THE TURING PLUS REPORT

R.C. Holt and J.R. Cordy

10 March 1985
(Revised 2 September 1987)
(Revised June 2018)

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

The Computer Systems Research Institute (CSRI) is an interdisciplinary institute formed to conduct research and development relevant to computer systems and their application. It is jointly administered by the Department of Electrical Engineering and the Department of Computer Science at the University of Toronto, and is supported in part by the Natural Sciences and Engineering Research Council of Canada.

Abstract

Turing Plus is a general purpose language that is particularly suited to systems programming. It is a compatible extension of the Turing programming language.

Major extensions include:

- separate compilation of program parts (modules, monitors and subprograms),
- generalized input/output (random access and binary flies),
- concurrency (processes and monitors), and
- exception handling (the quit statement and handler blocks).

There are a number of other extensions, including new numeric types (various sizes of integers and reals, and the natural number type), characters and fixed length character strings, bit manipulation, type cheats, an indirection operation, register variables, procedure variables and assembly language inserts.

Table of Contents

1 Introduction

Turing Plus is a compatible extension of the Turing programming language. Before defining the features of Turing Plus, we begin with an overview of Turing.

Overview of Turing

The Turing programming language, designed in 1982, can be characterized as "a super-Pascal with a no-frills syntax and an airtight formal definition."

It is a super-Pascal in that it includes essentially all the features of the Pascal language, as well as badly needed additional features such as modules, dynamic arrays, convenient strings, flexible input/output and exponentiation. Its no-frills syntax eliminates verbiage such as semicolons and program headers, yielding a language that is easier to use than BASIC.

Its airtight formal definition provides a mathematically precise specification of all aspects of the language, guaranteeing a degree of portability unapproachable in most other compiled languages.

The Turing language is defined by the Turing Report [Holt & Cordy 1983], and more thoroughly introduced in the Turing Programming Language [Holt & Cordy 1988]. Turing is the subject of the textbook Introduction to Computer Programming Using the Turing Programming Language [Holt & Hume 1984], which contains a copy of the Turing Report. The book The Turing Programming Language: Design and Definition [Holt et al. 1987] contains extensive information about the language, including its design goals and formal definition.

Extensions to Turing

Turing Plus adds to Turing a rich set of new features, including separate compilation, random access input/output, concurrency, exception handling, bit manipulation and assembly language inserts. The design of Turing Plus has drawn heavily from experience with Concurrent Euclid [Cordy & Holt 1980], and Turing Pius includes essentially all the of the features of Concurrent Euclid.

The unextended Turing language is an alternative to languages like Pascal and BASIC. Turing, as extended to become Turing Plus, is an alternative to languages like Modula 2, C and Ada.

Faithful execution and checking

Turing includes a a novel feature called faithful execution that ensures that execution is completely determined by the language specification. This implies that all observable actions of a Turing program are determined by the language specification, or else a controlled abort occurs. Such an abort can occur because of (1) violation of a language constraint, such as division by zero, or (2) exhaustion of computer resources, such as stack overflow.

Faithful execution guarantees portability from machine to machine (except for dependencies on resource exhaustion and nondeterminism due to floating point or random number generation). By setting a compiler flag, the user can choose to eliminate the checking code that enforces faithful execution. Turing without checking will execute the same as with checking (although a bit faster), unless there is resource exhaustion or a constraint violation. In that case, execution once again becomes dangerous and unpredictable, as is the case in machine-oriented languages such as C.

Visibility of the Underlying Implementation

Turing Plus recognizes the benefit of faithful execution and preserves as much of this feature as is consistent with the goal of allowing convenient access, as requested, to the underlying implementation. This access is called "visibility" and is divided into three increasingly dangerous levels.

The first level is data visibility, which allows access to the implementation (the bits) that representdata items. For example, in Turing Plus it is possible to consider the bits representing a real number to bean array of characters. Use of data visibility causes a program to be non-portable (in the general case), and its meaning is not formally defined. However, this level of visibility (called "dirty") never causes remote corruption of code or data.

The second level is address visibility. This allows the user to explicitly inspect and modify machine addresses. For example, *p := addr(x)* sets p to the address of variable x, and *int@(j+9OO) := 5* assigns 5 to the memory location at the addresss 900 beyond the value of x. The meaning of such an action can only be understood using intimate knowledge of the implementation, and will potentially change (corrupt) remote data and/or code. This "dangerous" use of addresses is impossible in Turing proper, but is always present in machine-oriented languages such as C. It is useful for certain systems programming purposes. It provides the expert with an extra degree of performance and convenience for various special purposes in systems programming.

The third level is code visibility. Code visibility allows explicit emission of particular machine code. This is even more treacherous than address visibility. Carefully designed address manipulation can be portable across closely related machine architectures, but specific emitted code is only meaningful for a given computer architecture. This feature allows access to underlying machine capabilities that are sometimes needed in systems programming.

Turing Plus provides the expert with convenient access to essentially all aspects of the underlying implementation. However, this access must be explicitly requested, for example, by using the addr operator, rather than being the default, as it is in machine-oriented languages like C. Most of the time, the Turing Plus programmer enjoys the productivity and convenience provided by Turing's basic features, and is exposed to the danger of C-like features only when those features are required and explicitly requested. The result is that the Turing Plus programmer can expect to be highly productive while still having controlled access to dirty and dangerous features.

Dirt and Danger

We will now give explicit definitions of the terms "dirty" and "dangerous". These definitions provide a classification of the various features of Turing Plus (and other languages). Basically, a feature is clean (mathematically defined), dirty (implementation dependent), or dangerous (may cause remote corruption), as follows:

1. Clean. Clean features have a clear, machine-independent semantics, expressible in mathematics.

2. Dirty. Dirty features have semantics that is implementation dependent, for example, a type cheat that treats a real as an int is dirty (but not dangerous).

3. Dangerous. Dangerous features can cause remote corruption of program or data. For example, poking into explicit memory locations is dangerous. Converting (via a type cheat) to a type containing a pointer is dangerous. All dangerous features are also considered dirty.

2 Implementations of Turing and Turing Pius

As of February 1987, Turing is implemented by an interpreter (written in Turing) and a compiler with versions called ttc and pttc (written in Concurrent Euclid). Turing Plus is implemented by a compiler called tpc (written in Turing Plus). The Turing interpreter supports none of the Turing Plus extensions. The ttc compiler supports two extensions, generalized input/output and conditional compilation (but not separate compilation, as ttc gains speed by skipping the link step). The pttc compiler ("p" stands for "production") is the same but supports separate compilation. The interpreter, ttc and pttc compilers run on Vax/Unix, SUNos, PC compatibles and IBM mainframes.

The portable Turing Plus compiler, tpc, supports the entire Turing language as well as Turing Plus extensions (except as noted in this report). This compiler was originally developed by Mark Mendell with help from Ron Wessels. As of February 1987, tpc runs on SUNs, emitting MC68000 code. It also emits C source code to allow it to be easily ported to systems with C compilers. A VAX version of Turing Plus is currently under construction.

Parameter ordering

There has been a conscious effort to make tpc compatible with standard calling conventions, namely those defined by C. The calling and parameter passing conventions used by tpc are the same as those used by C.Beware that the order of parameter passing used by ttc, pttc and Concurrent Euclid is the reverse of this order. Consequently subprograms separately compiled by pttc (and ttc) and tpc can only be linked if care is taken to reverse five parameter lists in either the pttc or the tpc version. There is also an incompatibility in that certain data items (booleans, strings, sets, and enums) are allocated more space by ttc (and pttc) than by tpc.

The versions of ttc and pttc on IBM PC compatibles differ from those on Unix in the naming of files. A source program named prag.t will have a linkable modules called prog.o under Unix but prog.obj on the PC, and an executable module called prog.x on Unix (but prog.exe on the PC).

3 Generalized Input/Output

This section defines a set of extensions to the input/output facilities of Turing to provide internal form (binary) input/output and random access files. The extensions take the form of a set of new statements similar to Turing's get and put statements.

The Open Statement

The **open** statement is introduced; it extends and replaces the capabilities of Turing's *open* predefined procedure. (For compatibility, the *open* procedure is retained.)

An *openStatement* is one of:

a. **open** : *fileNumberVariable*, *fileName*, *ioCapability* {, *ioCapability*}

b. **open** : *fileNumberVariable*, *argNumber*, *ioCapability* {, *ioCapability*}

The **open** statement attempts to open the named file for the specified input/output capabilities. If the file is opened successfully, the *fileNumberVariable*, whose type must be int, is set to the strictly positive file number to be used in accessing the file. Otherwise (if the open fails), *fileNumberVariable* is set to zero. The *fileName* is a string expression giving the file path of the file to be opened.

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

```
prog.x infile outfile
```

specifies execution of prog.x with program arguments *infile* and *outfile*. (Use prog.exe instead of prog.x in MS-DOS.)

The *argNumber* is the position of the argument on the command line. (The first argument is number 1.) The name of the file to be opened is the corresponding command line argument. If there is no such argument or if the file cannot be opened successfully, *fileNumberVariable* is set to zero. Otherwise, *fileNumberVariable* is set to the strictly positive file stream number to be used in accessing the file.

An *ioCapability* is one of:

get, **put**, **read**, **write**, **seek**, **mod**

A file can be accessed using only the statements corresponding to the input/output capabilities with which it was opened. The **tell** statement is additionally allowed if the file is opened for **seek**.

The **open** statement truncates the file to length zero if the *ioCapabilities* include **put** or **write** but not **mod**. In all other cases, **open** leaves the existing file intact. The **mod** *ioCapability* specifies that the file is to be left intact, i.e., that the file is to be modified. Each open positions to the beginning of a file. There is no mechanism to delete a file. To open for append, one has to open for **seek** (and for **write** or **put**) and then **seek** to the end.

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

Program argument files referenced by command line argument number and used in **put**, **get**, **read** or **write** statements need not be explicitly opened, but are implicitly opened with the capability corresponding to the input/output statement with which they are first used. (The *fileNumber* specifies the number of the argument.) As in Turing, a *fileNumber* of zero specifies the special error output stream. Turing Plus also extends Turing with the convention that *fileNumber* -1 specifies the standard output stream, and -2 the standard input stream.

The Close Statement

A *closeStatement* is:

close : *fileNumber*

The *fileNumber* must be an integer value giving the file number of an open file. Normally the *fileNumber* is an integer variable whose value was set by an **open** statement. After a close, the specified file is no longer available to be used in a **get**, **put**, **read**, **write**, **seek** or **tell** statement.

Intemal Form (Binary) Input/Output

Access to internal form (binary) files is based on the new statements **read** and **write**.

A *readStatement* is:

read : *fileNumber* [: *status*], *readItem* {, *readItem*}

The **read** statement reads one or more items in internal form from the specified file. The *fileNumber* must specify a file that is open with **read** capability (or a program argument file that is implicitly opened).

The optional *status* is an int variable that is set to implementation dependent information about the read. If *status* is returned as zero, the read was successful. Otherwise *status* gives information about the incomplete or failed read (which is not documented here).

A *readItem* is:

variableReference [: *requestedSize* [: *actualSize*]]

Each *readItem* specifies a variable to be read in internal form. The optional *requestedSize* is an integer value giving the number of storage units (usually bytes) of data to be read. The *requestedSize* should be less than or equal to the size of the item's internal form in memory (otherwise, a warning message is issued). If no *requestedSize* is given, then the size of the item in memory is used. The optional *actualSize* is an int variable that is set to the number of storage units (bytes) that are actually read.

A *writeStatement* is:

write : *fileNumber* [: *status*], *writeItem* {, *writeItem*}

The **write** statement writes one or more items in internal form to the specified file. The *fileNumber* must specify a file that is open with **write** capability. The *status* is the same as described for the **read** statement.

A *writeItem* is:

reference [: *requestedSize* [: *actualSize*]]

Each *writeItem* is a variable, or a named non-compile-time constant, to be written in internal form. The optional *requestedSize* is an integer expression giving the number of storage units (usually bytes) of data to be written. The *requestedSize* should be less than or equal to the size of the item's internal form in memory (otherwise a warning message is issued). If no *requestedSize* is given then the size of the item in memory is used. The optional *actualSize* is set to the number of storage units (bytes) actually written.

An array, record or union may be read and written as a whole. (Note: in Turing Plus, elements of packed arrays, records and unions cannot be transferred by **read** and **write**.)

Random Access Input/Output

Random access of both text (character) and internal form files is provided by the **seek** and **tell** statements.

A *seekStatement* is one of:

a. **seek** : *fileNumber*, *filePosition*

b. **seek** : *fileNumber*, *

The **seek** statement repositions the specified file such that the next input/output operation will begin at the specified point (*filePosition*) in the file. The *fileNumber* must specify a file that is open with **seek** capability. The *filePosition* is a non-negative integer offset in storage units from the beginning of the file, such as that returned from the **tell** statement. (The first position in the file is position zero.)

Form (b) specifies that the next operation is to begin at the position immediately following the current end of the file. A *filePosition* of zero specifies that the next operation is to start at the beginning of the file. Seeking to a position beyond the current end of the file and then writing automatically fills the intervening positions with the internal representation of zero.

A *tellStatement* is:

tell : *fileNumber*, *filePositionVariable*

The **tell** statement sets *filePositionVariable*, whose type must be int, to the integer offset in storage units from the beginning of the specified file at which the next input/output operation will begin if no intervening **seek** is executed. The *fileNumber* must specify a file that is open with **seek** capability (or a program argument file that is implicitly opened). The **tell** statement is useful for recording the file position of a certain piece of data for later access using **seek**.

4 Conditional Compilation

This section defines conditional compilation, which is the compile-time selection of sections of source text to make up a Turing program. The **#if** construct specifies the selection and has the form:

```
  #if expn1 then
     src1
 { #elsif expnN then          Any number of #elslf clauses
     srcN }
 [ #else                      Optional #else clause
     srcE ]
  #end if
```

Each of *src1, srcN,* and *srcE* represent a part of a Turing source program. If *expn1* is true then *src1* is selected, otherwise if *expn2* is true then *src2* is selected, and so on. If none of the expressions is true then *srcE* (if present) is selected.

We will give an example and then will specify the form and meaning of expressions (*expnN*):

```
#if stats and debug then
    var count, array 1.5 of real
    var message: string
#elsif tracing then
    put *Debugging message"
#end if
```

If both of *stats* and *debug* are true then the declarations of count and message will be selected, otherwise if *tracing* is true, then the **put** statement will be selected. Otherwise, nothing is selected.

Each of the expressions *(expn1, expnN*) consists of identifiers, the boolean operators **and**, **or** and **not** (but not ->), and parentheses. The short forms & and | are allowed in these expressions.

An identifier *id* is considered to be true if it is passed as a flag to the compiler in the form *-Did,* where *id* is the identifier. For example, under Unix or MS-DOS the following sets *stats* and *debug* to be true in the compilation of tests.

```
tpc –Dstats –Ddebug tests
```

Note that the Turing Plus compiler is invoked by *tpc*, but that the standard Turing compiler is invoked by *ttc* or *pttc*. Flag identifiers such as stats and debug are independent of any Turing program identifiers. For example, the identifier stats could also be the name of a variable in the Turing program.

Comments and literals are respected; this means that **#if**, **then**, etc. appearing in comments and literals are ignored. All of a **#if** construct must appear in a single source file, and cannot be partially contained in an included file. A **#if** construct must not include a **parent** or **child** directive. (These directives are discussed below.)

The source text can contain any number of **#if** constructs (within reason) and these constructs can be nested. It is not necessary for **#if** to appear first on a line. For example, it can be preceded on the line by spaces, tabs, or a Turing statement.

5 Separate Compilation

This section defines the Turing Plus separate compilation facility. This facility is designed to meet several criteria:

1. The syntax and semantics of separate compilation must be simple, convenient, and consistent with the spirit of the Turing language.

2. There must be no new semantics for modules, subprograms or scopes.

3. There must be a simple method of splitting most Turing programs into separately compiled pieces.

4. There must be an obvious way to place the separately compiled parts of a program into source files.

5. The syntax and semantics of the separate compilation facility should respect and emphasize the hierarchical structure of Turing programs.

6. It should be difficult to misuse the separate compilation facility.

7. It must provide for the sharing of global variables among separate compilations.

8. The facility must allow full compile-time type checking across separate compilations without requiring stored symbol tables.

9. It should be possible to use simple, existing program linkers.

The methods for separately compiling modules and monitors are identical (except that the keyword **monitor** replaces **module**), so we will use the term "module/monitor" when referring to either of these two constructs.

The reader may be tempted to believe that separate compilation in Turing and Turing Plus is intended as a way to provide general visibility of external variables by separately compiled subprograms. This view, which applies to less structured languages like Fortran and C, violates information hiding and is contrary to the philosophy of Turing. The best way to think of any Turing or Turing Plus program (separately compiled or not) is as a set of statements and declarations that obey Turing's scope rules and import/export lists.

An existing Turing or Turing Plus program and be divided into separately compiled parts in the following straightforward manner. Each module/monitor or subprogram that is to be separately compiled is moved into a separate source file and is replaced by a child directive designating that file. (As explained below, it is necessary to introduced certain clauses, including grant lists and parent clauses, before the separated parts are compiled.) It is helpful to remember that it is never necessary to use the **include** directive to accomplish this separate compilation.

If there already exist separately compiled modules or subprograms, they can effectively be made a part of a larger Turing program by designating them as *children*.

In general, a particular module/monitor or subprogram should be designated to be a **child** in only one place in a particular program. This is because a module/monitor or subprogram is considered to have only one declaration in a given program. Trouble can arise if a user thinks that each **child** designation of a module/monitor creates a distinct instance of the module/monitor. This is not the case, and the seemingly distinct children share a single set of statically allocated data. Worse yet, the present implementations of Turing and Turing Plus give no warning about this problem. The corresponding situation with subprograms does not cause trouble, because subprograms have no statically allocated data.

The Turing Plus compiler is consciously designed to allow easy access to subprograms written in other languages such as C. To link to an existing C subprogram, one designates it as a **child** (or as an external subprogram). Care must be taken to ensure that the representation of parameters is consistent between Turing Plus and the particular C implementation. Beware that C compilers have as yet no recognized standard for either linkage conventions or allocation of space to integers.

Before describing the separate compilation features in detail, we will give two examples. The first is the separate compilation of a procedure and the second is the separate compilation of a module.

Example of a Separately Compiled Procedure

This section shows how a procedure can be separately compiled. Consider this program, which is stored in a source file called "*hello.t*".

```
put "Start"

procedure greet (s: string)
    put s
end greet

greet ("Hello")
```

To separately compile the *greet* procedure we will divide the source file into two source files, called *main.t* and *greet.ch*. The *main.t* file contains:

```
put "Start"
child "greet.ch"
greet ("Hello")
```

The **child** directive indicates that a part to the program is contained in the separately compiled file called "*greet.ch*". The *greet.ch* file contains:

```
stub procedure greet (s: string)

body procedure greet
    put s
end greet
```

This file contains the *greet* procedure, into which has been inserted the keyword "**stub**" and the line "**body procedure** greet". The **stub** keyword indicates that this file is not a main program, but is rather a separately compiled program part.

The stub of the *greet* procedure is its header, "**procedure** greet (s: string)". This stub (or interface) contains all of the information needed for checking legitimate usage of the *greet* procedure. In the present example, this means checking that the main program calls the *greet* procedure with a parameter of type "string". When *main.t* is compiled, the stub of greet (down to the keyword "**body**") in the separate file *greet.ch* is inspected, but the body of the procedure is ignored.

The body of the procedure, which begins with the keyword "**body**", is sometimes called the “implementation”. When *greet.ch* is compiled, the compiler reads both the stub and the body and produces the object module (the machine language translation) of the *greet* procedure.

The Turing Plus compiler can be invoked to translate the main program *main.t* to an object module *main.o* using this command:

```
tpc –c main.t
```

(Use *pttc* instead of *tpc* for the production Turing compiler.) The -c flag specifies that a linkable object module rather than an executable load module is to be produced. This object module is

called *main.o.* The *greet* procedure can be separately compiled into an object module using the command:

```
tpc -c greet.ch
```

Under Unix, this produces the object module "*greet.o*". The two object modules can then be linked together using the command:

```
tpc main.o greet.o
```

The resulting executable load module is called *main.x* under Unix. The only difference under MS-DOS is that the object modules are called *main.obj* and *greet.obj* and the load module is called *main.exe.* There are other ways to use *tpc* to compile and link, but these will not be discussed here.

Example of a Separately Compiled Module

This section illustrates the separate compilation of a simple module. Consider this main program, which is in a source file called "*demo.t*".

```
put "Demo"

module stack
    export (push, pop)
    var top := 0

    var c: array 1..100 of int

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

    procedure pop (var i: int)
        i := c(top)
        top – top -1
    end pop
end stock

stack.push (27)
var j: int
stack.pop (j)
put j           % Output 27
```

To separately compile the module "*stack*", we use two separately compiled files, called "*demomain.t*" and "*stack.ch*".

The main program, contained in *demomain.t*, is as follows:

```
put "Demo"

child "stack.ch"

stack.push (27)
var j: int
stack.pop (j)
put j           % Output 27
```

The *stack.ch* file contains:

```
stub module stack
    export (push, pop)
    procedure push (i: int)
    procedure pop (var i: int)
end stack

body module stack
    var top := 0

    var c: array 1..100 of int

    body procedure push
        top := top + 1
        c(top) := i
    end push

    body procedure pop
        i := c(top)
        top – top -1
    end pop
end stock
```

As was the case for the *greet.ch* file, in the stack.ch file we distinguish between the stub (interface) of the *stack* module and its body (implementation). The stub contains all information needed to check for legitimate use of the module. The module's body, which follows the keyword "**body**", gives the implementation.

The main program can be compiled using:

```
tpc –c demomain.t
```

This compilation uses stack module’s stub in *stack.ch*, but ignores stack's body. The *stack* module can be separately compiled using the command:

```
tpc -c stack.ch
```

This compilation produces the object module *stack.o* for the *stack* module, using both the stub and the body. The two object modules can then be combined under Unix using the command:

```
tpc demomain.o stack.o
```

Under MS-DOS, the object modules are *demomain.obj* and *stack.obj.* We have introduced separate compilation by example using a procedure and a module. We will now describe Turing's separate compilation facility in detail.

The Parent-Child Hierarchy

Each Turing program consists of a hierarchy of nested modules/monitors and subprograms. For example, the program:

```
module m
    export (q)
    var v: int

    module n
        import (var v)
        export (p)
        procedure p (i: int)
            v := v + i
        end p
    end n

    procedure q (j: int)
        v := j
    end q

    q (0)
    n.p (10)
end m

m.q (5)
```

has a structure which can be pictured as:

```
                                  module n — procedure p
main program —- module m -—<
                                  procedure q
```

We call each module/monitor or subprogram which is directly nested within another module/monitor a "*child*". The module/monitor in which the child is nested is called the "*parent*" module/monitor. In the example, the module n is a child of module m, which is n's parent module. The procedure p is in turn a child of module n, and hence n is p's parent. Because subprograms cannot be nested in Turing, a subprogram cannot be a parent (although a main program can be). Each child module/monitor should have exactly one parent in each program that it is a part of. This means that in an entire program composed of separately compiled parts, exactly one child directive should specify a particular child file.

Separate Compilation of Modules/Monitors

A module/monitor can be separately compiled. The module/monitor is replaced by a **child** directive. The following shows this replacement to separately compile module n.

```
module m
    export (q)
    grant (var v)

    var v: int

    child "n.ch"

    procedure q (j: int)
        v := j
    end q

    q (0)
    n.p (10)
end m

m.q (5)
```

The **child** clause tells the compiler that "n" is separately compiled and its stub can be found in the file "n.ch". This file must begin with the stub specification for n, optionally followed by a body (implementation) of n. When compiling module m, the compiler will read only the stub of n, and not its body. This allows m access to the export list of n and allows the compiler to check that m's use of n is consistent with its interface. References to the exported identifiers of n are implemented as external references to be resolved at link time.

In order for a separately compiled unit to access identifiers declared in its parent module, the parent must list the identifiers in its grant clause (see the third line of the above example). The **grant** clause lists the symbols declared in the parent module which it is willing to share with its separately compiled children. The syntax of **grant** clauses is similar to that of import lists. Separately compiled children are allowed to import from the parent only those identifiers which the parent has explicitly granted.

The **grant** clause is optional. If a parent module does not contain a grant clause, then none of the identifiers declared in the parent module or imported by it may be used in its separately compiled children.

Child Files

The child file referenced in a **child** clause must contain the interface (stub) of the sub-module/ monitor. If there is only one implementation of the module/monitor, it is convenient to include the body in the stub file. For example, the child file for module n, ”n.ch", could contain:

```
parent “main.t"   % Specify file containing the main program and m

stub module n
    import (var v)
```

```
    export (p)
    procedure p (i: int)
end n

body module n
    body procedure p
        v := v + i
    end p
end n
```

Note that this file begins with a "**parent**" clause. The **parent** clause gives the name of the source file that contains the parent module of the separate compilation. This tells the compiler where to find the global symbols which are imported by the module interface; this allows the compiler to check that n's use of these symbols is consistent with their declaration in m.

The **parent** clause is optional; if omitted, the separate compilation is called an orphan, and it cannot import any global identifiers.

The interface file is a compilation unit and can be separately compiled. If the file does not contain a body, then the compiler will simply check that the interface is consistent internally and with its parent. If a body is included in the file, then the compiler will compile the body and produce object code which can be linked with the parent.

Alternate Implementations

At times it is necessary to have several alternate implementations of the same interface. This is done using a "body" file, which necessarily begins with the **body** keyword. For example, if the child file for module n was "n.ch " as above, an alternate body file "n.bd" might consist of:

```
body "n.ch“ module n
    var x: int

    body procedure p
        x := v
        v := x + i
    end p
end n
```

The file name "n.ch" gives the name of the file containing the module interface for which this file contains an alternate implementation. When an alternate body file is compiled, the compiler will compile only the interface part of the file "n.ch " and will ignore any implementation which may be present in the interface file. It will insure that the implementation is consistent with the interface and will produce object code for the alternate implementation which can be linked with the parent.

Separate Compilation of Subprograms

The separate compilation of subprograms (procedures and functions) is done in a way analogous to modules. For example, we can separately compile procedure q of our original example as follows:

```
module m
    grant (var v)
    var v: int

    child "n.ch"

    child "q.ch"

    q (0)
    n.p (10)
end m
```

The interface file "q.ch" for procedure q would be:

```
parent "main.t"

stub procedure q (j: int)
import (var v)

body procedure q
    v:= 2 * j
end q
```

The first three lines of this file give q's interface (up to the keyword **body**) and the last three lines give q's implementation.

The **parent** clause provides the compiler with the environment for q and allows it to check consistency between the child procedure and its parent module. The parameters are declared in the subprogram stub and not in the body. If no **import** clause appears it is assumed that the subprogram imports all identifiers granted by the parent. The import list in a subprogram interface must be a subset of the grant list of the parent. The optional **pre**, **init** and **post** clauses of the subprogram appear in the implementation.

Alternate implementations for q can be made using alternate body files, for example:

```
body "q.ch" procedure q""
    v := j + j
end q
```

Naming Conventions for Files

By convention, the suffix of a Turing main program file is ".t", the suffix for a child file containing both the stub and body is ".ch", the suffix for a file containing only a stub is ".st", and the suffix for a file containing only a body is ".bd". The existing Turing and Turing Plus compilers insist upon these suffixes.

Syntax of Separate Compilation Facility

This section gives the syntax for separate compilation. Although it is not explicitly shown here, monitors are separately compiled in a manner similar to modules.

A *compilation* is one of:

a. [*grantList*]
   program

b. [**parent** *fileName*]
   stub *moduleStub*
   [**body** *moduleImplementation*]

c. [**parent** *fileName*]
   stub *extendedSubprogramHeader*
   [**body** *subprogramImplementation*]

d. **body** *fileName moduleImplementation*

e. **body** *fileName subprogramImplementation*

A *moduleDeclaration* is augmented to have an optional *grantList* immediately following the optional *exportList*.

A *grantList* is:

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

There are two new forms of declaration:

a. **external** *subprogramHeader*
b. **child** *fileName*

Form (b) declares a module or subprogram.

A *moduleStub* is:
module id
[*importList*]
[*exportList*]
{ *declarationInStub* }
end id

A *declarationInStub* is one of:

a. *compileTimeConstantDeclaration*
b. *typeDeclaration*
c. *collectionDeclaration*
d. [**external** [*explicitStringConstant*]] *subprogramHeader*
e. **child** *fileName*

Note: a subprogram declared in the module stub must have a body in the module body file, unless it is declared as **external** or **child** in the stub.

A *moduleImplementation* is the same as a *moduleDeclaration* except that it cannot have an **import** or **export** list.

An *extendedSubprogramHeader* is:

subprogramHeader
[*importList*]

A *subprogramImplementation* is one of:

a. **procedure** id
   [**pre** *booleanExpn*]
   [**init** *idexpn* {, *idexpn* }]
   [**post** *booleanExpn*]
   statementsAndDeclarations
  end id

b. **function** id
   [**pre** *booleanExpn*]
   [**init** *idexpn* {, *idexpn* }]
   [**post** *booleanExpn*]
   statementsAndDeclarations
  end id

Example of Separately Compiling a Stack Module

This section contains an example of a separately compiled module. Note that the stack module shown here is identical to a previous example module.

The file "stackuse.t":

```
put "Enter 10 integers"

child "stack.ch"     % Specify interface to stack module

for i: 1..10
    var j: int
    get j
    stack.push (j)
end for

put "Here is list backwards"
for i: 1..10
    var j: int
    stack.pop (j)
    put j
end for
```

The file "stack.ch":

```
stub module stack     % Interface for stack
    export (push, pop)
        procedure push (i: int)
        procedure pop (var i: int)
end stack

body module stack     % Implementation of stack
    var top:- 0
    var c: array 1.100 of int

    body procedure push
        top := top + 1
        c(top) := i
    end push

    body procedure pop
        i := c(top)
        top := top - 1
    end pop
end stack
```

The file "stack.bd":

```
body "stack.ch" module stack   % Alternate implementation of stack
    var c: collection of
        record
            value: int
            prev: pointer to c
        end record
    var top := nil(c)

    body procedure push
        var oldtop := top
        new c, top
        c(top).value := i
        c(top).prev := oldtop
    end push

    body procedure pop
        var oldtop := top
        i := c(top).value
        top := c(top).prev
        free c, oldtop
    end pop
end stack
```

The alternate implementation *stack.bd* can be compiled into a linkable object module using the command:

```
tpc –c stack.bd
```

This produces "stack.o". If the object module for *stackuse.t* is already in the file *stackuse.o*, an executable load module can be created under Unix using the command:

```
tpc stackuse.o stack.o
```

Use *pttc* instead of *tpc* for the production Turing compiler. The corresponding MS-DOS command uses ".obj" instead of ".o".

6 Various Features of Turing Plus

This section defines the following new features of Turing Plus:

1. New numeric types (the *nat* type and various sizes of *int* and *real*)
2. Character types
3. Packed Types
4. Exclusive Or Set Operator
5. Type Cheats
6. Bit Manipulation
7. External Variables
8. The Indirection Operator
9. Addresses as Integers
10. Size and Addr Attributes
11. Use of Registers
12. Subprogram Variables
13. Assembly Language Inserts

New Numeric Types

The new "*nat*" type represents non-negative integers (the natural numbers), for example:

```
var c: nat := 4      % Cannot hold negative values
```

The new predefined functions *natstr* and *strnat* that are analogous to *intstr* and *strint*.

The new types *int1, int2* and *int4* are implementation-dependent subranges of the integers that are intended to occupy 1, 2, and 4 bytes respectively. These types represent (almost) symmetric subranges about zero, for example, int1 represents values in the subrange -128..127. Analogously, *natl*, *nat2* and *nat4* are subranges of the natural numbers intended to

occupy 1, 2 and 4 bytes respectively. These types represent subranges starting at zero, for example nat1 represents values in the range 0~255. Analogously, *real4* and *real8* represent real numbers using 4 and 8 bytes. The unsuffixed *real* type generally has the same representation as *real8*.

In most implementations, real arithmetic will be carried out with a *rreb* (relative roundoff error bound) of at most 1e-14 and and exponent range of at least -38..38, regardless of the size or significant digits of the operations. (In most byte-oriented implementations, this means 8-byte real arithmetic is used.)

Whenever an *int* value is required, a *nat* value is allowed (with implicit conversion) and vice versa. A *nat* or *int* value can also appear wherever a real value is required, with implicit conversion to *real*. Similarly, a *real4* or *real8* value is allowed wherever a real value is allowed. These conversions round to the nearest target precision value.

When *int* values are combined with *nat* values, the nat is converted to int. For byte-oriented machines, arithmetic for nat values of any size guarantees accuracy for values lying in *nat4*. Similarly int operations guarantee accuracy in *int4*.

For byte-oriented machines, *intN* (where N is 1, 2 or 4) is expected to provide the full set of possible values: -(2**(M-1)) .. (2**(M-1))-1, where M=8*N. Similarly, *natN* ranges over 0..(2**M)-1. For int and nat (without the N), a checking implementation may reserve a pattern, -(2**31) for int, 2**32 for nat, to represent uninitialized variables.

Subrange types, set types and enumerated types can have an explicit byte size specified, for example:

```
var cents: 0..99 : 1
var r: set of 0..99 : 2
type color enum (red, yellow, green) : 2
```

The specified size must be a compile time value of 1, 2 or 4. Note that a size cannot be specified for a type name (e.g., t:2 is illegal), array, record or union.

There are new explicit integer constants that can be written using any base from 2 to 36; these are written as *B#V* where B (the base) is 2,3, ... 36 (in base 10) and V gives the value in that base. If B is 2, V is a non-empty sequence of 0 and 1 digits; in general, each digit of V must be less than B. The digits corresponding to 10,11, ... are *a,b,* ... (or equivalently: *A,B,* ...). For example:

```
assert 2#10 = 2
assert 16#a = 10
```

A negative value is created by prefixing a minus sign, for example:

```
assert -16#a = -10
```

These constants cannot be read by **get** and these forms can not be created by **put**.

New predefined functions are introduced to convert between *nat* and *string* values:

natstr (n [, width]) *Convert nat to string*

strnat (s) *Convert string to nat*

In these, *n* is a natural value (including any non-negative integer), *s* is a string value, and *width* is a natural value giving the string length. The result of *natstr* is n represented as a string, with *width* increased if needed to represent the value. An omitted *width* specifies that the result is to be just long enough to represent the value. The result of *strnat* is the characters in *s* interpreted as the text representation of a natural number.

Character Types

We introduce types that correspond to (1) single characters and (2) fixed length strings of characters. For example,

```
var c: char := 'W'              % A single character
var f: char(5) := 'Hello'       % Fixed length of 5
```

Fixed length character strings such as *f* are called *char*(n), pronounced "character n".

Note that the type *char* and the type *string*(1) are distinct; *char* always represents exactly one character, whereas *string*(1) can represent either the null string or a string containing one character. The *char* type is an index type and can be used, for example, as subscripts, for ranges and case labels. In the following discussion we will use italic *string* to refer to Turing's varying length strings (with or without explicit maximum lengths).

The new types *char* and *char*(n) are designed so that they can be freely intermixed with each other and with the *string* type. This means that concatenation (+), comparisons, substring and length can be applied to any of these types. The **get** and **put** statements can also be used with these types.

The char Type

The *char* type is a scalar type; it and its subranges can be used as array subscripts, as selectors (tags) for case statements and for union types, and as the base type of a set. The *char* type can be used as for statement indexes, as arguments to *succ* and *pred*, and can be compared (=, not=, >, >=, <=). Characters can be read and written by **put** and **get**.

Example:

```
var c: char := 'H'
put c ..                    % Write one character
get c                       % Read one character

if c = 'i' or c = 'o' then
    put c                   % Hi or Ho
else
    put 'Huh?'              % Huh?
end if
```

Example:

```
% Read characters, counting each capital letter
var frequency: array 'A'..'Z' of nat
for d: 'A'..'Z'
    frequency(d) := 0
end for

loop          % Tabulate use of capital letters
    exit when eof
    var c: char
    get c     % Reads 1 character
    if c >= 'A' and c <= 'Z' then     % Ok for ASCII
        frequency(c) := frequency(c) + 1
    end if
end loop

for d: 'A'..'Z'    % Print frequency of 'A' to'Z'
    put d, frequency(d) : 10
end for
```

The Turing Report's definitions of *ord* and *chr* are modified to take advantage of the *char* type. This modification changes *chr* to return a value of type *char* (instead of *string*(1)), so *chr* can now return the values that correspond to *eos* and *uninitchar*. It changes *ord* so its formal parameter's type is *char*, so it can accept the characters *eos* and *uninitchar*. Thus, for byte-oriented machines, *chr* and *ord* now accept and return the full range of 256 ASCII values. Note that these modifications do not affect programs using the Turing Report's definitions of *ord* and *chr*.

The *char*(n) type. The *char*(n) type is considered to be a non-scalar. In a parameter list, a parameterType can be *char*(*), which accepts *char*(n) for any value of n. The types *char*(*) and *char*(n) when n is not a compile-time expression are called *dynamic char*(n). Dynamic char(n) has the same class of restrictions as are applied to dynamic arrays. Dynamic *char*(n) can be passed only to the parameter type *char*(*). It is not allowed to assign or compare dynamic *char*(n). It cannot be a named type and cannot appear in a record, union or collection.

Explicit constants

New explicit *char* constants are introduced having the form: a single quote, zero or more characters, and a final single quote, for example, 'Hello'. If there is exactly one character inside the quotes, the constant's type is *char*, otherwise it is *char*(n) where n is the number of enclosed characters. Since there is implicit conversion between the types *char* and *string*, the type of these constants is rarely significant, but does make a difference in declarations without an explicit type, for example,

```
const c := 'H'          % c is of type char
var d: char(2)          % d is of type char(2)
var e := 'Hi'           % e is of type char(2)
var f := "h"            % f is of type string (no fixed max length)
```

The back slash character (\) can be used in these constants, as in string explicit constants, to specify certain characters, such as \t or \T for tab (see Turing Report "Identifiers and Explicit Constants"). Backslash can now also be used to escape a single quote, as in,

```
const singleQuote := '\''    % A single quote
```

It is not necessary to precede a single quote by back slash in a double quoted (string) constant, nor vice versa, for example:

```
const name := "O'Reilly"
var sentence := 'He said, "Hi"'
```

Note that numeric values of characters can be given explicitly using the *chr* function, for example:

```
const greeting := 'Hi' + chr(0)     % Type is char(3)
```

The caret ("^") is used in explicit constants (*string*, *char* and *char*(n)) to specify control characters. The character immediately following the caret is defined as follows. If the character following the caret is a question mark, the result is an ASCII DEL (delete) character:

'^?' = '\d' (the ASCII DEL (delete) character, i.e., chr (16#7f))

Otherwise, the top three bits of the character are set to zero,

'^c' = chr(ord('c') & 2#11111) (where c is not a question mark)

For example:

```
const BS := '^H'
```

If the caret character itself is required in an explicit constant, it must be preceded by a back slash.

Automatic conversions

There are automatic conversions among the types *char*, *char*(n), and *string*. If these types are combined in a concatenation, the result type is:

a. *char*(n) when the operands are *char* or non-dynamic *char*(n)

b. *string* when either operand is *string* or dynamic *char*(n)

A *char* value is converted in an expression, as required, to be of type *char*(1) or string (with length of 1). A value of type *char*(n) is converted, as required, to be a *string* with length n. Note that the converted value is an expression and not a variable reference.

A value converted to type *string* must (1) not contain the *eos* (end of string) and *uninitchar* characters and (2) must not exceed the length limit (at least 255) of strings.

The result of the concatenate operator is considered a compile-time expression when both operands are compile-tune expressions.

Type equivalence and assignability

Type equivalence determines what variable types can be passed to **var** parameters. The rules for the new types are:

1. The type *char* is only equivalent to itself

2. The type *char*(n) is equivalent to *char*(m) iff n = m. Note that dynamic *char*(n) can only be passed to *char*(*). This means, for example, that *char*(n) cannot be passed to a **var** *string* parameter.

Assignability determines which values can be assigned to a variable of a given type or passed to a value (non-var) parameter. The rules for the new types are:

1. Values of type *string*, dynamic or non-dynamic *char*(n) and *char* that contain exactly one character are assignable to *char*.

2. Values of type *string*, *char*(n) and *char* that contain exactly n characters are assignable to *char*(n). However, it is never possible to assign to a dynamic *char*(n).

3. Values of type *string*, *char*(n) and *char* can be assigned to type *string* (and *string*(m)), but the length must not exceed the maximum length of the target string.

Two values of types *char*(n) and *char*(m), both non-dynamic, can be compared without conversion to *string* if n = m; if n not= m or at least one is a dynamic type, both are converted to type *string* before doing the comparison.

Limits on lengths

An implementation may limit the length of the type *char*(n) along with dynamic *char*(n), and the length of concatenation of the *char* types. This limit is expected to be large (recommended at least 32767). Note that the length of a non-dynamic *char*(n) can always be determined at compile time.

There is a separate limit for lengths of values of type *string*, which is to be at least 255. A value of type dynamic *char*(n) that is compared or concatenated automatically is converted to type *string* and cannot exceed this limit.

Substrings

Substrings with a single parameter, such as s(i), s(*) and s(*-1) are special cases as applied to *char*(n) including dynamic *char*(n). These substrings behave like subscripting of an array of characters; they return a reference of type *char*. This reference can be assigned to or passed to a **var** parameter of type *char*. Of course, the value being operated on must itself be a variable for this to be allowed.

Substrings applied to type *string*, or with two parameters, remain as they were in the Turing Report and cannot be assigned to or passed **var**. In other words, they are expressions (not variable references) of type *string*. They are subject to the maximum length of the string type.

The **get** statement is extended so it can be used to read *char* variables and fixed length strings. This statement does not allow a *width* specification for *char* variables.

Example:

```
var c: char
get c:1          % This is NOT allowed
```

Rather, one would use:

```
get c            % Read one character
```

Example:

```
var f: char(3)
get f            % Reads 3 characters
get f:2          % Reads 2 characters, into f(1) and f(2)
```

If end-of-file is encountered when reading c or f, the null character is used to complete the input. The value of null is implementation dependent, but is recommended to be *chr*(0).

The **put** statement applies to *char* and fixed length strings, for example:

```
var g: char(3) := 'Yes'
put g:4          % Prints YesB where B means blank
put g            % Prints Yes
```

Examples:

```
var c: char := 'F'
var s: string(10) := "Mellow "
var f: char(6) := 'Yellow'
var g := 'Hi'

put c            % Outputs: F
put s            % Outputs: Mellow
put f            % Outputs: Yellow
f(1) := c
put f            % Outputs: Fellow
s := g
put s            % Outputs: Hi
```

There are three new predefined procedures. In these descriptions the type *strchar* represents any of *string*, *string*(n) or *char*(n) type.

strmove (**var** target: strChar, source: strChar, loc: int)

The characters of source moved to target, starting at position *loc*.
If target is of type string, this procedure is equivalent to this statement:

target := target (1..loc-1) + source + target (loc + length(source) .. *)

It is required that *loc* >= 1 and *loc* + length(*source*)-1 <= length (*target*).
This operation does not change the length of *target*.

strdelete (**var** target: string (*), len, loc: int)

This removes *len* positions from *target* starting at position *loc*. This is equivalent to the statement

target := target (1..loc~1) + target (loc + len .. *)

It is required that *loc* >= 1 and *loc* + *len* - 1 <= length (*target*).

strreplace (**var** target: string (*), source: strchar, loc, len: int)

Replaces *len* characters starting at *loc* in *target* by *source*:

target := target (1..loc-1) + source + target (loc + len .. *)

It is required that *loc* >= 1, *len* >= 0 and the resulting length *loc*+*len*-1 <= upper (*target*).

Packed Data Types

Turing Plus allows the keywords **array**, **record** and **union** to be preceded by the keyword "**packed**". In a word-addressing architecture, packing may result in several small scalars (e.g. characters) to be stored within a single word. Turing Plus also allows packed to precede **enum** or explicit subranges (expression..expression). Dynamic arrays cannot be **packed**. Packed types are equivalent only to equivalent packed types. The elements of a packed array and the tags and fields of packed unions and records cannot be passed by reference and cannot be bound to. In a byte-addressing architecture, packing does not necessarily affect storage allocation.

The Exclusive Or Operator

The set operator **xor** (exclusive or, also called symmetric difference) is introduced. The other set operators are * (intersection), + (union) and - (set difference). Given sets a and b of type s, **xor** is defined by the equation

a **xor** b = (a + b) - (a * b)

In other words m is a member of a **xor** b iff m is a member of exactly one of sets a and b. The **xor** operator is a new *infixOperator* with the same precedence as +.

Type Cheats

Turing Plus provides a way to interpret the internal representation (bits) of one type as another. This means Turing type checking rules can be overridden. For example:

```
var i: int1
type (char, i) := 'B'      % Type cheat: treat i as char
```

The character 'B' is assigned to variable i, whose type is considered to be *char* (although it is really *int1*). This assignment is equivalent to either of the following:

```
i := type (int1, 'B')
i := ord ('B')
```

The use of type cheats is implementation dependent, and should be used only by persons acquainted with the particular implementation of Turing Plus. Programs using this construct are not necessarily portable to other implementations.

Example:

```
type byte: int1
type word: array 1 ..2 of byte
var j: int2 := 6
type (word, j) (2) := 9
```

This example considers the *int2* variable *j* to be an array of two bytes and assigns 9 to the second of these. The resulting value of *j* depends on the implementation, because the order of byte addressing within words varies from computer to computer. In general, a type cheat has the form:

type (*targetType*, *expn* [: *sizeSpec*])

If the *expn* is a variable reference and the *sizeSpec* is omitted, then the type cheat construct is considered to be a variable whose type is *targetType*. If the *expn* is a value that is not a variable or if *sizeSpec* is present, the type cheat construct is considered to be an expression value whose type is *targetType*.

The *sizeSpec* is a compile-time integer expression giving the size of the *expn's* value. It can be specified only for integer or natural values (where it must be 1, 2 or 4) or real values (where it must be 4 or 8).

The targetType must be one of:

a. [*moduleId* .] *typeId*
b. int, int1, int2, int4
c. nat, nat1, nat2, nat4
d. boolean
e. char [(*compileTimeExpn*)]
f. string [(*compileTimeExpn*)]
g. addressint

A type cheat is carried out in two steps. The first step converts the value if necessary to the size given by *sizeSpec*. The second step, when involves no generated code, interprets the value as the target type.

An implementation may prohibit certain type cheats. Memory alignment requirements of certain types may render some type cheats to be infeasible. It is dangerous to consider a value to have a target type larger than the value's type; it is recommended that the compiler issue a warning when this occurs. An implementation may prohibit certain type cheats on **register** scalar items.

Type Cheats in Parameters

A reference parameter can be automatically subject to a type cheat by inserting the keyword "**type**" before the parameter's type. The new form of *parameterType* is:

[**type**] *typeSpec*

Note that stars cannot be given as array bounds or string maximums when "**type**" is present.

For example:

```
procedure dump (a: type array 0.. 10000 of nat1, n: int)
    for i: 0 ..n-1
        put a (i) : 4
    end for
end dump
```

This procedure prints the values of n bytes starting at the address of formal parameter *a*. All parameters that use type cheats must be passed by reference. The actual parameter for a parameter type cheat must be a variable or non-scalar (so that it can be passed by reference).

The Type Cheat for Naturals (#)

There is a prefix operator, #, which is a short form for a class of common type cheats. It converts its argument to a natural number. The precedence of # is the same as unary minus. In general, "# *expn"* is the same as **type** (*natN*, *expn*) where N is determined as follows. If the *expn* is a variable or expression of size 1, 2 or 4 then N is the size of the variable, otherwise N is 4.

For example:

```
var c: char (3)
#c(2) := 24
```

The form #c(2) is short for **type** (*nat1*, c(2)).

For example, if c is a character, then #c = *ord*(c). Note that #c can be assigned to (its type is *nat1*), but *ord*(c) cannot be assigned to.

Example:

```
var i: int2 := -1
assert #i = 16#ffff      % Assumes 2's complement
#1 := 16#fffe
assert i = -2
```

Note that 16#fffe is a positive number, but when assigned to *i*, the effect is that i takes the value -2. Due to machine alignment requirements, some uses of # may not be feasible or may have unexpected results.

Bit Manipulation

Operators are introduced to manipulate the bits of natural values. These operators are mathematically defined in a machine-independent, portable manner, which does not depend on the internal representation of numbers. When combined with the # operator, these operators/functions provide convenient manipulation of bits in the bytes/words representing arbitrary types.

The convention is that bits are numbered from least significant to most significant bits (right to left) with the least significant bit being bit number zero. Overflow occurs for results exceeding the maximum value of the *nat* type.

Each of the following five bit manipulation operators requires operands of type *nat* and returns a result of type *nat*.

Shifting left: i **shl** j

Returns the value of i shifted left j bits. For example, 2#11 **shl** 2 = 2#1100.
We define i **shl** j = i*(2**j). Overflow occurs for results exceeding the maximum value of the *nat* type.

Shifting right: i **shr** j

Returns the value of i shifted right j bits. For example, 2#1001 **shr** 2 = 2#10.
We define i **shr** j = i div (2**j).

Bitwise and: i **and** j

Returns the bitwise conjunction of i and j. Note that **and** is also a boolean operator.

Bitwise or: i **or** j

Returns the bitwise disjunction (inclusive or) of i and j. Note that **or** is also a boolean operator.

Bitwise exclusive or: i **xor** j

Returns the bitwise (exclusive or) symmetric difference of i and j. The result has 1 in position p iff exactly one of i and j has 1 in position p.

The precedence of these operators is:

shl, **shr**	same as	*
and	same as	boolean **and**
or	same as	boolean **or**
xor	same as	infix +

The notation:

bits (c, *subrange*)

represents a subsequence of the bits in natural number *c* interpreted as a natural number.

This sequence is specified by the subrange, e.g., 2..4. The subrange is one of:

a. *typeSpec*
b. *compileTimeIntegerExpression*

The *typeSpec* must specify a subrange type; let L and M (for least and most significant) stand for the bounds of this subrange. It is required that L and M are compile-time integer values and that 0 <= L <= M. In form (b), L and M are both given by the *compiieTimeIntegerExpression*. Bits L through M of natural number *c* constitute the selected bit sequence. The form bits (c, *subrange*) can be used (1) wherever a *nat* value is allowed and (2) as the target of an assignment statement that is assigning a value in the range 0 .. 2**(M-L+1))-1. In case (1), we define the value of bits (c, L..M) as (c mod (2**(L+1))) div 2**M. In case (2), *c* must be a variable that is *nat* (or *nat1*, *nat2* or *nat4*). Note that L and M are not limited to small values, for example, bits (c, 400..407) := 0 is legal.

Examples:

```
type T12: 1..2
var d: nat2 := 2#1100
assert bits (d, T12) = 2#10
bits (d, T12) := 2#01
assert d = 2#1010
```

Combined Assignment Operations

Turing Plus allows all binary operators, such as +, **shr** and **or** to combine with := to make new assignment statements, for example:

```
i += 1        % i := i + 1
j -= k + 1    % j := j - (k + 1)
x shr= 2      % x := x shr 2
```

Extemal Variables and External Subprograms

Variables can be declared to be at absolute locations, for example:

```
external 16#9001 ttyData: char
ttyData := 'A'
```

This declares *char* variable *ttyData* to be located a hexadecimal address 9001 and assigns 'A' to it. Declaring variables to be at absolute addresses is particularly useful for device management in computer architectures with memory mapped device registers. The syntax for *variableDeclaration* is extended to include the form:

external [*addressSpec*] **var** id [: *typeSpec*] [:= *expn*]

where *addressSpec* is a compile-time expression in the type *addressint* or a compile-time string. As is the case for all Turing *variableDeclarations*, least one of the *typeSpec* and initializing expression must be present.

If the *addressSpec* is omitted, the identifier names an external variable; this name represents an implementation dependent method of locating a variable. For example:

```
external var ERRFLAG: int
if ERRFLAG = 0 then...
```

If the *addressSpec* is a string, this string is used as the variable's external name, for example:

```
external "x" var y: int
```

The external name to link to is x while the name used in the program is y.

Turing Plus extends the allowed use of external subprograms (described in the Turing Report) to allow their declaration within subprograms.

Variable Number of Parameters

Turing Plus adds a construct that allows a subprogram to have a variable number of parameters. This is specified by replacing the type of the last parameter in the subprogram header by dot-dot (..). For example:

```
procedure printf (fmt: string, a: ..)
```

Only the last (right-most) parameter can be so specified, and it must not be specified using **var**. Within the body of the subprogram, the parameter (a in the example) is visible as an *addressint*. Any number of parameters, of any type, can be passed to *a* and to positions following *a*, for example:

```
printf ("Weight of %6.1f%20s", x, t)
```

where x is a real value and *t* is a string value. Each of these parameters is considered to have the type *addressint*. It is intended that the parameters be passed and accessed in the same way that variable numbers of parameters are handled in the C language. This allows a Turing Plus program to call any C subprogram. It also allows any existing C program to be replaced by a Turing Plus subprogram. A common expected use of this feature is access to external C library functions such as *printf*, for example:

```
external procedure printf (format: string, values: ..)
```

The Indirection Operation

The indirection (or dereferencing) operation @ is used to access values of specified types in memory. For example, a value whose type is *nat1*, located at absolute address 246, is written as:

```
natl @ (246)      % Peek or poke location 246
```

The target type has the same possibilities as in type cheats. The location should be an expression returning a value of type *addressint*, described below.

Addresses as Integers

The *addressint* type is a predefined integer or natural type which corresponds to the machine data type used for addresses. On a 16-bit architecture, such as the PDP-11, *addressint* is equivalent to *nat2*. On the 32-bit Vax architecture, *addressint* is equivalent to *nat4*. Note that the *addressint* type is not equivalent to any pointer type.

Example:

```
type r:
    record
        a: int
        b: char(28)
        c: char(11)
    end record
var p: addressint
  ...
r@(p).b{7) := 'B'
```

This example shows an indirection operation, r@(p). This means that a variable of type *r* is considered to be at address *p* in memory, and a character in this variable is being modified.

Size and Addr Attributes

The implementation-dependent attribute *size* returns the size (in bytes for a byte-oriented computer) of a variable or of a *namedType*. For example:

```
var i: nat2
assert size(i) = 2
```

The implementation-dependent attribute *addr* returns the memory address, as type *addressint*, of a variable or non-scalar named constant. For example:

```
var x: real
var a: addressint := addr(x)
```

For example, *nat1*@(*addr*(v)) represents the first byte of variable or non-scalar constant v.

Use of Registers

The form of *variabieDeciaration* is extended to allow the forms:

var register id {, id } : *typeSpec* [:= *expn*]
var register id {, id } := *expn*

A variable declaration can use the keyword **register** only if it is in a subprogram. Register variables cannot be bound to or passed to reference parameters.

The form of *variableBinding* becomes:

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

The **register** keyword can also be used immediately preceding the declared identifiers in const and parameter declarations. The keyword register is a hint to the implementation to attempt to use a machine register to represent the item. The address (*addr*) of register variables cannot be used, and certain type cheats cannot be applied to them.

Subprogram Variables

Variables and constants can contain references to (the names of) subprograms. In the following, *t* is a procedure type, and *u* is a variable of type *t* initialized to be a reference to procedure *p*:

```
procedure p (var i: int, x: real) i := round(x) end p
type t: procedure q (var j: int, y: real)

var u: t := p      % Procedure variable u is initialized to p
u (i, 24.6)        % Call procedure u

var v := u         % Procedure variable v is initialized to u
v (j, 3.14159)     % Call procedure u
```

The implementation assigns the address of procedure p's code to u. Hence *addr*(u) yields the address of the pointer *u*, and not the address of the subprogram's code. The address of the code itself is given by #*u*.

Values of subprogram types can be called, assigned, compared for equality, and passed as parameters, but cannot be operands of numeric, boolean or string operators. To support subprogram types, there is a new type, which has exactly the same syntax as a subprogram header.

The new *typeSpec* is:

subprogramHeader

The name of the subprogram, its parameters, and its return value which appear in this *typeSpec* can not be referenced; they are present only to keep the form of the *subprogramHeader* consistent with headers of subprograms appearing elsewhere in the Turing Plus language syntax.

There is the restriction that a function's header used as a *typeSpec* must have at least one parameter. This restriction allows us the simple rule: a function or procedure name without parameter list, appearing as an expression, is considered to be the name of the subprogram (rather than a call to the subprogram).

Example:

```
function f (i: real): function g (j: real): real
    % f produces a function as result value
    if i > 0 then result sqrt
    else result sin
    end if
end f
```

```
var h: function x (y: real): real := f (4.7)    % h is sqrt
var x: real := h (16.0)                         % x = sqrt (16.0)
var i: real := f (4.7) (16.0)                   % i = sqrt (16.0)
assert x = z and x = sqrt (16.0)
```

Extension of Upper and Lower

The upper and lower attributes are extended to accept parameters of the form

[*moduleId* .] *typeId*

Assuming the type is non-opaque, these return, respectively, the upper and lower bounds of the specified type, which must be a subrange.

Control of Checking

In the Turing language (as opposed to Turing Plus), there is by default complete checking, meaning that all constructs execute according to their mathematical specification, or else a controlled abort occurs. This is called *faithful execution*. The user may choose to gain a degree of efficiency by using an option that eliminates checking. This option is specified by passing the -k flag to the ttc compiler, as in:

```
ttc -k prog.t
```

(Details on such flags can be obtained by the command ttc -A.)

Turing Plus does checking in a similar manner except that it supports new features that are not checkable. As a systems language, a Turing Plus compiler may also choose to omit certain checks in the name of expediency.

In Turing Plus, there are two keywords, **checked** and **unchecked**, that are used to request that certain portions of a program should or should not include checking code. This code checks such things as array subscripts, validity of pointers, initialization of variables, stack overflow and arithmetic overflow.

For example, in the following, the **unchecked** keyword requests the elimination of the checking code for the last two statements in the loop.

```
loop
    a (i) := 0
    unchecked
    b (i + j) := 0
    i := i + 1
end loop
```

The **checked** and **unchecked** keywords can occur wherever a statement can occur. They have effect up to the end of the current block (in the above example, up to "**end loop**").

Turing Plus implementations may check certain constructs even when **unchecked** is requested. For example, if integer overflow causes a hardware interrupt, then this overflow

checking may always remain in effect. The -O (optimize) compiler flag overrides the checked keyword and turns off all checking.

Unchecked Collections

In the Turing language (as opposed to Turing Plus) pointers are checked by default for validity when they subscript collections. The existing Turing compilers *pttc* and *ttc* do this checking by extending each pointer with a "time stamp" field. When the **new** statement creates an object in a collection, it generates a new time stamp. This stamp value is stored with the object and with the pointer that locates the object. Pointer assignment copies the time stamp as well as the actual pointer. The compiler generates extra code to make sure that a pointer's time stamp matches the stamp in the corresponding object.

The Turing Plus compiler provides the same checking by default, but it also provides a way to eliminate the time stamp field, so that pointers become compatible with pointers in languages such as C. This is specified by declaring the collection to be **unchecked**, as in

```
var c: unchecked collection of node
```

(Without the **unchecked** keyword, collections are considered to be **checked**.) Pointers of unchecked collections are represented as machine addresses, of the same size as *addressint*. This allows explicit pointer arithmetic. For example, the following increases pointer p by 200:

```
var p: pointer to c              % c is unchecked
type (addressint, p) += 200
```

Assembly Language Inserts

In Turing Plus the user can specify assembly language to be emitted (generated) using the **asm** statement, which has the form:

asm [*labelstr* :] *str* {, *expn* }

Both *labelstr* and *str* are compile-time string expressions. This construct emits one or more lines of assembly language inline into the generated code for the program. The emitted assembly code consists of:

1. Optionally, an assembly language label, given by *labelstr*, followed a machine dependent character (usually a colon) if appropriate for the target assembler.

2. The *str*, preceded and followed by a machine dependent separator (usually a tab or blank).

3. Representations of each *expn*, separated by a machine dependent separator (usually a comma).

If an *expn* is a compile-time string, its value is emitted. Any other *expn* is evaluated (if necessary), and assembly language notation representing its value is emitted. This evaluation may cause the emission of assembly code; all such emission for the *expn* will precede the line of assembly code specified by the **asm** construct.

Note that the *expn* can use *addr*, *size* and indirection (@) as required to specify an appropriate value to emit. An implementation may not be able to handle certain *expns*, for example, those consuming too many temporary registers. It is the programmer's responsibility to ensure that the types and sizes of expressions are appropriate for the given line of assembly language.

Examples:

```
var vlong, v2: int4
asm "movl", 5, vlong                    % vlong := 5
asm "movl", addr (vlong), v2            % v2 := addr (vlong)
asm "movl", int4@(v2), vlong            % vlong := vlong
asm "movl", size(vlong)+4, int4@(v2)    % vlong := 8
```

7 Exception Handling

Turing Plus adds exception handling, supported by (1) the **quit** statement, and (2) the **handler**, which is a block of declarations and statements occurring optionally at the beginning of a subprogram. The **quit** statement as well as any system detected failures cause a program or process to either abort or to give control to the most recently encountered (activated) exception handler.

Quit Statements and Exception Handlers

A new statement, called "**quit**", is introduced to explicitly cause a program (or process) to fail. The failure (or exception) either aborts the program (or process) or causes control to return to an exception handler (described below).

The *quitStatement* has the form:

quit [*guiltyParty*] [: *quitReason*]

The *quitReason* is an int expression. If the *quitReason* is omitted, a default value is chosen in the following way. If *guiltyParty* is omitted or is <, the default is 1. If *guiltyParty* is > and an exception handler is active, the default is the *quitReason* of the exception being handled; if no exception is being handled, the default is 1. In the case of program abort, an implementation may pass the *quitReason* (integer value) to the operating system.

The *guiltyParty* option is used to designate the point of failure, for example, to tell the debugger (if any) which line of the program is considered to be the cause of failure.

A *guiltyParty* is one of:

a. <
b. >

If *guiltyParty* is omitted, it is assumed that the exception results because of a failure occurring at the **quit** statement. If *guiltyParty* is <, then the failure is considered to have occurred at the point of call to the current subprogram. For example, if the current subprogram is implementing square root (*sqrt*) and is passed a negative argument, it can use < to specify that the caller provided a faulty argument.

GuiltyParty can also be >, which means that the failure already occurred and is being passed on (re-raised). To summarize, the three possibilities for designating the point of failure correspond to:

a. < The caller is cause of failure
b. > The exception being handled is the cause of failure
c. (none) The present quit is the cause of the failure.

Case (b) defaults to case (c) if it occurs when a handler is not active.

An exceptionHandler has the form:

handler (id)
statementsAndDeclarations
end handler

The id is an *int* constant whose value is the *quitReason* of the exception that is being handled.

A handler can appear only in the body of a subprogram or process, preceding the declarations and statements.

The new form of a *subprogramBody* is modified to be:

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

Exactly the same declarations and statements can appear in a handler as can appear in the subprogram body following the handler.

In the absence of exceptions, exception handlers have no observable effect. A particular handler is activated (meaning it becomes ready to handle an exception) when the subprogram or process containing it is entered. It remains active until (1) the subprogram (or process) in which it is located has completed, or (2) the handler is given control. Activation of a handler when a previous handler is already active will cause exceptions to be passed to the newly activated handler. In other words, handlers have a dynamic scope that begins when the exception handler is encountered (but skipped) and ends when (1) the subprogram (or process) containing the handler has completed execution or (2) the handler is given control.

When a handler is given control it becomes, in effect, a replacement for the declarations and statements following it. If the handler is in a function, it must terminate with a **result** statement giving the function's value or with a **quit**. If the handler is in a procedure, the handler must terminate with a **return**, a **quit**, or by encountering the end of the handler (which is equivalent to a return).

When a handler terminates with a **result** or **return** statement (or by reaching the handler's end in a procedure), the subprogram's **post** condition (if any) must be true. If termination of a handler is caused by **quit**, the post need not be true. Similarly, a **quit** outside of a handler does not need to establish the post condition of the associated subprogram.

Example of exception handling:

```
const stackOverflow := 500
const maxTop := 100
var top: 0..maxTop := 0
var stack: array 1..maxTop of int

procedure push (r: int)
    if top = maxTop then
        quit: stackOverflow
    end if
    top := top + 1
    stack(top) := i
end push

  ...

procedure parse

    handler (exceptionNumber)
        put "Failure number ", exceptionNumber
        case exceptionNumber of
            label stackOverflow:
                put "*** Stack has overflowed"

            ... other exceptions handled here ...

            label:          % Unexpected failures
                quit >      % Pass exception further
        end case
    end handler

    parseExpn    % Eventually push is called

end parse
```

The *parse* procedure contains a handler that is activated when parse is called. The parse procedure calls *parseExpn*, which indirectly calls *push*. When *push* executes its **quit** statement, control is returned to the most recently activated exception handler. Assuming no handlers were activated after the handler in the *parse* procedure, control is passed to parse's handler. The interrupted procedures, including *parseExpn* and *push*, are terminated and their local variables are deleted before the handler gains control. (In other words, the implementation stack is trimmed of their stack frames.)

System Generated Exceptions

In a "faithful" implementation of Turing, either the program will execute according to the language definition, or else the system will detect a failure. In the absence of exception handlers, these failures generally cause the program to abort. If one of these failures occurs

while an exception handler is active, then control is passed to the handler. The *quitReason* number passed is implementation dependent. These numbers are documented in the "*%exception*" standard include file. Note that the user may simulate a system exception by doing a **quit** with the corresponding system exception number.

When Turing programs are not "checked" by the implementation, the system may not detect failures. In some cases (dirty cases), failure to detect the failure may yield incorrect data, but with most implementations it should not cause remote corruption. In other cases (dangerous cases), failures may result in arbitrarily chaotic behaviour and remote corruption. For example, an undetected out-of-range subscript may corrupt arbitrary data or code.

The presence of a handler does not automatically cause checking for system exceptions. Rather, it is the "checked" option that specifies what exceptions the system is required to detect. When checking is not in effect, the implementation may assume that the programmer has proven that the exception cannot occur.

System Exceptions are Implementation-Dependent

Since the Turing language does not impose an order of evaluation on subexpressions, exceptions are not necessarily repeatable from implementation to implementation. For example, in this code:

```
i := 0
x := 24 div i + 24/i
```

the exception may be either integer division by zero or real division by zero, depending on order of evaluation.

If checking is turned off or if dangerous language features are used in the program, the program's behaviour becomes highly implementation dependent. Note that an implementation is allowed to do checking even if checking is turned off; of course a programmer may choose to take advantage of his or her knowledge of one particular implementation to produce an implementation dependent solution to a particular problem.

Trouble

If an exception occurs in certain situations and is subsequently caught by a handler, it may be difficult to continue the computation. For example, if an exception occurring in a monitor is caught outside of the monitor, it may never again be possible to re-enter the monitor. If an exception occurring in a process is not caught, the entire program is aborted. If an implementation allocates dynamic arrays on the heap, an exception leaving the scope of these arrays may prevent de-allocation of their allocated space.

Implementation Considerations for Exception Handling

Exception handling in Turing Plus is designed so that it has no effect on the performance of those parts of a program that do not use the feature. The only overhead in time or space occurs when a handler is activated or de-activated. Introduction of handlers should cause no overhead in subprograms not containing them.

Handlers can be supported by a LIFO linked list of nodes representing active handlers. For programs without concurrency the head of this list can be an implicit global variable. When there are processes, there is a list for each process, with the header in the process's descriptor

(which is used to hold other process status, such as registers, when the process is not executing).

The nodes of the list are essentially implicit local variables held in the runtime stack. Each time a handler is encountered, it is activated by pushing (a copy of) its node onto the stack, and linking it to be first in the list of nodes. The handler is deactivated by unlinking from the list. The allocation of a node on the stack can be done by assuming that its handler uses the implicit declaration:

```
var state:
    record
        ... fields to hold stack info ...
    end record
```

Note that before a handler is activated, its node is unlinked from the list, thereby guaranteeing that an exception in this handler does not re-activate (the same instance of) this handler.

Note that before a program or process begins execution, the header of its list of nodes must be initialized to locate a default (system) handler. For a non-concurrent main program, this default handler can abort the program, passing the *quitReason* to the operating system.

Some hardware, such as the Digital Equipment Vax, provides special subprogram calling instructions that support exception handling without explicit software intervention. Even without hardware support, it is possible to completely avoid time (CPU) overhead in activating and de-activating handlers by using tables that give intervals (windows) of program counter values that are covered by a particular handler. When an exception occurs, a window is searched for which brackets the current program counter (or brackets a return point from an active subprogram).

The exception that corresponds to running out of run-time stack can cause an unbounded exceptioning situation (repeated stack overflows) if care is not taken. However, this situation is avoided by using this strategy: activating a handler must not entail any further pushes onto the implementation stack. Effectively, all that needs to be done is to load registers from the latest node in the exception list, without doing any pushes onto the stack.

8 Concurrency

This section defines the lightweight concurrency features of Turing Plus. These features are explicitly designed to support multiple CPU systems as well as single CPU systems. They are essentially the same as those in Concurrent Euclid, with the addition of:

1. The **fork** statement, to explicitly activate processes.

2. Device monitors, which run at hardware priorities and which can contain interrupt handling subprograms.

3. Deferred conditions, such that the signaller continues before the awakened process.

Processes and the Fork Statement

The **fork** statement activates a new process, for example:

```
process speak (word: string)
    loop
        put word
    end loop
end speak

fork speak ("Hi")
fork speak ("Ho")
```

The process declaration creates a process template, which is activated twice in parallel in this example. One activation repeatedly prints "Hi", and the other "Ho".

The syntax of a *processDeclaration* is:

process id [*parameterList*] [: *compileTimeExpn*]
[*importList*]
[**pre** *booleanExpn*]
[**init** id := *expn* {, id := *expn* }]
[**post** *booleanExpn*]
[*exceptionHandler*]
declarationsAndStatements
end id

The optional *compileTimeExpn* in the header is used to specify the maximum size of the process's stack. A process declaration can appear wherever a module declaration is allowed. The declarations and statements in a process are the same as those in subprograms. A process cannot be declared in a monitor.

The syntax of the **fork** statement is:

fork [*moduleId* .] id [(*expn* {, *expn* })] [: *reference* [, *expn* [, *reference*]]]

The first optional *reference* at the end of the **fork** statement must be a boolean variable. The fork statement sets this reference to true if the fork succeeded. If it failed (presumably because stack space could not be allocated), this reference is set to *false*. If the fork fails, but this reference is omitted, an exception is raised. The optional *expn* specifies the size for the process's stack; this overrides the (optional) stack size given in the process declaration. The second optional reference must be a variable with the predefined type *addressint*, which is set to identify the process activation. It has an implementation dependent meaning; it (usually) locates the process's process descriptor.

Monitors

The form of *monitors* is similar to that of modules:

monitor id [: *compileTimeIntegerExpn*]
...
end id

The implementation will guarantee that only one activity at a time takes place in a monitor. This means that a process will be blocked if it calls a monitor that is already active. The blocked

process will not be allowed to proceed until the monitor is inactive (that is, until another process leaves the monitor). If the optional compile time integer expression is present, this is a device monitor, having its exclusive access enforced by a machine-dependent trick such as executing it at the hardware priority level given by the expression.

The body of a monitor has the same form as that of a module, except that (1) modules, monitors and processes cannot be declared inside monitors and (2) certain additional statements (**signal** and **wait**) are allowed to appear (only) in monitors.

A device monitor is restricted from calling other monitors (directly or indirectly) and from calling separately compiled modules. This restriction is imposed to eliminate the possibility of blocking a process with a non-zero hardware priority (as this would inadvertently allow multiple entry into a device monitor).

Conditions

Within a monitor, *conditions* can be declared using a new form of variable declaration.

var id {, id } : [**array** *subrangeType* {, *subrangeType* } **of**] **condition** [*conditionOption*]

A *conditionOption* is one of:

a. **priority**
b. **deferred**
c. **timeout**

The **priority** option requires that corresponding **wait** statements (see below) must include priorities. Forms (b) and (c) declare *deferred* conditions. With a **deferred** condition, the signalled process becomes ready to re-enter the monitor when it becomes inactive. This is as opposed to an immediate (non-deferred) condition, in which a **signal** statement immediately wakes up a sleeping process (as is the case in Concurrent Euclid). All conditions in a device monitor must be deferred.

The **timeout** option means that signalling is deferred with a timeout, and that an extra parameter to the **wait** statement must give a timeout interval. If a process waits longer than its timeout interval, it is automatically signalled. Beware that the *empty* function can be non-repeatable when applied to timeout conditions. For example, *empty* (c) may not equal *empty* (c) in a single expression.

Note that conditions cannot be named as types, and can only be declared in the main body of a monitor. They cannot be contained in records, unions, or collections, and cannot be declared inside statements (such as **begin** or **loop**) or subprograms.

Signal and Wait Statements

The syntax of the **signal** and **wait** statements is:

signal *conditionVariable*
wait *conditionVariable* [, *expn*]

In both cases the variable must be a condition variable. The **signal** statement wakes up one process from the specified condition queue, if such a process exists. If the condition is

deferred, the signaller continues in the monitor, and the awakened process is allowed to continue only when the monitor becomes inactive. If it is an immediate (non-deferred) condition, the awakened process immediately continues its execution in the monitor, and the signaller waits until the monitor is inactive to continue.

If it is a **priority** condition, the **wait** must include the optional non-negative priority *expn*, which will be used to order processes waiting for the condition, lower priorities first. An implementation may limit the range of priorities in a priority condition.

If it is a **timeout** condition, the **wait** must include the optional non-negative number timeout *expn*, which gives the timeout interval.

There is no guaranteed order of continuing among deferred awakened processes, processes signalling immediate conditions, and processes attempting to enter the active monitor.

The Pause Statement

The syntax of the *pauseStatement* is:

pause *expn*

The *expn* must be a non-negative value. The process is held up by the number of time units given by the expression. These units may vary from implementation to implementation, but 1 millisecond is recommended. In simulation versions of Turing Plus simulation time rather than real time may be used as the basis of the pause.

Predefined Subprograms

The following new predefined subprograms are introduced:

setpriority (p: nat)

This procedure sets approximate relative speed of this process. This speed cannot generally be counted on to guarantee critical access to shared variables. A smaller value of p means a faster speed. The argument of *setprioirty* may be limited to the range 0 .. 2*15 - 1.

getpriority: nat

This function returns the relative speed of the executing process.

empty (*conditionVariable*): boolean

The result of this function is true if no processes are waiting for the condition.

Interrupt Handling Procedures

A parameterless procedure exported from a device monitor can be specified to be an interrupt handling procedure by suffixing a device specification to its header.

procedure id [: *deviceSpecification*]

The *deviceSpecification* is a compile-time natural number that identifies, to the implementation, the class of interrupts that effectively call this procedure. Note that interrupt handling procedures may not be called from within a program.

The user is expected to follow two programming disciplines. The first is: an exception must never propagate out of an interrupt handling procedure. If such an exception should occur, it may be passed to the handler that was active at the time of the interrupt. The second is that an interrupt handling procedure must not execute a wait, either directly, or indirectly by calling another procedure. If this should happen, it may block the interrupted process.

Implementation Considerations

The concurrency features of Turing Plus have been designed to be very efficient. For a single CPU implementation, entering or leaving a monitor should require about two machine instructions in the usual rase of no contention. Similarly, a signal on an empty queue should require only about two machine instructions.

References

[Holt & Cordy 1983]
R.C. Holt and J.R. Cordy, The Turing Language Report. Technical report CSRI-153, Computer Systems Research Institute, University of Toronto, December 1983 (Revised August 1986).

[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.