next output based on row and column positions. See also the locatexy procedure which is used to locate the output based x and y positions, where x=0, y=0 is the left bottom of the screen.

The screen should be in a *"screen"* or *"graphics"* mode; see the setscreen procedure for details. If the screen is not in one of these modes, it will automatically be set *"screen"* mode.

See also setscreen and drawdot

locatexy graphics procedure

SYNTAX:

```
locatexy  (x, y : int)
```

DESCRIPTION: The locatexy procedure is used to move the cursor so that the next output from put will be at approximately *(x, y)*. The exact location may be somewhat to the left of *x* and below y to force alignment to a character boundary.

EXAMPLE: This program outputs "Hello" starting at approximately (100, 50) on the screen.

```
setscreen ("graphics")
locatexy (100, 50)
put "Hello"
```

DETAILS: The locatexy procedure is used to locate the next output based on x and y positions, where the position x=0, y=0 is the left bottom of the screen. See also the locate procedure which is used to locate the output based in row and column positions, where row 1 is the top row and column 1 is the left column.

The screen should be in a *"graphics"* mode; see the setscreen procedure for details. If the screen is not in a *"graphics"* mode, it will automatically be set to *"graphics"* mode.

See also setscreen and drawdot

statement

SYNTAX: A *loopStatement* is:

```
loop
    statementsAndDeclarations
end loop
```

DESCRIPTION: A loop statement causes the statements (and declarations) in it to be repeatedly executed. This continues until terminated by one of its enclosed exit statements (or by an enclosed return or result statement).

EXAMPLE: Output on separate lines: Happy, Happy, Happy, etc.

```
loop
    put "Happy"
end loop
```

EXAMPLE: Read words up to the word Stop.

```
var word : string
loop
    get word
    exit when word = "Stop"
end loop
```

DETAILS: A loop statement can contain more that one exit, or none at all (in which case it is an infinite loop). When the exit when is at the beginning of the loop, the loop works like Pascal's do while; when at the end, the loop works like Pascal's repeat until.

Just preceding the statements and declarations, you are allowed to write an "invariant clause" of the form:

invariant *trueFalseExpn*

This clause is equivalent to: assert *trueFalseExpn*.

lower bound of an array or string

SYNTAX:

lower *(arrayReference [, dimension]*): int

DESCRIPTION: The lower attribute is used to find the lower bound of an array. (See upper for finding the upper bound.) Since the lower bound of every array is necessarily know at compile time, lower is rarely used.

maximum function

SYNTAX:

max (*expn , expn*)

DESCRIPTION: The max function is used to find the maximum of two numbers (the two *expn's).* For example, max (5, 7) is 7. If both numbers are int the result is int, but if one or both of the numbers are real, the result is real. See also the min function.

EXAMPLE: This program outputs 85.72.

```
var x : real := 74.61
var y : real := 85.72
put max (x, y )          % outputs 35.72
```

EXAMPLE: This program inputs 10 numbers and outputs their maximum.

```
var m , t : real
get m                    % Input first number
for / : 2 .. 10          % Handle remaining 9 numbers
    geu
    m := max ( m, t)
end loop
put "The maximum is ", m
```

(maximum column) function

SYNTAX:

maxcol : int

DESCRIPTION: The maxcol function is used to determine the number of columns on the screen.

EXAMPLE: This program outputs the maximum column number.

```
put "Number of columns on the screen is ",
    maxrow
```

DETAILS: For IBM PC compatibles as well as most Unix dumb terminals, in *"text" or "screen"* mode, maxcol = 80. For the default IBM PC compatible *"graphics"* mode (CGA), maxcol = 40. See the locate procedure for an example of the use of maxcol.

fliaxcolor graphics function

SYNTAX:

maxcolor : int

DESCRIPTION: The maxcolor function is used to determine the maximum color number for the current mode of the screen. The alternate spelling is maxcolour.

EXAMPLE: This program outputs the maximum color number.

```
setscreen ("graphics")
put "The maximum color number is", maxcolor
```

DETAILS: The screen should be in a *"screen"* or *"graphics"* mode; if not, it will automatically beset to *"screen"* mode. See setscreen for details.

See drawdot and palette for examples of the use of maxcolor. See the color procedure which is used for setting the currently active color.

For IBM PC compatibles in *"screen"* mode, maxcolor = 15. For the default IBM PC compatible *"graphics"* mode(CGA), maxcolor = 3.

(maximum row) function

SYNTAX:

```
maxrow : int
```

DESCRIPTION: The maxrow function is used to determine the number of rows on the screen.

EXAMPLE: This program outputs the maximum row number.

```
put "Number of rows on the screen is ", maxrow
```

DETAILS: For IBM PC compatibles , maxrow = 25. For many Unix dumb terminals, maxrow = 24. See the locate procedure for an example of the use of maxrow.

graphics function

[PC and Mac only]

SYNTAX:

```
maxx : int
```

DESCRIPTION: The maxx function is used to determine the maximum value of x for the current graphics mode.

EXAMPLE: This program outputs the maximum x value.

```
setscreen ("graphics")

put "The maximum x value is", maxx
```

DETAILS: The screen should be in a *"graphics"* mode; if not, it will automatically be set to *"graphics"* mode. See setscreen for details.

See drawdot for an example of the use of maxx and for a diagram illustrating x and y positions. For the default IBM PC compatible graphics mode (CGA), maxx = 319.

graphics function

SYNTAX:

maxy : int

DESCRIPTION: The maxy function is used to determine the maximum value of y for the current graphics mode.

EXAMPLE: This program outputs the maximum y value.

```
setscreen ("graphics")

put "The maximum y value is ", maxy
```

DETAILS: The screen should be in a *"graphics"* mode; if not, it will automatically be set to *"graphics"* mode. See setscreen for details.

See drawdot for an example of the use of maxy and for a diagram illustrating x and y positions. For the default IBM PC compatible graphics mode (CGA), maxy =199.

minimum function

SYNTAX:

min (*expn , expn)*

pESCRIPTION: The min function is used to find the minimum of two numbers (the two *expris).* For example, min (5,7) is 5. If both numbers are int the result is int, but if one or both of the numbers are real, the result is real. See also the max function.

EXAMPLE: This program outputs 74.61.

```
var x : real := 74.61
var y : real := 85.72
put min (x, y)          % outputs 74.61
```

EXAMPLE: This program inputs 10 numbers and outputs their minimum.

```
var m , t : real
get m                   % Input first number
for / I 2 .. 10         % Handle remaining 9 numbers
    getf
    m := min ( m, t)
end loop
put "The minimum is ", m
```

mod remainder (modulo) operator

SYNTAX:

mod

DESCRIPTION: The mod *(modulo)* operator produces the remainder of one number divided by the another. For example, 7 mod 2 produces 1 and -7 mod 2 produces -1.

EXAMPLES: In this example, *eggCount* is the total number of eggs. The first put statement determines how many dozen eggs there are. The second put statement determines how many extra eggs there are beyond the last dozen.

```
var eggCount : int
get eggCount
put "You have ", eggCount div 12," dozen eggs"
put "You have", eggCount mod 12," left over"
```

DESCRIPTION: See also *infix operators, precedence* of operators and the div operator.

jX\odule declaration

SYNTAX: A *moduleDeclaration* is:

```
module id
    [ import ([ var ] id {, [ var ] id }) ]
    [ export ([ opaque ] id•{, [ opaque ] id } ) ]
    statementsAndDeclarations
end id
```

DESCRIPTION: A module declaration creates a package of variables, constants, types and subprograms, and sub-modules. The name of the module (id) is given in two places, just after module and just after end.

EXAMPLE: Implement a stack of strings:

```
module stack
    export (push, pop )

    var top: int := 0
    var contents: array 1 .. 100 of string

    procedure push ( s : string)
        top := top + 1
        contents (top) := s
    end push

    procedure pop (var s: string)
        s := contents (top)
        top := top -1
    end push
end stack

stack. push ("Harvey" )
var name : string
stack.pop (name)          % This sets name to Harvey
```

DETAILS: In other programming languages, the term used for a module is a *package, cluster* or *object.*

A module declaration is executed (and the module is initialised) by executing its declarations and statements; for example, the *stack* module is initialised by setting the *top* variable to 0. This initialisation executes all the statements and declarations in the module that are not contained in procedures or functions. The initialisation is completed before any procedure or function of the module can be called from outside the module. Such a call to a procedure or function simply executes the body of that procedure or function (the module is *not* initialised with each such call).

The import list gives the names of items declared outside the module that can be accessed inside the module; since *stack* has no import list, it is not allowed to access any names declared outside of it.

The export list is used to implement *information hiding,* which means isolating implementation details inside a module. The export list gives the names of items declared inside the module that can be used outside the module; for example, *push* and *pop* are exported from *stack.* Each such use of an exported item must be preceded by the module name and a dot, for example, *stack.push.* Names that are not exported, such as *top* and *contents,* cannot be accessed outside the module.

Only procedures, functions, constants and types can be exported; variables and (sub)modules cannot be exported.

The opaque keyword is used (only) to precede type names that have declarations in the module. Outside of the module, the type will be distinct from all others types, this means, for example, that if the opaque type is a record, its fields cannot be accessed outside of the module. Opaque types are used to guarantee that certain items are inspected and manipulated in only one place, namely, inside the module. These types are sometimes called *abstract data types.*

EXAMPLE: Use an opaque type to implement complex arithmetic.

```
module complex
    export (opaque value, constant, add,
                                ... other operations... )
    type value:
        record
            realPt, imagPt : real
        end record

    function constant
                    (realPt, imagPt: real): value
        var answer : value
        answer. realPt := realPt
        answer. imagPt := imagPt
        result answer
    end constant

    function add (L, R: value): value
        var answer : value
        answer .realPt := L .realPt + R .realPt
        answer .imagPt :=L . imagPt +R .imagPt
        result answer
    end add

... other operations lor complex arithmetic go here ...
end complex

var c,d-.complex .value :=complex.constant(1,5)
            % c and d become the complex number (1,5)
var e : complex .value := complex.add (c,d)
            % e becomes the complex number (2,10)
```

DETAILS: Module declarations can be nested inside other modules but cannot be nested inside procedures or functions. A module must not contain a bind as one of its (outermost) declarations. A return statement cannot be used as one of the (outermost) statements in a module.

The syntax of a *moduleDeclaration* presented above has been simplified by leaving out pre, invariant and post conditions; the full syntax is:

```
module id
    [ import ([ var ] id {, [ var ] id } ) ]
    [ export ([ opaque ] id {, [ opaque ] id } ) ]
    [ pre trueFalseExpn ]
    statementsAndDeclarations
    [ invariant trueFalseExpn ]
    statementsAndDeclarations
    [ post trueFalseExpn ]
end id
```


named type

SYNTAX: A *namedType* is one of:

(a) *typeId*
(b) *moduleId. typeId*

DESCRIPTION: A type can be given a name *(typeId)* and later this name can be used instead of writing out the type.

EXAMPLE: In this example, *phoneRecord* is a named type.

```
type phoneRecord :
    record
        name : string (20)
        phoneNumber : int
        address : string (50)
    end record

var oneEntry : phoneRecord
var phoneBook : array 1 .. 100 of phoneRecord
```

DETAILS: Form (a) is the most common kind of named type. Form (b) is used when the type name has been exported from a module.
Arrays whose bounds are not known at compile time cannot be named.

nargS number of arguments IPC, statement

SYNTAX:

nargs : int

DESCRIPTION: The nargs function is used to determine the number of arguments that have been passed to a program from the command line. For example, if the program is run from the Turing environment using

:rfilelfile2

then nargs will return 2. If a program called *prog.x* is run under Unix using this command

prog.x filel file2

the value of nargs will similarly be 2.

The nargs function is usually used together with the fetcharg function to access the arguments that have been passed to the program. See fetcharg for an example of the use of nargs.

SYNTAX: *AnewStatementis:*

new *collectionId, pointerVariableReference*

DESCRIPTION: A new statement creates (allocates) an element of a collection and assigns its location to the pointer variable.

EXAMPLE: Declare a list and allocate one of its nodes.

```
var list : collection of
    record
        contents : string (10)
        next : pointer to list
    end record
var first: pointer to list
new list, first     % Allocate an element of list,
                    % located by first.
```

DETAILS: The opposite of allocating an element of a collection, namely, deallocating it, is done by the free statement.

If there is no more space to allocate an element, new will set the pointer to be the nil value, in this example, to *nil (list).*

See also the collection declaration, the pointer type, and the free statement.

nil pointer to a collection

SYNTAX: "

nil *(collectionId)*

DESCRIPTION: The nil pointer for a collection does not locate any element of the collection. This pointer is distinct from pointers to actual elements of the collection and it can be compared to these pointers. It is also distinct from the uninitialised pointer value.

EXAMPLE: In this example, the pointer called *first* is set to the nil pointer of collection *c,* that is, to nil(c).

```
var c : collection of
    record
        name : string (50)
        next: pointer to c
    end record
var first: pointer to c := nil ( c)
```

DETAILS: See also collection declaration.

jlOt true/false (boolean) operator

SYNTAX:

not

DESCRIPTION: The not *(boolean negation)* operator produces the opposite of a true/false value. For example, not *(x >y)* is equivalent to ***x<=y.***

EXAMPLES:

```
var error : boolean := talse
var success : boolean
...
success := not error        % success becomes the
                            % opposite of error
```

DETAILS: The not operator takes true and produces false and takes false and produces true. The not operator can be written as ~. See also the boolean type, *prefix operators,* and *precedence* of operators.

opaque type

DESCRIPTION: When a type *T* is exported from module M using-the keyword opaque, the type *M.T* is distinct from all other types. Opaque types are used to guarantee that all updates to values of the type are done within module M.

See module declarations for an example of an opaque type used to implement complex arithmetic. See *equivalence* of types for the definition of the type matching rules for opaque types.

file statement

SYNTAX: An *openStatement is* one of:

(a)

open : *fileNumberVar, fileName , ioCapability*
{ , ioCapability }

open : *fileNumberVar, argNumber , ioCapability*
{ , ioCapability }

DESCRIPTION: The open statement connects the program to a file so the program can perform operations such as read on the file. In form (a), the open statement translates a *fileName*, such as "Master", to a file number such as 5. Form (b) , which is less commonly used, opens a file whose name is given by a program argument; this is described below.

The read statement uses the file number, not the file name, to access the file. When the program is finished using the file, it disconnects from the file using the close statement. Each *ioCapability* is the name of an operation, such as read, that is to be performed on the file.

EXAMPLE: This programs illustrates how to open, read and then close a file.

```
var fileName : string := "Master"   % Name
Var fileNo : int                    % Number of file
var inputVariable : string ( 100 )
open : fileNo, fileName, read

read : fileNo, inputVariable

close : fileNo
```

DETAILS: The open statement always sets the *fileNumber* to a positive number. If the open fails (generally because the file does not exist), the *fileNumber* is set to zero.

An *ioCapability* is one of:
get, put, read, write, seek, mod
A file can be accessed using only the statements corresponding to the input/output capabilities with which it was opened. Note: tell is allowed only if the open is for seek.

The open statement truncates the file to length zero if the *ioCapabilities* include put or write but not mod (which stands for modify). In all other cases, open leaves the existing file intact. The mod *ioCapability* specifies that the file is to be modified without being truncated. Each open positions to the beginning of a file. There is no mechanism to delete a file. To open for appending to the end of the file, one has to open for seek (and for write or put) and then seek to the end (see the seek statement).

Mixed mode files, which combine get and read (or put and write), are supported by some operating systems, such as Unix, but not by others, such as MS-DOS.

Form (b) of the syntax allows opening of a file whose name is given as a program argument on the command line. For example, under Unix, the command line

prog.x infile outfile

specifies to execute *prog.x* with program arguments *infile and outfile.* Similarly, in the Turing programming environment, the *run* command can accept program arguments. The *argNumber* is the position of the argument on the command line. (The first argument is number 1.) The name of the file to be opened is the corresponding program argument. If there is no such argument, or if the file cannot be opened successfully, *fileNumberVariable* is set to zero. See also nargs, which gives the number of arguments, and f etcharg, which gives the n-th argument string.

Program argument files referenced by argument number and used in put, get, read or write statements need not be explicitly opened, but are implicitly opened with the capability corresponding to the input/output statement in which they are first used. (The *fileNumber* gives the number of the argument.)

The operating system standard files (error, output and input) are accessed using file numbers 0, -1, and -2, respectively. These files are not opened explicitly, but are used simply by using form (b) with the number. Beware of the anomalous case of a failed open that gives you file number 0. A subsequent use of this number in a put will produce output that goes to the standard error stream, with no warning that the file you attempted to open is not actually being used.

See also the close, get, put, read, write, seek and tell statements.

There is an older and still acceptable version of open that has this syntax:

open (var *fileNumber :* Int, *fileName :* **string,** *mode :* **string)**

The *mode* must be "r" (for get) or "w " (for put).

or (boolean) operator

SYNTAX:

AorB

DESCRIPTION: The or (boolean) operator yields a result of true if at least one (or both) of the operands is true, or is a short circuit operator; for example, if *A* is true in *A* or B then *B* is not evaluated.

EXAMPLE:

```
var success: boolean := false
var continuing := true   % the type is boolean

continuing := continuing or success
```

DETAILS: *continuing* is set to false if and only if both *continuing* and *success* are falsae. Since Turing uses short circuit operators, once *continuing* is true, *success* will not be looked at.

See also *boolean* (which discusses true/false values), *explicitTrueFalseConstant* (which discusses the values true and false), *precedence* and *expn* (expression).

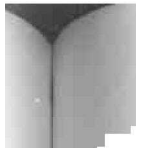
Ord character-to-integer function

SYNTAX:

ord (*ch :* string (1)) : int

DESCRIPTION: The ord function accepts an enumerated value or a string of length 1 and returns the position of the value in the enumeration or of the character in the ASCII (or EBCDIC for IBM mainframes) sequence. Values of an enumerated type are numbered left to right starting at zero. For example, ord ("A") is 65. The ord function is the inverse of chr, so for any character *c*, chr (ord *(c)) = c*.

See also the chr, intstr and strint functions.



palette graphics procedure [PC only]

SYNTAX:

palette *(p: ml)*

)ESCRIPTION: The palette procedure is used to change the palette number to *p*.

[EXAMPLE: This program shows all the colors of palette number 3 for an IBM PC compatible using CGA graphics. The first line of output, for color 0, will not be visible, because the background is also color 0.

```
setscreen ("graphics")
palette (3)
for colorNumber: 0 .. maxcolor
    color (colorNumber)
    put "Color number", colorNumber
end for
```

)ETAILS: The meaning of the palette depends on the display hardware on the computer. On IBM PC compatibles under CGA (the default graphics mode), there are palettes numbered 0 to 3. The main palettes are numbers 2 and 3. Here is the meaning of the color numbers under these CGA palette numbers .

Palette 2: 1 = cyan (blue), 2 = magenta (red), and 3 = white.
Palette 3: 1 = green, 2 = red, 3 = brown.

Palette number 0 is like 2 but not as bright. Palette 1 is like 3 but not as bright.

The palette procedure is meaningful only in a *"graphics"* mode. See setscreen for a description of the graphics modes.

See also whatpalette, which is used to determine the current palette number. See also drawdot and maxcolor.

paramDeclaration
parameter declaration

SYNTAX: A *paramDeclaration* is one **of:**

(a) [var] *id {, id } : typeSpec*

(t>) **procedure** ***id***
[(paramDeclaration {, paramDeclaration })]

(c) **function** ***id***
[(paramDeclaration {, paramDeclaration })]
: typeSpec

DESCRIPTION: A parameter declaration, which is part of the header of a procedure or function, specifies a formal parameter (see also procedure and function declarations). Form (a) above is the most common case. Forms (b) and (c) specify procedures and functions that are themselves passed as parameters.

EXAMPLES:

```
procedure putTitle (title: string)
    % The parameter declaration is:  title: string
    put  title
end putTitle

procedure x (var s : array 1 .. * of string (*))
    % Set each element of s to the null string
    for i : 1 .. upper (s)
        s (i) := ""
    end for
end x
```

DETAILS: Parameters to a procedure may be declared using var, which means that the parameter can be changed inside the procedure; for example, s is changed in the *x* procedure. If a parameter is declared without var, it cannot be changed. (This is different from the Pascal language, in which non-var parameters can be changed.) Parameters

to functions cannot be declared to be var.

Parameters declared var are passed by reference, which means that a pointer to the value is passed to the procedure, rather than passing the actual value. This implies that in the call *p (a* (0), in which array element fl(i') is passed to procedure *p*, a change to *i* in *p* does not change the element referred to by p's actual parameter. Every non-scalar (not integer, subrange, real, boolean, enumerated or pointer) parameter is passed by reference whether or not declared var. In all other cases (scalar non-var parameters) the parameter is passed by value (the actual value is copied to the procedure).

The upper bound of an array or string that is a formal parameter may be specified as star (*), as is done above for parameter s in procedure *x*. This specifies that size of the upper bound is inherited from the corresponding actual parameter. Parameters declared using star are called *dynamic* parameters.

The names of the formal parameters must be distinct from each other, from the procedure or function name, and from pervasive identifiers. However, they need not be distinct from names outside of the procedure or function.

EXAMPLE: Find the zero of function f. This example illustrates form (c), which is a parameter that is a function.

```
function findZero (function f ( x : real): real,
        left, right, accuracy: real): real
    pre sign (f (left)) not= sign (f ( right)))
            and accuracy > 0
    var L : real := left
    var R: real := right
    var M: real
    const signLeft := sign (f (left ))
    loop
        M := ( R + L) / 2
        exit when abs ( R - L) <= accuracy
        if signLeft =sign (f (M)) then
            L:=M
        else
            R := M
        end if
    end loop
```

```
    result M
end findZero
```

DETAILS: Forms (b) and (c) of *paramDedaration* are used to specify formal parameters that are themselves procedures or functions. For example, in *the findZero* function,/is a formal parameter that is itself a function.

play procedure

[PC only]

SYNTAX:

play (*music* : string)

DESCRIPTION: The play procedure is used to sound musical notes on the computer.

EXAMPLE: This program sounds the first three notes of the C scale.

```
play ("cde" )
```

DETAILS: The play procedure takes strings that contain characters that specify notes, rests, sharps, flats, duration. The notes are the letters a to g (or A to G). A rest is p (for pause). A sharp is + and a flat is -. The durations are 1 (whole note), 2 (half note), 4 (quarter note), 8 (eight note) and 6 (sixteenth note). The character > raises to the next octave and < lowers. For example, this is the way to play C and then C sharp one octave above middle C with a rest between them, all in sixteenth notes: play(">6cpc+"). Blanks can be used for readability and are ignored by play.

Under some systems such as Unix, the play procedure has no effect.

See also the playdone function which is used to see if a note has finished sounding. See also the sound procedure which makes a sound of a given frequency (Hertz) and duration (milliseconds).

playdone function

SYNTAX:

```
playdone: boolean
```

DESCRIPTION: The playdone function is used to determine when notes played by the play procedure have finished sounding.

EXAMPLE: This program sounds the first three notes of the C scale and outputs "All done" as soon as they are finished. Without the loop, the message would come out before the notes are finished.

```
play("cde" )
loop
    exit when playdone
end loop
put "All done"
```

DETAILS: Under some systems such as Unix, the playdone procedure is meaningless.

See also the play procedure. See also the sound procedure which makes a sound of a given frequency (Hertz) and duration (milliseconds).

pointer type

gYNTAX: A pointer-Type is:

```
pointer to collectionId
```

pESCRIPTION: A variable declared by a pointer type is used to located the elements of the collection whose name is *collectionId.* The new statement creates a new element of the collection and places the element's location in a pointer variable. The free statement destroys an element located by a pointer variable.

EXAMPLE: Create a collection that will represent a binary tree.

```
var tree : collection of
    record
        name : string (10)
        left, right: pointer to tree
    end record

var root: pointer to tree
new tree, root
tree (root ).name := "Adam"
```

DETAILS: In Turing, a pointer is effectively a subscript (an index) for a collections. Pointers can be assigned, compared for equality and passed as parameters.

See collections for more details about the use of pointers. See also, new and free statements.

post assertion

SYNTAX: An *postAssertion* is:

post *trueFalseExpn*

DESCRIPTION: A post assertion is a special form of an assert statement that is used in a procedure or function. It is used to give requirements that the body of the procedure or function is supposed to satisfy. These requirements are given by the *trueFalseExpn.* After the body has executed and just before the procedure or function returns, the *trueFalseExpn* is evaluated. If it is true, all is well and execution continues. If it is false, execution is terminated with an appropriate message. See assert statements and procedure and function declarations for more details. See also pre and invariant assertions.

EXAMPLE: This function is supposed to produce an integer approximation of the square root of integer *i.* The post condition requires that this result, which is called *answer,* must be within a distance of 1 from the corresponding real number square root.

```
function intSqrt (i: int) answer : int
    pre i >= 0
    post abs (answer- sqrt( i'))<= 1
    ... statements to approximate square root-
end intSqrt
```

DETAILS: A post assertion can also be used in a module declaration to make sure that the initialisation of the module satisfies its requirements; see module declaration for details.

assertion

SYNTAX: An *preAssertion* is:

pre *trueFalseExpn*

pESCRIPTION: A pre assertion is a special form of an assert statement that is used in at the beginning of a procedure or function. It is used to give requirements that the caller of the procedure or functions is supposed to satisfy. These requirements are given by the *trueFalseExpn.* The *trueFalseExpn* is evaluated. If it is true, all is well and execution continues. If it is false, execution is terminated with an appropriate message. See assert statements and procedure and function declarations for more details. See also post and invariant assertions.

EXAMPLE: This function computes the average of *n values.* Its pre condition requires that *n* must be strictly positive, to avoid the possibility of dividing by zero when computing the average.

```
function average (a: array 1 .. * of real,
        n: int): real
    pre n > 0                              .
    var sum: real := 0
    for i: 1 .. n
        sum := sum* a( i)
    end for
    result sum/n
end average
```

DETAILS: A pre assertion can also be used in a module declaration to make sure that requirements for initialisation of the module are met; see module declaration for details.

precedence of operators

DESCRIPTION: The order of applying operators in an expression such as 3 + 4*5 are determined by Turing's *precedence* rules. These rules state, for example, that multiplication is done before addition, so this expression is equivalent to 3 + (4 * 5).

Parenthesised parts of an expression are evaluated before being used; for example, in (1 + 2) * 3, the addition is done before the multiplication.

The precedence rules are defined by this table, in which operators appearing earlier in the table are applied first; for example multiplication is applied before addition:

(1) –
(2) *prefix* + and -
(3) *, /, div, mod*
(4) *infix* +,-
(5) <, >, =, <=, >=, not=, in, not in
(6) not
(7) and
(8) or
(9) => (*boolean implication)*

Operators appearing on a single line in this table are applied from left to right; for example, *a-b-c* is the same is *(a-b)-c*.

Here are some examples illustrating precedence, in which the left and right expressions are equivalent:

-1**2	-d**2)
*a+b*c*	
*a*blc*	
*bore*and*d*	*b* or *(c* and *d)*
x <*y* and*y* < z	*(x* < *y)* and (y < z)

The final example illustrates the fact that in Turing, parentheses are not required when combining comparisons using and and or; these would be required in the Pascal language.

See also *infix* and *prefix* operators and the int, real, string, boolean, set and enum types.

predecessor function

SYNTAX: . - ;

pred (*expn*)

DESCRIPTION: The pred function accepts an integer or an enumerated value and returns the integer minus one, or the previous value in the enumeration. For example, pred (7) is 6.

EXAMPLE: This part of a Turing program fills up array *a* with the enumerated values *red, yellow, green, red, yellow, green,* etc.

```
type colors : enum ( green, yellow, red )
var a : array 1 .. 100 of colors
var c : colors := colors .red
for /: 1 ..100
    a ( / ) := c
    if c = colors . green then
        c := colors . red
    else
        c:= pred (c)
    end if
end for
```

DETAILS: It is illegal to apply pred to the first value of an enumeration. See also the succ function.

prefix operator

SYNTAX: *AprefixOperator* is one of:

```
(a)   +      % Integer and real identity  (does not change value)
(b)   -      % Integer and real negation
(c)   not    % Not (Boolean negation)
```

DESCRIPTION: A *prefix operator* is placed before a value or *operand* to produce another value, for example, if the value of *x* is seven then *-x* is negative seven. There are also infix operators such as multiplication (*) and addition (+), which are placed between two values to produce a third value; see *infix operator*.

The + and - prefix operators can be applied only to numeric values (integer and real). The not prefix can be applied only to true/false (boolean) values; for example not *(x > y)* is equivalent to *x <= y*. The not operator produces true from false and false from true.

See also the int, real and boolean types, as well as *precedence* (for the order of applying operators) and *infix operators*.

procedure declaration

SYNTAX: A *procedureDecluration* is:

```
procedure id
    [ (paramDeclaration {, paramDeclaration }) ]
    statementsAndDeclarations
end id
```

DESCRIPTION: A procedure declaration creates (but does not run) a new procedure. The name of the procedure (id) is given in two places, just after procedure and just after end.

EXAMPLES:

```
procedure greetings
    put "Hello world"
end greetings

greetings                    % This outputs Hello world

procedure sayItAgain ( msg: string , n : int)
    for i : 1 .. n
        put msg
    end for
end sayItAgain

sayItAgain ("Toot" , 2)  % Toot is output twice

procedure double (var x: real)
    x := 2 * x
end double

vary : real := 3.14
double ( y )         % This doubles the value of y
```

DETAILS: The set of parameters declared with the procedure are called *formal* parameters; for example, in the *double* procedure, *x is* a formal parameter. A procedure is called (invoked) by a procedure *call statement* which consists of the procedure's name followed by the parenthesised list *of actual* parameters (if any); for exarhple, *double(y)* is a call having y as an actual parameter. If there are no parameters (see the *greet* procedure above), the call does not have parentheses. The keyword procedure can be abbreviated to proc.

Ordinarily, a procedure *returns* (finishes and goes back to the place where it was called) by reaching its end. However, the return statement in a procedure causes it to return immediately. Note that return can also be used in the main program to cause it to halt immediately.

Only parameters declared using var may be changed in the procedure, for example, *x* is changed in the *double* procedure. The upper bounds of arrays and strings that are parameters may be declared to be star (*), meaning the bound is that of the actual parameter. See *paramDedaration* for details about parameters.

Procedures and functions cannot be declared inside other procedure and functions.

The syntax of a *procedureDeclaration* presented above has been simplified by leaving out the import list, pre and post condition and init clause; the full syntax is

```
procedure id
    [ (paramDedaration {,paramDeclaration }) ]
    [ import ([ [var] id {, [var] id } ]) ]
    [ pre trueFalseExpn ]
    [ init id := expn {, id := expn } ]
    [ post trueFalseExpn ]
    statementsAndDeclarations
end id
```

See import list, pre condition, init clause and post condition for explanations of these additional features.

A procedure must be declared before being called; to allow for mutually recursive procedures, there are forward declarations of procedures with later declaration of each procedure body. See forward and body declarations for explanations.

statement

SYNTAX: *AprocedureCall* is;

```
procedureId [ ( expn { , expn} ) ]
```

DESCRIPTION: A procedure call is a statement that calls (invokes or activates) a procedure. If the procedure has parameters, a parenthesised list of expressions *(expns)* must follow the procedure's name *(procedureId).*

EXAMPLES:

```
procedure greet
    put "Hello"
end greet

greet       % This is a call to the greet procedure

procedure times ( var i: int, factor : int)
    i := factor * i
end times

var y: int
times ( j, 4)       % Multiply j by 4
```

DETAILS: The parameter declared in the header of a procedure, is a *formal* parameter, for example, i and *factor* above are formal parameters. Each expression in the call is an *actual* parameter, for example, / and 4 above are actual parameters.

If a formal parameter is declared using var, then the expression passed to that parameter must be a variable reference (so its value can potentially be changed by the procedure); this means for example that it would be illegal to pass ;'+3 as the first parameter to *times.* The variable reference and the formal parameter must have equivalent types (see *equivalence* for details).

Each actual parameter passed to a non-var formal parameter must be assignable to that parameter (see *assignability* for details). See also *procedureDeclaration.*

In this explanation of *procedureCall,* we have up to this point ignored the possibility of procedures exported from modules. If the procedure is being called from outside of a module from which has been exported, the syntax of the *procedureCall* is:

moduleId. procedureId [(expn {, expn})]

In other words, the module's name and a dot must precede the procedure's name.

program an entire Turing program

SYNTAX: *A program* is:

statementsAndDeclarations

DESCRIPTION: A Turing program consists of a list of statements and declarations.

EXAMPLES: This is a complete Turing program. It outputs *Alan M. Turing.*

```
put "Alan M. Turing"
```

EXAMPLES: This is a complete Turing program. It outputs a triangle of stars.

```
var stars: string := "*"
loop
    put stars
    stars := stars + "*"
end loop
```

EXAMPLES: This is a complete Turing program. It outputs *Hello* once and *Goodbye* twice.

```
procedure sayItAgain ( what: string, n : int)
    for /: 1 .. n
        put what
    end for
end sayItAgain

sayItAgain ("Hello", 1)
sayItAgain ("Goodbye", 2)
```

put statement

SYNTAX: *AputStatementis:*

put [: *fileNumber,} putItem { , putItem} [. .]*

DESCRIPTION: The put statement outputs each of the *putItems.* Ordinarily, after the final *putItem,* a new line is started in the output. However, if the optional dot-dot (..) is present, subsequent output will be continued on the current output line. With character graphics, the omission of dot-dot causes the remainder of the output line to be cleared to blanks.

Ordinarily, the output goes to the screen. However, if the *fileNumber* is present, the output goes to the file specified by the file number (see the open statement for details). Also, output can be redirected from the screen to a file by a command such as :r > fileName

EXAMPLE:

```
var n : int := 5
put "Alice owes me $", n
        % Output is: Alice owes me $5
        % Note that no extra space Is
        % output before an integer such as n.
```

EXAMPLES:

Statement	Output	Notes
put 24	24	
put 1/10	0.1	Trailing zeros omitted
put 100/10	10	Decimal point omitted
put 5/3	1.666667	6 fraction digits
put *sqrt* (2)	1.414214	6 fraction digits
put 4.86 *10**9	4.86e9	Exponent for > 1 e6
put 121 :5	bb121	Width of 5; "b" is blank
put 1.37 : 6 :3	b1.370	Fraction width of 3
put 1.37 :11 : 3 : 2	bb1.370e+00	Exponent width of 2
put "Say VHelloV"	Say "Hello"	
put "XX": 4, "Y"	*XXbbY*	Blank shown as *b*

EXAMPLE: A single blank line is output this way:

```
put ""                                  % Output null string
then new line
```

This put statement is sometimes used to close off a line that has been output piece by piece using put with dot-dot.

DETAILS: The general form of a *putItem* is one of:

(a) *expn [: uridthExpn [: fradionWidth [: exponentWidth]]]*
(b) skip

See the above examples for uses of *widthExpn, fradionWidth* and *exponentWidth;* for the exact meaning of these three widths, see the definitions of the functions *realstr, frealstr* and *erealstr.* The skip item is used to end the current output line and start a new line.

rand random real number procedure

SYNTAX:

rand (var *r* : real)

DESCRIPTION: The rand statement is used to create a pseudo-random number in the range zero to one. For example, if *x* is a real number, after rand(x), *x* would have a value such as 0.729548 or 0.352879.

EXAMPLE: This program repeatedly and randomly prints out "Hi ho, hi ho" or "It's off to work we go".

```
var r : real
loop
    rand (r)
    if r > 0.5 then
        put "Hi ho, hi ho"
    else
        put "It's off to work we go"
    end if
end loop
```

DETAILS: The rand statement sets its parameter to the next value of a sequence of pseudo-random real numbers that approximates a uniform distribution over the range 0<r <1.

Each time a program runs, rand uses the same pseudo-random number sequence. To get a different sequence (actually, to start the sequence at a different point), use the randomize procedure.

To use several sequences of repcatable pseudo-random number sequences, use the randseed and randnext procedures.

See also randint, randomize, randseed and randnext.

fanoint random integer procedure

SYNTAX:

randint (var *i* : int, *low, high* : int)

DESCRIPTION: The randint statement is used to create a pseudo-random integer in the range *low* to *high,* inclusive. For example, if *i* is an integer, after randintO', 1,10), *i* would have a value such as 7 or 2 or 10.

EXAMPLE: This program simulates the repeated rolling of a six sided die.

```
var roll: int
loop
    randint(i,1,6)
    put "Rolled", i
end loop
```

DETAILS: The randint statement sets its parameter to the next value of a sequence of pseudo-random integers that approximates a uniform distribution over the range *low* <i < *high* . It is required that *low* < *high.*

Each time a program runs, randint uses the same pseudo-random number sequence. To get a different sequence (actually, to start the sequence at a different point), use the randomize procedure.

To use several sequences of rcpeatable pseudo-random number sequences, use the randseed and randnext procedures.

See also rand, randomize, randseed and randnext.

randnext procedure

SYNTAX:

randnext (var *v :* real, *seq :* 1 .. 10)

DESCRIPTION: The randnext procedure is used when you need several sequences of pseudo-random numbers, and you need to be able to exactly repeat these sequences for a number of simulations. The randnext procedure is the same as rand, except *seq* specifies one of 10 independent and repeatable sequences of pseudo-random real numbers.

The randseed procedure is used to start one of these sequences at a particular point. See also randseed, randint, rand and randnext.

procedure

SYNTAX:

randomize

DESCRIPTION: This is a procedure with no parameters that resets the sequences of pseudo-random numbers produced by rand and randint, so different executions of the same program will produce different results.

EXAMPLE: This program simulates the repeated rolling of a six sided die. Each time the program runs, a different sequence of rolls occurs (unless there is quite a coincidence or you run the program a lot of times!)

```
randomize
var roll: int
loop
    rand (l)
    put "Rolled", l
end loop
```

DETAILS: If randomize is not used, each time a program runs, rand and randint use the same pseudo-random number sequences. To get a different sequence (actually, to start the sequence at a different point), use randomize.

To use several sequences of repeatable pseudo-random number sequences, use the randseed and randnext procedures.

See also randint, rand, randseed and randnext.

randseed procedure

SYNTAX:

randseed (seed : int, *seq* : 1 .. 10)

DESCRIPTION: The randseed procedure restarts one of the sequences generated by randnext. Each restart with the same seed causes randnext to produce the same sequence for the given sequence. See also randnext, randint, rand, and randomize.

file statement

SYNTAX: A *readStatement* is:

read *ifileNumber [istatus], readItem{, readItem}*

DESCRIPTION: The read statement inputs each of the *readItems* from the specified file. These items are input directly using the *binary* format that they have on the file. In other words, the items are not in source (ASCII or EBCDIC) format. In the common case, these items have been output to the file using the write statement.

By contrast, the get and put statement use source format, which a person can read using an ordinary text editor.

EXAMPLE: This example shows how to input a complete employee record using a read statement.

```
var employeeRecord:
    record
        name: string (30)
        pay: int
        dept: 0 .. 9
    end record
var fileNo: int
open : fileNo, "payroll", read

read : fileNo, employeeRecord
```

DETAILS: The *fileNumber* must specify a file that is open with read capability (or else a program argument file that is implicitly opened).

The optional *status* is an int variable that is set to implementation dependent information about the read. If *status* is returned as zero, the read was successful. Otherwise *status* gives information about the incomplete or failed read (which is not documented here). The common case of using *status* is when reading a record or array from a file and you are not sure if the entire item exists on the file. If it does not exist, the read will fail part way through, but your program can continue and diagnose the problem by inspecting *status.*

A *readItem* is:

variableReference [: *requestedSize [: actualSize]]*

Each *readItem* specifies a variable to be read in internal form. The optional *requestedSize* is an integer value giving the number of bytes of data to be read. The *requestedSize* should be less than or equal to the size of the item's internal form in memory (else a warning message is issued). If no *requestedSize* is given then the size of the item in memory is used. The optional *actualSize* is an int variable that is set to the number of bytes actually read.

An array, record or union may be read and written as a whole.

See also the write, open, close, seek, tell, get and put statements.

teal type
(the real number type)

SYNTAX:

; real

DESCRIPTION: The real number type is used for number that have fractional parts, for example, 3.14159. Real numbers can be combined by various operators such as addition (+) and multiplication (*). Real numbers can also be combined with integers (whole numbers, such as 23, 0 and -9), in which case the result is generally a real number. An integer can always be assigned to a real variable, with implicit conversion to real.

EXAMPLE:

```
var weight, x : real
var x: real := 9.83
var tax := 0.7          % The type is implicitly real because
                        %0.7 is a real number
```

DETAILS: See also *explicitRealConstant.* The int type is used when values that are whole numbers; see int for details.

Real numbers can be converted to integers using *ceil* (ceiling), *floor,* or *round.* Real numbers can be converted to strings using *erealstr, frealstr,* and *realstr;* these conversion functions correspond exactly to the formatting that is available when using the put statement with real numbers. Strings can be converted to real numbers using *strreal.* See descriptions of these conversion functions.

The predefined functions for real numbers include *min, max, sqrt, sin, cons, arctan, sind, cosd, arcand, In* and *exp.* See the descriptions fo these functions.

Pseudo random sequences of real numbers can be generated using *rand.* See the description of this procedure.

Real numbers in Turing are implemented using 8 byte floating point representation, which provides 14 to 16 decimal digits of precision and an exponent range of at least -38 .. 38. The PC and Macintosh versions of Turing have 16 decimal digits of accuracy because they use IEEE standard floating point representation.

realstr real-to-string function

SYNTAX:

realstr (*r* : real, *width* : int): string

DESCRIPTION: The realstr function is used to convert a **real** number to a string; for example, realstr (2.5el, 4)= *"bb25"* where *b* represents a blank. The string is an approximation to *r,* padded on the left with blanks as necessary to a length of *width.*

The *width* parameter must be non-negative. If the *width* parameter is not large enough to represent the value of *r* it is implicitly increased as needed. The displayed value is rounded to the nearest decimal equivalent with this accuracy, with ties rounded to the next larger value.

The string realstr *(r, width)* is the same as the string frealstr *(r, width, defaultfw*) when *r* =0 or when le-3 < *abs (r)* < Ie6, otherwise the same as erealstr *(r, width, defaultfw, defaultew),* with the following exceptions. With *realstr,* trailing fraction zeroes are omitted and if the entire fraction is zero, the decimal point is omitted. (These omissions take place even if the exponent parts is printed.) If an exponent is printed, any plus sign and leading zeroes are omitted. Thus, whole number values are in general displayed as integers.

Defaultfw is an implementation defined number of fractional digits to be displayed; for most implementations, *defaultfw* will be 6. *Defaultew* is an implementation defined number of exponent digits to be displayed; for most implementations, *defaultew* will be 2.

The realstr function approximates the inverse of strreal, although round-off errors keep these from being exact inverses.

See also the erealstr, frealstr, strreal, intstr and strint functions.

tecord type

SYNTAX: A *recordType* is:

```
record
      id {, id }: typeSpec
    {id {, id } : typeSpec }
end record
```

DESCRIPTION: Each value of a record type consists of fields, one field for each name (id) declared inside the record. In the following example, the fields are *name, phoneNumber* and *address.*

EXAMPLE:

```
type phoneRecord :
    record
        name : string (20)
        phoneNumber : int
        address : string ( 50)
    end record

var oneEntry : phoneRecord
var phoneBook : array 1 .. 100 of phoneRecord
var/:int
oneEntry .name := "Turing, Alan"

phoneBook (/'):= oneEntry        % Assign whole record
```

DETAILS: In a record, *id*'s of fields must be distinct. However, these need not be distinct from identifiers outside the record. Records can be assigned as a whole (to records of an equivalent type), but they cannot be compared. A semicolon can optionally follow each *typeSpec.*

Any array contained in a record must have bounds that are known at compile time.

repeat (make copies of string) function

SYNTAX:

repeat (s : string, *i* : int) : string

DESCRIPTION: The repeat function returns i copies of string *s* catenated together. For example, repeat ("X", 4) is "XXXX".

EXAMPLE: This program outputs "HoHoHo".

```
var word : string := "Ho"
put repeat (word, 3)
```

DETAILS: If *i* is less than or equal to zero, the null string"" is returned. The repeat function is often used for spacing of output; for example, this statement skips 20 blanks before outputting *x*.

```
put repeat ( " ",20), x
```


result statement

SYNTAX: A *resultStatement* is:

result *expn*

DESCRIPTION: A result statement, which must appear only in a function, is used to provide the value of the function.

EXAMPLE: This function doubles its parameter.

```
function double ( x : real) : real
    result 2*x
end double
put double (5.3 ) % This outputs 10.6
```

EXAMPLE: This function finds the position of a name in a list.

```
function find ( a : array 1 ..100 of string ): int
    for i: 1 .. 100
        if a (i) = name then
            result;
        end if
    end for
end find
```

DETAILS: The execution of a result statement computes the value of the expression *(expn)* and terminates the function, returning the value as the value of the function.

The expression must be assignable to the result type of the function, for example, in *double*, *2*x* is assignable to real. (See the *ssignmentStatement* for the definition of assignable.)

A function must terminate by executing a result statement and not be reaching the end of the function.

return statement

SYNTAX: A *returnStatement is:*

```
return
```

DESCRIPTION: A return statement terminates the procedure (or main program) in which it appears. Ordinarily, a procedure (or main program) terminates by reaching its end; the return statement is used cause early termination.

EXAMPLE: This procedure takes no action if the *errorHasOccurred* flag has been set to true.

```
procedure double
    if errorHasOccurred then
        return    % Terminate this procedure
    end if
    ... handle usual case in this procedure ...
end double
```

DETAILS: A return must not appear as a statement in (the outermost level of) a module, nor can it appear in a function.

round real-to-integer function

SYNTAX:

```
round ( r : real): int
```

DESCRIPTION: The **round** function is used to convert a real number to an integer. The result is the nearest integer to r. In case of a tie, the numberically larger value is returned. For example, round (3) is 3, round (2.85) is 3 and round (-8.43) is-8.

See also the floor and ceil functions.

Screen procedure

SYNTAX:

screen (*mode* **:** int)

DESCRIPTION: The screen procedure is used to set the mode of the screen. AH the options of screen has been incorporated into the newer procedure called setscreen; it is recommended that you use setscreen instead of screen. See the setscreen procedure for details.

The modes set by screen are:

0 Exit screen mode

1 Enter screen mode (implied by using other screen mode commands such as els)

2 Echo (as characters are typed, they appear on the screen)

-2 No echo (as characters are typed, they do not appear on the screen; however, characters read by get are always echoed)

3 Turns on cursor on IBM PCs and Apple Macintoshes, but has no effect under Unix

-3 Turns off cursor on IBM PC's and Apple Macintoshes, but has no effect under Unix

4 Line input

-4 Single character input

5 On input RETURN (control-M or ASCII 13 or "\r") becomes NEWLINE (control-J or ASCII 11 or "\n")
On output NEWLFNE becomes RETURN followed by NEWLINE

-5 On input RETURN remains unchanged
On output RETURN and NEWLINE remain unchanged

Seek file statement

SYNTAX: A *seekStatement is* one of:

(a) **seek** *ifileNumber, filePosition*
(b) **seek** *:fileNumber, **

DESCRIPTION: Random access of both source (ASCII or EBCDIC) and internal form (binary) files is provided by the seek and tell statements. The seek statement repositions the specified file so that the next input/output operation will begin at the specified point *(filePosition)* in the file.

The *fileNumber* must specify a file that is open with seek capability. The *filePosition* is a non-negative integer offset in bytes from the beginning of the file; in the common case, this is a number returned by the tell statement. (The first position in the file is position zero.)

Form (b) specifies that the next operation is to begin at the position immediately following the current end of the file. A *filePosition* of zero specifies that the next operation is to start at the beginning of the file. Seeking to a position beyond the current end of the file and then writing automatically fills the intervening positions with the internal representation of zero.

EXAMPLE: This example shows how to use seek to append to the end of a file.

```
var employeeRecord:
    record
        name: string (30)
        pay: int
    end record
var fileNo: int
open : fileNo, "payroll", write, seek, mod
Seek : fileNo, *     % Seek to the end of the file
write : fileNo, employeeRecord     % This record
                                   % is added to the end of the file
```

DETAILS: See also the read, write, open, close, tell, get and put statements. Another example use of seek is given with the explanation of the tell statement.

separator

between tokens in a program

DESCRIPTION: A Turing program is made up of a sequence of *tokens* (see *tokens),* such as var, *x,:,* and int. These tokens may have *separators* between them. A separator is a comment (see *comment),* blank, tab, form feed or an end of line.

Set type

SYNTAX: A *setType* is:

set of *typeSpec*

DESCRIPTION: Each value of a set type consists of a set of elements. The *typeSpec,* which is restricted to being a subrange or an enumerated type, gives the type of these elements.

EXAMPLE: The *srmllSet* type is declared so that it can contain any and all of the values 0,1 and 2. Variable s is initialized to be the set containing 1 and 2.

```
type smallSet : set of 0 .. 2
var s : smallSet := smallSet (0,1 )
...
if 2 in s then ...
```

DETAILS: In classical mathematics, the set consisting of 0 and 1 is written as (0,1). This is written in Turing using *a set constructor* consisting of the name of the set type followed by a parenthesized list of elements, which in this example is *smalllnt* (0,1). The empty set is written, for example, as *smalllnt ().* The full set is written as *smalllnt* (all), so *smalllnt* (all) = *smalllnt* (0,1,2).

Sets can be assigned as a whole (to sets of an equivalent type). See also *equivalence* of types.

«Hr

The operators to combine two sets are union (+), intersection (*), set subtraction (-), equality (=), inequality (not=), subset (<=), strict subset (<), superset (>=), and strict superset (>). Only sets with equivalent types (equal bounds on their index types) can be combined by these operators. The operators to see if an element is or is not in a set are in and not in; for example, the test to see if 2 is in set *s* is written in the above example as: *2 Ins.*

The *indexType* of a set type must contain at least one element, for example, the range 1 .. 0 would not be allowed. See also *indexType.* The compiler may limit the *typeSpec* to at most a small range, for example, to no more than 31 possible elements.

See also *precedence* of operators for the order of applying set opeations.

setConstructor

SYNTAX: A *setConstructor* is:

setTypeId (membersOfSet)

DESCRIPTION: Each value of a set type consists of a set of elements. In classical mathematics, the set consisting of 0 and 1 is written as (0,1). This is written in Turing using a *set constructor* consisting of the name of the set type *(setTypeId)* followed by a parenthesized list of elements.

EXAMPLE: The *smallSet* type is declared so that it can contain any and all of the values 0,1 and 2. Variable s is initialized to be the set containing 1 and 2. The set (0,1) is written in this Turing example as *smalllnt* (0,1).

```
type smallSet : set of 0 .. 2
var s : smallSet := smallSet ( 0,1 )
...
if 2 in s then ...
```

DETAILS: The form of *membersOfSet* is one of:

(a)	expn Cexpn)	*List of members of set*
(b)	all	*All member of index type of set*
(c)		*Nothing, meaning the empty set*

The empty set is written, for example, as *smalllnt* (). The full set is written as *smalllnt* (all), so *smalllnt* (all) = *smalllnt* (0,1,2). See also the set type.

The syntax of *setConstructor* as given above has been simplified by ignoring the fact that set types can be exported from modules. When a set type is exported and used outside of a module, you must write the module name, a dot and then the type name. For example, the set constructor above would be written as *m.smallSet(\,2)*, where *m* is the module name.

graphics procedure

[PC, Mac and Unix only)

SYNTAX: ;; -

setscreen (s : string)

EXAMPLE: Here are example uses of the setscreen procedure. In many cases, these will appear as the first statement of the program. However, they can appear any place in a program.

```
setscreen( "graphics" )     % IBM CGA graphics
setscreenj "graphics:e16" ) % IBM EGA graphics
setscreen( "screen" )       % Jo use locate
setscreen( "nocursor" )     % Turn off cursor
setscreen( "noecho" )       % Do not echo keys
```

DESCRIPTION: The setscreen statement is used to change the mode of the screen as well as the way of doing input and output. The parameter to setscreen is a string, such as "graphics". The string contains one or more options separated by commas, such as "text,noecho".

DETAILS: Many of the options to setscreen are specific to IBM PC compatible computers and Apple Macintoshes and may have no meaning on other systems such as Unix.

Where the options to setscreen are mutually exclusive, they are listed here with the default underlined. Here are the options:

"text", "screen", "graphics" - Sets mode to the given mode; *"text"* is the default character output mode; *"screen"* is character graphics mode, allowing the locate procedure; *"graphics*" is *"pixel graphics"* mode. By default, "graphics" is CGA graphics on IBM PC compatibles. On Unix dumb terminals, *"graphics"* is meaningless. A suffix can be given, as in *"graphics:h16"*, to specify another version of pixel graphics (assuming the corresponding hardware is available). The set of *"graphics"* options is given on the next page. On Apple Macintoshes, the graphics screen is 480x275 by default. The size of the screen can also be set with a suffix in the form "gr«phtcs:150;250", which would set the graphics output window to be 150x250 pixels. Macintoshes also allow you to set the window size in *screen* mode. The default is 25 rows by 80 columns but this can be changed with *"screen:W;100"* to be 10 rows by 100 columns.

Mode	maxx+1	maxy+1	**maxcolor+1**	
"graphics"	320	200	4	(CGA)
"graphicsrmono"	320	200	4	(gray)
"graphics:hmono"	640	200	2	
"graphics:16"	320	200	16	
"graphics:h!6"	640	200	16	
"graphics:e!6"	-640	350	16	
"graphics:v2"	640	480	2	
"graphics: vl 6"	640	480	16	
"graphics:m256"	320	200	256	
"graphics:150;250"	150	250	2	(Mac only)

Warning: in Version 4.2 of Turing for IBM PC compatibles, it is not possible to change modes directly from one *"graphics"* mode to another, but rather you must change from a *"graphics"* mode to *"text"* mode and then to the next *"graphics"* mode.

"cursor", "nocursor" - Causes the cursor to be shown (or hidden). There is never a cursor showing in *"graphics"* mode. On Unix dumb terminals, the cursor cannot be hidden. There is an optional suffix for *"cursor"* that determines the shape of the cursor. In CGA graphics, the cursor is constructed out of horizontal lines numbered 0,1,2, up to 7, with 0 being the top. The suffix gives the range of lines to be used for the cursor, for example, *"cursor.5,7"* specifies a cursor consisting of lines 5 through 7. In general, this form is *"cursor:startline;endline"*, where startline and endlinc are integer literals such as 5 and 7. On the Apple Macintosh, it is possible to set the cursor size from 0-10.

"echo", "noecho" - Causes (or suppresses) echoing of characters that are typed. Echoing is commonly turned off in interactive programs to keep typed characters from being echoed at inappropriate places on the screen.

"line", "char" - Causes a whole line *(line)* to be read at once or else a single character *(char)*. [PC and Unix only]

"retmap". "noretmap" - Causes RETURN on input to be mapped to NEWLINE, and NEWLINE on output to be mapped to RETURN followed by NEWLINE. Using *"noretmap"* stops this mapping. [PC and Unix only]

DETAILS: The setscreen procedure supports all the features of the older screen procedure; although the screen procedure is still supported, it is recommended that setscreen be used instead.

See also drawdot, drawline, drawoval, drawarc, whatdotcolor, color, colorback, takepic and drawpic.

sign function

SYNTAX:

sign (*r* : real): -1 .. 1

DESCRIPTION: The sign function is used to determine whether a number is positive, zero or negative. It returns 1 if r > 0,0 if *r* = 0, and -1 if r < 0. For example, sign (5) is 1 and sign (-23) is -1.

EXAMPLE: This program reads in numbers and determines if they are positive, zero or negative:

```
var x : real
get x
case sign (x) of
    label  1 : put "Positive"
    label  0 : put "Zero"
    label -1 : put "Negative"
end case
```

Sin sine function (radians)

SYNTAX:

sin (*r :* real): real

DESCRIPTION: The sin function is used to find the sine of an angle given in radians. For example, sin (0) is 0.

EXAMPLE: This program prints out the sine of pi/6,2*pi/6,3*pi/6, up to 12*pi/6 radians

```
constp/:=3.14159
for/:1 ..12
    const angle := i *pi / 6
    put "Sin of", angle, " is ", sin (angle)
end for
```

DETAILS: See also the sind function which finds the sine of an angle given in degrees. (2 * pi radians are the same as 360 degrees.)

sine function (degrees)

SYNTAX:

sind (r : real): real

DESCRIPTION: The sind function is used to find the sine of an angle given in degrees. For example, sind (0) is 0.

EXAMPLE: This program prints out the sine of 30,60,90, up to 360 degrees.

```
for /: 1 .. 12
    const angle := / * 30
    put "Sin of", angle, " is ", sind ( angle)
end for
```

DETAILS: See also the sin function which finds the sine of an angle given in radians. (2 * pi radians are the same as 360 degrees.)

Sizepic graphics function

[PC only.

SYNTAX:

sizepic (*x1,* *y1, x2,* *y2 : mi):* int

DESCRIPTION: The sizepic function is used to determine the size buffer needed to record a picture from the screen (see description of takepic). This gives the minimum number of elements of the int array used by takepic . The buffer is used by drawpic to make copies of the picture on the screen.

EXAMPLE: This program outputs the size of array needed to hold a picture with left bottom comer at x=10, y=20 and right top comer at x=50, y=60.

```
setscreen ("graphics")

put "The size of the array needs to be",
    sizepic (10,20,50,60)
```

DETAILS: See takepic for an example of the use of sizepic and for further information about buffers for drawing pictures.
The screen should be in a *"graphics"* mode; see the setscreen procedure for details. If the screen is not in a *"graphks"* mode, it will automatically be set to *"graphics"* mode.

See also drawpic.

See also setscreen, maxx, maxy, drawdot, drawline, drawbox, and drawoval.

Skip (used in get statement)

SYNTAX:

```
skip
```

DESCRIPTION: Using skip as an input item in a get statement causes the current input to be ignored until a non-whitespace token is encountered. Whitespace includes all blanks, tabs, form feeds and newlines.

EXAMPLE: The skip input item is most frequently used to skip past an end-of-line (newline) character to see if the end of the input file has been reached. Thus, it is most frequently seen paired with *eofin* a loop body as follows

```
loop
    get skip
    exit when eof
    get...
    ...
end loop
```

DETAILS: The skip bypasses all whitespacc characters including any trailing newlines and blank lines. By skipping these characters, a true end-of-file condition can be detected. Otherwise, the end-of-file is obscured by the exisitng whitespace until a following get which will fail since there is no trailing data.

EXAMPLE: Another use of skip is to correctly identify the start of a long string (usually to be read in *line* or *counted* mode) by skipping the whitespace and trailing newline as follows

```
var i: int
var line: string
loop
    get i, skip, line:'
    ...
end loop
```

DETAILS: The first item in the get statement reads an integer by skipping all whitespace and reading digits until whitespace is encountered. The input stream is then left with the whitespace as the next input character. The skip then skips past the whitespace effectively beginning the next input at the next non-whitespace character. This truncates leading blanks and has another, potentially more important, effect. If the integer is the last data on a line and the string is on a following line, the skip is necessary to avoid setting *line* to a null string value.

See also get statement; and **loop** statement.

I

Skip (used in **put** statement)

SYNTAX:

skip

DESCRIPTION: Using skip as an output item in a put statement causes the current output line to be ended and a new line to be started.

EXAMPLE: This example, To *be* is output on one line and *Or not to be* on the next.

```
put "To be", skip, "Or not to be"
```

DETAILS: Using skip is equivalent to outputting the newline character "\n".

SOUnd statement

[PC only]

SYNTAX:

sound (*frequency, duration* : int)

DESCRIPTION: The sound statement is used to cause the computer to sound a note of a given frequency for a given time. The frequency is in cycles per second (Hertz). The time duration is in milliseconds. For example, middle A on a piano is 440 Hertz, so sound(440,1000) plays middle A for one second.

EXAMPLE: This program sounds the frequencies 100,200 up to 1000 each for half a second.

```
for /: 1 ..10
    put /
    sound(100*/, 500)   % Sound note (or 1/2 second
end for
```

DETAILS: See also the play statement, which plays notes based on musical notation; for example, play("8C") plays an eighth note of middle C. See also the delay, clock, sysclock, wallclock, time and date statements.

On IBM PC compatibles, the hardware resolution of duration is in units of 55 milliseconds. For example, sound(440,500) will delay the program by about half a second, but may be off by as much as 55 milliseconds.

SCJft square root function

SYNTAX:

sqrt (*r* : real): real

DESCRIPTION: The sqrt function is used to find the square root of a number. For example, sqrt (4) is 2.

EXAMPLE: This program prints out the square roots of 1,2,3,... up to 100.

```
for /: 1 .. 100
    put "Square root of",/," is", sqrt (/)
end for
```

DETAILS: It is illegal to try to take the square root of a negative number. The result of sqrt is always positive or zero.

The opposite of a square root is the square; for example, the square of *x* is written is x**2.

standard type

SYNTAX: A *standardType* is one of:

(a) **int**
(b) **real**
(c) **string** [**(***maximumLength***)]**
<d) **boolean**

DESCRIPTION: The standard types can be used throughout a program. They should not be included in an import list. See also int, real, string and **boolean.**

statement

SYNTAX: A *statement* is one of:

(a) *assignmentStatement* % *variableReference := expn*
(b) *openStatement* **% open ...**
(c) *closeStatement* % *close ...*
(d) *putStatement* % *put ...*
(e) *getStatement* % *get ...*
(f) *readStatement* % *read ...*
(9) *writeStatement* % *write...*
(h) *seekStatement* % *seek...*
(i) *tellStatement* % *tell...*
(j) *forStatement* % *for ... end for*
(k) *loopStatement* % *loop ... end loop*
(l) **exit [when** *trueFalseExpn]*
(m) *ifStatement* % *if... end if*
(n) *caseStatement* % *case ...endcase*
(o) **assert** *trueFalseExpn*
(P) **begin**
statementsAndDeclarations
end
(q) *prOCedureCall* % *procedureId [(parameters)]*
(0 **return**
(s) **result** *expn*
(t) new *collectionId, pointerVariableReference*
(u) **free** *collectionId, pointerVariableReference*
(v) tag *unionVariableReference, expn*

DESCRIPTION: A *statement* (or *command)* causes a particular action, for example, the *putStatement*

```
put "Hello"
```

outputs *Hello.* See the descriptions of the individual statements for explanations of their actions. Each statement can optionally by followed by a semicolon (;).

EXAMPLES:

```
width   :=24                % Assignment statement
put "Hello world"           % Put statement
exit when i = 100           % Exit statement
assert width < 320          % Assen statement
```

DETAILS: You can use a result statement only in a function. You can use a return statement only to terminate a procedure or the main program (but not to terminate the initialization of a module). See also result and return.

There are a number of predefined procedures, such as *drawline*, which are not listed as statements above; uses of these are considered to be procedure calls, which is one form of statement.

statementsAndDedarations

SYNTAX: *StatementsAndDedarations* are:

{*statementOrDeclaration* }

DESCRIPTION: *StatementsAndDedarations* are a list of statements and declarations. For example, a Turing program consists of a list of statements and declarations. The body of a procedure is a list of statements and declarations.

Each *statementOrDeclaration* is one of:

(a) *statement*
(b) *declaration*

See also *statement* and *declaration.*

EXAMPLES: This list of statements and declarations is a Turing program that outputs *Hello Rrank.*

```
var name : string
name := "Frank"
put "Hello", name
```

String comparison

SYNTAX: A *stringComparison* is one of:

(a) *stringExpn* = *stringExpn*
(b> *stringExpn* **not=** *stringExpn*
(c> *stringExpn* > *stringExpn*
(d> *stringExpn* < *stringExpn*
(e) *stringExpn* >= *stringExpn*
(0 *stringExpn* <= *stringExpn*

DESCRIPTION: Strings *(stringExpns)* can be compared for equality (= and not=) and for ordering (>, <, >= and <=).

EXAMPLES:

```
var name: string := "Nancy"
var HcenceNumber : string (6 )
HcenceNumber :="175AJN"
```

DETAILS: Two strings are considered to be equal (=) if they have the same length and are made up, character by character, of the same characters; otherwise they are considered to be unequal (not=).

Ordering among strings is essentially alphabetic order. String *S* is considered to come before string T, that is S < *T,* if the two are identical up to a certain position and after that position, either the next character of S comes before the next character of *T* or else there are no more characters in S and T contains more characters.

S >T (S comes after T) means the same thing as 7 <S. S >=T means the same thing as *S>TorS =T. S* <=T means the same thing as S<TorS=T.

The ordering among individual characters is given by ASCII, which specifies among other things that letter capital L comes alphabetically before capital letter *M* and similarly for small (lower case) letters.

On IBM mainframe computers, the EBCDIC specification of characters may be used instead of ASCII.

string type

SYNTAX: A *stringType* is:

string [**(** *maximumLength*)]

DESCRIPTION: Each variable whose type is a *stringType* can contain a sequence (a string) of characters. The length of this sequence must not exceed the *stringType's* maximum length.

EXAMPLES:

```
var name: string
name := "Nancy"
var HcenceNumber : string (6)
HcenceNumber :="175AJN"
```

DETAILS: Strings can be assigned and they can be compared for both equality and for ordering; see also *string comparison* and *assignment statement.*

Strings can be catenated (joined together) using the + operator and separated into substrings; see *catenation* and *substring.* String functions are provided to find the length of a string, to find were one string appears inside another, and to make repeated copies of a string all joined together; see *length, index,* and *repeat.*
See *explicitStringConstants* for exact rules for writing string values such as "Nancy".

A string type written without a maximum length is limited to holding at most 255 characters.

The *maximumLength* of a string, if given as a part of the type, must be known at compile time, and must be at least 1 and at most 255. The maximum length of a string is given by *upper,* for example, *uppeilicenceNumber)* is 6; see also *upper.*

In the declaration of a string that is a var formal parameter of a procedure or function, the *maximumLength* can be written as a star (*), in which case the maximum length is taken to be that of the corresponding actual parameter, as in:

```
procedure deblank (var s : string(*)).
```

The star can also be used when the parameter is an array of strings.

Strint string-to-integer function

SYNTAX:

```
strint (s : string ): int
```

DESCRIPTION: The strint function is used to convert a string to an integer. The integer is equivalent to string s. For example, strint("-47") = -47.

String s must consist of a possibly null sequence of blanks, then an optional plus or minus sign, and finally a sequence of one or more digits.

The intstr function is the inverse of strint, so for any integer *i*, strint (intstr (i)) = *i*.

See also the chr, ord and intstr functions.

Strreal string-to-real function

SYNTAX:

```
strreal ( s: string ): real
```

DESCRIPTION: The strreal function is used to convert a string to a real number; for example, strreal ("2.5el") will produce an approximation to the number 25.0.

String s must consist of a possibly null sequence of blanks, then an optional plus or minus sign and finally an explicit unsigned real or integer constant.

The realstr, erealstr and frealstr functions approximate the inverse of strreal, although round-off errors keep these from being exact inverses.

See also the realstr, erealstr, frealstr, intstr and strint functions.

*

subrangeType

SYNTAX: *A subrangeType* is:

```
expn .. expn
```

DESCRIPTION: A subrange type defines a set of values, for example, the subrange 1.. 4 consists of 1,2,3 and 4.

EXAMPLES:

```
var i : 1 .. 10                 % i can be 1,2... up to 10
type xRange : 0 .. 319          % Define integer subrange
var pixels : array xRange of int
                                % Array elements are
                                % numbered 0, 1,... 319
for k : xRange                  % k ranges from 0 to 319
    pixels ( k ) := 0
end for
```

DETAILS: A subrange must contain at least one element; in other words, the second expression *(expn)* must be at least as large as the second expression.

The lower bound of a subrange must be known at compile time. The upper bound is allowed to be a run-time value only in one situation and that is when the it gives the upper bound of an array being declared in a variable declaration, in other words when declaring a *dynamic* array.

In the most common case, subranges are a subset of the integers, as in 1.. 10. You can also have subranges of enumerated types.

Substring of another string

SYNTAX: A *substring* is one of:

```
(a)  stringReference  (leftPosition .. rightPosition )
(b)  stringReference  ( charPosition )
```

DESCRIPTION: A substring selects a part of another string. In form (a) the substring starts at the left position and runs to the right position. In form (b), the substring is only a single character.

EXAMPLES:

```
var word: string := "bring"
put Word (2 .. 4)           % Outputs rin
put word (3)                % outputs i
put Word (2 .. *)           % Outputs ring; the star (*) means
                            % the end of the string.
put WOrd (* - 2 .. * - 1 )  % Outputs in
```

DETAILS: The leftmost possible position in a string is numbered 1. The last position in a string can be written as a star (*); for example, *word* (2 .. *) is equivalent to *word* (2 .. *length(word)*).

Each of *leftPosition*, *rightPosition*, and *charPosition* must have one of these forms:

(a) *expn*
(b)*
(c) * - *expn*

The exact rules for the allowed values of *leftPosition* and *rightPosition* are:

(1) *leftPosition* must be at least 1,
(2) *rightPosition* must be at most *length(stringReference)*, and
(3) the length of the selected substring must zero or more.

This specifically allows null substrings such as *word(\,G)* in which *rightPosition* is 0 and *word(6$)* in which *leftPosition* is one more that *length(stringReference)*.

Note that substrings are not assignable. The statement var s: string s(3) := "a" is illegal in Turing.

SYNTAX:

succ (*expn*)

DESCRIPTION: The succ function accepts an integer or an enumerated value and returns the integer plus one, or the next value in the enumeration. For example, succ (7) is 8.

EXAMPLE: This part of a Turing program fills up array *a* with the enumerated values *green, yellow, red, green, yellow, red,* etc.

```
type colors : enum (green, yellow, red )
var a: array 1 .. 100 of colors
var c : colors := colors .green
for i: 1..100
    a(i) := c
    if c = colors. red then
        c:= colors. green
    else
        c:= succ (c)
    end if
end for
```

DETAILS: It is illegal to apply succ to the last value of an enumeration. See also the pred function.

Irafll

SYNTAX:

sysclock (var *c* **: int)**

DESCRIPTION: The sysclock statement is used on a multitasking system such as Unix to determine the amount of time that has been used by this program (process). Variable *c* is assigned the number of central processor milliseconds assigned to this program. This is of little use on a personal computer.

EXAMPLE: On a Unix system, this program tells you how much time it has used.

```
var timeilsed : int
sysclock (timeUsed)                    ,
put "This program has used ", timeUsed,
        " milliseconds of CPU time"
```

DETAILS: See also the delay, time, clock, wallclock and date statements.

System statement

nix only

SYNTAX:

system (*command :* string, var *ret* : int)

DESCRIPTION: The system statement is used to execute the shell (operating system) *command*, as if it were typed at the terminal. The return code is in *ret*. A return code of 0 (zero) means no detected errors. A return code of 127 means the command processor could not be accessed. A return code of 126 means the command processor did not have room to run on the PC.

EXAMPLE: This program creates a directory listing when run under DOS on an IBM PC compatible computer; the same program will run under Unix by changing *"dir"* to *"Is"*.

```
var success : int
system ( "dir", success )
if success not= 0 then
    if success = 127 then
        put "Sorry, can't find 'dir'"
    elsif success = 126 then
        put "Sorry, no room to run 'dir'"
    else
        put "Sorry, 'dir' did not work"
    end if
end if
```

DETAILS: See also the nargs, fetcharg and getenv functions.

tag statement

SYNTAX: A *tagStatement* is:

tag *unionVariableReference, expn*

DESCRIPTION: A tag statement is a special purpose assignment that is used for changing the tag of a union variable.

EXAMPLE: In this example, the tag field of union variable *v* is set to be *passenger*, thereby activating the *passenger* field of *v*.

```
type vehicleInfo :
    union kind : passenger .. recreational
        label passenger :
            cylinders : 1..16
        label farm :
            farmClass -.string (10)
        label :          % No fields for "otherwise" clause
    end union
var v : vehicleInfo

tag V , passenger        % Activate passenger part
```

DETAILS: A tag statement is the only way to modify the tag field of a union variable (other than by assigning an entire union value to the union variable).

It is not allowed to access a particular set of fields of a union unless the tag is set to match the corresponding label value. See also union types.

takepic graphics procedure

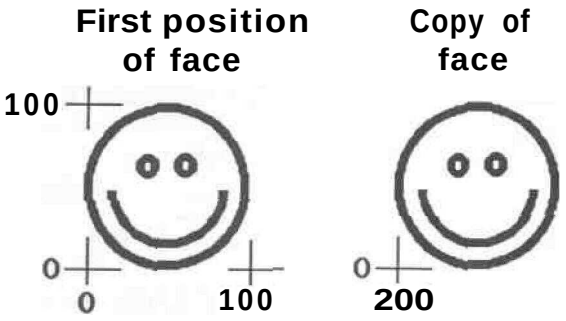
tPCo$_{inly]}$

SYNTAX:

takepic (*x1,* *y1, x2,* ***y2 :*** **int,**
var *buffer:* ***array*** **1 .. * of int)**

DESCRIPTION: The takepic procedure is used to record the pixel values in a rectangle, with left bottom and right right corners of *(xl, yl)* and *(x2,* y2), in the buffer array. This requires a sufficiently large buffer (see sizepic). The drawpic procedure is used to make copies of the recorded rectangle on the screen.

EXAMPLE: After drawing a happy face, this program copies the face to a new location.




```
setscreen ("graphics")
... draw happy face in the box (0,0) to (100,100)...
% Create buffer big enough to hold happy face
var face: array 1 .. sizepic(0,0,100,100) of int
% Copy picture into the buffer, which is the face array
takepic(0,0,100,100, face)
% Redraw the picture with its left bottom at (200,0)
drawpic(200,0, face,0)
```

DETAILS: The integer values that takepic places in the buffer can be read or written (using the read and write statements). Unfortunately, assignment (by :=) and put of the individual integer values in the buffer will fail in the case in which a value happens to be the pattern used to represent the uninitialized value (the largest negative number the hardware can represent).

The screen should be in a *"graphics"* mode; see the setscreen procedure for details. If the screen is not in a *"graphics"* mode, it will automatically be set to *"graphics"* mode.

See also sizepic and drawpic.

See also setscreen, maxx, maxy, drawdot, drawline, drawbox, and drawoval.

tell file statement

SYNTAX: An *tellStatement is:*

tell *:fileNumber, filePositionVar*

DESCRIPTION: The tell statement sets *filePosition Vur,* whose type must be int, to the current offset in bytes from the beginning of the specified file. The *fileN'umber* must specify a file that is open with seek capability (or else a program argument file that is implicitly opened). The tell statement is useful for recording the file position of a certain piece of data for later access using seek.

EXAMPLE: This example shows how to use tell to record the location in a file of a record. This location is later used by seek to allow the record to be read.

```
var employeeRecord:
    record
        name: string (30)
        pay: int
        dept: 0 .. 9
    end record
var fileNo: int
var location: int
open : fileNo, "payroll", write, seek

tell : fileNo, location          % Make note of this location
write : fileNo, employeeRecord   % write record
                                 %at this location

seek : fileNo, location          % GO back to location
read : fileNo, employeeRecord    % Read the record
                                 % that was previously written
```

DETAILS: See also the read, write, open, close, seek, get and put statements.

time (hours, minutes, seconds) procedure

[PC, Mac and Unix only]

SYNTAX:

time (var *t* : string)

DESCRIPTION: The time statement is used to determine the current rime of day. Variable *t* is assigned a string in the format *"hh:tnm:ss"*. For example, if the time is two minutes and 47 seconds after nine A.M., *t* will be set to *"09:02:47"*. Twenty-four hour time is used; for example, eleven thirty P.M. gives the string "23:30:00".

EXAMPLE: This program greets you and tells you the time of day.

```
var timeOfDay : string
time (timeOfDay)
put "Greetings!! The time is ", timeOfDay
```

DETAILS: See also the delay, clock, sysclock, wallclock and date statements.

Be warned that on some computers such as IBM PC compatibles or Apple Macintoshes, the rime may not be set correctly in the operating system; in that case, the time procedure will give incorrect results.

token

DESCRIPTION: A *token* is essentially a word, a number or a special symbol such as :=. In a Turing program there are four kinds of tokens: keywords such as get, identifiers such as *incomeTax*, operators and special symbols, such as + and :=, and explicit constants, such as 1.5 and "Hello". Some keywords, such as index, are reserved and cannot be used in programs to name variables, procedures, etc.

A get statement, such as

```
get incomeTax
```

uses *token-oriented* input. This means that white space (blanks, tabs, etc) are skipped before reading the input item. See the get statement for details.

EXAMPLES: In this example, the tokens are var, *x*, :, real, *x*, := and 9.84.

```
var x : real
x := 9.84
```


true

boolean value (as opposed to false)

SYNTAX:

```
true
```

DESCRIPTION: A boolean (true/false) variable can be either true or false (see boolean type).

EXAMPLE:

```
var passed : boolean := true
var mark: int
for : 1 ..10
    get mark
    passed := passed and mark >= 60
end for
if passed = true then
    put "You passed all ten subjects"
end if
```

DETAILS: The line if *passed=true* then can be simplified to if *passed* then with no change to the meaning of the program.

type declaration

SYNTAX: A *typeDeclaration* is:

type *id : typeSpec*

DESCRIPTION: A type declaration gives a name to a type. This name can be used in place of the type.

EXAMPLES:

```
type nameType : string (30)
type range : 0 .. 150
type entry :
    record
        name: nameType
        age: int
    end record
```

DETAILS:

The keyword pervasive can be inserted just after type. When this is done, the type is visible inside all subconstructs of the type scope. Without pervasive the type is not visible inside modules unless explicitly imported. Pervasive types need not be imported. You can abbreviate pervasive as a star (*).

typeSpec

type specification

SYNTAX: A *typeSpec* (type specification) is one of:

```
(a) int
    real
(c) boolean
(d) stringType          % Example: string (20)
(e) subrangeType        % Example: 1 .. 150
(f) enumeratedType      % Example: enum (red, green, blue)
<g) arrayType           % Example: array 1.. 150 of real
(h) setType             % Example: set of 1 .. 10
(0 recordType           % Example: record ... end record
G) unionType            % Example: union ... end union
(k) pointerType         % Example: pointer to collectionVar
(i) namedType           % Example: colorRange
```

DESCRIPTION: A type specification determines the allowed values for a variable or constant. For example, if variable *x* is an integer (its *typeSpec* is int), the possible values for *x* are numbers such as -15,0,3 and 348207. If x is a real number (its *typeSpec* is real), then its possible values include 7.8, -35.0, and 15el2. If x is a boolean, its possible values are true and false. If A: is a string, its possible values include "Hello" and "Good-bye".

EXAMPLES:

```
var numberOfSides   int
var x, y : real
type range : 0 .. 150     % The typeSpec here is 0.. 150
type entry :              % Here is a record typeSpec
    record
        name: string (25)
        age: range
    end record
```

union type

SYNTAX: A *unionType* is:

```
union [ id ] : indexType
    label labelExpn { , labelExpn}:
                {id {, id } : typeSpec}
    {label labelExpn { , labelExpn}:
                {id {, id } : typeSpec}}
    [ label:    {id {, id } : typeSpec} ]
end union
```

DESCRIPTION: A union type (also called a variant record) is like a record in which there is a run-time choice among sets of accessible fields. This choice is made by the tag statement, which deletes the current set of fields and activates a new set.

EXAMPLE: This union type keeps track of various information about a vehicle, depending on the kind of vehicle.

```
const passenger := 0
const farm := 1
const recreational := 2

type vehicleInfo :
    union kind : passenger .. recreational
        label passenger :
                cylinders : 1..16
        label farm :
                farmClass :string ( 1 0 )
        label :     % No fields for "otherwise" clause
    end union
var v : vehicleRecord

tag v, passenger        % Activate passenger part
v.cylinders := 6
```

DETAILS: The optional identifier following the keyword union is the name of the *tag* of the union type. If the identifier is omitted, the tag is still

considered to exist, although its value cannot be accessed. The tag must be of an index type, for example 1..7. You should limit the range of this index type, as the compiler may have a limit (at least 255) on the maximum range it can handle.

Each *labelExpn* must be known at compile time and must lie within the range of the tag's type. The fields, including the tag, of a union value are referenced using the dot operator, as in *v.cylinders* and these can be used as variables or constants. A field can be accessed only when the tag matches one of the label expressions corresponding to the field. The tag can be changed by the tag statement and but it cannot be assigned to, passed to a var parameter or bound to using var.

In a union, *id's* of fields, including the tag, must be distinct. However, these need not be distinct from identifiers outside the union. Unions can be assigned as a whole (to unions of an equivalent type), but they cannot be compared. A semicolon can optionally follow each *typeSpec.*

Any array contained in a union must have bounds that are known at compile time.

Upper bound of an array or string

SYNTAX:

upper (*arrayReference [, dimension]*): int

DESCRIPTION: The upper attribute is used to find the upper bound of an array or string. (See lower for finding the lower bound.)

EXAMPLE: In a procedure, see if the bound of array parameter *a* is large enough so it can be subscripted by *i*. If so set *a(i)* to zero.

```
procedure test (var a : array 1 .. * of real)
    if i <= upper ( a) then
        a (i):=0.0
    end if
end test
```

DETAILS: In a similar way, if s is a string, its upper bound (not length!) is given by upper (s). If an array has more than one dimension, as in var *b* : array 1..10,1 ..60 of int, you must specify the dimension, for example, upper *(b,* 2) returns 60.

var declaration

SYNTAX: A *variabkDedaration* is one of:

(a) var *id { , id } [: typeSpec] [:= initializingValue]*
(b) *collectionDeclaration*

DESCRIPTION: A variable declaration creates a new variable (or variables). Only form (a) will be explained here; see *collectionDeclaration* for explanation of form (b). The *typeSpec* of form (a) can be omitted only if the initializing value is present.

EXAMPLES:

```
vary, k  ". int := 1          %/ and k are assigned value 1
Var/ := "Sample"              % The type of t is string
var v : array 1 .. 3 of string (6) :=
        init("George","Fred","Alice")
```

DETAILS: The initializing value, if present, must be an expression or else a list of items separated by commas inside init (…). The syntax of *initializingValue* is one of:

(a) *expn*
(b) init (*initializingValue* {, *initializingValue}*)

Each init (…) corresponds to an array, record or union value that is being initialized; these must be nested for initialization of nested types.

If the *typeSpec* is omitted, the variable's type is taken to be the (root) type of the initializing expression, for example, int or string. The *typeSpec* cannot be omitted for dynamic arrays or when the initializing value is of the form init (...). The values inside init (…) must be known at compile time.

See also collection, bind, procedure and function declarations, parameter declarations, and import lists for other uses of the keyword var.

variableReference
use of a variable

SYNTAX: *A variableReference* is:

variableId { componentSelector }

DESCRIPTION: In a Turing program, a variable is declared and given a name *(variableId)* and then used. Each use is called a *variable reference.*

If the variable is an array, collection, record or union, its parts *(components)* can be selected using subscripts and field names (using *componentSelectors).* The form of a *componentSelector* is one of:

(a) (*expn (, expn])*
(b) *.fieldld*

Form (a) is used for subscripting (indexing) arrays and collections. The number of array subscripts must be the same as in the array's declaration. A collection has exactly one subscript, which must be a pointer to the collection. Form (b) is used for selecting a field of a record or union.

EXAMPLES: Following the declarations of *k, a* and *r,* each of *k, a (k)* and *r.name* are variable references.

```
var k : int
var a : array 1 .. 100 of real
var r :
    record
        name : string (20)
        phone : string (8 )
    end record
...
k:=5
a (/c):= 3.14159
r. name := "Steve Cook"
```

DETAILS: A variable reference can contain more than one component selector, for example, when the variable is an array of records; for an example, see the record type. See also *constantReference* and var declaration.

Wallclock seconds since 1970 [PC, Mac and Unix only]
procedure

SYNTAX:

```
wallclock (var c : int)
```

DESCRIPTION: The wallclock statement is used to determine the time in seconds since 00:00:00 GMT (Greenwich Mean Time) January 1,1970.

EXAMPLE: This program tells you how many seconds since 1970.

```
var seconds : string
wallclock (seconds)
put "The number of seconds since 1970 is ",
                seconds
```

DETAILS: See also the delay, time, clock, sysclock and date statements.

Be warned that on some computers such as IBM PC compatibles or Apple Macintoshes, the time may not be set correctly in the operating system; in that case, the wallclock procedure will give incorrect results. Also, on IBM PC compatibles, the call is dependent on having the time zone TZ variable correctly set. On an IBM PC, the default time zone is set to PST (6 hours from GMT).

On the Apple Macintosh, wallclock procedure returns the number of seconds since 00:00:00 local time Jan. 1,1970.

Whatcolor text color graphics function

[PC only]

SYNTAX:

```
whatcolor : int
```

DESCRIPTION: The whatcolor function is used to determine the current (foreground) color, ie., the color used for characters that are output using put. The alternate spelling is whatcolour.

EXAMPLE: This program outputs the currently active color number. The color of the message is that is this color.

```
setscreen ("graphics")

put "This writing is in color number", whatcolor
```

DETAILS: The screen should be in a *"screen"* or *"graphics"* mode; see setscreen for details.

See also the color procedure, which is used to set the color. See also colorback and whatcolorback.

whatcolorback color of background graphics function

[PC only]

SYNTAX:

```
whatcolorback: int
```

DESCRIPTION: The whatcolorback function is used to determine the current background color. The alternate spelling is whatcolourback .

EXAMPLE: This program outputs the currently active backgournd color number. The background color of the message is determined by this number.

```
setscreen ("screen")

put The background of this writing"
put "is in color number", whatcolorback
```

DETAILS: The screen should be in a *"screen"* or *"graphics"* mode. Beware that the meaning of background color is different in these two modes; see colorback for details.

See also color and whatcolor.

whatdotcolor graphics function

mly]

SYNTAX:

```
whatdotcolor ( x, y: int) : int
```

DESCRIPTION: The whatdotcolor function is used to determine the color number of the specified pixel. The alternate spelling is whatdotcolour

EXAMPLE: This program draws a line which bounces off the edges of the screen and makes a beep when it finds a pixel that has already been colored.

```
setscreen ("graphics")
var x, y: int := 0
var dx, dx: int := 1
loop
    if whatdotcolor (x, y) not= 0 then
f       sound (400, 50)
    end if
    drawdot (x,y, 1)
    x := x + dx
    y:=y+dy
    if x = 0 or x = maxx then
        dx := -dx
    end if
    if y = 0 or y = maxy then
        dy := -dy
    end if
end loop
```

DETAILS: The screen should be in a *"graphics"* mode; if not, it will automatically be set to *"graphics"* mode. See setscreen for details.

See also drawdot, which is used for setting the color of a pixel. See also maxx and maxy which are used to determine the number of pixels on the screen. See also sound which causes the computer to make a sound.

whatpalette graphics function [PC only]

SYNTAX:

```
whatpalette : int
```

DESCRIPTION: The whatpalette function is used to determine the current palette number.

EXAMPLE: This program outputs the current palette number.

```
setscreen ("graphics")

put "The current palette number is ", whatpalette
```

DETAILS: The whatpalette function is meaningful only in a *"graphics"* mode.

See also the setscreen procedure for a description of the graphics modes. See also the palette statement, which is used to set the palette number.

whattextchar graphics function [PC only]

SYNTAX:

whattextchar : string (1)

DESCRIPTION: The whattextchar function is used to determine the character on the screen at the current location.

EXAMPLE: This program outputs a message and then changes the foreground color (the color of the letters) of the message to color number 1 and the background color (surrounding each letter) to color number 7. The actual message (each letter) is not changed.

```
setscreen ("screen")
const message := "Happy New Year!!"
put message

for column : 1 .. length (message)
    locate (1, column)
    COlor ( 1 )                 % Color of letter
    COlorback ( 7 )             % Color around letter
    put Whattextchar ..         % Use same letter
end for
```

DETAILS: The whattextchar function is meaningful only in *"screen"* mode. In *"graphics"* mode, the concept of *text* on the screen is replaced by the concept of *pixels* on the screen.

See also setscreen which describes modes. See also color, whattextcolor, anad whattextcolorback.

whattextcolor graphics function [PC only]

SYNTAX:

whattextcolor: int

DESCRIPTION: The whattextcolor function is used to determine the color of the character on the screen at the current location. The alternate spelling is whattextcolour.

EXAMPLE: This program prints out a message with each letter in a random color and then prints the same message on the next line in exactly the reverse colors.

```
setscreen ("screen")
var clr: int
const message := "Happy New Year!!"
for column : 1 .. length (message)
    randint ( clr, 1 , maxcolor)
    color (clr)
    locate (1, column)
    put  message (i)..
end for
locate (1,1)
for column : 1 .. length (message)
    locate (1 , length (message) + 1 - c)
    clr := whattextcolor
    color  (clr)
    locate (2, column)
    put  message (i)..
end for
```

DETAILS: The whattextcolor function is meaningful only in "screen" mode. In *"graphics"* mode, the concept of *text* on the screen is replaced by the concept of *pixels* on the screen.

See also setscreen which describes modes. See also color, whattextchar, and whattextcolorback.

whattextcolorback graphics function

[PC only]

SYNTAX:

whattextcolorback: int

DESCRIPTION: The whattextcolorback function is used to determine the background color of the character on the screen at the current location. The alternate spelling is whattextcolourback.

EXAMPLE: This program prints out a message with each letter in a random background color and then prints the same message on the next line in exactly the reverse background colors.

```
setscreen ("screen")
var dr: int
const message := "Happy New Year!!"
for column : 1 .. length (message)
    randint ( clr, 1 , maxcolor)
    colorback( clr)
    locate (1, column)
    put  message  (i)..
end for
locate (1,1)
for column : 1 .. length (message)
    locate (1 , length (message)  + 1 - c)
    clr := whattextcolorback
    colorback (clr)
    locate (2, column)
    put message (l)..
end for
```

DETAILS: The whattextcolorback function is meaningful only in *"screen"* mode. In *"graphics "* mode, the concept of *text* on the screen is replaced by the concept of *pixels* on the screen.

See also **setscreen** which describes modes. See alsocolor, whattextchar, and whattextcolor.

Write file statement

SYNTAX: A *writeStatement* is:

write *: fileNumber [-.status], writeItem{, writeItem}*

DESCRIPTION: The write statement outputs each of the *writeItems* to the specified file. These items are output directly using the *binary* format that they have in the computer. In other words, the items are not in source (ASCII or EBCDIC) format. In the common case, these items will later be input from the file using the read statement.

By contrast, the get and put statement use source format, which a person can read using an ordinary text editor.

EXAMPLE: This example shows how to output a complete employee record using a write statement.

```
var employeeRecord:
    record
        name: string (30)
        pay: int
        dept: 0 .. 9
    end record
var fileNo: int
open : fileNo, "payroll", write

write : fileNo, employeeRecord
```

DETAILS: An array, record or union may be read and written as a whole. *The fileNumber* must specify a file that is open with write capability (or else a program argument file that is implicitly opened).

The optional *status* is an int variable that is set to implementation dependent information about the write. If *status* is returned as zero, the write was successful. Otherwise status gives information about the incomplete or failed write (which is not documented here). The common case of using status is when writing a record or array to a file and you are not sure if there is enough room on the disk to hold the item. If there is not, the write will fail part way through, but your program can continue and diagnose the problem by inspecting *status*.

A *writeItem* is:

reference [-.requestedSize [wctualSize]]

Each *writeItem* is a variable, or a named non-compile-time constant, to be written in internal form. The optional *requestedSize* is an integer expression giving the number of bytes of data to be written. The *requestedSize* should be less than or equal to the size of the item's internal form in memory (else a warning message is issued). If no *requestedSize* is given then the size of the item in memory is used. The optional *actualSize* is set to the number of bytes actually written.

See also the write, open, close, seek, tell, get and put statements.

Appendices

Appendix A

A List of Predefined Functions and Procedures

abs	arctan	arctand	ceil	chr
clock	close	els	color	colorback
colour	colourback	cos	cosd	date
delay	drawarc	drawbox	drawdot	drawfill
drawline	drawoval	drawpic	eof	erealstr
exp	fetcharg	floor	frealstr	getch
getenv	getpid	hasch	index	intreal
intstr	length	In	locate	locatexy
lower	max	maxcol	maxcolor	maxcolour
maxrow	maxx	maxy	min	nargs
nil	open	ord	palette	play
playdone	pred	rand	randint	randnext
randomize		randseed	realstr	repeat
round	screen	setscreen	sign	sin
sind	sizepic	sound	sqrt	strint
strreal	succ	sysdock	system	takepic
time	upper	wallclock	whatcolor	
whatcolorback	whatcolour	whatcolourback		
whatdotcolor	whatdotcolour	whatpalette		
whattextchar	whattextcolor	whattextcolorback		
whattextcolour	whattextcolourback			

A List of Keywords

all	and	array	assert	begin
bind	body	boolean	case	close
collection	const	decreasing	div	else
elsif	end	enum	exit	export
external	false	fen	for	forward
free	function	get	if	import
in	init	int	invariant	label
loop	mod	module	new	nil
not	of	opaque	open	or
pervasive	pointer	post	pre	proc
procedure	put	read	real	record
result	return	seek	set	skip
string	tag	tell	then	to
true	type	union	var	when
write				

Appendix B

A List of Predefined Functions and Procedures By Category without Arguments

Math

abs	absolute value function
arctan	arctangent with result in radians
arctand	arctangent with result in degrees
cos	cosine of angle in radians
cosd	cosine of angle in degrees
exp	exponentiation function
In	natural logarithm function
max	maximum value function
min	minimum value function
	returns the sign of the argument
	sine of angle in radians
sind	sine of angle in degrees
sqrt	square root function

Type Conversion

From Int

intreal	convert integer to real
intstr	convert integer to string

From Real

ceil	ceiling of argument producing integer
floor	floor of argument producing integer
round	round argument to nearest integer
realstr	convert real to string
frealstr	convert real to string
erealstr	convert real to string withexponent

From String

strint	convert string to integer
strreal	convert string to real

To/From ASCH

chr	convert integer to ASCII equivalent
ord	convert character to ASCII equivalent

Time

clock	time in milliseconds since process began
date	date in "DD Mmm YY" format
sysclock	CPU time in milliseconds used by process
time	time in "HH:MM:SS" format
wallclock	time in seconds since 00:00:00 GMT Jan 1,1970

System

getenv	return the environment string
getpid	return the process id
nargs	return the number of command line args
fetcharg	return the specified command line arg
system	execute operating system command
delay	pause program for duration in milliseconds

Sound

play	play musical notes
playdone	determine if play command finished
sound	play specified sound

Character Graphics

els	clear the screen
color	set text color
colour	set text colour
colorback	set text background color
colourback	set text background colour
locate	place cursor at specified character position
maxcol	returns the number of screen text columns
maxrow	returns the number of screen text rows
maxcolor	returns the maximum screen text color
maxcolour	returns the maximum screen text colour
screen	set the mode of the screen
setscreen	set the mode of the screen
whatcolor	return the currently active text color
whatcolour	return the currently active text colour
whatcolorback	return the current text background color
whatcolourback	return the current text background colour
whattextchar	returns the character at cursor position
whattextcolor	returns the character color at cursor
whattextcolour	returns the character colour at cursor
whattextcolorback	returns the character background color at cursor
whattextcolourback	returns the character background colour at cursor

Pixel Graphics

els	clear the screen
color	set text color
colour	set text colour
colorback	instantly change background color
colourback	instantly change background colour
drawarc	draws an arc on screen
drawbox	draws a box on screen
drawdot	draws a dot on screen
drawfill	fills in a figure of color borderColor
drawline	draws a line on screen
drawoval	draws an oval on screen
takepic	copies a rectangular area from screen into buffer
drawpic	copies a rectangular area from buffer onto the screen
sizepic	determine the necessary size of a buffer to hold area from screen
locate	place cursor at specified character position
locatexy	place cursor at specified pixel position
maxcol	returns the number of screen text columns
maxrow	returns the number of screen text rows
maxcolor	returns the maximum screen color
maxcolour	returns the maximum screen colour
maxx	returns the maximum x pixel value
maxy	returns the maximum y pixel value
palette	sets the colour palette to use
screen	set the mode of the screen
selscreen	set the mode of the screen
whatcolor	return the currently active text color
whatcolour	return the currently active text colour
whatcolorback	return the current background color
whatcolourback	return the current background colour
whatdotcolor	return the color of a pixel
whatdotcolour	return the colour of a pixel
whatpalette	return the current palette number

Strings

index	find position of patt within string s
length	returns the length of the string
repeat	copies string s concatenated i times

Files

eof	returns whether stream is at end of file

Character Input

getch	get a single character from the keyboard
hasch	returns whether there is a keystroke in the keyboard buffer

Attributes

lower	returns the lower bound of array in specified dimension
upper	returns the upper bound of array in specified dimension

Random Numbers

rand	generate a random number from 0 to 1
randint	generate a random integer from low to high inclusive
randnext	produce next pseudo-random number in specified sequence
randomize	resets random number sequence
randseed	restarts the specified sequence of pseudo-random numbers with seed

Enumerated Types

pred	returns the argument minus one or the previous value in the enumerated sequence
succ	returns the argument plus one or the next value in the enumerated sequence

Appendix C

A List of Predefined Functions and Procedures By Category with Arguments

Math

abs(x:int):int	absolute value function
abs (x: real): real	
arctan (x : real): real	arctangent with result in radians
arctand (x: real): real	arctangent with result in degrees
cos (angle : real): real	cosine of angle in radians
cosd (angle: real): real	cosine of angle in degrees
exp (r: real): real	exponentiation function (e^r)
In (r: real): real	natural logarithm function ($\ln_e r$)
max (x, y: int): int	maximum value function
max (x, y: real): real	
min (x, y: int): int	minimum value function
min (x , y: real): real	
sign (r: real):-!..!	returns the sign of the argument
sin (angle : real): real	sine of angle in radians
sind (angle : real): real	sine of angle in degrees
sqrt (r: real): real	square root function

Type Conversion

From Int

intreal (i: int): real	convert integer to real
intstr (i, width : int): string	convert integer to string

From Real

ceil (r: real): int	ceiling of argument producing integer
floor (r: real): int	floor of argument producing integer
round (r: real): int	round argument to nearest integer
realstr (r: real, width: int): string	convert real to string
frealstr (r: real, width, fractionWidth : int): string	convert real to string
erealstr (r: real, width, fractionWidth, exponentWidth: int): string	convert real to string withexponent

From String

strint (s : strung): int	convert string to integer
strreal (s : string): real	convert string to real

To/From ASCH

chr (i: int): string (1)	convert integer to ASCII equivalent
ord (ch: string (1)): int	convert character to ASCII equivalent

Time

clock (var c : int)	time in milliseconds since process began
date (var date : string)	date in "DD Mmm YY" format
sysclock (var c : int)	CPU time in milliseconds used by process
time (var t: string)	time in "HH:MM:SS" format
wallclock (var c : int)	time in seconds since 00:00:00 GMT Jan 1,1970

System

getenv (symbol: string): string	return the environment string
getpid: int	return the process id
nargs: int	return the number of command line args
fetcharg (i: int, s: string)	return the specified command line arg
system (command: string, var ret: int)	execute operating system command
delay (duration: int)	pause program for duration in milliseconds

Sound

play (music: string)	play musical notes
playdone: boolean	determine if play command finished
sound (freq, duration: int)	play specified sound

Character Graphics

els	clear the screen
color (clr: int)	set text color
colour (clr: int)	set text colour
colorback (clr: int)	set text background color
colourback (clr: int)	set text background colour
locate (row, col: int)	place cursor at character position (row,col)
maxcol: int	returns the number of screen text columns
maxrow: int	returns the number of screen text rows
maxcolor: int	returns the maximum screen text color
maxcolour: int	returns the maximum screen text colour
screen (mode: hit)	set the mode of the screen
setscreen (s: string)	set the mode of the screen
whatcolor: int	return the currently active text color
whatcolour: int	return the currently active text colour
whatcolorback: int	return the current text background color
whatcolourback: int	return the current text background colour
whattextchar: string (1)	returns the character at cursor position
whattextcolor: hit	returns the character color at cursor
whattextcolour: int	returns the character colour at cursor
whattextcolorback: int	returns the character background color at cursor
whattextcolourback: int	returns the character background colour at cursor

Pixel Graphics

els — clear the screen
color (clr: int) — set text color
colour (clr: int) — set text colour
colorback (clr: int) — instantly change background color
colourback (clr: int) — instantly change background colour
drawarc (x, y, xRadius, yRadius , initialAngle, finalAngle, clr: int) — draws an arc on screen centred at (x,y)
drawbox (xl, yl, x2, y2, clr: int) — draws a box on screen
drawdot (x, y, clr: int) — draws a dot on screen at (x,y)
drawfill (x, y, fillColor, border-Color: int) — fills in a figure of color borderCoIor
drawline (xl, yl, x2, y2, clr: int) — draws a line on screen
drawoval (x, y, xRadius, yRadius, clr: int) — draws an oval on screen centred at (x,y)
takepic (xl, yl, x2, y2: int, var buffer: array 1..* of int) — copies a rectangular area from screen into buffer
drawpic (x, y: int, buffer: array 1..* of int, picmode: int) — copies a rectangular area from buffer onto the screen
sizepic (xl, yl, x2, y2 : int): int — determine the necessary size of a buffer to hold area from screen
locate (row, col: int) — place cursor at character position (row, col)
locatexy (x, y: int) — place cursor at pixel position (x,y)
maxcol: int — returns the number of screen text columns
maxrow: int — returns the number of screen text rows
maxcolor: int — returns the maximum screen color
maxcolour: int — returns the maximum screen colour
maxxrint — returns the maximum x pixel value
maxy: int — returns the maximum y pixel value
palette (p : int) — sets the colour palette to use
screen (mode: int) — set the mode of the screen
setscreen (s: string) — set the- mode of the screen
whatcolor: int — return the currently active text color
whatcolour: int — return the currently active text colour
whatcolorback: int — return the current background color
whatcolourback: int — return the current background colour
whatdotcolor (x, y: int): int — return the color of pixel at (x,y)
whatdotcolour (x, y: int): int — return the colour of pixel at (x,y)
whatpalette : int — return the current palette number

Strings

index (s, part: string): int — find position of patt within string s
length (s : string): int — returns the length of the string
repeat (s : string, i: int): string — copies string s concatenated i times

Files

eof (streamNo: int): boolean — returns whether stream is at end of file

Character Input

getch (var ch: string (1)) — get a single character from the keyboard
hasch: boolean — returns whether there is a keystroke in the keyboard buffer

Attributes

lower (array [, dimension]): int — returns the lower bound of array in specified dimension
upper (array [, dimension]): int — returns the upper bound of array in specified dimension

Random Numbers

rand (var r: real) — generate a random number from 0 to 1
randint (var i: int, low, high : int) — generate a random integer from low to high inclusive
randnext (var v: real, seq : 1..10) — produce next pseudo-random number in specified sequence
randomize — resets random number sequence
randseed (seed : int, seq: 1..10) — restarts the specified sequence of pseudo-random numbers with seed

Enumerated Types

pred (enumerated value): enumerated value
pred (i: int): int — returns the argument minus one or the previous value in the enumerated sequence
succ (enumerated value): enumerated value
succ (i: int): int — returns the argument plus one or the next value in the enumerated sequence

Appendix D

IBM PC Keyboard Codes

The ascii value of the character returned by getch

	0	(space)	32	@	64	`	96
Ctrl-A	1	!	33	A	65	a	97
Ctrl-B	2	"	34	B	66	b	98
	3	#	35	C	67	c	99
Ctrl-D	4	$	36	D	68	d	100
Ctrl-E	5	%	37	E	69	e	101
Ctrl-F	6	&	38	F	70	f	102
Ctrl-G	7	'	39	G	71	g	103
Ctrl-H / BS	8	(	40	H	72	h	104
Ctrl-I / TAB	9	)	41	I	73	i	105
Ctrl-J / CR	10	*	42	J	74	j	106
Ctrl-K	11	+	43	K	75	k	107
Ctrl-L	12	,	44	L	76	l	108
Ctrl-M	13	-	45	M	77	m	109
Ctrl-N	14	.	46	N	78	n	110
Ctrl-0	15	/	47	O	79	o	111
Ctrl-P	16	0	48	P	80	p	112
Ctrl-Q	17	1	49	Q	81	q	113
Ctrl-R	18	2	50	R	82	r	114
Ctrl-S	19	3	51	S	83	s	115
Ctrl-T	20	4	52	T	84	t	116
Ctrl-U	21	5	53	U	85	u	117
Ctrl-V	22	6	54	V	86	v	118
Ctrl-W	23	7	55	W	87	w	119
Ctrl-X	24	8	56	X	88	x	120
Ctrl-Y	25	9	57	Y	89	y	121
Ctrl-Z	26	:	58	Z	90	z	122
Ctrl-[/ESC	27	;	59	[	91	{	123
CtrlA	28	<	60	\	92	|	124
Ctrl-]	29	=	61	]	93	}	125
Ctrl-^	30	>	62	^	94	~	126
Ctrl-_	31	?	63	_	95	Ctrl-BS	127

Alt-9	128	Alt-D	160	F6	192	Ctrl-F3	224
Alt-0	129	Alt-F	161	F7	193	Ctrl-F4	225
Alt-	130	Alt-G	162	F8	194	Ctrl-F5	226
All—	131	Alt-H	163	F9	195	Ctrl-F6	227
Ctrl-PgUp	132	Alt-J	164	F10	196	Ctrl-F7	228
	133	Alt-K	165		197	Ctrl-F8	229
--	134	Alt-L	166		198	Ctrl-F9	230
	135		167	Home	199	Ctrl-FlO	231
	136		168	Up Arrow	200	Alt-Fl	232
	137		169	PgUp	201	Alt-F2	233
	138		170		202	AU-F3	234
	139		171	Left Arrow	203	Alt-F4	235
	140	Alt-Z	172		204	AU-F5	236
	141	Alt-X	173	Right Arrow	205	AU-F6	237
	142	Alt-C	174		206	AU-F7	238
Back TAB	143	Alt-V	175	End	207	AU-F8	239
Alt-Q	144	Alt-B	176	Down Arrow	208	AH-F9	240
Alt-W	145	Alt-N	177	PgDn	209	AU-F10	241
Alt-E	146	Alt-M	178	Ins	210		242
Alt-R	147		179	Del	211	Ctrl-L Arrow	243
Alt-T	148		180	Shift-Fl	212	Ctrl-R Arrow	244
Alt-Y	149		181	Shift-F2	213	Ctrl-End	245
Alt-U	150		182	Shift-F3	214	Ctrl-PgDn	246
Alt-I	151		183	Shift-F4	215	Ctrl-Home	247
Alt-O	152		184	Shift-F5	216	Alt-1	248
Alt-P	153		185	Shift-F6	217	Alt-2	249
	154		186	Shift-F7	218	Alt-3	250
	155	Fl	187	Shift-F8	219	Alt-4	251
	156	F2	188	Shift-F9	220	Alt-5	252
	157	F3	189	Shift-FlO	221	Alt-6	253
Alt-A	158	F4	190	Ctrl-Fl	222	Alt-7	254
Alt-S	159	F5	191	Ctrl-F2	223	Alt-8	255

Ctrl-®, Ctrl-C and Ctrl-Break will terminate a Turing Program

"'}

Appendix E

Turing Operators

Mathematical Operators

Operators on Integers and Reals

Operator	Operation	Result Type
Prefix +	Identity	As Operands
Prefix -	Negative	As Operands
+	Addition	As Operands
-	Subtraction	As Operands
*	Multiplication	As Operands
/	Division	As Operands
div	Integer Division	Integer
mod	Remainder	Integer
**	Exponentiation	As Operands
<	Less Than	Boolean
>	Greater Than	Boolean
=	Equals	Boolean
<=	Less Than or Equal	Boolean
>=	Greater Than or Equal	Boolean
not=	Not Equal	Boolean

Boolean Operators

Operators on Booleans

Operator	Operation	Result Type
Prefix not	Negation	Boolean
and	And	Boolean
or	Or	Boolean
=>	Implication	Boolean

Set Operators

Operators on Sets

Operator	Operation	Result Type
+	Union	Set
-	Set Subtraction	Set
*	Intersection	Set
=	Equality	Boolean
not=	Inequality	Boolean
<=	Subset	Boolean
<	Strict Subset	Boolean
>=	Superset	Boolean
>	Strict Superset	Boolean

Special Set Operators

Operators on Members and Sets

in	Member of Set	Boolean
not in	Not Member of Set	Boolean

Operator Precedence

0)	**
(2)	prefix + and -
(3)	*, /, div, mod
(4)	+, -
(5)	<, /, >, /, =, /, <=, >=, not=, in, not in
(6)	not
(7)	and
(8)	or
(9)	=>

Appendix F

Turing File Statements

File Commands

open	open a file
close	close a file
put	write alphanumeric text to a file
get	read alphanumeric text from a file
write	binary write to a file
read	binary read from a file
seek	move to a specified position in a file
tell	report the current file position
eof	check for end of file

File Command Syntax

open: streamNo, fileName, ioCapability { , ioCapability}
ioCapabilility is one of get, put, read, write, seek, mod
put or write capability will cause any existing file to be truncated to zero length unless the mod capability is also specified.
seek capability is needed to use seek or tell commands.
close: streamNo
get: streamNo, getltem { , getltem}
put: streamNo, putltem { , putltem}
read: streamNo [: fileStatus], readltem { , readltem}
write: streamNo [: fileStatus], writeltem { , writeltem}
seek: streamNo, f ilePosition or seek : streamNo, *
tell: streamNo, filePositionVar
(function call) eof (streamNo)

Appendix G

Turing Control Constructs

FOR

```
for [ decreasing ] variable : startValue.. endValue
        ... statements...
        exit when condition
        ... statements ...
end for
```

LOOP

```
loop
        ... statements...
        exit when condition
        ... statements...
end loop
```

IF

```
if condition then
        ... statements...
{elsif condition then
        ... statements...}
[ else
        ... statements... ]
end if
```

CASE

```
case expn of
        {label expn { , expn} :
                ... statements ... }
        [label:
                ... statements... ]
end case
```

Any number of exit and exit when constructs can appear at any place inside for.. end for constructs and loop .. end loop constructs

Appendix H

Using The Printer From Turing on IBM PC's and Compatibles

There are three methods for producing output on a printer from within Turing programs running on IBM PC's and compatibles. The first is to direct output to the screen as usual and to use the SHIFT-PRINTSCREEN from your keyboard to print a copy of the information on the screen. Note that this will only print information that is on the screen at the time you press the PRINTSCREEN. Any information that has scrolled off the top of the screen is lost and information not yet displayed will not be printed at this time. Also, this is the only way in Turing to print pixel graphics output.

The second method is to specify that all standard output is to be directed to the printer. Under MS-DOS, the filename PRN is reserved to mean the printer. Thus you can use output redirection to run your program and route the output to file PRN. For example, instead of using the Fl key to run the program, type: :r > prn. See page 13 for more information on the :r command and input/output redirection.

The third method is frequently the closest to what a typical user wants. Output is directed to the screen or to the printer under program control. To do this, you must open file PRN for put and you must explicitly direct output to the appropriate location within your program. Here is an example program:

```
var prnStream : int
open :prnStream, "PRN", put
% verify that the printer is available
assert prnStream > 0

put "You will see this line on the screen."
put rprnStream, "This line is printed."
```

Run the program normally and output will appear as directed by each put statement.