THE TURING PROGRAMMING LANGUAGE

Turing, a new general purpose programming language, is designed to have Basic's clean interactive syntax, Pascal's elegance, and C's flexibility.

R. C. HOLT and J. R. CORDY

Turing is a general purpose programming language designed for convenient development of reliable, efficient programs. Its language design goals include ease of learning, concise and expressive syntax, graceful and efficient treatment of errors, control of program complexity, mathematically precise language definition, and small, fast implementation. (See the accompanying box, "Historical Development of Turing," on p. 1412.)

Turing is designed to be a *life cycle* language, with applications ranging from teaching programming concepts to children (where Turing is an alternative to languages like Basic and Logo) to development of serious software (where Turing is an alternative to languages like Pascal). The Turing Plus extension of Turing supports a greater range of applications, including systems programming (where the language is an alternative to C, Modula, and Ada).

Turing's design supports *faithful execution*; this means that a program is either executed according to the language's semantics or else halted with a message describing the nature of the failure. Turing is designed to be amenable to formal semantic definition.

The development of Turing included creating the Turing Report [13] (the informal definition of Turing), the Turing textbook [16] (for teaching programming using Turing), the formal definition of Turing [19] (a mathematical specification of the language), a portable Turing compiler and interpreter (as of 1987, running on Digital Vax minicomputers, IBM mainframes, IBM PC compatibles and Sun/68000 workstations) with associated interactive programming environments. As well, the Turing Plus extension is defined by its own report [17] and is implemented by a self-compiling portable compiler currently running on Sun workstations and Vaxes and generating MC68000 assembler, Vax assembler or C source. The quality of code produced for Turing, when optimization is requested, is essentially the same as that for C.

This article focuses on the structure of the Turing language giving examples of its use and explains how Turing has been formally defined. The extension to Turing Plus is also discussed, and then compared with Ada and Modula 2.

OVERVIEW OF THE TURING LANGUAGE

Turing can be thought of as a *super Pascal*, in that it contains essentially all of Pascal's features, while adding needed features such as dynamic arrays, modules, and varying length strings. The main features of Turing are summarized in Figure 1 (The reader may want to look ahead to Figure 3, which lists the additional features of Turing Plus). We begin by briefly discussing the structure of Turing, contrasting its features with those of other languages like Pascal and Modula 2 and then giving examples of these features.

A Turing program consists of statements and declarations, and these contain expressions. Statements in-

clude assignments to variables, loops, **if** statements and **case** statements. Declarations are used to name items such as variables and subprograms. Turing's scope rules, which determine the accessibility and lifetime of these names, are inherited from Pascal, which, in turn, inherits these rules from Algol. However, Turing modifies these rules in certain ways, most importantly by having **import** and **export** lists in modules to constrain the usual scope rules. Another modification is that Turing disallows redeclaration of names in a major scope (a subprogram or module) on the grounds that these redeclarations are commonly errors.

Names (identifiers) are like those in Pascal, except Turing allows underscores, long names (up to 50 characters), and is case-sensitive (so *A* is distinct from *a*). Expressions in Turing are much like those in Pascal with the following notable exceptions. First, Turing has two more operators: ** (exponentiation) and => (implication). Second, Turing's precedence rules allow certain expressions that are prohibited by Pascal; for example, in Turing one can write "$i = 1$ **or** $j = 1$," while the equivalent expression would require extra parentheses in Pascal.

DATA TYPES AND OPERATORS
- Integers, real numbers, booleans, enumerated types, sets, arrays, records.
- Pointers and collections with **new** and **free** statements.
- Unions (secure variant records) with a **tag** statement.
- Varying length strings.
- Named types (as in Pascal).
- Operators: +, −, *, /, **div**, **mod**, ** (exponentiation), <, >, <=, >=, =, **not=**, **not**, **and**, **or**, => (implication), **in**, **not in**.

DATA DECLARATIONS
- **var** and **const** declarations.
- **bind** declaration (replacement of Pascal's **with**).

INPUT/OUTPUT
- **get**, **put** statements with formatting and end-of-file detection.
- *open*, *close* procedures to access files by name or command line position.

CONTROL CONSTRUCTS
- **if** statements with **elsif** clauses and **else** clause.
- **case** statements with otherwise clause.
- **loop** and **for** statements with exits

SUBPROGRAMS AND MODULES
- Procedures with **return** statements.
- Functions with **result** statements and with scalar or nonscalar results.
- Dynamic array and string parameters.
- Subprograms as parameters.
- Modules with **import** and **export** lists for information hiding.

ASSERTIONS
- **assert**, **pre**, **post**, **invariant** (giving correctness requirements).

PREDEFINED SUBPROGRAMS
- Mathematical functions: *abs*, *max*, *min*, *sign*, *sqrt*, *sin*, *cos*, *arctan*, *sind*, *cosd*, *arctand*, *ln*, *exp*.
- Type transfer functions: *ceil*, *round*, *intreal*, *chr*, *ord*, *intstr*, *strint*, *erealstr*, *frealstr*, *realstr*, *strreal*.
- Random number generation: *rand*, *randint*, *randomize*, *randnext*, *randseed*.

FIGURE 1. Features of Turing

Two aspects of Turing's syntax make its programs have a simpler appearance than comparable Pascal programs. The first is that Turing programs require no header or trailer; instead of starting with a line like "program test(input, output)," a Turing program begins immediately with its first statement or declaration. The second is that semicolons are completely optional in Turing programs, and generally are not used. At the same time, Turing programs are format-free, meaning that declarations and statements can be split across lines or can appear several to a line with no special notation.

Turing's comments begin with /* and end with */ and can extend for several lines. More commonly *single line comments* are used, which begin with % and end at the end of the current line. Turing allows declarations to be placed wherever statements can appear. Each declaration effectively creates a new scope, which lasts until the end of the enclosing construct, e.g., to the end of the enclosing **then** clause, loop, or subprogram.

Variables, including nonscalars, can be initialized in their declarations. For example, "**var** *i*: **int** := 7" declares *i* to be an integer with an initialized value of 7. If the declaration contains an initial value, the type can be omitted; for example, "**var** *i* := 7" is an equivalent declaration. The omitted type is inferred from the type of the initial value. Constants are declared as in Pascal; for example, "**const** pi := 3.14159" creates a constant. Turing's constants are more general than Pascal's in that they need not be compile-time values. For example, given variables *x* and *y*,

const hypot := sqrt(*a* ** 2 + *b* ** 2)

creates a run-time constant (a named, computed value that cannot change in its scope).

Turing's varying length strings are much like those in PL/1 and various dialects of Pascal and Basic. As will be illustrated in an example, Turing has a particularly concise method of specifying substrings. The operator for string catenation is +.

Turing's arrays can be dynamic, meaning their size may not be known until run time. The notation upper(*a*) gives the upper bound of array *a*. Subscripting, as in Fortran, uses parentheses, so *a*(*i*) is the element of array *a* selected by subscript *i*.

Turing supports dynamic allocation, using **new** and **free** statements, in the general style of Pascal. However, Turing uses **collections** to collect related items that are to be allocated. Pointers select elements of collections, so *c*(*p*) is the element of collection *c* selected by pointer *p*.

Turing's version of variant records, called **unions**, provides a type safe method of overlaying storage and changing the structure of data at run time. A special statement, the **tag** statement, is used to change the tag data in a union.

Turing's control constructs include the **if** and **case** statements. The **if** statement allows multiple **elsif** clauses for cascaded selection, as well as an **else** clause.

The **case** statement allows an otherwise clause. Turing has a counted **for** loop, whose counter is local to the loop. Pascal's **while** and **repeat** statements are replaced by an indefinite **loop**. The **exit** statement is used to terminate execution of both **loop** statements and **for** statements. All nested constructs are syntactically ended with the keyword **end** followed by the name of the construct; for example, an **if** statement has the form: **if . . . end if**.

Turing has stream oriented input/output statements called **get** and **put**. These allow transfer of values of basic types (integers, real numbers, and strings). They support flexible formating, including field width, number of fractional digits and number of exponent digits. The *open* and *close* procedures support access to files by file name or by command line position. An end-of-file function is used to detect that a file contains no more characters.

Like Pascal, Turing has two kinds of subprograms: procedures and functions. Turing provides a **return** statement to allow early return from a procedure. There is also the **result** statement that produces the value of a function. Functions can produce nonscalar results, such as records and strings. Parameters can be dynamic, meaning arrays, and strings of various sizes can be passed to a given subprogram. Subprograms can be passed as parameters.

The module construct of Turing is similar to the corresponding feature in the Modula language. It can be thought of as an envelope that encloses declarations of data and subprograms, and allows only those explicitly *exported* from the module to be visible outside. Modules allow the program to hide information about details of an implementation, preventing inadvertent access to these details. Modules can be nested.

Turing is designed to encourage the use of program correctness methods. It includes a set of *assertions*, namely **assert**, **pre**, **post**, and **invariant**, which contain boolean expressions that should be true when executed. They specify correctness requirements for the

Historical Development of Turing

Turing was born of an effort to find an acceptable language for instructional computing in the Computer Science Department at the University of Toronto. In 1982 the department was faced with the task of selecting a language to replace PL/1, which had been used to teach programming in the department for a decade. PL/1's advantages include a varied and powerful set of language facilities, good texts and manuals, efficient compilers (including SP/k and PL/C [1], and reasonably wide acceptance, but it suffers from a number of shortcomings: it is a large, clumsy, and sometimes ill-defined language. Pascal [23], by contrast, is quite elegant. During the last decade, Pascal compilers and textbooks have become widely available, so Pascal was a prime candidate to succeed PL/1. Pascal, however, suffers from its own shortcomings [25, 28]. Compared to PL/1, Pascal has poor character string manipulation, limited numeric facilities, no local scopes (no declarations in **begin** blocks), and no dynamic arrays. In many ways, PL/1 is a more serious language than Pascal.[1]

As of 1982, the Concurrent Euclid (CE) language [2], [14]—also developed at the University of Toronto—was being used extensively in systems programming. Because CE has a precise mathematical foundation, incorporates up-to-date software concepts such as modules, and enhances reliability and efficiency, it also became a candidate, along with Pascal, to succeed PL/1. Unfortunately, CE does not have the convenient level of programming facilities needed for introductory programming; its input/output is complex, its string handling rudimentary, and its syntax verbose.

After considerable discussion and investigation, the department decided to develop a new language for instructional computing. Originally the language was called New Euclid, because it borrows heavily from CE and Euclid [26], but as the new language evolved, it took on a personality of its own. Consequently, it was given its own name, Turing, in honor of Alan M. Turing [11], one of the pioneering geniuses of computer science. From August 1982 until August 1983, an intensive research and development effort brought the Turing language into existence. (See "Turing: An Inside Look at the Genesis of a Programming Language" [15] for an account of the human interest side of this effort.) During this period, the notation of the language was designed and polished (by R.C. Holt and J.R. Cordy), a textbook was written (by R.C. Holt and J.N.P. Hume), and Turing compilers were developed for the Digital Vax/11 and the IBM/370. The compiler team consisted of J.R. Cordy, R.C. Holt, M.P. Mendell, S.G. Perelgut, S.W.K. Tjiang, and P.A. Matthews, later joined by A.F.X. Curley, C.B. Hall, and R. Wessels. The Turing compiler was written in CE using the S/SL compiler construction technique [12]. Further support for the language, including an interpreter and interactive environment, has been developed by J.R. Cordy, T.C.N. Graham, C.B. Hall, M. Robertson, and N. Eliot [4–6].

By the fall semester 1983, 3,000 University of Toronto students were using Turing on five computers (an IBM 3033 and four Vaxes). By fall 1984, an IBM PC version of Turing was also being used in computer science courses. With the addition of Turing Plus features, the language is now also used in upper level courses in data processing, systems software, compiler writing, and operating systems.

Turing has since been adopted for teaching or research in a number of other universities, including York University (Toronto), University of the Pacific (Stockton, California), Columbia University (New York), Concordia University (Montreal), and Queen's University (Kingston, Ontario). It has replaced Pascal or Basic in a number of Ontario high schools, where it is being used on IBM PCs. High school teachers report that Turing is easier to learn than Basic.

Further support for the language, including interactive programming environments and interpreters, has been developed at the University of Toronto and Queen's University [4], [5], [6]. Turing has also been used as the basis for further language design. For example, T.E. Hull developed Numerical Turing, which supports dynamic choice of floating point precision [20].

[1] The Pascal being referred to here is the language defined by the International Standards Organization [22]. not one of the many incompatible implementations of the language.

Turing program. These are (optionally) evaluated at run time, and cause program abortion if false. Using these assertions and Turing's formal semantics, a program can, at least in principle, be proven correct.

Turing also features *short forms* for frequently used constructs. For example, "$i := i + 1$" can be shortened to "i += 1" and the keyword **procedure** can be shortened to **proc**.

In Pascal, like most programming languages, certain things are disallowed, but the processor (compiler and run-time system) need not detect them. The Pascal standard calls these things "errors" [22]. Use of an uninitialized value in Pascal is an example an error. Turing language design is notable for eliminating all such errors. In other words, if the Turing Report disallows something, the compiler or run-time system is required to detect any violations. For example, the run-time system is required to detect uninitialized variables. With the elimination of "errors," Turing has the property of *faithful execution*, meaning the execution of a program always behaves exactly according to the language report, or else a message reports the nature of the problem. Things are actually a bit more flexible than has been explained; in particular, the Turing user can optionally eliminate the checking for such errors, in which case they may not be detected.

EXAMPLES OF FEATURES OF TURING

Now that we have given an overview of Turing, we will present some examples that illustrate features of the language. No attempt will be made to precisely define Turing; rather, we want to show the language's style of usage. The Turing textbook [16] provides an introduction to programming using Turing, the Turing Report [13] gives a specification of the language, and "Design Goals for the Turing Language" [18] describes why Turing features have their particular form and meaning.

Two Short, Complete Programs

The complete Turing program to print *Hello* is simply

```
put "Hello"
```

To output a sequence of *Hello*'s and *Goodbye*'s on separate lines, the complete Turing program is

```
loop
  put "Hello"
  put "Goodbye"
end loop
```

It is instructive to consider the equivalent in Pascal:

```
program test(output);
  begin
    while true do
      begin
        writeln("Hello");
        writeln("Goodbye")
      end
    end.
```

Pascal requires header and trailer lines. It also requires semicolons at the ends of some lines, for example after the line that outputs *Hello*, but not after other lines. If a student accidently replaces the final period by a semicolon, the program is no longer legal. If he or she puts a semicolon after **do**, the meaning of the program is drastically altered into an infinite loop with no output.

Dynamic Arrays

Here is an example in which the size of an array is not known at compile time:

```
var n: int
put "Give size"
get n
var a: array 1 . . n of real
. . . use dynamic array a . . .
```

Variable *a* is a dynamic array, whose size is determined at run time by the value of *n*, which is input in the third line.

Varying Length Strings

This example illustrates Turing's varying-length strings, which standard Pascal does not have:

```
var fruit: string
fruit := "pine"
fruit := fruit + "apple"   % fruit becomes pineapple
put fruit(1 . . 2) + fruit(6) + fruit(*)   % Outputs pipe
```

The second line sets fruit to be *pine* and then the third line catenates *apple* onto the end to set it to *pineapple*. The third line illustrates a Turing comment, beginning with a percent sign (%) and continuing to the end of the line. The final line catenates three substrings of fruit, its first two characters, its sixth character, and its last (*) character, to produce *pipe*.

Printing a Triangle of Stars

This example is a complete Turing program that prints a triangle of stars:

```
var s := "*"
loop
  put s
  exit when length(s) = 60
  s := s + "*"
end loop
```

The first line declares *s* to be a variable and initializes it to "*", implicitly declaring *s* to be of type **string**. The **put** statement outputs the value of *s* (a string of stars), and the assignment statement catenates another star onto the end of *s*. The **exit** statement terminates the loop when the string of stars is 60 characters long. By default, the maximum length of a string is 255 characters. This maximum can be explicitly controlled; for example, the type of *s* could be given as **string**(60). If the maximum length is exceeded, the program is halted (assuming checking is enabled) and an appropriate message is output.

Searching a File

The next example program reads lines and outputs any that contain the pattern *iron*, along with the number of the line. For example, it might print out:

```
6 and then the iron filings are mixed with sulfur,
    producing
23 touch of irony in his voice, exposing his years
    of anguish
```

The complete Turing program is

```
% Read file, print lines containing "iron"
var line: string
var lineNumber := 0
loop
  exit when eof
  lineNumber := lineNumber + 1
  get line: *   % Read an entire line
  if index(line, "iron") not = 0 then
  % Iron occurs in line?
put lineNumber: 4, " ", line
  end if
end loop
```

In the declaration for lineNumber (the third line), the type is integer, because lineNumber is initialized to the integer zero. The **get** statement uses a format of star (*) which means that the entire line is to be input. There is an extensive set of convenient input formats, which can be used to input a token (word) at a time or a given number of characters. The index function, used in the **if** statement, returns a nonzero value giving the position of *iron* in the input line (or zero if *iron* does not exist in the line). The **put** statement uses a format to print out lineNumber in a field of width four. By default, each **put** statement ends the current output line (by outputting a new line character), but if dot-dot (. .) appears at the end of the **put** statement, the line will be left to be completed by further output.

This program can read from the standard input stream (such as the terminal) or it can as well be redirected to read from a file. Turing includes facilities to open files by name. The **get** and **put** statements are used to access these files.

Summing an Array

The next example illustrates functions, dynamic array parameters, and **for** statements:

```
% Read and sum a list of numbers
var arraySize: int
get arraySize   % Read in size for array a
var a: array 1 . . arraySize of real
                        % Run-time upper bound
function sum(b: array 1 . .* of real): real
  var total: real := 0.0
  for j: 1 . . upper(b)   % For each array element
    total := total + b(i)
  end for
  result total
end sum
. . . compute elements of array a . . .
put "Sum of array:", sum(a)
```

The star(*) in the header of the sum function specifies that the size of formal parameter *b* is to be taken from the actual parameter, which is *a* in this example. The **result** statement terminates execution of the function and provides its value which is total.

The **for** statement serves as a declaration of its counter, *j* in this example; no previous declaration of *j* is needed or allowed. Notice that the form of a **for** statement is analogous to a declaration such as **var** *i*: 1 . . 19. The scope of *j* is the body of the **for** statement. Within the body, *j* is considered to be a run-time constant, meaning that it cannot be changed. Its value changes only between iterations, when it is automatically incremented by one. One form of the **for** statement counts down and another takes a named subrange type and iterates across the type.

Just as *j* is considered to be a constant, so is array parameter *b*. This means that once the function begins execution, *b*'s value is set and cannot change during the activation of the function. This is different from Pascal, in which *b* would act as a local variable. A parameter in a procedure can be declared as **var**, meaning that the value can be changed; these changes also occur in the corresponding actual parameter.

The limit of the **for** statement, upper(*b*), gives the upper bound of array *b*.

Assertions

The next example illustrates the use of **pre**, **post**, and **assert** constructs. These assertions are used for specifying the purpose of programs and checking at run time that key assumptions are being satisfied. This function accepts an integer called *cents*, e.g., 5, and returns its equivalent dollar string, i.e., $0.05.

```
function intDollar(cents: int) dollars: string
  pre cents >= 0
  post dollars(1) = "$" and dollars(* −2) = "." and
    cents = strint(dollars(2 . . * −3) +
                                  dollars(* −1 . . *))
  var t: string := intstr(cents)
                        % Convert cents to a string
  if length(t) = 1 then
    t := "00" + t
  elsif length(t) = 2 then
    t := "0" + t
  end if
  assert length(t) >= 3
  result "$" + t(1 . . * −2) + "." + t(* −1 . . *)
end intDollar
```

The **pre** assertion documents the fact that this function should not be called with negative values (otherwise it can produce anomalous results such as $0.−6). The **post** assertion documents requirements for the function's result. In the header for the function, the declaration of dollars gives a name to this result. The scope of dollars is limited to the **post**, because that is the only place that the function has a value. The **post** specifies that the result dollars should start with $, should have a decimal point two characters before the last character,

and should have the same value as cents when $ and decimal point are removed and the string is converted to be an integer. Together, a **pre** and **post** can specify the purpose of a subprogram. We say the subprogram is *correct* if, given a true **pre** expression, the subprogram body necessarily leads to a true **post** expression.

The intstr and strint predefined functions are part of the complete set of Turing-type transfer functions that convert among the types **int**, **real**, and **string.**

In this example, the purpose of the **if** is to make *t* at least three characters long, so there will be at least one digit preceding the inserted decimal point. The **assert** statement documents this purpose with its expression length(*t*) >= 3. This is an example of where a comment, such as

```
% t should now have a length of at least 3
```

can be replaced by an equivalent *machine interpretable comment* (in other words, by an assertion). During testing and maintenance, this assumption about the length of *t* will be checked by generated code. Once testing is complete, if execution time is critical, the code for all assertions and checking can be removed by compiler option, giving performance that matches machine oriented languages like C. In certain applications, such as secure operating systems, it may be appropriate to retain this run-time checking in the production version of the software.

Binding to a Variable

The next example illustrates the use of Turing's *bind* declaration.

```
type r:
  record
    s, t: string(15)
    n: int
  end record
var a: array 1 . . 500 of r
var i, j: int
. . .
bind var x to a(i)
x.s := "C.E. Shannon"
```

The **bind** specifies that *x* is to be another name for *a*(*i*). The **var** keyword specifies that *x*, and hence *a*(*i*), may be changed in the scope of *x*; for example, the last statement assigns to a field of *x*. If **var** is omitted, the declared item cannot be changed. The binding is by reference in that changes to *i* in the scope of *x* do not affect the element to which *x* is bound. Several items can be bound to in a single declaration, for example, **bind var** *x* **to** *a*(*i*), **var** *y* **to** *a*(*j*). The equivalent construct in Pascal

```
with a[i], a[j]
```

is problematic in that the field names of the two records are identical, so only the fields of *a*[*j*] are visible.

Linked Lists

The next example illustrates the use of Turing's pointers. These are equivalent to Pascal's pointers, but in Turing, each pointer is restricted to locate items in a particular *collection*. These collections can be thought of as arrays that are initially empty and have elements added to them by the **new** statement and deleted by the **free** statement.

```
% FIFO, singly linked list of names
var list: collection of
  record
    next: pointer to list
    name: string(30)
  end record
var first, last: pointer to list := nil(list)   % Empty list

procedure append(p: pointer to list)
  if first = nil(list) then
    first := p
  else
    list(last).next := p
  end if
  last := p
  list(p).next := nil(list)
end append
. . .
var item: pointer to list
new list, item   % Allocate a node
list(item).name := "A.M. Turing"
append(item)   % New node goes onto queue
```

In this example, list is a collection of records whose fields contain names and pointers to other such records. The declarations of first, last and item create pointers for this collection. Conceptually, these pointers are subscripts of list, as in list(item), and they select particular elements of list. The notation corresponding to list(item) in Pascal is item@, where @ may be written as an up arrow. The initialization of first and last assigns them the value nil(list), which is the nil pointer for this collection. The **new** statement creates an element or node in the list collection and then the string "A.M. Turing" is assigned to the string field of this element. The call to the append procedure adds this element to the singly linked list.

Turing's faithful execution guarantees that every undefined run-time action, such as using a dangling pointer, is detected. Consider this example:

```
free list, item   % Delete a node
list(item).name := "J. VonNeuman"
                              % Use of a dangling pointer
```

Once the element located by item has been deleted by **free**, it no longer makes sense to attempt to assign to that element's name field. With checking enabled in Turing, this violation is automatically detected at run time.

Variant Records

The next example illustrates Turing's variant records. These are called *unions* and are analogous to Pascal's corresponding feature, but with the advantage of allowing efficient run-time checks for legitimate access to the labeled alternatives.

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

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

var v: vehicleRecord := init(vehicle.farm, "dairy")
                        % Initialize tag and farmClass
. . .
v.farmClass := "vegetable"
                              % Checks that tag is farm
. . .
tag v, vehicle.passenger
v.cylinder := 4   % Checks that tag is passenger
```

The first line declares an enumerated type called vehicle. Each value of this type is written as the type name followed by a dot and the enumerated name, for example, vehicle.farm. This prefixing of the type name, which is not done in Pascal, clears up questions about the scope of names of enumerated values in complex situations—for example, when the type appears inside another type, is exported, or contains names that collide with other names. In Turing, the naming rules for enumerated values are essentially the same as the rules for fields of records.

The vehicle type is used as the type of the *tag field* of the union, which is called *kind*. Each label of the union is a value of this type. The union value *v* is initialized in its declaration by an **init** construct that sets the tag to vehicle.farm and the farmClass to "dairy." This initialization is optional. The **init** construct is also used to initialize arrays and records in their declarations. Changes to the tag field can only be done by the **tag** statement, which is effectively an assignment statement specialized for this purpose. When checking is enabled, the **tag** statement destroys (uninitializes) all union fields, such as cylinders and farmClass, before setting the tag field.

Any access to a union field is preceded by an optional run-time check to guarantee that the tag is set to activate this field. This is analogous to the subscript checking that precedes access to array elements. This check is necessary for faithful execution, for without it, as happens in Pascal, representations of one type may be interpreted as values of another. The name of the tag can be omitted in a union declaration, but its type must still be given and the **tag** statement is still required when changing from one set of active union fields to another.

Modules

The final example illustrates the module construct, which is available in languages like Modula 2 and Ada, but is missing from standard Pascal. Modules, sometimes called *packages* or *clusters*, are perhaps the most important feature introduced into programming languages in the last decade and should be taught even in introductory programming. The module construct has a pleasingly simple form in Turing:

```
module stack   % Implements a stack of integers
  export(push, pop)

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

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

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

The details of the implementation of the stack module are not visible outside the module. Only the push and pop procedures, which are exported in the second line, are visible outside. The module construct supports "information hiding," an essential software engineering concept that allows us to separate the specification of a module such as stack from its implementation.

FORMAL DEFINITION OF TURING

One of the major design goals of Turing was that the language should be amenable to formal definition. As each language feature was designed, it was scrutinized to make sure that existing language formalization techniques could be applied to it making it possible to produce a formal definition that specifies essentially all aspects of the language. We will now briefly discuss the purpose of this definition and the form it takes. (For a more extensive discussion of these topics and more references, the reader may wish to see [19].)

Why Formalize?

Although the Turing Report [13] provides, in principle, a complete but informal definition of the language, there are good reasons for providing a formal definition of the language as well.

The foremost reason is that mathematical notation provides a degree of precision that simply is not possible in natural language descriptions. A formal definition also assists the implementor. For example, the Turing's context-free syntax has been automatically translated into a parser by an LR(k) parser generator, and Turing's formal context conditions have served as a guideline to a programmer implementing semantic analysis for a Turing compiler. In principle, the definition could be used to prove the correctness of a Turing implementation, although that is probably not feasible at present.

A formal definition is also helpful to the Turing pro-

grammer. Turing's formal semantics can serve as a basis for either proving programs correct or for methodologies for developing correct programs [7, 8, 10].

Turing's Formal Syntax

The formal definition of Turing is divided into the two parts: *syntax* and *semantics*. The first part, syntax, determines which strings of characters are (legal) Turing programs, and the second specifies the meaning of these programs.

The syntax of Turing effectively specifies the "form" of programs, without being concerned about their meaning. The reader is warned that the term *syntax* is sometimes used elsewhere to mean context-free syntax. Here we use the term in a wider and more traditional sense to mean the specification of the set of strings in a given language.

Turing's formal syntax is defined by a boolean function called *syn* that maps a string *s* of characters to true iff the string is a Turing program. The syn function is defined in terms of three other functions: *lex*, which defines Turing's lexical structure; *cfs*, which defines its context-free structure; and *cc*, which defines its context conditions (also called context-sensitive syntax or static semantics):

$$\text{syn}(s) =_{\text{def}} \text{lex}(s) \;\&\; \text{cfs}(s) \;\&\; \text{cc}(s)$$

This is illustrated in Figure 2. Simply speaking, lex tests to see if words in the program are well-formed, cfs tests to see if these words are strung together into a reasonable sequence, and cc checks to see if the sequence can be given a reasonable meaning. The cc check is largely concerned with type checking, for example, preventing a string value from being assigned to an integer variable. A string that satisfies lex, cfs, and cc is considered to be a (legal) Turing program. We will now describe the methods used to define lex, cfs, and cc.

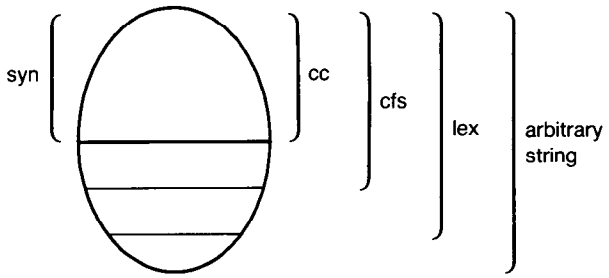

FIGURE 2. Strings that Satisfy CC, CFS, and LEX

Turing's Lexical Structure

The lexical (or word or token) structure of Turing is defined by specifying how to construct a lexically legal string. This is done by first defining classes of tokens. For example, the class of *integers*, which are the unsigned literal integer constants, is defined as the set of all non-null strings of digits (0 to 9). The remaining token classes, namely reals, strings, keywords, identifiers, and special symbols, are similarly defined as sets of strings. The set of *separators* that appear between tokens is defined. The lex function also specifies how separators are used between tokens.

Turing's Context-Free Syntax

The context-free syntax of Turing is specified in the traditional way using BNF. This syntax is shortened using extra notation, {. . .} for repetitions and [. . .] for optional items, and by giving operator precedence by a table rather than by extra productions. As an example of Turing's cfs, here is the context-free specification of statements:

A *statement* is one of the following:

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

The entire formal cfs definition is about 10 pages long with about 150 right-hand sides of productions.

Turing's Formal Context Conditions

It is common to formally specify the context-free syntax of a programming language, but relatively rare to formally specify its context conditions [24]. Usually this specification is done informally using phrases such as "a variable must be declared before it can be used."

Turing's formal context conditions, cc, are written in Rosselet's ADL notation [27]. ADL is notable for allowing the division of a formal definition into comprehensible parts. This division mimics the modularity of well-structured compilers [3], which are made maintainable by dividing them into parts. The cc definition contains a rule for each production in Turing's *abstract context-free syntax*. The abstract cfs is like Turing's (concrete) cfs except that details that are inconsequential to context conditions are omitted for brevity's sake. Each rule accepts the "syntactic subtree" representing the production's right-hand side, as well as contextual information, and returns information about this right-hand side. In the simplest case, this information is a Boolean value specifying whether the subtree is statically legal. In more interesting cases, such as in the rules for expressions, the rule returns a value representing the type of the expression.

Here is an example rule, which accepts an expression subtree E of the form $E_1 + E_2$ and returns the type of the expression. The production in question is $E ::= E_1 + E_2$.

```
if isNumeric(ty1) & isNumeric(ty2)
  then NumericType(ty1, ty2)
elsif EquivType(ty1, ty2) & tyTag1
  in{String, Set} then ty1
end if
```

The first case considered is when both subexpressions are numeric. It is tested using the isNumeric function which is specified in ADL. If this case succeeds, the type of the expression is the common numeric type of the two subexpressions. This type is given by NumericType, which returns an integer type for two integer operands and a **real** type for one or more real operands. The other case is when the plus sign means either string catenation or set union. If this case succeeds, the expression's type is returned as the type of the operands. If neither case succeeds, the rule fails to produce a result. This failure signals that the expression is statically illegal. This signal indicates as well that the entire program is illegal.

Turing's Formal Semantics

Once we have a definition of the syntax of Turing, we need to provide semantics for each string that syn accepts. In other words, we want to map each Turing program to its meaning. This is done by giving a function called sem that maps each Turing program t to its meaning m. The sem function is defined using "axiomatic semantics," an approach developed by Hoare, Dijkstra, and others that directly supports formal program correctness techniques.

Another equivalent definition of sem is given using "operational" semantics. This alternate definition of sem specifies the step-by-step changes in state (variable values) as a program executes and is useful for the implementor of a code generator, because it specifies what a Turing program must do at run time.

The existence of a formal definition implies that Turing is a mathematical notation as well as programming language, having a form and meaning that is completely machine-independent. We will not explore this formal definition further, but will instead now consider the extension of the language, Turing Plus.

TURING PLUS

Turing Plus is a compatible extension of Turing, meaning that any Turing program is also a Turing Plus program. Turing Plus can be thought of as an alternative to C, Modula 2, and Ada. The features that are added to Turing (see Figure 3) include separate compilation, concurrency, exception handling, and a variety of features to provide the programmer access to the underlying implementation. We will use the term *Turing proper* when we wish to emphasize that we are talking about Turing without these extensions.

VARIOUS FEATURES
- Natural numbers (the **nat** type).
- Sized numbers, e.g., **int1** (one byte integer), **nat2** (two byte natural number) and **real8** (eight byte real number).
- Bit manipulation, e.g., shifts.
- Subprograms as variables.
- Characters and fixed length character strings, e.g., **char** and **char**(15).
- Explicit type cheats, e.g., **type**(*c*, **nat1**).
- Address arithmetic.
- Indirection operator, e.g., **int@**(16#8ab36).

CONCURRENCY
- Process declarations.
- The **fork** statement, starting a new thread using a process declaration.
- Monitors, for critical sections.
- **wait** and **signal** statements, to block and wake up processes.
- Immediate and deferred condition queues.
- Time-out condition queues.
- Interrupt handling procedures and device monitors.

SEPARATE COMPILATION
- Stubs (interfaces) and bodies (implementations) for subprograms, modules and monitors.
- Parent and child directives (for enclosing and enclosed scopes).
- Complete checking of separately compiled parts.
- Linkage to other languages such as C and assembler.

EXCEPTION HANDLING
- Handlers (optional, in subprograms).
- Signaling an exception (the **quit**) statement.
- The guilty party (source of an exception).

GENERALIZED INPUT/OUTPUT
- Binary input/output.
- Random access input/output.

CONDITIONAL COMPILATION
- Compile time selection of segments of source.

FIGURE 3. Features Introduced by Turing Plus

Design Philosophy

Turing Plus provides detailed but controlled access to essentially all aspects (bits and bytes) of the run-time implementation, including pointer arithmetic and explicit emission of machine instructions. In other words, Turing Plus is a *permissive* language.

Turing proper is a *puritanical* language in that it is completely defined mathematically, and, given that checking is in force, it is not possible for Turing programs to directly access the underlying implementation. Dangerous features, such as C's pointer arithmetic and the type cheating that is done with Pascal's variant records, have been eliminated by careful language design.

The question is: How can a puritanical language, Turing, be gracefully extended into a permissive language, Turing Plus? Turing Plus maximizes checking and automatic error detection, in agreement with the philosophy of Turing proper; at the same time, Turing Plus provides the programmer with ways to explicitly override this checking.

Various Features of Turing Plus

Integers, reals and strings, which are among Turing's basic data types, are extended to provide more flexibil-

ity and efficiency. The *natural number* or **nat** type, called cardinal in Modula 2 and unsigned in C, is introduced to represent nonnegative integers. The programmer can control data sizes explicitly, for example each of **int1**, **int2**, **int4**, **nat1**, **nat2**, **nat4**, **real**4, and **real8** occupy the number of bytes specified by the final digit of the type name.

Bit manipulation is provided in terms of operations on natural numbers. These operations include shifting left and right, bit string extraction and insertion, as well as **and**, **or**, and exclusive **or**.

The **char** type, similar to the corresponding type in Pascal, is introduced. It is extended to a type denoted as **char**(*n*); this type is a fixed length character string whose length is *n*. The type **char**(*n*) is similar to Pascal's rather limited string type. Predefined subprograms are provided for efficient manipulation of **char** and **char**(*n*) values. Implicit type conversions allow **string**, **char** and **char**(*n*) types to be freely intermixed.

Turing Plus provides explicit type cheats, for example:

```
var c: char
var n: nat1
...
n := type(c, nat1)   % An explicit type cheat
```

Ordinarily, Turing Plus considers **nat1** (a natural number fitting in a byte) and **char** to be incompatible. In the last line of the above example, the programmer explicitly requested that the bits representing *c* be interpreted as **nat1**. Type cheats, such as this one, do not generate code or change the internal data representation. Natural numbers are a common target of type cheats, so a special syntax using the # operator is provided for conversions to **nat**. For example, the last line of the example can be written as: *n* := #*c*. Arbitrary type cheats among data types are allowed, for example, from a **real** to an array of **int1**. Some type cheats are meaningless or dangerous; even so they will not generally evoke a warning message. Ideally programmers will not use type cheats unless they know what their effects will be.

Arbitrary manipulation of machine addresses is allowed, for example:

```
var p: addressint      % An integer of address size
p := addr(v)           % Get address of variable v
p := p + 16#8AB36      % Increase p by hex 8AB36
int2@(p) := 124        % Poke 124 into double byte
                       % at p
```

A type of integer called **addressint** supports all the usual integer operations. On 32-bit machines, **addressint** is essentially the same as **nat4**. As the example illustrates, addr is used to extract the address of variables. Integer constants with various bases, such as base 16 in 16#8AB36, are supported. The final line in this example illustrates the *indirection* feature. This feature takes an integer such as *p* and a type such as **int2**; it treats the bytes at location *p* as a variable of type **int2**.

Turing proper supports only sequential files. Turing Plus generalizes input/output to support a variety of facilities including random access to specified byte positions in a file.

Concurrency

The basic set of concurrency features of Turing Plus are clean (implementation-independent) and suitable for portability of programs across multi-CPU configurations exemplified by Tunis, Sequent, and Encore. The concurrency features of Turing Plus are based on those of CE (Concurrent Euclid), as refined and extended according to experience with CE.

Process declarations are introduced. Each process declaration, which has the general form of a procedure declaration, creates a template for a concurrent thread of control (for a process). The **fork** statement activates a thread using a specified process declaration. Here is a complete program that nondeterministically prints out interspersed *Hi*'s and *Ho*'s:

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

fork speak("Hi")   % Start saying: Hi Hi Hi ...
fork speak("Ho")   % Start saying: Ho Ho Ho ...
```

The threads activated by **fork** are called *lightweight processes*; they are lightweight in that their interprocess communication is extremely efficient. Turing Plus's processes are "true processes," meaning that they physically run in parallel if multiple CPUs are available and that CPU slicing is used to simulate this parallelism on a single CPU system, in contrast with the co-routines or pseudo processes of Modula 2, which never implicitly give up control and which are not immediately suitable for multi-CPU configurations.

Interprocess communication is provided in a machine-independent way by monitors as described in [14]. Monitors provide mutually exclusive access to shared data. A monitor is essentially a special purpose module in which only one process can be active at a time; a process attempting to enter an active monitor is blocked until no other process is running in the monitor. *Conditions* can be declared in monitors; these are essentially queues of processes. A process executes a **wait** statement to block itself and place itself on one of these queues. Another process should eventually execute a **signal** statement on that queue to wake up the first process. Here is an example monitor that implements a semaphore.

```
monitor semaphore
  export(P, V)

  var n := 0
      % Semaphore count, which is a shared variable
  var q: condition   % Process queue

  procedure P
    if n = 0 then
```

```
        wait q   % Sleep if count is zero
      end if
      n -= 1
    end P

    procedure V
      n += 1
      signal q   % Wake up a process if one is sleeping
    end V
  end semaphore
```

By default, conditions are *immediate*, meaning that a signalled process immediately starts running. Turing Plus also provides *deferred* conditions, in which the signaller continues and the signalled process does not run until the monitor becomes unoccupied. There are also **timeout** conditions, which are implicitly signalled if an explicit signal does not occur within a specified time.

Turing Plus is designed to be a good tool for writing low level systems software such as operating systems, network controllers, and embedded systems. For example, the TUNIS implementation [14] of Unix has been programmed in Turing Plus. In this kind of application, concurrency as well as efficient access to low level machine facilities is a necessity.

Turing Plus provides access to memory mapped device registers by allowing variables to be specified to be located at absolute machine addresses. Of course, this access can be defeated if the program is run under control of a system whose protection hardware limits addressibility.

Device monitors are a specialized version of monitors that maintain mutual exclusion using machine-specific tricks, notably the setting of hardware priority levels. Special entries into these monitors, called *interrupt handling procedures*, are directly linked to the machine's interrupt vectors providing high performance for interrupt-intensive applications. Turing Plus is intended to provide comparable performance to hand-assembled interrupt handlers and better performance than is possible in languages like C and Ada that are less well attuned to low level machine functions.

Separate Compilation

Turing Plus allows subprograms and modules to be compiled separately and linked together using a standard linker, such as that provided by Unix or MS-DOS. Monitors, as a special case of modules, can also be separately complied. Separate compilation is completely checked. This implies that the number of and types of actual parameters of each call to a separately compiled subprogram are checked for consistency with the subprogram's formal parameters.

Turing Plus views a separately compiled program the same way it views a program compiled all at once. In other words, a separately compiled program has the same semantics as the single program that would explicitly contain all the separately compiled units. Separate compilation implies no new scope rules or control constructs.

To separately compile a *unit* (subprogram or module), its *stub* or interface is separated from its *body* or implementation. The stub contains the information needed to check for proper interfacing to the unit. The stub for a subprogram is essentially its header. The stub for a module includes the headers of exported subprograms and declarations of exported items such as types. With a few minor exceptions, any subprogram or module from an existing program can be removed and separately compiled. Units that already exist in a library can be combined into a new program. When a unit is separated from its surrounding program, the unit is called a *child* and the surrounding program its *parent*. Any variables that a parent wishes to allow its children to access must be specified in a **grant** list.

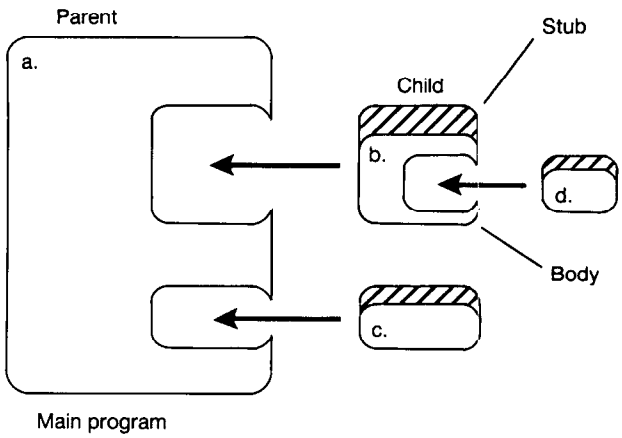

FIGURE 4. Separate Compilation Units

In the example illustrated in Figure 4, separately compiled units *b* and *c* are considered to be nested directly inside the main program. Separately compiled unit *d* is in turn considered to be nested inside unit *b*. In a given program, all separately compiled units are considered to be nested (directly or indirectly) inside the main program.

The main program could have this form:

```
grant(i, var j)
  % Variables the main program shares with
  % children
var i: int
child "b"   % Brings in b's stub
...
var j: int
...
child "c"   % Brings in c's stub
```

The **grant** directive specifies that variables *i* and *j* declared in the main program are to be shared with separately compiled children. The **var** keyword preceding *j* specifies that these children are allowed to modify *j*, but not *i*. A **grant** directive is used in (sub)units as well as in the main program for sharing items with (grand)children. The **child** "*b*" directive locates the file containing the stub for *b*. Similarly **child** "*c*" locates *c*'s

stub. When the main program is compiled, the stubs for *b* and *c* are inspected, but not their bodies.

The file defining *c* stub could be:

```
% Separately compiled procedure c
parent "a"
% Locates parent(main program),
% for declarations of i and j
procedure c(k: int)   % Stub for procedure c
% Body for c
procedure c
  j := i**2 + k
end c
```

The **parent** directive specifies that child *c* is embedded in a surrounding unit called *a*. When the main program, *a*, is compiled, the compiler uses the **child** "*c*" directive to locate and read in *c*'s stub (but not *c*'s body). When unit *c* is compiled, the compiler uses the **parent** "*a*" directive to locate and read the main program; the compiler reads it up to the **child** "*c*" directive but no further; this portion of *a* contains those declarations that are visible to procedure *c*.

A child that does not use any of declarations from its parent is called an *orphan* and does not need to begin with a **parent** directive. Assuming unit *b* is an orphan module, it could have this form:

```
% Separately compiled orphan module b
module b
  export(t, f)
  type t: string
  function f(s: t): t
                    % Header of exported subprogram
end b
```

File *b* contains the interface to module *b*. The body of a child is optionally contained with its stub, but has been omitted in the case of *b*. It may be that module *b* is actually programmed in another language such as C. It also may be that *b* is written in Turing Plus, in which case it could be contained in a file *e* such as this:

```
% File e containing body of module b
body "b"   % Locate the stub for module b
% Body for module b
module b
  child "d"   % A child of b, which is a function
  function f   % Parameters of f are given in b's stub
    result d(s + s)   % Body of f, uses d
  end f
end b
```

The body of a module must give the bodies for exported subprograms as well as any unexported items local to the module. In this example, *b* has a child called *d*. The stub and body for *d* are given like any other separately compiled unit.

A given interface can have several alternate implementations. This is done in Turing Plus by having several files of the form just shown for the body of *b* prefixed with the same directive **body** "*b*".

Exception Handling

Turing Plus provides a simple and efficient version of exception handlers that is not very different from the corresponding feature in Ada. Each subprogram optionally begins with a *handler*. When a subprogram containing a handler is called, its handler is activated; when this subprogram returns, its handler is deactivated. If an *exception* such as division by zero is detected, the current computation is halted and control is given over to the most recently activated handler. When the handler completes its execution, control returns to the point of call to the subprogram containing the handler. Essentially, the handler replaces the aborted action of the subprogram. It is not possible to return to the point of failure.

The form of a procedure containing a handler is:

```
procedure p(. . . p's parameters . . .)
  handler(code)   % Code gives exception number
    . . . statements to handle exception . . .
  end handler
    . . . usual statements for procedure p . . .
end p
```

When a handler is given control, it is given an integer code specifying the nature of the failure. The code could specify, for example, division by zero or subscript out of bounds. The body of the handler will often contain a case statement to analyze this code, but this is not required, and anything that can appear in an ordinary subprogram body can be written for a handler body.

Besides system-detected exceptions, there are explicit exceptions that are signalled by the **quit** statement, which has these forms:

```
quit : n      % Guilty party is the present code
quit <: n     % Guilty party is the caller
quit >: n     % Guilty party is a previous exception
```

The **quit** statement aborts the current computation and hands control to the most recently activated handler or to the implicit system handler if no handler is explicitly active. The **quit** statement contains an optional exception code *n*, which specifies the nature of the failure. The first **quit** statement shown above, without the < or > symbols, is the usual one. If there is no explicitly active handler, the implicit handler will report the exception number and the line number of this **quit** statement.

The second and third forms of **quit** contain the < and > symbols. The second statement, containing <, is best explained in terms of an example. The example is the implementation of sqrt (the predefined square root function). If a negative argument is passed to sqrt, it wants to quit, but it wants to place the blame on the caller, who supplied the bad argument. The < sign places the blame on the caller and the implicit handler prints an error message containing the line number of the caller rather than reporting a failure in sqrt. The third statement, containing >, is meaningful only inside

handlers. It specifies that the handler wants to place the blame on the previous exception, so that the implicit handler prints the line number of that exception rather than the line number in the current handler. This version of **quit** is used to allow a handler to inspect an exception and to do any necessary cleanup of data structures before passing the exception along to the next handler.

COMPARISON WITH MODULA 2 AND ADA

Modula 2 [29] and Ada are often mentioned as possible successors to Pascal. Since Turing is also a possible successor, we will briefly contrast Turing with those two languages.

Modula 2 is a relatively new language that is attracting an increasing number of users. It supports systems programming by providing features such as concurrency and suppression of type checking. This class of features was purposely omitted from Turing (although included in Turing Plus).

Modula 2 improves Pascal's syntax by, for example, introducing **end** as a closing for **if** statements, but its syntax is wordy compared to Turing's. Modula 2 is not designed for interactive programming or for use by children. Like Pascal, it omits many features that are needed to make the language more generally applicable, such as convenient input/output, varying length strings, dynamic arrays, exponentiation, and so on. Modula 2 was designed without a formal definition and has no concept of faithful execution.

Ada [21] is a language designed for the U.S. Department of Defense to program software for embedded computers (computers that are part of larger systems such as aircraft). Ada can be thought of as an overblown version of Modula 2. It contains systems programming features, which are omitted from Turing. The most commonly voiced complaint about Ada is that it, like PL/1, is an extremely complex language that is very difficult to master and to compile.

Ada provides few solutions to the problems that Turing is intended to solve. For example, Ada lacks convenient input/output and has no varying length strings. Like Modula 2, Ada is not designed for teaching children and (compared to Turing) has a wordy syntax. The Turing Plus extension of Turing could be an alternative to Ada.

CONCLUSION

This article has introduced the Turing language and its extension to Turing Plus. Together these form a programming language of unique breadth of application, from introductory programming for children to concurrent programming on bare board microcomputers. The language aspires to provide the simplicity of Basic, the elegance of Pascal, the flexibility of C, and the generality of Ada. At the same time, it is a relatively easy language to compile and supports applications requiring high performance.

Acknowledgments. Many people have been involved in the Turing project since its inception in 1982. J. N. P. Hume has been a central participant and has written much of the teaching material for Turing. E. C. R. Hehner has been a constant source of ideas and constructive criticism. J. A. Rosselet's Ph.D. thesis [27] formed the basis of Turing formal context conditions [19]. P. A. Matthews inspired the formal definition and laid out the formal semantics [19]. M. P. Mendell and S. G. Perelgut have been the key to the implementation success, with essential contributions by a number of others including A. F. X. Curley, S. W. K. Tjiang, C. B. Hall, T. C. N. Graham, M. Robertson, N. Eliot, and R. Wessels. T. E. Hull has encouraged and supported the development and has developed Numerical Turing [20] that supports **real** numbers with dynamically varying precision. The design of the Turing language has borrowed extensively from the Euclid language [26]. The research leading to this work has been generously supported by the Natural Sciences and Engineering Research Council of Canada, Bell Northern Research Ltd, and the Ontario Ministry of Education.

REFERENCES

1. Conway, R.W., and Wilcox, T.R. Design and implementation of a diagnostic compiler for PL/1. *Commun. ACM 6*, 3 (Mar. 1973), 169–179.
2. Cordy, J.R., and Holt, R.C. Specification of Concurrent Euclid. Tech. Rep. CSRI-133, Computer Systems Research Institute, University of Toronto, August 1981.
3. Cordy, J.R. A diagrammatic approach to processing programming language semantics, Tech. Rep. CSRI-67, Computer Systems Research Institute, University of Toronto, March 1976 (later published In *Proceedings of SIGPLAN 79 Symposium on Compiler Construction. SIGPLAN Not. 14*, 8 (Aug. 1979), 39–49.
4. Cordy, J.R., and Graham, T.C.N. TTV: A programming environment that's as smart as you want it to be. In *Proceedings of the 5th Canadian Symposium on Instructional Technology* (Ottawa, Canada, May 1986). NRC (National Research Council), 1986.
5. Cordy, J.R., and Graham, T.C.N. Design of an interpretive environment for TURING. In *Proceedings of SIGPLAN 87 Symposium on Interpreters and Interpretive Techniques. SIGPLAN Not. 22*, 7 (July 1987).
6. Cordy, J.R., Eliot, N., and Robertson, M. TURINGTOOL: A knowledge-based user interface to aid in the maintenance task, Tech. Rep. 87-183, Department of Computing and Information Science, Queen's University at Kingston, July 1987.
7. Dijkstra, E.W. *A Discipline of Programming*. Prentice-Hall, Englewood, Cliffs, N.J., 1976.
8. Gries, D. *The Science of Programming*. Springer-Verlag, New York, 1981.
9. Habermann, A.N. Critical comments on the programming language Pascal. *Acta Informatica* 3, 1973, 47–57.
10. Hehner, E.C.R. *The Logic of Programming*. Prentice-Hall International Series in Computer Science, London, 1984.
11. Hodges, A. *Alan Turing: The enigma*. Simon & Schuster, New York, 1983.
12. Holt, R.C., Cordy, J.R., and Wortman, D.W. An introduction to S/SL: syntax semantics language. *ACM Trans. Prog. Lang. Syst. 4*, 2 (Apr. 1982) 149–178.
13. Holt, R.C., and Cordy, J.R. The TURING language report. Tech. Rep. CSRG-153, Computer Systems Research Institute, University of Toronto, December 1983.
14. Holt, R.C. *Concurrent Euclid, The Unix System and Tunis*. Addison-Wesley, Reading, Mass. 1983.
15. Holt, R.C. TURING: An inside look at the genesis of a programming language. *Computerworld 18*, 20 (May 1984).
16. Holt, R.C., Hume, J.N.P. *Introduction to Computer Science Using the TURING Programming Language*. Reston, Prentice-Hall, Englewood Cliffs, N.J., 1984.
17. Holt, R.C., and Cordy, J.R. The TURING PLUS report. Tech. Memo, Computer Systems Research Institute, University of Toronto, 1985.
18. Holt, R.C. Design goals for the TURING programming language. Tech. Rep. CSRI-187, Computer Systems Research Institute, University of Toronto, 1986.
19. Holt, R.C., Matthews, P.A., Rosselet, J.A., and Cordy, J.R. *The TURING Programming Language: Design and Definition*. Prentice-Hall Englewood Cliffs, N.J., 1987.

20. Hull, T.E., Abraham, M.S., Cohen, M.S., Curley, A.F.X., Hall, C.B., Penny, D.A., and Sawchuk, J.T.M. Numerical TURING. *ACM SIGNUM Newsletter 20*, 3 (July 1985), 26–34.
21. Ichbiah, J. et al. Rationale for the design of the Ada programming language. *ACM SIGPLAN Not., 14*, 6 (June 1979).
22. *ISO Specification of the Computer Language Pascal.* International Standards Organization, 1981.
23. Jensen, K., and Wirth, N. *Pascal User Manual and Report.* 2d ed., Springer-Verlag, New York, 1974.
24. Kastens, U., Hutt, B., and Zimmerman, E. GAG: *A Practical Compiler Generator.* Springer-Verlag, Berlin, 1982.
25. Kernighan, B.W. *Why Pascal Is Not My Favorite Programming Language.* Computer Science Report 100, Bell Laboratories, Murray Hill, N.J., July 1981.
26. Lampson, B.W., Horning, J.J., London, R.L., Mitchell, J.G., and Popek, G.J. Report on the programming language Euclid. *ACM SIGPLAN Not. 12*, 2 (Feb. 1977) (The revised language is described in Rep. CSL-81-12, Xerox Palo Alto Research Center, October 1981.)
27. Rosselet, J.A. Definition and implementation of context conditions for programming languages. Tech. Rep. CSRG-162, Computer Systems Research Institute, University of Toronto, July 1984.
28. Welsh, J., Sneeringer, W.J., and Hoare, C.A.R. Ambiguities and insecurities in Pascal. *Software—Practice and Experience 7*, 6 (Nov. 1977), 685–696.
29. Wirth, N. Modula: A programming language for modular multiprogramming systems. *Software—Practice and Experience, 7*, 1 (Jan.–Feb. 1977), 3–35.

CR Categories and Subject Descriptors: D.2.4 [**Software Engineering**]: Program Verification—*validation*; D.2.6 [**Software Engineering**]: Programming Environments; D.2.m [**Software Engineering**]: Miscellaneous; D.3.2 [**Programming Languages**]: Language Classifications—*Turing*; D.3.3 [**Programming Languages**]: Language Constructs

General Terms: Design, Languages, Management, Standardization, Verification

Additional Key Words and Phrases: Programming language comparisons, programming language history

ABOUT THE AUTHORS:

RICHARD C. HOLT is a professor in the Computer Systems Research Group at the University of Toronto. He is the author of a dozen books on programming languages and operating systems. His ongoing research involves the Turing programming language and the Tunis (Unix compatible) operating system. Author's present address: R. C. Holt, Computing Systems Research Institute, University of Toronto, Sanford Fleming Bldg., 10 King's College Road, Toronto, Canada M5S 1A4.

JAMES R. CORDY received his Ph.D. in computer science in 1986 from the University of Toronto and is presently assistant professor of computing and information science at Queen's University at Kingston, Canada. He is co-designer of the programming languages Concurrent Euclid, Turing and Turing Plus, the Turing programming environment, and the S/SL compiler specification language. He presently conducts research in user interfaces, code generation, and programming language design tools. Author's present address: J. R. Cordy, Department of Computing and Information Science, Queen's University, Kingston, Canada K7L 3N6.