THE TURING LANGUAGE REPORT

R.C. Holt and J.R. Cordy

Technical Report CSRI-153
December 1983
(Revised July 1985)
(Updated August 1986)

Computer Systems Research Institute
University of Toronto
Toronto, Canada
M5S 1A1

Abstract

This report specifies the Turing language. Turing is a general purpose programming language that is well suited for teaching programming. It is designed to support the development of reliable, efficient programs. It incorporates language features that decrease the cost of program maintenance and that support formal verification.

Turing is designed to be supported by a user-friendly compiler and run-time system of modest size; these should insulate the user from vagaries of the underlying hardware and operating system. Turing is a Pascal-like language and incorporates almost all of Pascal's features. Turing alleviates many difficulties with Pascal; for example, Turing provides convenient text string handling, information hiding using modules, type-safe variant records (unions), as well as dynamic parameters and arrays.

A tutorial introduction to Turing is given in the text book “Introduction to Computer Science using the Turing Programming Language” [Holt & Hume 1984]. Turing compilers for various systems, including VAX/Unix, IBM PC/XT and IBM CMS are available from CSRI, University of Toronto. The language's formal definition is contained in the book “The Turing Programming Language: Design and Definition” [Holt et al. 1987], and the language appears as the cover article in the December 1988 edition of the Communications of the ACM [Holt & Cordy 1988].

Acknowledgements

The design of the Turing language has drawn heavily from previous program ming languages and from the advice and help of a number of people. The heaviest debt is to the Euclid family of programming languages, including Concurrent Euclid, which in turn draws extensively from Pascal.

The language design was greatly helped by concurrent development of its formal semantics, its first compilers and a text book. J.A. Rosselet formally specified the language's context checking using his ADL notation. P.A. Matthews and R.C. Holt developed the language's axiomatic semantics. J.R. Cordy, A.F.X. Curley, R.C. Holt, M.P. Mendell, S.G. Perelgut, and S.W.J. Tjiang developed the portable Turing compiler. This compiler is written in Concurrent Euclid, using the S/SL compiler-writing system; as of July 1983, it has been implemented for the DEC VAX, IBM 370, Motorola MC68000 and IBM PC/XT.

T.E. Hull encouraged the creation of the language and guided the development of its numeric features. J.N.P. Hume co-authored the text book and helped simplify certain language constructs. E.C.R. Hehner helped clarify and simplify procedures and functions. I.S. Weber, P.A. Matthews and T.C.N. Graham managed the computer typesetting of the Turing Report.

The original development period of Turing, from its inception, to its specification and formalization, to using its text book with compilers in classrooms, was 12 months.

The development of Turing was made possible by the financial support of the Natural Sciences and Engineering Research Council of Canada, and Bell Northern Research Limited.

Table Of Contents

1 Introduction, Terminology, and Notation

Turing is intended to be a general purpose language, meaning that it is potentially the language of choice for a wide class of applications. Because of its combination of convenience and expressive power it is particularly attractive for learning and teaching. Because of its clean syntax, Turing programs are relatively easy to read and easy to write.

The language improves reliability by disallowing error-prone constructs. It provides numerous compile-time and run-time checks to catch bugs before they cause disaster. These checks guarantee that each Turing program behaves according to the Turing language definition, or else a warning message is printed.

To support maximal efficiency, there is an option to remove run-time checking. This option allows well-tested, heavily used Turing programs to be extremely efficient. Each construct in Turing is designed to have an obvious, efficient implementation on existing computer hardware.

The design of Turing aims to address the verification and security difficulties with Pascal-like languages. For example, compile-time checks are used to prevent aliasing of variables and side effects in functions. Aliasing due to pointers has been eliminated using the concept of collections. Variant records (unions) have been made type safe by means of a tag statement that explicitly selects among the types of values to be represented.

Perhaps the most important programming construct developed in the last decade is the module or cluster, which enforces information hiding and supports data abstraction. Turing incorporates this feature, with the result that construction of large programs as a set of nearly independent parts is relatively straightforward.

Turing is well suited to interactive programming; it is intended for use on personal computers as well as on traditional main-frame computers.

The language is designed to be easily and efficiently implemented. Experience has shown that a production quality portable Turing compiler can be constructed in a few man-months.

1.1 Terminology and Basic Concepts

This section informally introduces basic terms, such as "scope" and "constant", used in describing the Turing language.

Variables and constants.

A *variable* is a named item whose value can be changed by an assignment statement.

Example:

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

This example uses comments that begin with a percent sign (%) and are concluded by the end of the line. Various items such as variables are given names; these names are called identifiers.

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

```
const c := 25
const x := sin (y)**2      % y Is a variable
```

In this example, c's value is known at compile time, so it is a compile-time (or *manifest*) constant. (See also "Compile-Time Expressions"). The value of x is computed at run-time; it is a run-time (or nonmanifest) constant. Since x's value depends on variable y, different executions of the construct containing x's declaration may produce different values of x. During the lifetime of each x, the value of that particular x remains constant, even though y may change.

An *explicit* (or *literal*) *constant* is a constant that denotes its own value; for example, the following are an explicit integer constant, an explicit boolean constant and an explicit string constant:

```
219
true
"Have a nice day"
```

Scopes and visibility.

The textual lifetime of a named item is called its *scope*. For example, the scope of x in the following is the body of the begin statement in which it is declared.

```
begin
    var z: real
    ... body of begin statement ...
end
```

A declared item's scope begins at its declaration and continues to the end of the construct in which the declaration occurs. An item's declaration textually precedes any use of the item (except in those cases in which the identifier is explicitly preceded by the forward keyword).

The visibility (scope) rules of Turing are basically the same as those of the Algol/Pascal family of languages. This means that a declared item is visible (can be named) throughout its scope, including in subconstructs of the scope.

Most constructs (variables, types, constants, subprograms and modules) cannot be named using an identifier that is already visible. That is, for most constructs, redeclaration of identifiers is disallowed. (The exceptions are names of parameters, values of enumerated types, and names of record and union fields.)

There is one construct (the module) in an item's scope that does not automatically inherit the ability to access the item. Modules must explicitly import items that are to be accessed in the module body.

A subprogram may optionally use an import list to specify items used in the subprogram's body.

Example:

```
1    var i: int
2    procedure increase (increment: int) % Increase i by increment
3        import (var i)
4        i := i + increment
5    end increase
6    increase (4)      % Increase i by 4
```

In this example, variable i is imported on line 3; it is imported *var* (as in var-iable) indicating that it can be changed in the procedure (in line 4). If i were imported without the var keyword, then it could be inspected, but line 4, which changes i, would not be allowed. A variable that is imported non-var is called a *read-only* variable. If a subprogram does not use an import list, it is considered to implicitly import any items that it actually references. By contrast, a module is always required to explicitly import any global items that are accessed in the module's body.

Subprograms.

Turing has two kinds of subprograms - *procedures* and *functions*. A procedure is called using a procedure call statement. Calls to a function occur in expressions and return a value to be used in the expression.

A subprogram header may optionally declare formal parameters. For example, line 2 of the above example declares "increment" as a formal parameter. Each call to the subprogram must supply actual parameters corresponding to the formal parameters; for example in line 6 above, "4" is an actual parameter.

Modules.

A *module* is a construct used for packaging items together, including subprograms and variables, with information hiding. Access to internal items is controlled by an "export" list.

Example:

```
module stack                  % Implement a stack of integers
    export (push, pop)        % Entries to stack are push and pop

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

    procedure push (i: int)
        top := top + 1
        contents (top) := i
    end push

    procedure pop (var i: int)
        i := contents (top)
        top := top – 1
    end pop
end stack

stack.push (14)      % Push 14 onto the stack
```

Since the module exports only push and pop, these are the only items in the module that can be accessed outside of the module. To access an exported item, one prefixes its name by the module's name, as in *stack.push*.

Side effects.

If executing a construct changes values of items outside of the construct, we say that it has *side effects*. For example, the *increase* procedure given as an example above has the side effect of changing the value of variable i. In order to preserve their mathematical meaning and assist in verification, Turing prevents functions from having side effects. The method of guaranteeing that functions have no side effects is by disallowing them from having *var* parameters (these are parameters that can be changed), by disallowing them from importing items *var*, and by disallowing them from directly or indirectly importing procedures that import items *var*.

Since expressions cannot have side effects, in Turing all calls to a function with the same values of parameters and of imported variables necessarily return the same value.

Example:

```
    x := f (24)      % Call function f with 24 as parameter
    y := f (24)      % Function f does not import x
```

After the execution of these two Turing statements, it is necessarily true that x = y.

Aliasing.

Given distinct visible identifiers x and y, aliasing is said to exist if a change to the value of x would also change the value of y. In the following, suppose i and j are aliases for the same variable.

```
    i := 1
    j := 2
```

After execution of these statements, i and j (which are actually the same variable) will both have the value 2. Aliasing greatly complicates formal program verification, and confuses the programmer. For these reasons, Turing bans aliasing. This ban is enforced by placing constraints on those language constructs that may allow variables to be renamed. In Turing, the only constructs that rename variables are reference parameters to subprograms, and the *bind* construct. Constraints on the use of these two constructs guarantee that once the new identifier is visible, either the old identifier is inaccessible, or both identifiers are read-only.

Execution with side effects and aliasing.

As has just been explained, the Turing language nominally prohibits side effects and aliasing. However, an implementation may extend the language to allow execution of programs violating these restrictions, given that appropriate error messages are issued. Programs with these violations have a well-defined operational (execution) semantics that is described by this Report, but such programs are not defined by the formal semantics of the Turing language.

Dynamic arrays.

An array is said to be *dynamic* if its size is not known at compile time, i.e., if its bounds are necessarily computed at run-time. Turing allows dynamic arrays.

Example:

```
    get n                               % Read value into variable n
    var a: array 1 ..n of real
```

This creates an array (really a vector) called a of n real values.

Dynamic parameters of subprograms.

Dynamic parameters allow arrays of varying sizes to be used as arguments to a subprogram. For example, the following function sums the first i elements of an array.

```
    1    function sum (b: array 1..* of real, i: int) : real
    2        var total: real := OD
    3        for j: 1..i
    4            total := total + b (j)
    5        end for
    6        result total
    7    end sum
    8    x := sum (a, 10)
```

Line 8 calls the *sum* function to add up the first 10 elements of array a. Line 1 uses "*" as the upper bound of parameter b to specify that b's upper bound is inherited from the upper bound of its corresponding argument (actual parameter). The final keyword *real* on line 1 specifies that the function returns a real value. Lines 3-5 are a *for* statement; it adds the first i elements of array b to total. Line 6 returns the value of total as the value of the function.

Checking and Faithful Implementations.

An implementation of Turing is said to be "faithful" if it meets the following criterion: the results of executing any Turing program will be determined by the source program together with the Turing specification (this Report) or else execution will abort with a message indicating (1) the reason for the abort, and (2) the location in the program where it was executing at the time of the abort. A checking implementation guarantees that execution is faithful. An abort may occur only because of (1) violation of a language-defined run-time constraint, such as a subscript out of bounds, or (2) resource exhaustion. Resource exhaustion may occur for a number of reasons, including lack of memory for calling a procedure, excessive run time, or excessive output.

Turing run-time constraints are categorized as (1) language constraints or (2) implementation constraints. Language (or validity) constraints disallow those actions that are clearly meaningless, such as division by zero, subscript out of bounds, or a false value in an assert statement. Implementation constraints disallow actions which have a language defined meaning but which are infeasible due to hardware or efficiency reasons, such as: limited range of int or limited exponent range of real implementations. Note that there are compile-time as well as run-time implementation constraints; the limited range of *case* statement labels is an example of a compile-time implementation constraint.

A non-checking implementation of Turing may omit any or all of the run-time checks required for faithful implementation; this omission of checking allows Turing programs to have execution efficiency comparable to programs written in machine-oriented languages like C. A non-checking implementation may assume that the user has written his program in such a way that there will be no violations of run-time constraints nor resource exhaustion; these assumptions may be used for improving the quality of generated code. It is recommended that a non-checking implementation should provide documentation of the run-time constraints that are not enforced and resource exhaustions that may not be detected.

The formal definition of Turing assumes that the real type corresponds exactly to the real numbers of mathematics. However, it is expected that implementations of Turing will actually implement real using the floating point unit of the host computer. To enhance portability, the section "Implementation Constraints on Integer, String and Real Types" in this Report gives suggested minimum standards for floating point precision and exponent range, as well as standards for implemented ranges of integers and maximum string lengths.

This section has introduced basic terminology and concepts. The remaining sections give the detailed specification of the Turing language.

1.2 Identifiers and Explicit Constants

An *identifier* consists of a sequence of at most 50 letters, digits, and underscores beginning with a letter; all of these characters are significant in distinguishing identifiers. Upper and lower case letters are considered to be distinct in identifiers and keywords; hence j and J are different identifiers. Keywords must be in lower case. Keywords and predefined identifiers must not be redeclared, that is, they are reserved words.

An *explicit string constant* is a sequence of zero or more characters surrounded by double quotes. Within explicit string constants, the back slash character (\) is an escape to represent certain characters as follows: \" for double quote, \n or\N for end of line character, \t or\T for tab, \f or\F for form feed (new page), \r or\R for carriage return, \b or\B for backspace, \e or\E for escape, \d or \D for delete, and \\ for back slash. Explicit string constants must not cross line boundaries. Within explicit string constants, the following two characters are disallowed: *eos* and *uninitchar*.

The *eos* (end of string) character is an implementation-dependent character that an implementation may use to mark the ends of strings. The *uninitchar* is an implementation dependent character that an implementation may use to mark a string that has not yet been assigned a value. (See "Implementation Constraints on Integer, String and Real Types" for recommended values of *eos* and *uninitchar*.)

Character values are ordered by either the ASCII or EBCDIC collating sequence, depending on the host computer (see "Character Collating Sequence").

An *explicit integer constant* is a sequence of one or more decimal digits, optionally preceded by a plus or minus sign.

An *explicit real constant* consists of three parts: an optional plus or minus sign, a significant figures part, and an exponent part. The significant figures part consists of a sequence of one or more digits optionally containing a decimal point. The exponent part consists of the letter e (or E) followed optionally by a plus or minus sign, followed by one or more digits. If the significant figures part contains a decimal point then the exponent part is optional. The following are examples of explicit real constants:

```
    2.0     0.     .1     2e4    -56.1e+27
```

An explicit integer or real constant that begins with a sign is called a *signed constant*; without the sign, it is called an *unsigned constant*.

The *explicit boolean* constants are *true* and *false*.

1.3 Comments and Separators

An end-of-line comment begins with the character % and ends at the end of the current line. A bracketed comment is any sequence of characters not including comment brackets surrounded by the comment brackets /* and */. Bracketed comments may cross line boundaries. A separator is a comment, blank, tab, form feed or end of line.

The tokens of the Turing language are the identifiers, keywords, explicit unsigned integer and real constants, explicit string and boolean constants, the operators and the special symbols. (See "Collected Keywords and Predefined Identifiers" and "Collected Operators and Special Symbols".) Each token in a Turing program can be preceded by any number of separators. Separators must not appear within any tokens, except as characters in explicit string constants. (See also "Recognizing Tokens".) Ends of lines must not appear within an explicit string constant; note that long string constants can be broken at line boundaries and connected by the concatenation operator (+).

In a Turing program, the sign that begins an explicit signed integer or real constant can have separators between it and the following unsigned constant. The sign and the unsigned constant are considered to form a signed constant only if the sign is a prefix operator, and not an infix operator (according to the syntax of Turing). For example, in the following, -4 is an unsigned constant, but -7 is not:

```
    if x -7 > -4 then ...
```

1.4 Syntactic Notation

In the remainder of this Report, the following syntactic notation is used:

`{ item }`	means zero or more of the item
`[ item ]`	means that the item is optional

Be warned: although this Report uses braces { ... } and brackets [...] as syntactic notation, another use of braces and brackets appears in the section "Short Forms". That section explains the use of braces and brackets as short forms for the Turing loop and if statements respectively.

Keywords and special characters are given in **boldface**. Nonterminals, e.g., *typeSpecification*, are given in italics. The following abbreviations are used:

id	for	*identifier*
expn	for	*expression*
typeSpec	for	*typeSpecification*

2 Programs and Declarations

2.1 Programs

A *program* consists of a sequence of declarations and statements.

A *program* is:

{ *declarationOrStatementInMainProgram* }

A program is executed by executing its declarations and statements. Here is a complete Turing program that prints: Alan Turing.

```
put "Alan Turing"
```

A *declarationOrStatementInMainProgram* is one of:

a. *declaration* [;]
b. *statement* [;]
c. *collectionDeclaration* [;]
d. *subprogramDeciaration* [;]
e. *moduleDeclaration* [;]

Each declaration or statement may optionally be followed by a semicolon.

2.2 Declarations

A *declaration* is one of the following:

a. *constantDeclaration*
b. *variableDeclaration*
c. *typeDeclaration*

Each of these declarations creates a new identifier (or identifiers); each new identifier must be distinct from other visible identifiers. That is, redeclaration of visible identifiers is not allowed. The effect of the 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 a subprogram or module, the end of a **begin**, **loop** or **for** statement, the end of a **then**, **elsif** or **else** clause of an **if** statement, or the end of a **case** statement alternative. An identifier must be declared textually preceding any references to it; the exception to this rule is the form "**forward** id", occurring in **import** lists and collection declarations.

2.3 Constant Declarations

A *constantDeclaration* is one of:

a. **const** [**pervasive**] id := expn
b. **const** [**pervasive**] id : *typeSpec* := *initializingValue*

An *initializingValue* is one of:

a. *expn*
b. **init** { *initializingValue* {, *initializingValue* } }

Examples:

```
const c := 3                    % The type of c is int
const s := "Hello"              % The type of s is string
const x := sin(y)**2            % The type of x is real
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)
% Assigns: c (1,1) := 1; c (1,2) := 2; c (2,1) := 3; c (2,2) := 4
```

A *constantDeclaration* introduces a name whose value is constant throughout the scope of the declaration. If the *typeSpec* is omitted, the type of the constant is taken to be the (root) type of the expression, which must not be a dynamic array. An initializing expression that does not appear inside an **init** construct may be a compile-time or run-time expression (but not a dynamic array), and must be assignable to the constant's type. Named non-scalar values are always considered to be run-time values. (See also "Compile-Time Expressions" and "Type Equivalence and Assignability").

The **init** construct is used only to initialize arrays, records and unions. All values in an **init** construct must be compile-time expressions. Note that **init** may be nested inside another **init** to initialize records, unions or arrays that contain other records, unions or arrays. The number of elements inside an **init** construct must equal the number of elements in the type of the constant being initialized. For a union, the **init** must contain first the tag value and then the field values corresponding to this tag value (see the discussion of union types in "Types and Type Declarations").

Constants declared using **pervasive** are visible in all subconstructs of the constant's scope. Such constants need not be explicitly imported.

2.4 Variable and Collection Declarations

A *variableDeclaration* is one of:

a. **var** id {, id } := *expn*
b. **var** id {, id } : *typeSpec* [:= *initializingValue*]

Examples:

```
var j, k: int := 1          % j and k are assigned initial value 1
var t := "Sample output"    % The type of t is string
var v: array 1..3 of string (6) := init ("George",“Fred","Alice")
```

A *variableDeclaration* creates a new variable (or new variables). In form (a), the variable's type is taken to be the (root) type of the expression; this type must not be a dynamic array.

Form (b) allows the declaration of dynamic arrays, whose upper bounds are run-time expressions. However, the lower bounds are constrained to be compile-time expressions. Given that n is a variable, here is an example of the declaration of a dynamic array:

```
    var w: array 1..n, 1..n of real
```

Run-time bounds are only allowed as illustrated in this example, i.e., as the upper bounds of an array declared using form (b). Note that dynamic arrays can never appear in records, unions, or collections. Each upper bound must be at least as large as its corresponding lower bound. A dynamic array cannot be initialized in its declaration and cannot be a named type.

A *collectionDeclaration* is one of:

a. **var** id {, id } : **collection of** *typeSpec*
b. **var** id {, id } : **collection of forward** id

A collection can be thought of as an array whose elements are dynamically created and deleted at run-time. Elements of a collection are referenced by subscripting the collection name with a variable of the collection's pointer type. (See the discussion of pointers in "Types and Type Declarations".) This subscripting selects the particular element of the collection located by the pointer variable.

The keyword forward is used to specify that the type id of the collection elements will be given by a later declaration in the collection's scope. The later declaration must appear at the same level (in the same list of declarationsAndStatements) as the original declaration. This allows the declaration of recursively cyclic collections, for example, when a collection contains pointers to another collection which in turn contains pointers to the first collection. A collection whose element type is **forward** can be used only to declare pointers to it until the type's declaration is given. The forward type id is inaccessible until its declaration is given.

Elements of a collection are created and deleted dynamically using the statements **new** and **free**; see "Types and Type Declarations" for an example. The statement "**new** C,p" creates a new element in the collection C and sets p to point to it; however, if, due to resource exhaustion, the new element cannot be created, p is set to the value *nil*(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.

Suppose pointer q is equal to pointer p and the element they locate is deleted via "**free** C,p". In that case we say that q 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 uninitialized value; it cannot be assigned, compared, used as a collection subscript, or passed to **free**.

The value *nil*(C) is the null pointer for the collection.

Collections cannot be assigned, compared, passed as parameters, bound to, or named by a **const** declaration. Collections must not be declared in subprograms.

2.5 Bind Declarations

A *variableBinding* is:

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

A *variableBinding* is used to give a new name to a variable (or a part of a variable). This declares an identifier that is itself considered to be a variable. The new variable is considered to be "read-only"unless preceded by **var** (see "Restrictions on Constants and Read Only Items").

Example:

```
    bind var x to a(i), y to r.j
```

This declares x and y, which are considered to be variables; x is essentially an abbreviation for a(i) and y is essentially an abbreviation for r.j; y is read-only. Changing the value of i during the scope of the **bind** does not change the value denoted by x. In order to avoid aliasing, variable a is inaccessible and r is read-only until the end of the scope of the **bind**.

A *variableBinding* cannot occur as a declaration in the (main) program, except nested inside constructs, such as subprograms and the **begin** statement. A module must not contain as one of its fields a *variableBinding*. (This restriction is made to prevent re-entry into the scope of an existing **bind**.)

Turing does not allow aliasing. Hence, the "root" identifier of the *variableReference* (the first identifier in the reference) of a **var bind** becomes inaccessible for the scope of the binding. Even though the root identifier is inaccessible, it cannot be redeclared in its scope. See also "Restrictions Preventing Aliasing". To allow binding to different parts of a variable, each root identifier remains accessible until the end of the list of bindings in the variableBinding. The new bound identifiers do not become visible until the end of this list. In order to avoid aliasing, bindings that are var to different parts of the same variable must be non-overlapping.

Turing has been designed so that each bind can be implemented "by reference" (by using the address of the target variable). An implementation that allows aliasing should warn when aliasing is present and should implement binds by reference.

3 Types

3.1 Types and Type Declarations

A *typeDeclaration* is:

type [**pervasive**] id: *typeSpec*

A *typeDeclaration* gives a name to a type. The type name can subsequently be used in place of the full type definition.

Named types may optionally be declared **pervasive**. Type names declared using pervasive are visible in all subconstructs of the scope in which they are declared. Such types need not be explicitly imported.

A *typeSpec* is one of the following:

a. *standardType*
b. *subrangeType*
c. *enumeratedType*
d. *arrayType*

e. *setType*
f. *recordType*
g. *unionType*
h. *pointerType*
i. *namedType*

A *standardType* is one of:
a. int
b. real
c. boolean
d. string [(compileTimeExpn)]

The standard types are always visible in all scopes, and can not be explicitly imported.

The optional compile-time expression in a string type is a strictly positive integer value giving the string's maximum length. If the string's maximum length is omitted, the string is (ideally) considered to have no limit on its length. However, it is expected that most implementations will impose a standard default limit. This limit is recommended to be at least 255.

Example:

```
   var s: string := "Hello there"
```

Parameters can be declared to be dynamic strings, with maximum lengths declared as "*"; see "Subprograms".

A scalar type is an integer, real, boolean, enumerated type, subrange or pointer. The non-scalar types are: strings, sets, arrays, records, and union types.

An *index type* is a subrange, enumerated type or a named type which is an index type. Index types can be used as array subscripts, as selectors (tags) for case statements and union types, and as the base types of sets.

A *subrangeType* is:

compileTimeExpn .. *expn*

The two expressions give the lower and upper bounds of the range of values of the type. The two expressions must be both integers or both of the same enumerated type. The lower bound must be less than or equal to the upper bound. The second expression must be a compile-time expression in all cases except when it gives the upper bound of a dynamic array being defined in a variableDeclaration. (See "Variable and Collection Declarations".)

Example:

```
    var i: 1..10 := 2      % i can be 1,2 ... up to 10
```

An *enumeratedType* is:

enum (id {, id })

The *enumeratedType* declares an ordered sequence of identifiers whose associated values are distinct, contiguous and increasing. See the definitions of the ord, succ, and pred functions in "Predefined Functions".

Example:

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

The values are denoted by the name of the enumerated type followed by a dot followed by one of the enumerated identifiers; for example: *color.green*. Enumerated types and their subranges are index types.

An *arrayType* is:

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

Each *indexType* must be a subrange type, an enumerated type or a named type which is an indexType. Note that variables and parameters can be declared to be dynamic arrays, with run-time upper bounds; see "Subprograms". A dynamic array type must not be given a name, and must not occur in a record, union or collection. Each *indexType* gives the range of a subscript. The *typeSpec* gives the type of the elements of the array.

Elements of an array can be referenced using subscripts (see "Variables and Constants") and themselves used as variables or constants. Arrays can be assigned (but not compared) as a whole.

A *setType* is:

set of *indexType*

The *indexType* is called the *base type* of the set. An implementation may limit the number of items in the base type; this number will be at least 31. A variable of a set type can be assigned as value subsets of the entire set. See "Set Operators and Set Constructors".

Example:

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

A *recordType* is:

```
record
   id {, id } : typeSpec [ ; ]
 { id {, id } : typeSpec [ ; ] }
end record
```

Variables declared using a record type have the fields given by the declarations in the *recordType*. Fields of a record may be referenced using the dot operator (see "Variables and Constants") and themselves used as variables or constants. Record variables can be assigned (but not compared) as a whole.

A *union type* (or *variant record*) is like a record in which there is a run-time choice among sets (called alternatives) of accessible fields. This choice is made by the tag statement, which deletes the current set of fields and activates a new set.

A *unionType* is:

```
union [ id ]: indexType of
    label compileTimeExpn {, compileTimeExpn } : { id {, id } : typeSpec [ ; ] }
  { label compileTimeExpn {, compileTimeExpn } : { id {, id } : typeSpec [ ; ] } }
  [ label : { id {, id } : typeSpec [ ; ] } ]
end union
```

Example:

```
type vehicle: enum (passenger, farm, recreational)

type vehicleRecord:
    union kind: vehicle of
        label vehicle.passenger:
            cylinders: 1..16
        label vehicle.farm:
            farmClass: string (10)
        label:
             % No fields for "otherwise" alternative
    end union

var v: vehicleRecord := init (vehicle.farm, "dairy")

% Set tag and farmClass
tag v, vehicle.passenger      % Activate passenger alternative
v.cylinders := 6
```

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 a non-checking implementation would not need to represent it at run-time.

Each expression following **label** is called a *label value*. These must be distinct compile-time expressions in a given union type. Each label value must be assignable to the tag's type, which is an index type. Each labeled set of declarations is called an *alternative*. The final optional alternative with no label expressions is called the *otherwise alternative*.

An implementation may limit the range of the tag's type; the range limit, from the minimum label value to the maximum inclusive, will be at least 256.

The fields and tag of a union may be referenced using the dot operator (see "Variables and Constants"), and the fields can be used as variables or constants. Access or assignment to a field of an alternative is allowed only when the tag's value matches one of the alternative's label values (or matches none of the union's label values for the otherwise alternative). A checking implementation must guarantee this match at each access or assignment to a field. A tag cannot be assigned to and must not be the object of a **var** bind nor passed to a **var** parameter. A union's tag value can be changed using the **tag** statement (see "Statements"). In a checking implementation, the **tag** statement will actually change (uninitialize) any existing field values. A non-checking implementation will not necessarily change the fields. Note that union types under a checking implementation are "type safe"; so, changing the tag will not automatically change values of one alternative to be values of another alternative. A union's tag can also be changed by assigning to the entire union.

The identifiers declared as fields of a record, tag and fields of a union type, or values of an enumerated type must be distinct from each other. However, they need not be distinct from other visible identifiers. Fields of records and unions must not be dynamic arrays.

A *pointerType* is:

pointer to *collectionId*

Variables declared using a *pointerType* are pointers to dynamically created and deleted elements of the specified collection; see "Variable Declarations". Pointers are used as subscripts of the specified collection to select the element to which they point. The selected element can be used as a variable or constant. Pointers may be assigned, compared for equality and passed as parameters.

Example:

```
var list: collection of forward node

type node:
    record
        contents: string (10)
        next: pointer to list
    end record

var first: pointer to list := nil (list)
var another: pointer to list

new list, another      % Create new list element
list (another).contents := "Belgium"
list (another).next := first
first := another
```

The **forward** directive can be avoided in self-referencing collections, as this example illustrates:

```
var list: collection of
    record
        contents: string (10)
        next: pointer to list
    end record
```

That is, the name of a collection is visible inside the collection.

A *namedType* is:

[*moduleId* .] *typeId*

The *typeId* must be a previously declared type name. Type names exported from a module are referenced using the dot operator.

3.2 Type Equivalence and Assignability

This section defines the terms *type equivalence* and *type assignability*. Roughly speaking, an actual parameter can be passed to a var formal parameter only if their types are *equivalent*, and a value can be assigned to a variable (or a value passed to a non-var parameter) only if the value's type is *assignable* to the variable's type. Type equivalence can be determined at compile time; assignability sometimes cannot be determined until run time (when the target type is a subrange or a string with a maximum length).

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

(a) the same standard type,
(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
(f) 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, and so on.)

Outside of the exporting module M an ***opaque*** *type* with identifier T is not equivalent to any other type (but M.T is equivalent to type identifiers declared outside M that name M.T). By contrast, if type U is exported non-opaque from M, then type M.U is equivalent to the type that U names inside M. The parameter or result type of an exported subprogram or an exported constant is considered to have type M.T outside of M iff the item is declared using the type identifier T. The opaque type M.T is distinct from any other type that M exports or imports. A value of opaque type M.T can be assigned, but cannot be compared, subscripted, or field selected, and cannot be an operand of any operator. All that can be done with a value of an opaque type is to assign it or to pass it as a subprogram parameter. These rules for determining if opaque type M.T is equivalent to another type U imply that one never needs to look inside module M to see if M.T is equivalent to U; in other words, opaque type M.T is a new, distinct type created by module M. See "Modules" for an example using opaque types.

Each textual instance of a type definition for an enumerated, record or union type creates a new type that is not equivalent to any other type definition.

The int type is not considered to be equivalent to any integer subrange. The string type without explicit maximum length is not considered to be equivalent to any string type having an explicit maximum length.

The *root type* of any integer expression is int, that of any enumerated value is the defining enumerated type, that of any string expression is string (without specified maximum length), and that of any other value is the value's type. The root type of a named type is the root type of the type that is named. The root type of an integer subrange is int and that of an enumerated subrange is the original enum type. The root type of opaque type M.T is M.T.

Whenever a named value (variable, constant or function) is used where an expression is required, the type of the expression is considered to be the *root type* of the named value. For example,

```
var x: 1..10 := 5
y := x    % The type of expression x is int
```

A value is *assignable* to a type (called the *target type*) if

(a) the value's root type is equivalent to the target's root type, or
(b) the value is an integer and the target is real.

These two requirements can be enforced at compile time. In case (a) there is also a run-time requirement that a value assigned to a target subrange is contained in the subrange, and a value assigned to a target string does not exceed the target's maximum length. In case (b) the integer is implicitly converted to real. Throughout the language, wherever a real expression is required, an integer expression is allowed and is converted to real by an implicit call to the predefined function *intreal*.

The type of a reference passed to a var parameter must be equivalent to the formal's type. The type of an expression passed to a non-var (constant) parameter must be assignable to the formal's type. A dynamic actual array parameter (a parameter with run-time computed upper bounds) can be passed only to a formal array parameter with * declared for upper bounds. Similarly a dynamic actual string parameter can be passed only to string (*). See also the discussion of parameterType in "Subprograms". Dynamic strings can be assigned, but dynamic arrays cannot. In an assignment v := e, a variable initialization, a const declaration, or an initialization init v := e, e must be assignable to v and neither v nor e can be dynamic arrays.

Examples:

```
type smallint: 1..10
var t: smallint

var j: 1..100  % Variable j is assignable to i when in range 1..10

type smallarray: array 1..10 of real

var a: array smallint of real  % Equivalent to smallarray

type rec:
    record
        f: string
        g: real
    end record

var r1: rec
var r2: rec

var r3:      % Not equivalent to r1 and r2
    record
        f: string
        g: real
    end record
```

Variables i and j have the same root type (int), so one can be assigned to the other, given that the assigned value is in the target's declared range. Array a's type is equivalent to the type *smallarray*; so a could be assigned to a variable or passed to a formal parameter of type *smallarray*. The types of rl and r2 are equivalent and one can be assigned to the other. However, r3's type is not equivalent to the type of rl and r2.

Here is an example illustrating the equivalence rules for **opaque** types:

```
type T1: int

module m
   import (T1)
   export (T2, opaque T3, opaque T4, T5)
      type T2: T1
      type T3: T1
      type T4: T3
      type T5: T4
end m
```

Inside module M, types T1 through T5 are equivalent, but outside of M, types T1, M.T2 and M.T5 are equivalent but types M.T3 and M.T4 are distinct (opaque) types.

4 Subprograms and Modules

4.1 Subprograms

A *subprogram* is a *procedure* or a *function*.

A *subprogramDeclaration* is one of:

a. *subprogramHeader*
   [*importList*]
   subprogramBody

b. **forward** *subprogramHeader*
   forwardImportList

c. **body procedure** id
   subprogramBody

d. **body function** id
   subprogramBody

A *subprogramHeader* is one of:

a. **procedure** id [(*parameterDeclaration* {, *parameterDeclaration* })]

b. **function** id [(*parameterDeclaration* {, *parameterDeclaration* })] [id] : *typeSpec*

A *procedure* is invoked by a procedure call statement, with actual parameters if required. A *function* is invoked by using its name, with actual parameters if required, in an expression. If a subprogram p has no parameters, a call to it does not have any parentheses, i.e., the call is of the form "p" and not "p ()".

A procedure may return explicitly by executing a return statement or implicitly by reaching the end of the procedure body. A function must return via a result statement (see "Statements").

Subprograms may optionally take parameters, the types of which are defined in the header. The names of the parameters, as well as the name of the subprogram, are visible inside the subprogram, but not visible in the type specifications of the parameters and function result.

Parameters to a procedure may be declared using **var**, which means the parameter is considered to be a variable inside the procedure. An assignment to a **var** formal parameter changes the actual parameter, as well as the formal parameter. The element of an array passed to a parameter is not affected by changes to the subscript, for example, given the call:

```
p (a (i))
```

an assignment to i in the body of p does not affect p's actual parameter. Parameters declared without using **var** are considered to be constants. Functions are not allowed to have any side effects and cannot have **var** parameters.

The identifiers declared in a parameter list must be distinct from each other, from pervasive identifiers, from the subprogram name, and from the result identifier (if present). However, they need not be distinct from other identifiers visible outside of the subprogram.

A *parameterDeclaration* is one of:

a. [**var**] id {, id } : *parameterType*

b. *subprogramHeader*

Form (a) is used to declare formal parameters that are var (variable) or non-var (constant). An actual parameter that is passed to a var formal must be a variable (a reference) whose type is equivalent to the formal's type. An actual parameter that is passed to a non-var formal parameter must be a value that is "assignable" to the format's type. (See "Type Equivalence and Assignability".)

A *reference* parameter is any non-scalar or **var** parameter. Parameters that are not reference parameters are called *value* parameters; a value parameter is any scalar non-**var** parameter. See "Restrictions Preventing Aliasing" for constraints on the use of **var** and reference parameters.

Example:

```
function odd (i: int) : boolean     % i is a value parameter
    result (i mod 2) not= 0
end odd
```

Example:

```
var messageCount: int := 0

procedure putMessage (msg: string) % msg is a reference parameter
    messageCount := messageCount + 1
    put "Message number ", messageCount, ":", msg
end putMessage
```

The second kind of *parameterDeclaration*, form (b), specifies a *parametric* subprogram. The corresponding actual parameter must name a subprogram. The actual parameter subprogram must have parameters and result type equivalent to those of the formal parameter. Parametric subprograms are called like other subprograms. Example:

```
% Find zero of parametric function f
function findZero (function f (r: 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
```

A *parameterType* is one of:

a. *typeSpec*
b. string (*)
c. **array** *compileTimeExpn* .. * {, *compileTimeExpn* .. * } **of** *typeSpec*
d. **array** *compileTimeExpn* .. * {, *compileTimeExpn* .. * } **of** string (*)

In forms (c) and (d), the upper bounds of the index types of a dynamic array parameter are declared as “*”, in which case any array whose element type and index types' lower bounds are equivalent to the parameter's can be passed to the parameter. In forms (b) and (d) the maximum length of a dynamic string is declared as “*”. For **var** parameters or arrays of strings, the maximum length is taken to be that of the actual parameter. For non-array, non-**var** formal parameters, the type string (*) is taken to mean simply string. Note that multiple parameters declared using one dynamic parameter type do not necessarily have the same upper bounds and string maximum lengths; instead each parameter inherits the sizes of its actual parameter.

The upper bounds and maximum lengths of dynamic parameters can be accessed using the upper attribute; see "Attributes".

The Turing language has been designed so that value parameters can be passed "by value" (by passing expression values to the subprograms) and reference parameters can be implemented "by reference" (by passing addresses instead of values). Given that aliasing and side effects are prohibited (as nominally required by this Report), passing parameters by reference or by value-result will be logically equivalent. If an implementation allows aliasing or side effects it should warn that these are present and should use "pass by reference" exactly for those parameters designated as reference parameters in this Report.

An *importList* is:

import ([[**var**] id {, [**var**] id }])

A *forwardImportUst* is:

import ([[*varOrForward*] id {, [*varOrForward*] id }])

A *varOrForward* is one of:

a. **var**
b. **forward**

If a subprogram has an **import** list, it uses this to specify identifiers visible outside the subprogram that are to be visible in the subprogram's body. Identifiers not in the list will not be visible in the body. Pervasive identifiers need not be imported (they are implicitly imported). If a subprogram does not have an **import** list, it is considered to implicitly import all identifiers that textually appear as uses inside its body. An implicitly imported identifier is considered to be imported **var** if it is assigned to, bound using **var**, passed to a **var** parameter, or acted on by a **tag**, **new** or **free** statement inside the subprogram. The identifier is also considered to be imported **var** if it is a module's name and a procedure of this module is called in the subprogram.

Note that the subprogram name and the parameter names are always visible inside the subprogram, and must not appear in the **import** list. By contrast, a module name is not visible inside the module, and must not be imported.

Modules have import lists, with the same meaning as in subprograms, with the following difference: a module with no explicit import list is considered to have the empty import list: **import** ( )

Only identifiers that name variables or modules can be imported **var**. (See "Restrictions on Constants and Read Only Items.")

Functions are not allowed to have side effects and cannot import anything **var**. This restriction is transitive; hence a function cannot import a procedure that directly or indirectly imports anything **var**. Input/output is considered to be a side effect; hence functions cannot use **get** or **put** statements; they must not directly or indirectly call procedures that use these statements or that call the predefined input/output procedures. A parametric procedure that is passed as an actual parameter to a function must not directly or indirectly import anything **var** or directly or indirectly use the **get** and **put** statements or predefined input/output procedures. (Note that although Turing nominally prohibits function side effects, an implementation may extend the language to allow them given that appropriate warning messages are issued.)

An identifier must not be repeated in an import list. It is permissible to import pervasive identifiers and predefined identifiers, although it is redundant to do so.

When a subprogram p is passed as a parametric subprogram to subprogram q, any variables imported **var** directly or indirectly by p are considered to be var parameters passed to q. Any variables imported directly or indirectly non-**var** by p are considered to be read-only reference parameters passed to q. This is done to prevent potential aliasing as a result of the call.

The result type of a function is given by the *typeSpec* that follows the function's (optional) parameter declarations. The expression in a function's result statement must be assignable to the function's result type. Note that the result type can be a non-scalar, but must not be a dynamic array or dynamic string. The optional identifier preceding this *typeSpec* is the name of the function's result. This identifier can only be referenced in the function's **post** assertion.

A *subprogramBody* is:

> [**pre** *booleanExpn*]
> [**init** id := *expn* {, id := *expn* }]
> [**post** *booleanExpn*]
> *declarationsAndStatements*
> **end** id

The identifier following **end** must be the name of the subprogram. The **pre** expression must be true when the subprogram is called; the **post** expression must be true when it returns.

The **init** clause defines constants (the identifiers to the left of each assignment operator :=). These can only be accessed in the **post**, **assert** and **invariant** assertions.

A **forward** subprogram is a subprogram whose body declaration will be given later in its scope. (This is the only situation in which the keyword **body** is used as a prefix for a subprogram declaration.) The body declaration must appear at the same level (in the same list of *declarationsAndStatements*) as the **forward** declaration. The prefix **forward** in an **import** list can be applied only to subprograms. The use of forward in an import list refers to a subprogram declared later at the same level (in the same list of declarationsAndStatements).

Before a subprogram can be called and before its body appears, and before it can be passed as a parametric subprogram, its header as well as headers of subprograms directly or indirectly imported by it must have appeared. A function must not import a **forward** procedure. (This restriction is imposed to simplify checks for side effects in functions). Forward subprograms allow subprograms to be mutually recursive.

Example of mutual recursion:

```
% Evaluate an input expression e of the form t { + t } where
% t is of form p { * p } and p is of form ( e ) or is an
% explicit real constant.

% For example, the value of 1.5 + 3.0 * ( 0.5 + 13 ) halt is 15
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 "{ + t }"
       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 (nextValue)
        tValue := tValue * nextValue
    end loop
end term

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

The declaration of a subprogram or module must not appear inside a subprogram or statement. The declaration of a collection must not appear inside a subprogram.

4.2 Modules

A module defines a package of variables, constants, types, subprograms, and sub-modules. The interface of the module to the rest of the program is defined by its import and export clauses.

A *moduleDeclaration* is:

module id
  [*importList*]
  [**export** ([**opaque**] id {, [**opaque**] id })]
  [**pre** *booleanExpn*]
  { *declarationOrStatementInModule* }
  [**invariant** *booleanExpn*
   { *declarationOrStatementInModule* }]
  [**post** *booleanExpn*]
end id

A *declarationOrStatementInModule* is one of:

a. *declaration* [;]
b. *statement* [;]
c. *collectionDeclaration* [;]
d. *subprogramDeclaration* [;]
e. *moduleDeclaration* [;]

A module declaration is executed (and the module is initialized) by executing its declarations and statements. See "Terminology and Basic Concepts" for an example of a module. The identifier following the end of a module must be the module's name.

The *importList* gives identifiers visible outside the module that are to be visible inside the module. See the description of import clauses in "Subprograms".

Exported identifiers are identifiers declared inside the module which may be accessed outside the module using the dot operator. Unexported identifiers cannot be referenced outside the module. Only subprograms, constants and types can be exported. Variables and modules must not be exported. The **opaque** keyword can be used only to prefix names of types. Outside the module an opaque type is distinct from all other types; see "Type Equivalence and Assignability". An identifier must not be repeated in an export list.

Example:

```
module Complex     % Implements complex arithmetic

    export (opaque value, constant, add, ... other operations )

    % The "value" type is opaque, so information about the
    % representation of complex values is private to this module

    type pervasive value:  % "value" is visible throughout module
        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 for complex arithmetic go here ...

end Complex

var u,v: Complex.value := Complex.constant (1.0, 2.0)
var w: Complex.value := Complex add (u ,v)
```

See "Restrictions Preventing Aliasing" for constraints on reference parameters in calls to enter a module.

The module's **pre** expression must be true when execution of the module declaration begins. The **post** expression must be true when the initialization of the module (execution of its declaration) is finished. The initialization of the module must make the **invariant** expression true, and it must be true whenever an exported subprogram is called or returns. The **invariant** clause must appear before the headers of exported subprograms. It is good style to limit each invariant expression so it does not refer, directly or indirectly, to imported variables or modules; this style implies that the value of the expression cannot change except when the module is active.

Module declarations may be nested inside other modules but must not be nested inside subprograms. A module must not contain as one of its declarations a variableBinding.

4.3 Restrictions on Constants and Read Only Items

A variable or module is read-only in a subprogram or module into which it is imported non-var. An identifier declared non-var in a bind construct is also read-only. Exported procedures of a read-only module cannot be referenced (called or passed as parametric procedures).

All components of a constant are considered constant, and all components of read-only variables are considered read-only.

Constants and read-only variables are restricted as follows. They cannot be assigned to, bound to **var**, further imported **var**, or passed to **var** parameters. A constant or read-only union cannot be the object of a **tag** statement. A read-only collection cannot be the object of a **new** or **free** statement.

4.4 Restrictions to Prevent Aliasing

Given distinct visible identifiers x and y, aliasing is said to exist if a change to the value of variable x would change the value of y. Aliasing is possible only when variables are renamed. In Turing, renaming of variables occurs in only two constructs: reference parameters and bind. Aliasing is prevented by placing restrictions on these two constructs.

(Note that variables imported by a parametric subprogram p are considered to be reference parameters to the subprogram to which p is passed; see "Subprograms".)

To explain these restrictions, we first define the terms "direct importing" and "indirect importing”. A subprogram or module directly imports the items in its import clause. (A subprogram that does not have an explicit import clause is considered to have an implicit import clause giving the identifiers it actually accesses; see "Subprograms".) Each item is imported non-var (which means read-only for imported variables and modules) or **var**.

A subprogram or module p indirectly imports all items that are directly or indirectly imported by items directly imported by p. The direct and indirect imports of a read-only module are all considered to be non-var.

Note: when procedure p exported from module m is called from outside the module, we consider that p has the same import list as m. The same is true for exported functions except that all items are considered to be imported non-var.

Aliasing due to the first construct (reference parameters) is prevented by restrictions (a) and (b), as follows:

Restriction (a):
A (part of) a variable is not allowed to be passed to a **var** parameter if the called subprogram or module has another means of accessing (the same part of) the variable. This access can occur in two ways. The first is by a direct or indirect import of the variable by the called subprogram or module. The second is by passing (an overlapping part of) the same variable to another reference parameter in the same call.

Restriction (b):
A (part of) a variable is not allowed to be passed to a reference parameter if the called subprogram or module has another means of changing (the same part of) the variable. The possibility of changing the variable can occur in two ways. The first is by a direct or indirect **var** import of the variable by the called subprogram or module. The second is by passing (an overlapping part of) the same variable to another **var** parameter of the same call.

Aliasing due to the second construct (bind) is prevented by restrictions (c) and (d).

Restriction (c):
A **var** bind of y to x makes x inaccessible for the scope of y, and a non-var bind of y to x makes x read-only for the scope of y.

Restriction (d):
A var bind to x disallows calls to subprograms or modules that directly or indirectly import x, and a non-var bind to x disallows calls to subprograms or modules that directly or indirectly import x **var**.

Function calls never cause aliasing because functions cannot import variables **var** (either directly or indirectly) and cannot have **var** parameters.

These restrictions to prevent aliasing are necessary for proving the correctness of a Turing program using the formal definition of Turing. However, an implementation may extend the language by allowing violations of these restrictions, given that appropriate messages are issued.

5 Statements and Input/Output

DeclarationsAndStatements are:

{ *declarationOrStatement* }

A *declarationOrStatement* is one of:

a. *declaration* [;]
b. *statement* [;]
c. *variableBinding* [;]

A *statement* is one of:

a. *variableReference* := *expn*
b. *procedureCall*
c. **assert** *booleanExpn*
d. **return**
e. **result** *expn*
f. *ifStatement*
g. *loopStatement*
h. **exit** [**when** *booleanExpn*]
i. *caseStatement*
j. **begin**
        declarationsAndStatements
    end
k. **new** *collectionId*, *variableReference*
l. **free** *collectionId*, *variableReference*
m. *forStatement*
n. **tag** *variableReference*, *expn*
o. *putStatement*
p. *getStatement*

A declaration inside an **if**, **loop**, **case**, **for** or **begin** statement must not be a module or subprogram; all other kinds of declarations, including **bind**, are allowed.

Form (a) is an assignment statement. The expression is evaluated and the value assigned to the variable. The *variableReference* is a reference that refers to (part of) a variable; see "References". The expression must be assignable to the variable type; see "Type Equivalence and Assignability".

Form (b) is a procedureCall, which is a reference of the form:

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

An exported procedure is called outside the module in which it was declared using the dot operator. See "Subprograms" and "Modules".

Form (c) is an **assert** statement. The boolean expression must be true whenever the assert statement is executed. A checking implementation evaluates the assertion at runtime and aborts the program if it is false.

Form (d) is a **return** statement, which causes immediate return from a program or a procedure. A program or procedure returns either via a return statement or implicitly by reaching the end of the program or procedure. Functions and module bodies may not contain **return** statements.

Form (e) is a **result** statement, which can only appear in a function and causes immediate return from the function giving the function's value. The result expression must be assignable to the result type of the function given in the function's header. Execution of a function must conclude by executing a **result** statement and not by reaching the end of the function.

Form (f), an *ifStatement* is:

```
  if booleanExpn then
     declarationsAndStatements
 { elsif booleanExpn then
     declarationsAndStatements }
 [ else
     declarationsAndStatements ]
  end if
```

The boolean expressions following the keyword **if** and each **elsif** are successively evaluated until one of them is found to be true, in which case the corresponding statements following **then** are executed. If none of the expressions evaluates to true, then the statements following **else** are executed; if no **else** is present then execution continues following the **end if**.

Form (g), a *loopStatement* is:

```
  loop
     [ invariant booleanExpn ]
     declarationsAndStatements
  end loop
```

The statements within the loop are repeated until terminated by one of its **exit** statements or an enclosed **return** or **result** statement. The boolean expression in the invariant must be true whenever execution reaches it; a checking implementation will abort if it is false.

Form (h) is a loop **exit**. When executed, it causes an immediate exit from the nearest enclosing **loop** or **for** statement. The optional boolean expression makes the exit conditional. If the expression evaluates to true then the exit is executed, otherwise execution of the loop continues. An exit statement can appear only inside loop and for statements.

Form (i), a *caseStatement* is:

case *expn* **of**
  label *compileTimeExpn* {, *compileTimeExpn* } :
    declarationsAndStatements
  { **label** *compileTimeExpn* {, *compileTimeExpn* } :
    declarationsAndStatements }
  [**label** :
    declarationsAndStatements]
end case

The optional final clause with no expression between **label** and : is called the *otherwise alternative*. The **case** expression is evaluated and used to select one of the alternatives for execution. The selected alternative is the one having a label whose value equals the case expression. If the case expression value does not equal any of the label values then the otherwise clause is executed. If no otherwise alternative is present, the case expression must equal one of the label values. When execution of the selected alternative is completed, execution continues following the **end case**, not to the next alternative.

The root type of each label must be int or an index type and must be equivalent to the root type of the case selector expression. Label expressions must be compile-time expressions. Label values in a **case** statement must be distinct. An implementation may limit the range of case label values to insure efficient code; this range, from the minimum label value to the maximum, will be at least 256.

Form (j) is a **begin** statement. Begin statements can be used to limit the scope of declarations.

Forms (k) and (l) are **new** and **free** statements, for creating and deleting elements of a collection respectively (see "Variable and Collection Declarations").

Form (m), a *forStatement* is one of:

a. **for** [id] : *forRange*
    [**invariant** *booleanExpn*]
    declarationsAndStatements
  end for

b. **for decreasing** [id] : *expn* .. *expn*
    [**invariant** *booleanExpn*]
    declarationsAndStatements
  end for

The statements enclosed in the **for** statement are repeated for the specified range, or until the loop is terminated by one of its **exit** statements or an enclosed **return** or **result** statement.

A *forRange* is one of:

a. *expn* .. *expn*
b. *namedType*

Where the *namedType* must be a (non-opaque) subrange or enumerated type, and is not permitted if "**decreasing**" is present. Form (b) of *forRange* is equivalent to form (a) using the type's lower and upper values. The range is given by the value of the two expressions when the **for** statement is executed. The types of the two values must be of the same index type, or both of type int. For the first iteration, id has the left expression's value; for successive iterations, id is increased by one (or decreased by one if **decreasing** is present), until in the last iteration id equals the right value. If the left value exceeds the right (or is less than the right when **decreasing**), there are no iterations.

For each repetition, id is set to a new value in the range; these are contiguous values that are increasing, unless decreasing is specified in which case they are decreasing. The **for** statement is a declaration for id, which must be distinct from other visible identifiers. The scope of id is from the beginning to the end of the **for** statement. If the id is not present, the **for** statement behaves the same way, except that the value corresponding to the id cannot be accessed.

For each repetition, id is a constant and its value cannot be changed. The boolean expression in the **invariant** must be true whenever execution reaches it; a checking implementation will abort if it is false.

Statement form (n) is a **tag** statement. The variable of the statement must be a union. The union's tag is changed to be the value of the expression, which must be assignable to the tag's type. See "Types and Type Declarations" for further description and an example of usage.

5.1 Input and Output

Put and get statements are used to read/write items to/from streams (sequential files of characters).

A *putStatement* is:

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

A *streamNumber* is a non-negative integer expression. Omitting the stream number from **put** or **get** statements results in the default input stream (stdin) for **get** and the default output stream (stdout) for **put**. (See also "Predefined Procedures" for open and close, which are used to associate streams with stream numbers.) The written or read items must be strings or numbers (integer or real). The default input and output streams cannot be selected using a *streamNumber*. There is a run-time constraint that a particular stream can be read from or written to, but not both.

By convention stream 0 is considered to be a special error output stream (stderr). By convention the streams numbered 1 to n are attached to files specified externally by the user as command line arguments when the program is run.

Since functions cannot have side effects, they are not allowed to contain **get** and **put** statements or to directly or indirectly call procedures that contain **get** or **put** statements.

A *putItem* is one of:

a. *expn* [: *widthExpn* [: *fractionWidth* [: *exponentWidth*]]]
b. **skip**

From left to right in a **put** statement, either the expn's value of the *putItem* is appended as text to the output stream, or **skip** starts a new line. A new line is also started at the end of the list of *putItems*, unless the list is followed by "..", in which case this new line is not started. This allows the next **put** statement to continue the current output line.

If the *widthExpn* is omitted, then the value is printed in a field just large enough to hold the value. The *fractionWidth* and *exponentWidth* are allowed only for integer and real values.

For string value s, integer value i and real value r, the text output for the *putItem* given on the left is defined by the string expression on the right:

```
s : w               s + repeat (" ", w -length (s))
i                   intstr (i, 0)
i : w               intstr (i, w)
r                   realstr (r, 0)
r : w               realstr (r, w)
r : w : fw          fealstr (r, w ,fw)
r : w : fw : ew     erealstr (r, w ,fw, ew)
```

See "Predefined Functions" for definitions of the functions used on the right.

Example **put** statements and their output:

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`	Assumes fwdefault = 6
`put sqrt(2)`	`1.414214`	
`put 4.86,10**9`	`4.86e9`	Exponent printed for >= 1e6
`put 121:5`	`bbl21`	Width of 5; "b" is a blank
`put 1.37:6:3`	`b1.370`	Fraction width of 3
`put 1.37:11:3:2`	`bbl370e+00`	Exponent width of 2
`put "O'Brian"`	`O'Brian`	
`put "X=", 5.4`	`X=5.4`	
`put "XX": 4, "Y"`	`XXbbY`	Blank shown here as "b"

A *getStatement* is:

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

A *getItem* is one of:

a. *variableReference*
b. **skip**
c. *variableReference* : *
d. *variableReference* : *widthExpn*

Forms (a) and (b) support token-oriented input, in which white space is ignored; (c) supports whole line-oriented input, and form (d) supports individual character-oriented input. In form (a) the *variableReference's* root type must be integer, real or string, while forms (c) and (d) allow only strings. The value read into a string must not contain an *eos* or *uninitchar* character (see "Identifiers and Explicit Constants").

Form (a) first skips white space (defined as the characters blank, tab, form feed, new line, and carriage return); then it reads the sequence of non-white space characters as a token. A token consists of either (1) one or more non white space characters, up to but not including either a white space character or end of file, or else (2) if the token's first character is a quote ("), then an explicit string constant. (See also "Identifiers and Explicit Constants".) Explicit string constants can only be input for string *variableReferences*. When the *variableReference* is a string, the value of the explicit string constant or the characters of the token are assigned to the variable. If it is an integer, the predefined function *strint* converts the token to an integer before assigning to the variable. Analogously for reals, *strreal* converts the token to real before assigning it to the variable. It is an error to use form (a) if no token remains in the stream.

In form (b), the **skip** option skips white space, stopping when encountering any non-white space character or end of file. This option is used to detect whether further tokens exist in the input; if no more tokens exist in the input, all characters of the file are skipped and the *eof* predefined function becomes true.

The following input stream:

```
    Alice 216 "World champion"
```

is used in this example:

```
    var name, fame: string
    var time: int
    get name, time, fame
    % name = "Alice", time = 216, and fame = "World champion"
```

Example:

```
    % Read and sum a sequence of numbers
    var sum: real := 0.0
    var x: real
    loop
        get skip          % Skip to eof or next token
        exit when eof     % eof is explained in "Predefined Functions"
        get x
        sum := sum + x
    end loop
    put "Sum is: ", sum
```

Form (c) reads the rest of the characters of the current input line (not including the trailing new line character) and assigns them to the *variableReference*, which must be a string. The trailing new line character is read and discarded. (Note: it may be that the final line of a stream is not terminated by a new line character; in this case form (c) reads the remaining characters to end of file.) It is an error to use form (c) if no characters remain in the stream (i.e., if *eof* is true for the stream).

Form (d) is similar to form (c) except (1) at most *widthExpn* (a non-negative integer) characters are read, (2) the new line character at the end of a line is part of the string assigned to the *variableReference*, and (3) attempting to read past the end of stream is allowed and returns the remaining characters (if any, possibly returning the null string).

Example:

```
var s, t, u: string
get s : *       % Reads entire first input line,
                %   discarding trailing new line character
get t : 20      % Reads at most 20 characters; t may end with "\n"
get u : 1       % Reads next single char (or null string for eof)
```

Example:

```
% Read and print entire input stream a line at a time
var line: string
loop
    exit when eof
    get line : *      % Read entire line
    put line
end loop
```

Example:

```
% Read and print entire input stream a character at a time
var c: string (1)
loop
    exit when eof
    get c : 1      % "\n" is read into c as a character
    put c ..       % Output lines are ended when c="\n"
end loop
```

6 References and Expressions

6.1 References

The syntax for a *reference* includes variable references and constant references, as well as procedure call statements, function calls, values of enumerated types, attributes, and parametric subprograms. A *variableReference* is a *reference* that denotes a variable or part of a variable.

A *variableReference* is a:

reference

A *reference* is:

[*moduleId* .] id { *componentSelector* }

Where a *componentSelector* is one of:

a. (*expn* {, *expn* })
b. . id

Form (a) of *componentSelector* allows subscripting of arrays and collections. The value of each array subscript expression must be in the declared range of the corresponding index type of the array. The number of array subscripts must be the same as the number of index ranges declared for the array. A collection must have exactly one subscript and this must be a pointer to the collection.

Form (a) also allows calls to functions. The number of expressions must be the same as the number of declared parameters of the function. Each expression must be assignable to the corresponding formal parameter of the function.

Form (b) of componentSelector allows field and tag selection for records and unions. (Fields of a record or variable and a union's tag are referenced using the dot operator). It also allows access to items exported from a module.

A value of an enumerated type is a special case of form (b), namely, id.id, where the first id is the name of the type and the second id must be one of the identifiers given in the **enum** type definition.

6.2 Expressions

An *expn* (expression) represents a calculation that returns a value. A *booleanExpn* is an *expn* whose value is *true* or *false*.

Turing is a strongly typed language, meaning that there are a number of constraints on the ways values can be used. The following sections explain how values are mapped by operators to produce new values.

An *expn* is one of the following:

a. *reference*
b. *explicitConstant*
c. *substring*
d. *setConstructor*
e. *expn infixOperator expn*
f. *prefixOperator expn*
g. (*expn*)

Form (a) includes (1) references to constants and variables including subscripting and field and tag selection, (2) function calls and (3) values of enumerated types. See “References". Form (b) includes explicit boolean, integer, real and string constants; see "Identifiers and Explicit Constants". Form (c) is a substring operation; see "String Operators and Substrings". Form (d) is a set constructor; see "Set Operators and Constructors".

In form (e), an *infixOperator* is one of:

a. +	(integer and real addition; set union; string concatenation)
b. -	(integer and real subtraction; set difference)

c. *	(integer and real multiplication; set intersection)
d. /	(real division)
e. **div**	(truncating integer division)
f. **mod**	(remainder)
g. **	(integer and real exponentiation)
h. <	(less than)
i. >	(greater than)
j. =	(equal)
k. <=	(less than or equal; subset)
l. >=	(greater than or equal; superset)
m. **not**=	(not equal)
n. **and**	(boolean conjunction)
o. **or**	(boolean inclusive or)
p. ->	(boolean implication)
q. **in**	(member of set)
r. **not in**	(set non-membership)

In form (f), a *prefixOperator* is one of:

a. +	(integer and real identity)
b. -	(integer and real negation)
c. **not**	(boolean negation)

The order of precedence is among the following classes of the operators, in decreasing order of precedence:

1. **
2. prefix +, -
3. *, /, **dlv**, **mod**
4. infix +, -
5. <, >, =, <=, >=, **not**=, **in**, **not in**
6. **not**
7. **and**
8. **or**
9. ->

Expressions are evaluated according to precedence, left to right within precedence. Note that exponentiation is grouped from left to right.

For example, each expression on the left below is equal to the expression on the right.

```
a - b — c         a - (b - c)
-2**2             - (2**2)
a + b * c         a + (b * c)
x < y and b       (x < y) and b
b or c and d      b or (c and d)
```

6.3 Numeric Operators

The numeric (integer and real) operators are +, -, *, /, **dlv** (truncating division), **mod** (remainder) and ** (exponentiation).

The **dlv** operator is defined by:

x **div** y = trunc (a / b)

where "/" means exact mathematical division and *trunc* truncates to the nearest integer in the direction of zero. The result is of type int. The operands can be integer or real. Note that with real operands, **div** may produce an integer overflow.

The **mod** operator is defined by:

x **mod** y = x — (y * (x **div** y))

If x and y are both of root type int, the result type is int, otherwise the result is real. Note that **mod** applied to real operands is useful for range reduction; for example, for x > 0, *sin* (x) can be computed as *sin* (x **mod** (2*pi)). Note that **mod** with int operands never produces an overflow, but with real operands, it may produce a real underflow.

The / operator requires real or integer operands and produces a result of type real.

Whenever a real value is required, an integer value is allowed and is converted to real by an implicit call to the *intreal* predefined function; see "Predefined Functions". Note that this rule implies that the / operator can accept two integer operands, but both will be converted real. The operators +, -(infix and prefix), * and ** require integer or real operands; if one or both operands are real, the result is real, otherwise the result is int. The right operands of dlv and / must not be zero. If both operands of ** are of root type int, the right operand must not be negative. If the left operand is real and the right is of root type int, the right operand must be non-zero when the left is negative. If both operands are real, the right operand must be strictly positive when the left is negative and otherwise must be zero or positive.

Examples:

```
    7 / 2 = 3.5          -7 / 2 = -3.5
    7 dlv 2 = 3          -7 div 2 = -3
    7 mod 2 = 1          -7 mode 2 = -1
    7**2 = 49            -7**2 = -49
```

A checking implementation is expected to detect division and **mod** by zero, zero to the zero power, integer overflow, and real overflow and underflow.

6.4 Comparison Operators

The comparison operators are <, >, =, <=, >=, and **not**=. These operators yield a boolean result. Both operands of a comparison operator must have the same root type; see "Type Equivalence and Assignability". Only strings, sets, and scalars (values whose root type is int, real, boolean, enumerated or pointer) can be compared. Arrays, records and unions cannot be compared.

Booleans and pointers can be compared only for equality (= and **not**=). See "String Operators and Substrings" for a description of string comparison.

6.5 Boolean Operators

The boolean operators are **and** (conjunction), **or** (inclusive or), -> (implication) and **not**. These require boolean operands and return a boolean result. Note that a -> b has the same meaning as (**not** a) **or** b. The boolean operators are conditional; that is, if the result of the operation is determined by the value of the left operand then the right operand is not evaluated. In the following, division by zero is avoided, because the right operand of and is executed only if the left operand is true:

```
    if count not= 0 and sum / count > 60 then ...
```

6.6 String Operators and Substrings

The only string operator is + (concatenation); it requires string operands and returns a string result. An implementation may limit the allowed length of string values; this limit will be at least 25S.

The ordering of strings is determined by left to right comparison of pairs of corresponding characters until an end of string or a mismatch is found. See "Character Collating Sequence". The string with the greater of the mismatched characters is considered greater. If no mismatch is found and onestring is longer than the other, the longer string is considered greater. Note that strings of differing lengths are never considered to be equal, and there is no implicit "blank padding" of the ends of strings. The following function recursively defines the "greater than" string relation in terms of comparison of strings of length one.

```
    function greaterthan (s ,t: string) : boolean
        if length (s) = 0 or length(t) = 0 then
            result length (s) > length (t)
        elslf s (1) = t (1) then
            result greaterthan (s (2.. *), t (2.. *))
        else
            result j (1) > t (1)
        end if
    end greaterthan
```

The *length* predefined function returns the number of characters in a string value; see "Predefined Functions".

A substring selects a contiguous sequence of the characters in a string. For example, if L = 3, R = 5 and s = "string", then the substring of s from position L to position R, written s (L .. R), is “rin". A single character can also be selected, for example, s (L)="r", and in general, for any integer expression e, s (e) = s (e .. e).

String positions L and R can also be written as * [- expn], where the asterisk represents the length of the string. That is, position selector * [- expn] is an abbreviation for *length* (s) [- expn]. For example, if s = "string", then s (*) = "g", s (4 ..*) = "ing", and s (*-2 .. *—1) = "in".

The general form of a substring is:

stringReference (*substringPosition* [.. *substringPosition*])

where *stringReference* is a reference of type string, and each *substringPosition* is one of:

a. *expn*
b. * [- *expn*]

Each *substringPosition* expression must be of root type int.

The following restrictions apply to L and R:

L >= 1 and R <= *length* (*reference*) and R - L + 1 >= 0

Note that *length* (s (L .. R)) = R - L + l. Note that for L >= 1 and L <= length (s) + 1, s (L, L - 1) is the null string, i.e., the string of length zero. A substring is an expression (not a variable), and it so cannot be assigned to.

6.7 Set Operators and Set Constructors

The set operators are + (set union), - (set difference), * (set intersection), <= and >= (set inclusion), and in and not in (set membership). Sets can also be compared for equality using = and not=. The set operators +, - and * take operands of equivalent set types and yield a result of the same type. The set operators <= and >= take operands of equivalent set types and yield a Boolean result. The operators in and not in take a set as right operand and an expression of the set's base type as left operand. They yield a Boolean result.

A *setConstructor* is one of:

a. *setTypeReference* ( )
b. *setTypeReference* (**all**)
c. *setTypeReference* (*expn* {, *expn* })

Where each *setTypeReference* is a reference of the form [*moduleId* .] *setTypeId*.

Form (a) represents the empty set of the set type. Form (b) represents the complete set. Form (c) is a set containing the elements specified by the expressions, each of which must be of the base type of the set.

6.8 Compile-TIme Expressions

A *compile-time expression* is an expression whose value can, in principle, be computed at compile time. The following are compile-time expressions:

1. Explicit integer, real, boolean and string constants, as well as enumerated values of the form *id1.id2* where *id1* is the name of an enumerated type
2. Set constructors containing only compile-time element values (or **all**)
3. Named constants that name scalar compile-time expressions
4. The result of the integer operators prefix + and -, infix + and -, *, **dlv** and **mod** when the operands are compile-time integer expressions
5. The built-in functions *chr* and *ord* when the actual parameter is a compile-time expression
6. The result of the string concatenate operator (+) when both operands are compile-time string expressions

Note that a compile-time expression can be invalid, for example, 1/0, and is still considered to be a compile-time expression. Expressions that do not satisfy this definition are called *run-time expressions*.

6.9 Predefined Functions

The following are pervasive, predefined functions.

eof (i: int): boolean

Accepts a non-negative stream number (see description of **get** and **put** statements) and returns true if and only if there are no more characters in the stream. This function must not be applied to streams that are written to (using **put**). The parameter and parentheses can be omitted (i.e., as "*eof*"), in which case the stream is taken to be the default input stream.

pred (expn)

Accepts an integer or an enumerated value and returns the integer minus one, or the previous value in the enumeration. *Pred* must not be applied to the first value of an enumeration.

succ (expn)

Accepts an integer or an enumerated value and returns the integer plus one, or the next value in the enumeration. *Succ* must not be applied to the last value of an enumeration.

String Functions

length (s: string): int

Returns the number of characters in the string. The string must be initialized.

index (s, patt: string) : int

If there exists an i such that *s (i .. i + length (patt) - 1) = patt*, then the smallest such i is returned, otherwise zero is returned. Note that 1 is returned if *patt* is the null string.

repeat (s: string, i: int) : string

If i > 0, returns i copies of s concatenated together, otherwise returns the null string. Note that *for all j >= 0, length (repeat (t, j)) = j * length (t).*

Mathematical Functions

abs (expn)

Accepts an integer or real value and returns its absolute value. The type of the result is int if the *expn* is an of root type int; otherwise it is real.

max (expn, expn)

Accepts two numeric (real or integer) values and returns their maximum value. If both are of root type int, the result is an int; otherwise it is real.

min (expn, expn)

Accepts two numeric (real or integer) values and returns their minimum value. If both are of root type int, the result is an int; otherwise is is real.

sign (r: real) : -1 .. 1

Returns -1 if r < 0, 0 if r = 0, and 1 if r > 0.

sqrt (r: real) : real

Returns the positive square root of r, where r is a non-negative value.

sin (r: real) : real

Returns the sine of r, where r is an angle value expressed in radians.

cos (r: real) : real

Returns the cosine of r, where r is an angle value expressed in radians.

arctan (r: real) : real

Returns the arctangent (in radians) of r.

sind (r: real) : real

Returns the sine of r, where r is an angle expressed in degrees.

cosd (r: real) : real

Returns the cosine of r, where r is an angle expressed in degrees.

arctand (r: real) : real

Returns the arctangent (in degrees) of r.

ln (r: real) : real

Returns the natural logarithm (base e) of r.

exp (r: real): real

Returns the natural base e raised to the power r.

Type Transfer Functions

floor (r: real) : int

Returns the largest integer less than or equal to r.

ceil (r: real) : int

Returns the smallest integer greater than or equal to r.

round (r: real) : int

Returns the nearest integer approximation to r. Rounds to larger value in case of tie.

intreal (i: int) : real

Returns the real value equivalent of i. No precision is lost in the conversion,
so *floor (intreal (j)) = ceil (intreal (j)) = j*. To guarantee that these equalities hold,
an implementation may limit the range of i.

chr (i: int): string (1)

Returns the i-th character of the system's character collating sequence as a string of length one, where the first character corresponds to 0, the second to 1, and so on. See "Character Collating Sequence". The selected character must not be *uninitchar* (a reserved character used to mark uninitialized strings) or *eos* (a reserved character used to mark the end of a string). See "Identifiers and Explicit Constants".

ord (expn)

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 system's character collating sequence. Values of an enumerated type are numbered left to right starting at zero.
See "Character Collating Sequence".

intstr (i, width: int): string

Returns a string equivalent to i, padded on the left with blanks as necessary to a length of width; for example, *intstr* (14, 4) = "bb14" where b represents a blank.
The optional width parameter must be non-negative; if omitted, it is assumed to be 1.
If width is not large enough to represent the value of i, the length is automatically increased to the minimum needed. The string returned by *intstr* is of the form:

{ blank } [-] digit { digit }

The leftmost digit is always non-zero unless the value of i is zero, in which case there is a single zero digit.

strint (s: string) : int

Returns the integer equivalent of the string s. String s must consist of a possibly null sequence of blanks, followed by an optional plus or minus sign, and a sequence of one or more digits. Note that for integer i and non-negative w, *strint (intstr (i, w)) = i.*

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

Returns a string (including exponent) approximating r, padded on the left with blanks as necessary to a length of width; for example, *erealstr* (2.5e1, 9, 2, 2) = "b2.50e+01" where b represents a blank. The *width* must be a non-negative int value. If the width is not large enough to represent the value of r, it is implicitly increased to the minimum needed. The *fractionWidth* parameter is a 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, it is increased to the minimum 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 of the digits are zeroes.

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

Returns a string approximating r, padded on the left with blanks if necessary to a length of *width*. The number of digits of fraction to be displayed is given by *fractionWidth*; for example, *frealstr* (2.5e1, 5, 1) = "b25.0" where b represents a blank. The width must be non-negative. If the *width* parameter is not large enough to represent the value of r, it is implicitly increased to the minimum 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 } [-] digit { digit } . { digit }

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

realstr (r: real, width: int) : string

Returns a string approximating r, padded on the left with blanks if necessary to a length of *width*, for example, realstr (2.5e1, 4) = "bb25" where b represents a blank. 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 to the minimum 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 1e-3 <= abs (r) < 1e6, otherwise it is 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 part 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 by default; 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.

strreal (s: string) : real

Returns a real approximation to string s. String s must consist of a possibly null sequence of blanks, followed by an optional plus or minus sign and an explicit unsigned real or integer constant.

6.10 Attributes

There are pervasive attributes that are properties of variables rather than properties of values. For example, the "*upper*" attribute of a string variable gives its maximum length. Note that assigning a value to a variable does not change the variable's attributes. Example:

```
    var s: string (10) := "Eggs"
    var t: string (6) := "Bacon"
    s := t
```

At all times, *upper* (s) = 10 and *upper* (t) = 6. The available attributes are:

lower (reference [, dimension])

Accepts an array and returns the lower bound of the array.

upper (reference [, dimension])

Accepts an array and returns the upper bound of the array; also accepts a string and returns its maximum length.

In *lower* and *upper*, *dimension* is a compile-time integer expression, which is present iff the reference is a multi-dimensioned array. It specifies which dimension, where the first is 1, the second is 2 and so on. The reference does not need to have been assigned a value.

nil (collectionId)

Accepts a collection and returns the collection's null pointer.

6.11 Predefined Procedures

The following procedures are pervasive and predefined.

Randomization Procedures

rand (**var** r: real)

Sets r to the next value of a sequence of pseudo random real numbers that approximates a uniform distribution in the range 0 < r < 1.

Example:

```
    var r: real
    loop     % Randomly print a sequence of phrases
        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
```

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

Sets i 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*.

randomize

A parameterless procedure that resets the sequences of pseudo random numbers produced by *rand* and *randint*, so different executions of the same program may produce different results.

randnext (**var** v: real, seq: 1 .. 10)

This procedure is the same as *rand*, except that *seq* specifies one of 10 independent and repeatable sequences of pseudo random real numbers. This can be useful when trying to reproduce a randomly occurring bug.

randseed (seed: int, seq: 1 .. 10)

This 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 number.

Input / Output Procedures

(Note: these predefined procedures have been replaced by equivalent corresponding statements in Turing+.)

open (**var** streamNumber : int, fileName: string, mode: string)

The *filename* gives the name of a file that is to be read from or written to. The *streamNumber* parameter is set to the stream number to be used for the file in **get** or **put** statements. The *mode* must be "r" (for read) or "w" (for write) indicating whether the stream is to be read from or written to. If the open fails, *streamNumber* is set to zero.

close (streamNumber: int)

This procedure disassociates the stream number from the stream it is presently designating.

All of the predefined procedures (*rand, randint, randomize, randnext, randseed, open* and *close*) have side effects. As a result, functions are not allowed to call them directly or indirectly by calling procedures that call them.

6.12 The Uninitialized Value

The value of a scalar, string or set that is not initialized must not be used (fetched) in evaluating an expression. For example, before any of the following are executed, variable x must have been assigned a value.

```
    const c := x
    y := x + y
    p (x + y)
```

A scalar, string or set need not be initialized before being passed to a **var** (reference) parameter, but must be initialized before being passed to a non-var (value) parameter. These rules imply that once a particular scalar, string or set variable is initialized, it will stay initialized.

A variable that has been declared and not assigned to (and is not initialized in its declaration) is considered to be uninitialized. When an element of a collection is created by the **new** statement, it is uninitialized. All fields of a union become uninitialized when the **tag** statement is applied to the union. Part or all of an array, record or union variable may become uninitialized when the variable is assigned to, according to the initialization of the value being assigned.

When conditional evaluation of an expression does not require the value of a particular variable, the variable need not be initialized. For example, in the following, if i = 10 then x need not be initialized:

```
    exit when i = 10 or x = 5
```

A non-scalar that is not a string or set can be assigned, used as the value of a **const**, passed to a parameter (var or non-var) or returned as a function result without being initialized. Note: the non-scalars in question (i.e. non-scalars that are not strings or sets) are arrays, records and unions. Scalar, string or set components of non-scalar types must be initialized before being used (fetched).

An initialized component of a non-scalar can become uninitialized due to assigning to the entire containing non-scalar variable or changing the tag of the containing union.

A checking implementation of Turing is expected to enforce these restrictions on the use of uninitialized variables.

6.13 Character Collating Sequence

Certain Turing language features, notably string comparison and the *chr* predefined function, depend on the system's character collating sequence. This is the sequence that determines the ordering among character values. There are two widely used collating sequences: ASCII and EBCDIC. A Turing implementation is expected to use one of these, with preference given to ASCII. Note that a Turing program that is correct assuming one of these sequences is not necessarily correct assuming the other.

The *ord* function maps a character value to its corresponding ASCII or EBCDIC value, which will be in the range 0 .. 255. For standard ASCII characters, the range is limited to 0 .. 127. Therefore, subject to limits on the domain of *succ*, the following equations hold:

For all characters c:
  chr (ord (c)) = c, and *succ (c) = chr (ord (c) + 1)*

The ASCII and EBCDIC sequences share the important property that digits are contiguous.

For all c in the range "0" .. "8"
  succ (c) = chr (ord (c) + 1)

Therefore, if s is the string of length 1 corresponding to integer i then *ord* (s) = *ord* ("0") + i, for example, *ord* ("3") = *ord* ("0") + 3.

In ASCII, letter characters are also contiguous:

ord ("A") = *ord* ("B") - 1
ord ("B") = *ord* ("C") - 1
...
ord ("a") = *ord* ("b") - 1
ord ("b") = *ord* ("c") - 1
...

For example, the following function converts a string of digits to an integer.

```
function digitsint (s: string) : int
    const L := length (j)
    const digit := ord (s (L)) - ord ("0")
    if L = 1 then
        result digit
    else
        result 10 * digitsint (s (1 .. L-1)) + digit
    end if
 end digitsint
```

Unfortunately, in EBCDIC the letters are not contiguous; there are gaps between letters I and J and between R and S. The test to see if ASCII character c is a capital letter is:

```
"A" <= c and c <= "Z"
```

But for EBCDIC character c, we must use:

```
("A" <= c and c <= "I") or
    ("J" <= c and c <= "R") or
        ("S" <= c and c <= "Z")
```

Consult standard definitions of ASCII and EBCDIC collating sequences for more details.

7 Source Inclusion Facility

Other source files may be included as part of a program using the "include" construct.

An *includeConstruct* is:

include *explicitString*

The *explicitString* gives the name of a source file whose text is to be to be included in the compilation. The **include** construct is replaced in the program source by the contents (source text) of the specified file. Include constructs can appear anywhere in a program and can contain any valid source code fragment. Included source files can themselves contain include constructs.

8 Short Forms

The following forms can be used as alternatives for the syntax given in the language specification. These alternatives shorten frequently used constructs. Note: Not all Turing implementations support all short forms.

```
  Long form                          Short form

v := v + (expn)                    v += expn
v := v - (expn)                    v -= expn
v := v * (expn )                   v * = expn

if expn then statements            [ expn : statements
elsif expn then statements           | expn : statements
else statements                      | : statements
end if                             ]

case expn of                       case expn of
label labels : statements            | labels : statements
label : statements                   | : statements
end case                           end case

union id: typeSpec of              union id: typeSpec of
label labels : statements            | labels : statements
label : statements                   | : statements
end onion                          end union

loop                               {
    statements                         statements
end loop                           }

exit                               >>
exit when expn                     >>: expn

return                             >>>
result expn                        >>>: expn
```

```
for optionalId: expn .. expn   {+ optionalId: expn .. expn
    statements                         statements
end for                        }

for decreasing optionalId:     {- optionalId : expn .. expn
        expn .. expn
    statements                         statements
end for                        }

and                            &
not                            ~
put                            !
get                            ?
array indexTypes of            { indexTypes }
procedure                      proc
function                       fcn
pervasive                      *
```

Example using short forms:

Long form **Short form**

```
function gcd (i, j: int): int        fcn gcd (i, j: int): int
    var x: int := i                      var x := i
    var y: int := j                      var y := j
    loop                                 { >>: x = y
        exit when x = y                      [ x > y: x -= y
        if x > y then                          | :    y -= x
            x := x - y                       ]
        else                             }
            y := y - x                   >>>: x
        end if                       end gcd
    end loop
    result x
end gcd
```

9 Collected Keywords and Predefined Identifiers

Keywords of Turing:

all	**and**	**array**	**assert**	**begin**
bind	**body**	**boolean**	**case**	**collection**
const	**decreasing**	**div**	**else**	**elsif**
end	**enum**	**exit**	**export**	**false**
fen	**for**	**forward**	**free**	**function**
get	**if**	**import**	**in**	**init**
int	**invariant**	**label**	**loop**	**mod**
module	**new**	**not**	**of**	**opaque**
or	**pervasive**	**pointer**	**post**	**pre**
proc	**procedure**	**put**	**real**	**record**

result	**return**	**set**	**skip**	**string**
tag	**then**	**to**	**true**	**type**
union	**var**	**when**		

Predefined Identifiers of Turing:

abs	*arctan*	*arctand*	*ceil*	*chr*
close	*cos*	*cosd*	*eof*	*erealstr*
exp	*floor*	*frealstr*	*index*	*intreal*
intstr	*length*	*ln*	*lower*	*max*
min	*nil*	*open*	*ord*	*pred*
rand	*randint*	*randnext*	*randomize*	*randseed*
realstr	*repeat*	*round*	*sign*	*sin*
sind	*sqrt*	*strint*	*strreal*	*succ*
upper				

10 Collected Operators and Special Symbols

The operators and special characters of Turing are:

```
,    ..   .    :    ;    *    **
/    +    -    <    >    >>   >>>
=    <=   >=   ->   ~    !    ?
&    (    )    {    }    [    ]    |
```

Six of these are introduced by the short forms:

```
>>   >>>  ~    !    ?    &
```

Note that ~= is considered to be two tokens, ~ and =.

11 Recognizing Tokens

The tokens of the Turing program are recognized by scanning characters from left to right, skipping white space, until the beginning of a token is found. The token is recognized by maximal scanning meaning the token is extended on the right as long as the additional characters (potentially) form more of the token. White space, if any, is then skipped up until the beginning of the next token.

In the following fragment, "10" and "then" are distinct tokens even though they are not separated by white space:

```
if a > 10then     % Acceptable
```

In the next example, by maximal scanning "4e" is the (ill-formed) initial part of a real constant, so this fragment is not acceptable:

```
if a > 10then x := 4else     % Not acceptable
```

Turing has one exception to the maximal scanning rule, and this is: two adjacent dots are always considered to be the token "..". For example, in:

```
var a: array 1..10 of real
```

the fragment "1..10" is considered to be the three tokens "1", ".." and "10" and not the two tokens "1." and ".10".

12 Implementation Constraints on Integer, String, and Real Types

Ideally, there should be no implementation constraints on Turing programs (see description of the language and implementation constraints in "Terminology and Basic Concepts"). If there are no implementation constraints, then we call the language Ideal Turing. In Ideal Turing, the int type has an unbounded range of values, and the real type has infinite precision and infinite exponent range; that is, int and real correspond exactly to the mathematical concepts of the integers and the real numbers. Similarly, in Ideal Turing the type string (without an explicit length), comprises all sequences of characters, with no limit on length.

Programs written in Ideal Turing can be thought of as mathematical formulae, rather than as instructions for a computer. For example:

```
function factorial (i: int) : int
    pre i >= 0
    if i = 0 then
        result 1
    else
      result i * factorial (i-1)
    end if
end factorial
```

In Ideal Turing, this function gives a definition of "*factorial*" for all non-negative integers; but note that in a particular implementation of Turing, integer overflow will occur for large values of i.

In most practical implementations of Turing, int will be limited to a range of integers: *minint* .. *maxint*, and string lengths will be limited to *maxstr*. To support program portability, it is recommended that in all implementations, *minint* <= —(2**31—1), *maxint* >= 2**31-1, and *maxstr* >= 255. It is recommented that *eos* and *uninitchar* correspond, respectively, to the characters with ordinals 0 and 128 respectively.

In an implementation, each non-zero real r may be represented by a floating point number of the form:

r = f * *radix***e

where:

f is the significant digits part,
e is the exponent, and
radix is the number base of the representation.

It is assumed that f is normalized, i.e., if f is not zero then

1/*radix* <= *abs* (f) and *abs* (f) <= 1.0

If f is zero then e is also zero. The number of digits of precision of f (in the given *radix*) may be limited to *numdigits*.

In most practical implementations, the exponent e will be limited to the range *minexp* .. *maxexp*. To support program portability, it is recommended that the equivalent base 10 range of exponents be at least -38 .. -38, i.e., that *minexp* * *ln* (*radix*)/*ln* (10) <= -38 and *maxexp* * *ln* (*radix*)/*ln* (10) >= +38.

Floating point operators provide an approximate but repeatable result corresponding to the exact mathematical result. For operators +, -, * and / with operands x and y, let f be the floating point result and let m be the exact mathematical result. For non-zero m the relative round off error is defined as:

abs ((m-f) / m)

(If m is exactly zero, f should also be zero.) Each implementation of Turing is to specify the value of *rreb* (relative round off error bound) such that the round-off error for +, —, * and / never exceeds *rreb*. To support program portability, it is recommended that *rreb* be at most 1e-14.

For implementations using rounding floating point operators, such as the DEC VAX, *rreb* is:

rreb = 0.5 * *radix***(—*numdigits* + 1)

For implementations using chopping floating point operators, such as the IBM 370, *rreb* can be given as:

rreb = *radix***(—*numdigits* + 1)

Unfortunately, there are some implementations of floating point in which neither rounding or chopping is consistently carried out; in this case, *rreb* is larger than would be calculated by these formulas.

Each implementation of Turing will provide a standard include file called limits which will contain definitions of *minint*, *maxint*, maxstr, *radix*, *minexp*, *maxexp*, *numdigits* and *rreb*. This file can be included in a Turing program by writing **include** "*%limits*". This include file will also contain the definition of functions that access and modify exponents of floating point values:

getexp (r: real) : real

Returns exponent e of r. If r = 0, then e = 0.

setexp (r: real, e: int) : real

Returns value of r with the exponent changed to e.
The value of e must be in the range *minexp* ..*maxexp*.

13 External Subprograms

There is a language extension which allows Turing programs to call subprograms written in other languages. The syntax for this extension is:

external [*overrideName*] *subprogramHeader*

The optional *overrideName* must be an explicit string constant. When it is omitted, the name used for external linking is the subprogram's identifier. If the *overrideName* is present, it is used as the linking name. An implementation may limit the number of characters in the linking name. Conventions for order and method of parameter passing is implementation dependent.

14 Changes In the Taring Language

The Turing language as defined in this Report is slightly modified from the version of the original Turing Report. These modifications have been made so that certain features are more convenient for the user. This version of the Report is somewhat modified and expanded from the original Report to clarify certain descriptions. The description of Turing constructs, in terms of context-free rules, has been made to satisfy LR(1). The modifications are:

1. Upper and lower case in identifiers and keywords are now distinct.
   For example, *r* and *R* are now considered to be distinct identifiers.
   Keywords must now be used strictly in lower case.

2. Sets are now considered to be non-scalars instead of scalars. This allows efficient implementation of long sets, by allowing passing of sets by reference.

3. Collection names are now visible inside their declarations, so self-referencing collections need not use the forward directive.

4. The definitions of *realstr* and *frealstr* were modified so that with *realstr*, small values do not show their exponents. This means that the *putItem* i:w:f creates nice columns of numbers even if i < 10**-3.

5. The range of an increasing for statement can now be given as a named type (subrange or enumerated).

6. In a **for** loop, the (optional) **invariant** now must come before the loop body, instead of after.

7. The substring feature now allows each position to be selected by the form * [- expn].
   The asterisk represents the length of the string. For example, if s ="abc" then
   s (*) = "c" and s (1 ..*—1) = "ab"

8. The *round* function now rounds ties to the next larger integer instead of to an even value.

9. The *intstr* function now allows omission of the width parameter, so if i = 25, then *intstr* (i) = "25". Note that *intstr* (i) = *intstr* (i, 1).

10. It is now clarified that the dimension parameter of upper and lower is to be present iff the first parameter is a multi-dimensioned array.

11. The rules restricting use of ** (exponentiation) are clarified to the following three cases.

a. i**j	requires j >= 0, allows i < 0
b. x**i	allows x < 0 and i < 0
c. x**y, i**y	requires x >= 0 and i >= 0, allows y < 0

    where i and j are of root type int and x and y are real.

12. Semicolons are now allowed following declarations in records and unions.

13. The restriction that a module's invariant must not reference imported variables or modules has been removed.

14. Modules and subprograms can no longer be pervasive.

15. A bind can no longer be declared in the (main) program (except as nested in constructs such as subprograms and begin.)

16. Collections can no longer be declared in subprograms.

17. It is clarified that the forward type id of a collection is inaccessible until declared.

18. External subprograms are introduced as an extension to Turing.

19. The short form for import (:) is no longer defined.

20. Carriage return characters are now considered to be white space.

Appendix A

The Context-Free Syntax of Turing

In our notation, tokens (also called terminals) such as loop and procedure, are written in **boldface**, while non-terminals (called syntactic variables in the Algol 60 Report), such as *declaration* and *setType*, are written in *italics*. A set of production rules describes how to replace the non-terminal program by zero or more tokens to yield a token sequence which satisfies the context-free syntax of Turing.

A production rule consists of a single non-terminal (which names the production) and a list of sequences containing tokens and non-terminals, with the meaning that any occurrence of the non-terminal can be replaced by one of the sequences. A sequence in the production can contain the open and close brackets [and], and the open and close braces { and }, which enclose tokens and non-terminals that are optional (if enclosed in brackets) or can occur zero or more times (if enclosed in braces). That is, if *item* is a sequence of tokens, non-terminals, brackets, and braces, then:

[*item*] means that item is optional, and
{ *item* } means that item can occur zero or more times.

As an example, consider the following production rule:

An *importList* is one of:

a. **import** ( )
b. **import** ([**var**] id {, [**var**] id })

If id can be replaced by an identifier, then the following are some of the token sequences described by this rule.

```
import ( )
import ( push, pop, top )
import ( var element, left, var right)
```

Note that this definition of the context-free syntax omits the short forms of Turing.

1 Programs and Declarations

A *program* is:

{ *declarationOrStatementInMainProgram* }

A *declarationOrStatementInMainProgram* is one of:

a. *declaration* [;]
b. *statement* [;]
c. *collectionDeclaration* [;]
d. *subprogramDeclaration* [;]
e. *moduleDeclaration* [;]

A *declaration* is one of the following:

a. *constantDeclaration*
b. *variableDeclaration*
c. *typeDeclaration*

A constantDeclaration is one of:

a. **const** [**pervasive**] id := *expn*
b. **const** [**pervasive**] id : *typeSpec* := *initializingValue*

An *initializingValue* is one of:

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

A *variableDeclaration* is one of:

a. **var** id {, id } := *expn*
b. **var** id {, id } : *typeSpec* [:= *initializingValue*]

A *collectionDeclaration* is one of:

a. **var** id {, id } : **collection of** *typeSpec*
b. **var** id {, id } : **collection of forward** id

A *variableBinding* is:

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

2 Types

A *typeDeclaration* is:

type [**pervasive**] id : *typeSpec*

A *typeSpec* is one of the following:

a. *standardType*
b. *subrangeType*
c. *enumeratedType*
d. *arrayType*
e. *setType*
f. *recordType*
g. *unionType*
h. *pointerType*
i. *namedType*

A *standardType* is one of:

a. int
b. real
c. boolean
d. string [(*compileTimeExpn*)]

A *subrangeType* is:

compileTimeExpn .. *expn*

An *enumeratedType* is:

enum (id {, id })

An *arrayType* is:

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

A *setType* is:

set of *indexType*

An *indexType* is one of:

a. *subrangeType*
b. *enumeratedType*
c. *namedType*

A *recordType* is:

```
record
   id {, id } : typeSpec [ ; ]
 { id {, id } : typeSpec [ ; ] }
end record
```

A *unionType* is:

```
union [ id ] : indexType of
   label compileTimeExpn {, compileTimeExpn } : { id {, id } : typeSpec [ ; ] }
 { label compileTimeExpn {, compileTimeExpn } : { id {, id } : typeSpec [ ; ] } }
 [ label : { id {, id } : typeSpec [ ; ] } ]
end union
```

A *pointerType* is:

pointer to *collectionId*

A *namedType* is:

[*moduleId* .] *typeId*

A *collectionId, moduleId,* or *typeId* is an:

id

3 Subprograms and Modules

A *subprogramDeclaration* is one of the following:

a. *subprogramHeader*
   [*importList*]
   subprogramBody

b. **forward** *subprogramHeader*
   forwardImportList

c. **body procedure** id
   subprogramBody

d. **body function** id
   subprogramBody

A *subprogramHeader* is one of:

a. **procedure** id [(*parameterDeclaration* {, *parameterDeclaration* })]

b. **function** id [(*parameterDeclaration* {, *parameterDeclaration* })] [id] : *typeSpec*

A *parameterDeclaration* is one of:

a. [**var**] id {, id } : *parameterType*
b. *subprogramHeader*

A *parameterType* is one of:

a. *typeSpec*
b. string (*)
c. **array** *compileTimeExpn* .. * {, *compileTimeExpn* .. * } **of** *typeSpec*
d. **array** *compileTimeExpn* .. * {, *compileTimeExpn* .. * } **of** string (*)

An *importList* is:

import ([[**var**] id {, [**var**] id }])

A *forwardImportList* is:

import ([[*varOrForward*] id {, [*varOrForward*] id }])

A *varOrForward* is one of:

a. **var**
b. **forward**

A *subprogramBody* is:

    [**pre** *booleanExpn*]
    [**init** id := *expn* {, id := *expn* }]
    [**post** *booleanExpn*]
    declarationsAndStatements
end id

A *moduleDeclaration* is:

module id
    [*importList*]
    [**export** ([**opaque**] id {, [**opaque**] id })]
    [**pre** *booleanExpn*]
    { *declarationOrStatementInModule* }
    [**invariant** *booleanExpn*
      { *declarationOrStatementInModule* }]
    [**post** *booleanExpn*]
end id

A *declarationOrStatementInModule* is one of:

a. *declaration* [;]
b. *statement* [;]
c. *collectionDeclaration* [;]
d. *subprogramDeclaration* [;]
e. *moduleDeclaration* [;]

4 Statements and Input/Output

declarationsAndStatements are:

{ *declarationOrStatement* }

A *declarationOrStatement* is one of:

a. *declaration* [;]
b. *statement* [;]
c. *variableBinding* [;]

A *statement* is one of the following:

a. *variableReference* := *expn*
b. *procedureCall*
c. **assert** *booleanExpn*

d. **return**
e. **result** *expn*
f. *ifStatement*
g. *loopStatement*
h. **exit** [**when** *booleanExpn*]
i. *caseStatement*
j. **begin**
    declarationsAndStatements
  end
k. **new** *collection!d* , *variableReference*
l. **free** *collectionId* , *variableReference*
m. *forStatement*
n. **tag** *variableReference* , *expn*
o. *putStatement*
p. *getStatement*

A *procedureCall* is a:

reference

An *ifStatement* is:

if *booleanExpn* **then**
  declarationsAndStatements
{ **elsif** *booleanExpn* **then**
  declarationsAndStatements }
[**else**
  declarationsAndStatements]
end if

A *loopStatement* is:

loop
  [**invariant** *booleanExpn*]
  declarationsAndStatements
end loop

A *caseStatement* is:

case *expn* of
  label *compileTimeExpn* {, *compileTimeExpn* } :
    declarationsAndStatements
  { **label** *compileTimeExpn* {, *compileTimeExpn* } :
    declarationsAndStatements }
  [**label** :
    declarationsAndStatements]
end case

A *forStatement* is one of:

```
a.  for [ id ] : forRange
        [ invariant booleanExpn ]
        declarationsAndStenements
    end for

b.  for decreasing [ id ] : expn .. expn
        [ invariant booleanExpn ]
        declarationsAndStenements
    end for
```

A *forRange* is one of:

a. *expn* .. *expn*
b. *namedType*

A *putStatement* is:

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

A *putItem* is one of:

a. *expn* [: *widthExpn* [: *fractionWidth* [: *exponentWidth*]]]
b. **skip**

A *getStatement* is:

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

A *getItem* is one of:

a. *variableReference*
b. **skip**
c. *variableReference* : *
d. *variableReference* : *widthExpn*

A *streamNumber, widthExpn, fractionWidth*, or *exponentWidth* is an:

expn

5 References and Expressions

A *variableReference* is a:

reference

A *reference* is:

id { *componentSelector* }

A *componentSelector* is one of:

a. (*expn* {, *expn* })
b. . id

A *booleanExpn* or *compileTimeExpn* is an:

expn

An *expn* is one of the following:

a. *reference*
b. *explicitConstant*
c. *substring*
d. *setConstructor*
e. *expn infixOperator expn*
f. *prefixOperator expn*
g. (*expn*)

An *explicitConstant* is one of:

a. *explicitUnsignedIntegerConstant*
b. *explicitUnsignedRealConstant*
c. *explicitStringConstant*
d. true
e. false

An *infixOperator* is one of:

a.	+	(integer and real addition; set union; string concatenation)
b.	-	(integer and real subtraction; set difference)
c.	*	(integer and real multiplication; set intersection)
d.	/	(real division)
e.	**div**	(truncating integer division)
f.	**mod**	(remainder)
g.	**	(integer and real exponentiation)
h.	<	(less than)
i.	>	(greater than)
j.	=	(equal)
k.	<=	(less than or equal; subset)
l.	>=	(greater than or equal; superset)
m.	**not**=	(not equal)
n.	**and**	(boolean conjunction)
o.	**or**	(boolean inclusive or)
p.	->	(boolean implication)
q.	**in**	(member of set)
r.	**not in**	(set non-membership)

In form (f), a *prefixOperator* is one of:

a. +	(integer and real identity)
b. -	(integer and real negation)
c. **not**	(boolean negation)

All infix operators (including **) associate left-to-right. The precedence of operators is as follows, in decreasing order of precedence (tightest binding to loosest binding):

1. **
2. prefix +, -
3. *, /, **dlv**, **mod**
4. infix +, -
5. <, >, =, <=, >=, **not**=, **in**, **not in**
6. **not**
7. **and**
8. **or**
9. ->

References

[Holt & Cordy 1988]
R.C. Holt and J.R. Cordy, The Turing Programming Language.
Communications of the ACM 31(12), December 1988, pp. 1410-1423.

[Holt & Hume 1984]
R.C. Holt and J.N.P. Hume, Introduction to Computer Science Using the Turing Programming Language. Brady, 1984, 404 pp.

[Holt et al. 1987]
R.C. Holt, P.A. Matthews, J.A. Rosselet, and J.R. Cordy, The Turing Programming Language: Design and Definition. Prentice Hall, 1987, 325 pp.