THE DESIGN OF EDISON

PER BRINCH HANSEN

(1981)

This paper describes the considerations behind the design of the programming language Edison including the reasons why a large number of well-known language features were excluded. It also discusses the linguistic problems of writing a concise language report.

1 THEMES

This paper describes the considerations behind the design of the programming language Edison (Brinch Hansen 1981). Edison is deeply influenced by the advances made by Pascal (Wirth 1971) and its successors Concurrent Pascal (Brinch Hansen 1975) and Modula (Wirth 1977). In my attempt to learn from these previous efforts and improve on them I had to focus on both the virtues and defects of these languages. The criticism may appear very unfair coming from a programmer who has benefited immensely from the innovations of Pascal ten years ago. But such criticism is necessary to gain a clearer perspective of a programming tradition that has become so natural to me that it inhibits my ability to search for better methods.

In designing the programming language Edison I tried to do two things: (1) to achieve simplicity by questioning the necessity of a number of well-known language concepts, and (2) to gain new insight by deliberately approaching language design with a philosophy that is completely different from the spirit in which Concurrent Pascal was designed.

The first viewpoint gradually led to the omission of many language features that I had previously thought were valuable features of the programming languages Pascal and Concurrent Pascal, namely

P. Brinch Hansen, The Design of Edison. *Software—Practice and Experience 11*, 4 (April 1981), 363–396.

reals
subrange types
variant records
files
pointers
unnamed types
goto statements
case statements
repeat statements
for statements
with statements
cycle statements
init statements
multiple class instances
parameterized classes
monitors
process modules
process queues

If I were to do it again, I would go further and eliminate functions as well.

When you see the list of what is not in the language, you may wonder what is left. The answer is—not much!

The process of eliminating language features took place over a period of more than a year and was not an easy one. Old habits are hard to break! What took the longest though was the simple discovery of new syntactical structures for well-known concepts, such as type declarations. The experience of using a good programming tool tends to make one unable to discover better ones.

The search for a different design philosophy can best be illustrated by contrasting Concurrent Pascal and Edison.

In Concurrent Pascal, program modularity is supported by the rather complicated concepts of processes (which combine modularity and concurrent execution) and monitors (which combine modularity and synchronized execution). A large class of time-dependent programming errors are eliminated at compile-time by checking that processes do not refer directly to the same variables. The only form of communication among processes is by means of monitor procedure calls. Within monitors another class of synchronization error is automatically eliminated by a mutual exclusion of monitor calls enforced during program execution. In addition, the compiler detects recursive calls to prevent deadlocks.

This rigorous approach to security was based on the belief that race

conditions would be extremely difficult to locate at run-time due to their irreproducible nature, and that they must be detected by a compiler before a concurrent program is even tested. The resulting programs have been simple enough to publish and have been more reliable than the hardware they run on (Brinch Hansen 1977).

The security of Concurrent Pascal was achieved by careful selection of a small number of complicated concepts. Although this approach can be quite successful in practice, it is not without pitfalls. It requires a great deal of practical experience in operating system design and a gambler's instincts to select the combination of shared variables, synchronized procedure calls, and the Simula class concept as the only possible mechanism of process communication in a programming language—when the much simpler concept of a semaphore (in theory) suffices. If you make the right guess, fine—otherwise, such a language will be of little practical use.

The RC 4000 multiprogramming system had the conceptual advantage of a clear and consistent structure, but it was not always successful in practice. When the system was ready, it soon became apparent that the transmission of small messages of fixed length was not enough to support the design of a general operating system. Since this was the only form of communication available to concurrent processes, system programmers began in some cases to ignore the system by using a single process to simulate several coroutines for which they could design more suitable forms of interaction (Brinch Hansen 1970).

Another (somewhat unexpected) consequence of careful *ad hoc* design is that the number of specialized mechanisms tends to multiply to cover the programming needs. In Concurrent Pascal, modules occur in three varieties (processes, monitors, and classes), and in Modula there are four kinds (processes, interface modules, device modules, and other modules).

These observations led to a separation of the concepts of modularity, concurrency, and synchronization in Edison. The result is a more flexible language based on fewer concepts in which one can achieve the same security as in Concurrent Pascal by adopting a programming style that corresponds to the processes and monitors of Concurrent Pascal. Or one can use the language to express entirely different concepts. On the other hand, it is also possible to break the structuring rules and write meaningless programs with a very erratic behavior. I have adopted this more general and less secure approach to programming to learn from it. It is still too early to make firm conclusions about consequences of such a compromise.

The desire to replace specialized mechanisms with more general ones that can be combined freely led to the adoption of the following rule: Whenever a mechanism is needed for a particular purpose the most general variant of that mechanism will be selected.

An example is provided by the well-known need to enable programs to call certain procedures implemented by an operating system, for example to access files. In Sequential Pascal the headings and parameter types of such procedures are described by a prefix to every program. When an operating system (written in Concurrent Pascal) calls a program (written in Sequential Pascal) an anonymous table with the addresses of these procedures is automatically generated for the program. The program prefix is the only mechanism for using procedures as parameters in Sequential Pascal.

In Edison the same problem is solved more generally by allowing procedures to be parameters of other procedures and by letting library programs in Edison take the form of procedures.

The more general approach has already paid off. The Mono operating system written in Concurrent Pascal includes procedures for writing integers and text on a variety of media. Each of these procedures in turn calls another procedure to write a single character on a given medium. Since the latter procedure varies from medium to medium it was necessary to write a different version of the former procedure for each kind of medium. When this operating system was rewritten in Edison it was sufficient to program one version of each procedure and pass as a parameter to that procedure another procedure for writing single characters on the desired medium. Generality pays off in unexpected ways!

Another example of the preference for generality in Edison is the general nesting of procedures and modules which is already supported by Pascal and Modula, but is forbidden in Concurrent Pascal. I find nested procedures as hard to read as prose that makes frequent use of parenthetic remarks. Such procedures were therefore outlawed in Concurrent and Sequential Pascal. The compilers took advantage of the exclusive use of two block levels by generating efficient instructions for the addressing of local and global variables.

In designing Edison I found that this restriction made it impossible to describe the effect of library programs by the simple rule of conceptually inserting the program text where the corresponding program declaration appears in the calling program. Since both the calling program and the library program could use two levels of nesting this textual substitution might

create an invalid program with three levels of nesting. So general nesting was allowed merely as a means of simplifying the language description. Once this decision was made I discovered the obvious: the compiler can still recognize references to variables in a local block and the immediately surrounding block and generate special instructions for addressing them.

It is with both regret and delight that I finally in 1980 find myself appreciating language concepts that were obvious to the designers of Algol 60 twenty years ago (Naur 1962). In the following, a number of other design decisions are explained in some detail.

2 ABSTRACT DATA TYPES

The invention of syntactic structures to describe self-contained parts of larger programs is one of the major achievements of the last decade. In Concurrent Pascal, these structures are called processes, monitors, and classes. In Modula and Edison they are known as modules.

One of the main purposes of a program module is to implement the concept of an abstract data type—a data type that can only be operated upon by a fixed set of operations. In Concurrent Pascal, abstract data types are implemented by means of a secure variant of Simula classes (Dahl 1972). In Edison I decided to use a very different idea inspired by Modula.

Consider the problem of unpacking integer values from a sequence of disk pages. In Modula this can be done by means of the following module (Algorithm 1). The define clause shows that the module exports an entitity named next to the surrounding block. The use clause indicates that the module uses (or imports) entities named page, pagelength, and get from the surrounding block.

The data structure of the module consists of two local variables which assume the value of a disk page and the index of the last value unpacked from that page.

The procedure called next gets another disk page from the disk (if necessary) and unpacks the next integer value from the page. The initial operation described at the end of the module gets the first page from the disk and sets the index to zero.

The compiler ensures that the only operation performed on the data structure (following the initial operation) is the well-defined operation called next. This kind of module is appealing because it ensures the integrity of the data structure entirely by means of scope rules: the variables named data and index are local to the module and are unknown outside the module.

The protection of the data structure is achieved at compile-time without run-time support.

```
module symbolinput;
  define next;
  use page, pagelength, get;

  var data: page; index: integer;

  procedure next(var value: integer);
  begin
    if index = pagelength then
      get(data); index := 0
    end;
    index := index + 1;
    value := data[index]
  end next;

begin get(data); index := 0
end symbolinput
```

Algorithm 1

In Edison the same module looks as follows (Algorithm 2). The define clause has been replaced by an asterisk in front of the exported procedure (a notation borrowed from Pascal Plus; see Welsh 1980). This simplifies the visual image of the module a bit and makes it unnecessary for the compiler to check whether all entities mentioned in the define list are indeed declared within the module. On the negative side one must now scan the module visually to discover the operations it implements. The imported entities are known simply by virtue of the ordinary scope rules of nested blocks. So there is no use clause either in Edison.

This kind of module is an ideal structuring tool when it controls the access to a single instance of a data structure that is hidden inside the module as in the previous example. The initial operation ensures that the data structure is brought into a consistent, initial state before it is operated upon by the surrounding block. The module concept is less convenient when several instances of an abstract data type are needed. The following example shows an Edison module that implements push and pop operations on stacks of integer values (Algorithm 3).

```
module "symbol input"
  var data: page; index: int

* proc next(var value: int)
  begin
    if index = pagelength then
      get(data); index := 0
    end;
    index := index + 1;
    value := data[index]
  end

begin get(data); index := 0 end
```

Algorithm 2

In the surrounding block one or more stacks can be declared

```
var s, t: stack
```

and used as follows

```
push(s, 15)      pop(t, value)
```

The initial operation of the module can no longer guarantee that all stacks are empty to begin with (since the stacks are declared in the surrounding block after the module). The surrounding block must do that by performing the operations

```
newstack(s)      newstack(t)
```

on the stacks before using them to push and pop integer values.

Now, if the asterisk in front of the record type named stack would export not only the record type, but also its field, then it would be possible to perform meaningless operations on stacks outside the module, for example

```
s.size := −3
```

The rule that record fields cannot be exported from modules was introduced in the language to ensure that the module procedures describe the only

```
module
  array table[1:100](int)
* record stack(contents: table; size: int)

* proc push(var x: stack; y: int)
  begin x.size := x.size + 1;
    x.contents[x.size] := y
  end

* proc pop(var x: stack; var y: int)
  begin y := x.contents[x.size];
    x.size := x.size - 1
  end

* proc newstack(var x: stack)
  begin x.size := 0 end

begin skip end
```

Algorithm 3

possible operations on stacks (apart from assignments and comparisons of the values of whole stacks).

In Concurrent Pascal, a stack can be described more elegantly by the following type declaration (Algorithm 4). An instance of this data type is declared and used as follows:

```
var s: stack

init s      s.push(15)      s.pop(value)
```

But the class concept is more complicated both to explain and implement because it combines the concepts of data types, procedures, and modules into a single, indivisible unit. By contrast, the module concept of Edison merely modifies the scopes of a set of named entities of any kind. With the exception of fields, any named entity can be exported from a module by placing an asterisk in front of its declaration.

3 MONITORS

The significance of the monitor concept was that it imposed modularity on synchronizing operations used by concurrent processes. A monitor intro-

```
type stack =
class
  var contents: array [1..100] of integer;
    size: 0..100;

  procedure entry push(y: integer);
  begin size := size + 1;
    contents[size] := y
  end;

  procedure entry pop(var y: integer);
  begin y := contents[size];
    size := size − 1
  end;

begin size := 0 end
```

Algorithm 4

duces an abstract data type and describes all the operations on it by means of procedures that can be called by concurrent processes. If several processes attempt to execute monitor procedures simultaneously these procedures will be executed one at a time (Brinch Hansen 1973; Hoare 1974).

In Concurrent Pascal, a message buffer for transmitting characters from one process to another can be declared as follows (Algorithm 5). The buffer is represented by a message slot of type character and a boolean indicating whether it is full or empty. Two variables of type queue are used to delay the sending and receiving processes until the buffer is empty and full, respectively.

Initially, the buffer is empty. The procedure named put delays the calling process (if necessary) until there is room in the buffer for another message. When that condition is satisfied, a character is placed in the buffer making it full. Finally, the receiving process is continued if it is waiting for the message. (If the receiver queue is empty the continue operation has no effect.)

The get operation is similar to the put operation.

Since only one operation must be performed at a time on the monitor variables the following rules apply to operations on queues: When a process delays its completion of a monitor operation another monitor operation can be performed by another process, and, when a process continues the execu-

```
type buffer =
monitor
  var slot: char; full: boolean;
    sender, receiver: queue;

  procedure entry put(c: char);
  begin if full then delay(sender);
    slot := c; full := true;
    continue(receiver)
  end;

  procedure entry get(var c: char);
  begin if not full then delay(receiver);
    c := slot; full := false;
    continue(sender)
  end;

begin full := false end
```

Algorithm 5

tion of a delayed process, the former process automatically returns from the monitor procedure in which it executed the continue operation.

The whole monitor concept is a very intricate combination of shared variables, procedures, process scheduling, and modularity.

Here is Modula's variant of the monitor concept for a single buffer instance (Algorithm 6). Although monitors and queues are called interface modules and signals in Modula the concepts are essentially the same.

Now, process queues were originally proposed by me as an engineering tool to reduce the overhead of process scheduling on a single-processor computer (Brinch Hansen 1972). Each queue is used to delay a process until the monitor variables satisfy a particular condition, such as

not full or full

When a process delays itself in a queue until a condition holds it depends on another process to continue its execution when the condition is satisfied.

In 1972, Tony Hoare published a much more elegant mechanism for process scheduling in the form of the *conditional critical region*

with v **when** B **do** S

```
interface module buffer;
  define get, put;

  var slot: char; full: boolean;
    nonempty, nonfull: signal;

  procedure put(c: char);
  begin if full then wait(nonfull);
    slot := c; full := true;
    send(nonempty)
  end put;

  procedure get(var c: char);
  begin if not full then wait(nonempty);
    c := slot; full := false;
    send(nonfull)
  end get;

begin full := false end buffer
```

Algorithm 6

which delays a process until a variable v satisfies a condition B and then executes a statement S. The execution of conditional regions on a given variable v will take place strictly one at a time in some order (Hoare 1972).

This beautiful concept requires a process to reevaluate a boolean expression B periodically until it yields the value true. The fear that this reevaluation would be too costly on a single processor motivated the introduction of queues (also called signals or conditions) as an engineering compromise between elegance of expression and efficiency of execution.

With the new inexpensive microprocessor technology now available, I feel that the much simpler concept of conditional critical region should be preferred. Occasional inefficiency is of minor importance on a microprocessor.

Hoare's original proposal made it possible for operations on different shared variables, such as

with v_1 **when** B_1 **do** S_1

and

with v_2 **when** B_2 **do** S_2

to take place simultaneously. For Edison I decided to use the simplest form of the conditional critical region

when B **do** SL **end**

where SL is a statement list of the form $S_1; S_2; \ldots; S_n$. The execution of all *when statements* will take place strictly one at a time. If several processes need to evaluate (or reevaluate) scheduling conditions simultaneously they will be able to do so one at a time in some fair order (for example cyclically).

Measurements of concurrent systems have shown that in a well-designed system each process spends most of its time operating on local data and only a small fraction of its time exchanging data with other processes. The additional overhead of expression evaluation will therefore most likely be quite acceptable. This performance issue can, of course, only be settled by measurements of running Edison programs. But it is an intellectual gamble that I feel quite comfortable about.

A monitor in Edison is simply a module in which the procedure bodies consist of single when statements as illustrated by the buffer example (Algorithm 7).

```
module "buffer"
  var slot: char; full: bool

* proc put(c: char)
  begin
    when not full do
      slot := c; full := true
    end
  end

* proc get(var c: char)
  begin
    when full do
      c := slot; full := false
    end
  end

begin full := false end
```

Algorithm 7

Notice that Edison does not include the monitor concept. A monitor is constructed by using a programming style that combines the simpler concepts of modules, variables, procedures, and when statements. In addition, processes are no longer required to manipulate scheduling queues, since the when statements implement the necessary delays of processes. This form of a monitor is very close to the original proposal which I called a "shared class" (Brinch Hansen 1973).

Once this module has been programmed correctly, the compiler will ensure that the buffer only is accessed by the synchronized get and put operations. The module is now as secure as any monitor.

But the programmer is no longer tied to the monitor concept, but can use simpler concepts, such as semaphores (Algorithm 8).

```
module
* record semaphore(value: int)

* proc wait(var s: semaphore)
  begin
    when s.value > 0 do
      s.value := s.value − 1
    end
  end

* proc signal(var s: semaphore)
  begin
    when true do
      s.value := s.value + 1
    end
  end

* proc newsem(var s: semaphore; n: int)
  begin s.value := n end

begin skip end
```

Algorithm 8

Semaphores can then be used to implement a multislot buffer in which sending and receiving can take place simultaneously from different slots (Algorithm 9).

If the mutually exclusive operations are as simple as wait and signal

```
module
* record semaphore(value: int)

* proc wait(var s: semaphore)
  begin
    when s.value > 0 do
      s.value := s.value − 1
    end
  end

* proc signal(var s: semaphore)
  begin
    when true do
      s.value := s.value + 1
    end
  end

* proc newsem(var s: semaphore; n: int)
  begin s.value := n end

begin skip end
```

Algorithm 8

(rather than entire monitor procedures) then the conditional critical regions may well be more efficient in some cases than conventional monitors with scheduling queues.

4 PROCESSES

In the RC 4000 system I dealt with the complexities of processes that may appear and disappear at any time during program execution. In Concurrent Pascal, I tried the opposite approach of processes that exist forever after their creation. This works quite well for operating systems and real-time programs which perform the same tasks over and over. In addition, it simplifies store allocation dramatically.

For Edison I selected a compromise between these two extremes. Processes described by a *concurrent statement* of the form

cobegin 1 **do** SL_1
also 2 **do** SL_2
...
also n **do** SL_n **end**

can appear and disappear dynamically—but only at the same time!

The concurrent statement was published with the following syntax by Dijkstra in 1968

parbegin $S_1; S_2; \ldots; S_n$ **parend**

I merely replaced the semicolon (which normally denotes sequential execution) with the word *also* (to indicate simultaneity).

The process constants 1, 2, ..., n were introduced to make it possible to select a particular processor in a multiprocessor system (usually because the given processor is the only one that is connected to a particular peripheral device). On a single-processor computer the process constants can either be ignored or used to define the storage requirements of processes. (If they are ignored the simplest implementation strategy is to divide the available storage space evenly among the processes.)

Consider now two processes that exchange a sequence of characters terminated by a period through a buffer. In Concurrent Pascal the producer can be declared as follows

```
type producer =
process (buf: buffer);
var x: char;
begin read(x);
  while x <> '.' do
  begin buf.put(x); read(x) end;
  buf.put(x)
end
```

To vary the theme a bit the consumer will be programmed in Modula

```
process consumer;
var y: char;
begin get(buf, y);
  while y <> '.' do
    write(y); get(buf, y)
  end;
  write(y)
end consumer
```

In both cases, a process is described by a special kind of module. These syntactic forms enable compilers to check that one process does not refer to the variables of another process—an extremely dangerous kind of programming error.

In Edison, the same operational security can be achieved by adopting a programming style in which processes are described by procedures which are called by concurrent statements (Algorithm 10). Since each procedure call creates fresh instances of the local variables of the given procedure, and since these variables are only accessible to the calling process, this solution is as secure as any.

```
proc producer
var x: char
begin read(x);
  while x <> '.' do
    put(x); read(x)
  end;
  put(x)
end

proc consumer
var y: char
begin get(y);
  while y <> '.' do
    write(y); get(y)
  end;
  write(y)
end

cobegin 1 do producer
also 2 do consumer end
```

Algorithm 10

On the other hand, the programmer can also write concurrent programs, such as the following in Edison

```
var x, y: char
begin read(x);
  while x <> '.' do
    y := x;
    cobegin 1 do write(y)
    also 2 do read(x) end
  end;
  write(x)
end
```

Since the compiler provides no assistance in checking whether these processes refer simultaneously to the same variables without proper synchronization, such programs must be written with extreme care.

The added flexibility (and insecurity) of Edison compared to Concurrent Pascal will be viewed by some as a step backwards and by others as a challenge. To me it is simply an experiment that will either confirm or contradict my current feeling that programming languages cannot be expected to support complex abstractions, but should instead make it reasonably convenient to adopt programming styles which use simpler concepts to construct the more complex ones. Needless to say this makes the design of a programming language a delicate balance between the anarchy of assembler language and the straightjacket of highly specialized languages.

5 SCOPE RULES

Although Pascal is block structured it is not clear from the report whether this means that the scope of a named entity is the entire block in which the entity is declared or whether the entity is known from the point of its declaration to the end of the given block (Jensen 1974).

The former interpretation is used in the Algol 60 report from which the block concept originates (Naur 1962). The latter meaning is implemented by most Pascal compilers and is the one described in the Edison report.

The requirement that a named entity must be declared before it is used makes it possible for a compiler to build a name table and verify that the names are used correctly during a single scan of the program text. In most cases this convention is quite natural since we are used to reading text in the order in which it is written. Occasionally, however, programmers are mystified by the compiler's refusal to accept an Edison program with the following structure

```
array line [1:80] (char)
  ...
proc program(
  proc writetext(text: line))
array line [1:80] (char)
var x: line
  ...
begin ... writetext(x) ... end
```

The compiler insists that the statement

```
writetext(x)
```

contains a type error. The problem is an inadvertant use of two declarations of the type called line in nested blocks that are separated by more than a hundred lines of text and therefore do not appear on the same page of text.

To the compiler these declarations introduce two different data types with the same name. The scope of the first line type extends from its declaration in the outer block up to the declaration of the second line type in the inner block. The second line type is valid in the rest of the program. This makes the parameter of the procedure writetext of the first type while the argument of the procedure call is of the second type.

The record fields in Pascal do not follow the normal scope rules. If two variables are declared as follows

```
var x: record y, z: char end; y: boolean
```

then the name y denotes a boolean variable when it stands alone but refers to a character field when it occurs in the variable symbol $x.y$. Although this convention seems natural enough when it is illustrated by example, it is nevertheless an exception that adds to the complexity of an already quite subtle set of naming rules.

In looking at the first Edison compiler I found that its most complicated part, the semantic analysis, introduced about 400 distinct names of which less than 40 were field names of records. I decided that anyone who can invent 360 different names can surely invent another 40. Consequently, in Edison the scope of a field name (like any other name) extends from its declaration to the end of the block which contains the declaration. Although a field name y can only be used in a field variable symbol of the form $x.y$ it cannot be redeclared with another meaning in the same block.

The reasons for including modules which restrict the scope rules further in the language have already been discussed in previous sections. Later I will discuss the difficulties of explaining scope rules precisely in prose.

6 DATA TYPES

The most significant advance made by Pascal ten years ago was the introduction of a set of data types which for the first time made systems programming in an abstract programming language practical. The influence of Pascal on Edison in the choice of data types is considerable. Nevertheless, it must be admitted that the finer conceptual details of the data types are somewhat complicated and are imprecisely described in the Pascal report.

In Edison I have tried to simplify and clarify some of the issues concerning data types. A type is a named set of values. Every value is of one and only one type, so the data types are disjoint.

The *subrange types* of Pascal can be used to introduce types that are either disjoint, contained in other types, or even overlapping, as illustrated by these examples

var x1: 1..10; x2: 11..20;
x3: 5..9; x4: 0..15

This raises complicated issues, such as: Is the expression $x1 + x2$ valid, and if so, what is its type (1..10, 11..20, 1..20, or perhaps 12..30)? Is a value of type 1..10 also of type 0..15? Are some of the values of type 0..15 also of type 1..10 while others are also of type 11..20? Are the values of these subrange types in general compatible with the integer values or are they distinct types?

The elimination of subrange types from Edison makes these issues irrelevant.

Subrange types may originally have been introduced in Pascal to define the possible values of array indices. In Edison the concept of *range* serves the same purpose. A range is not a type. It is a finite set of successive, elementary values of the same type from a lower bound to an upper bound (both included). An array type declaration, such as

array table [1:100] (int)

describes a range of possible indices from 1 to 100. The indices of the array elements are of type integer.

The syntactic rules of Pascal and Modula make type definitions of the form

type temperature = integer;
speed = integer

legal but do not define the meaning of this. Are temperature and speed just different names for the standard type integer? In that case, the use of type synonyms serves no purpose since the addition of temperatures to speeds is now a valid operation (because they are just integers). If, on the other hand, temperature and speed are distinct types then we have conceptual confusion in the language. For although the types are now incompatible, we would presumably expect the arithmetic operations and the ordering relations to apply to both of them. But the set of values to which these operations apply is per definition the set of integers. And, to introduce several different kinds of incompatible "integers" nullifies in one stroke one of mankind's most important concepts.

Since the syntax of Edison does not include the rule

Type name = Type name

the problem never arises.

The conceptual clarity of operational security that is gained from the use of types is considerable. But, as all abstractions, types derive their power from an oversimplified view of the security problem: all integer values are considered compatible irrespective of the physical or conceptual properties they represent. One cannot hope to use the type concept to capture the subtle distinctions between temperatures and speed any more than it can describe the constraints among related values of the same type, such as the requirement that the sum of free and used pages on a disk must equal the total number of available pages.

In Pascal and Modula type declarations begin with a name as in

T1 = **record** x: char; y: integer **end**;
T2 = **array** [1..10] **of** char

The compiler must therefore scan three symbols before it can determine whether the type is a record or an array. This is an ugly exception in a recursive descent compiler that otherwise only needs to look at a single symbol to determine which syntactic form it is the beginning of.

Error recovery is also complicated by the above syntax. After a syntax error, such as

```
T1 = record x: char; y: integer ned;
T2 = array [1..10] of char
```

the transcribed keyword *end* will now be interpreted as a misplaced field name *ned*, and so will the type name T_2 that follows the semicolon. The result is a burst of misleading error messages referring to all uses of variables of type T_2.

In Edison each type declaration begins with a word symbol (instead of a name), for example

```
record T1 (x: char; y: int)
array T2 [1:10] (char)
```

After a syntax error, such as

```
record T1 (x: char; y: int]
array T2 [1:10] (char)
```

the compiler will skip the right bracket and correctly recognize the word *array* as the beginning of a new type declaration.

It is true that assignments and procedure calls suffer from the same problem of error recovery since they both begin with names and also may contain names as operands. But the accidental erasure of a statement during compilation is not as serious as the erasure of a declaration since other statements cannot refer to the missing one by name.

It is characteristic of a superior tool like Pascal that its notation is felt to be so natural that there seems to be no reason for the programmer to look for a better one. So it was without much thought that I originally used Pascal's notation for type declarations in Edison. As I was testing an Edison compiler written in Pascal the problems of error recovery became apparent. But even then it took a while before I was mentally prepared to propose an alternative syntactic notation.

With one exception (strings) the fixed length of array types in Pascal has never bothered me in the design of operating systems and compilers. But *character strings* pose special problems that are cleverly hidden by the use of *ad hoc* means in Pascal and Concurrent Pascal. (The Modula report is too vague on this point for me to understand it.)

By their nature character strings are of different lengths, for example

```
'tape'    'emono'
```

How then does one write a procedure that outputs a string to some device (or file) one character at a time? In Pascal you can try the following

```
procedure writetext(text: phrase);
var i: integer; c: char;
begin i := 0;
  while i < n do
  begin i := i + 1;
    write(text[i])
  end
end
```

where

```
type phrase = array [1..n] of char
```

Now, if $n = 4$ the call

```
writetext('tape')
```

is valid, but the call

```
writetext('emono')
```

is not since the string argument is now of length 5.

Why then is it not felt as an intolerable problem in Pascal? Because Pascal includes a standard procedure named write which is cleverly designed to accept strings of any length as shown below

```
write('tape')    write('emono')
```

But this procedure cannot be written in the language itself.

Since Concurrent Pascal is a language for operating system design, a standard procedure such as write cannot be built into the language. For one of the purposes of Concurrent Pascal is to design a filing system that includes a write procedure.

This difficulty is circumvented by the following *ad hoc* rule in the Concurrent Pascal report: "An argument in a procedure call corresponding to a string parameter may be a string of any length." This rule makes it possible to write a procedure that outputs a string of any length terminated by a given delimiter, say #

```
procedure writetext(text: phrase);
var i: integer; c: char;
begin c := text[1]; i := 1;
  while c <> '#' do
  begin write(c); i := i + 1;
    c := text[i]
  end
end
```

This procedure can print any string of at most n characters as illustrated by the calls

writetext('tape#') writetext('emono#')

In both cases, however, the actual parameter value will be a string of n characters consisting of the given character string followed by some characters of unknown value (an implementation detail that programmers fortunately seldom discover).

In designing the array constructors of Edison these problems were carefully considered. An array constructor, such as

phrase('t', 'a', 'p', 'e', '#', ' ', ' ')

denotes an array value of type phrase (assuming that $n = 7$). The elements of the array value are given by a list of characters.

In Edison a character string, such as

'tape# '

can be used as an abbreviation for a character list

't', 'a', 'p', 'e', '#', ' ', ' '

that contains the same sequence of characters separated by commas. The previous array constructor can therefore also be written as follows

phrase('tape# ')

In general, an array constructor must contain an expression for each element of the array type (unless it is a string type). In the latter case, the above constructor may be written simply as

```
phrase('tape#')
```

This is an abbreviation for an array value consisting of the given character values followed by spaces to make the string of length n.

A graphic character, such as #, is denoted '#' in Edison. A *control character* is given by its ordinal value in the character set. In the ASCII character set, the control character new line is character number 10. In Concurrent Pascal, Modula, and Edison the new line character is denoted

```
'(:10:)'      10C      char(10)
```

respectively. In Concurrent Pascal the control characters may be given names, such as

```
const nl = '(:10:)'
```

But the name nl cannot be used within a character string

```
'tapenl#'
```

because it is indistinguishable from the graphic letters n and l. This leads to the following awkward notation

```
'tape(:10:)#'
```

in Concurrent Pascal. Whether the string

```
'tape10C#'
```

has the intended meaning in Modula I cannot tell from the report. In Edison, the inclusion of character names in string constructors is straightforward

```
const nl = char(10)

phrase('tape', nl, '#')
```

In Pascal and Modula the absence of constructors makes the use of constant tables so awkward that one soon adopts a programming style in which decisions are made by case statements rather than table lookup. The ability to initialize tables by a single assignment statement in Edison should once again make table-driven decision-logic an attractive alternative.

The Concurrent Pascal compiler, which is written in Sequential Pascal, makes extensive use of the *variant records* and *pointers* of Pascal (Hartmann 1977). Since these concepts are both complicated and insecure to use I decided to write an Edison compiler without using them. The resulting compiler was much easier to understand. Evidently these tools were only used previously because they were there.

Only in one part of the compiler did I feel the need for variant records. During semantic analysis a name index is used to retrieve the attributes of a named entity from a table. Since the attributes depend on the kind of entity the name refers to, the name table is best described as an array of variant records of the following Pascal type

```
nameattr =
  record link: integer;
    case kind: namekind of
      constant:
        (consttype, constvalue: integer);
          ...
      procparam, procedur:
        (proclevel, procaddr, proctype,
          param: integer)
  end
```

In Edison the name table is described as an array of elements of the type

```
record nameattr (kind: namekind; link: int;
  none1, none2, none3, none4, none5: int)
```

The variants of the type are described by separate record types

```
record constattr (constkind: namekind;
  constlink, consttype, constvalue,
  none6, none7, none8: int)
    ...
record procattr (prockind: namekind;
  proclink, proclevel, procaddr, proctype,
  param, none11)
```

The above records are padded with fields to make them all of the same length.

The compiler first uses a name index x to select a general name description of type nameattr. If the kind field is equal to constant, the name description is then retyped to be of type constattr as illustrated by the following program piece

```
var x, value: int

if names[x].kind = constant do
   value := names[x]:constattr.constvalue
      ...
```

If v denotes a variable of some type T_1 then $v{:}T_2$ denotes the same variable considered to be of another type T_2 of the same length. The retyped variable $v:T_2$ has a value of type T_2 with the same stored value as the value of v. A selection of a field f of the retyped variable is denoted $v:T_2.f$.

I cannot recall any other system program that requires the use of variant records. If they are that rare in practice then dynamic retyping seems preferable to introducing a very complicated data type in Edison with a matching set of constructors and control statements. But, if the use of variant records is more frequent than I thought, then their elimination from Edison must be regarded as a mistake. Until the experimental data are available from a wide range of applications, the elimination of variant records seems a worthwhile experiment.

The pointers of Pascal have been omitted from Edison for the same reason that variant records were eliminated. It is a complicated concept that appears to be a concession to the current practice of programming even though its full implications are not well understood. The need to define cyclical data structures in terms of different record types that can point to one another breaks the general rule that a type must be declared before it can be used as a subtype of another type. Furthermore, like the tag fields of variant records, pointer variables must be initialized to nil automatically to enable a processor to detect the use of a meaningless pointer. Otherwise, the effect of an assignment can even change the code of a program (just as in assembler language). No other kind of variable poses this serious problem.

The dynamic allocation of storage accessed through pointers is quite complicated particularly when it is combined with the storage allocation of concurrent processes. And there is no secure way to release the storage again and make sure that it can no longer be accessed through existing pointer values. Since the aim of Edison is utter simplicity rather than a compromise dictated by tradition the pointer types of Pascal could only be excluded from the language.

Most operations on a pair of data values are valid only if the values are of the same type. Unfortunately, the precise meaning of *type equivalence* is not defined in the Pascal and Modula reports. The Concurrent Pascal

report states that two types are compatible if one of the following conditions is satisfied:

1. Both types are defined by the same type or variable declaration, or
2. Both types are subranges of a single enumeration type, or
3. Both types are strings of the same length, or
4. Both types are sets of compatible base types. The empty set is compatible with any set.

Although this clarifies the matter, it is not particularly simple to remember or implement.

Every data type used in an Edison program has a name—either a standard name (int, char, bool) or a name introduced by a type declaration.

The nameless data types of Pascal and Modula, as in

var x: **array** [1..100] **of** integer

do not exist in Edison and nor do subrange types.

In Edison each set constructor includes the name of its type and so does a string constructor. Even the empty set has a type name!

Consequently, the issue of type equivalence is settled by the simple rule that two types are the same only if they are denoted by the same name and have the same scope.

7 STATEMENTS

Where Pascal and Modula use 11 and 9 different sequential statements, respectively, Edison supports only 5 (including the empty statement)

skip
assignment
procedure call
if statement
while statement

Inspired by Dijkstra's guarded commands (1975) the syntactic forms of the *if* and *while* statements have been generalized and made more uniform

if B_1 **do** SL_1	**while** B_1 **do** SL_1
else B_2 **do** SL_2	**else** B_2 **do** SL_2
...	...
else B_n **do** SL_n **end**	**else** B_n **do** SL_n **end**

where the $B's$ and $SL's$ denote boolean expressions and statement lists. But, in contrast to guarded commands, the if and while statements of Edison are deterministic since the boolean expressions are evaluated in the order written (and not in unpredictable order).

Programs, such as compilers, that accept both correct and incorrect input often describe a choice among several different actions on valid inputs followed by a single action to be taken on invalid input. For example

```
if mode = constant do
  constant_factor(typ, endfactor)
else mode in typekinds do
  constructor(typ, endfactor)
else mode in varkinds do
  variable_factor(typ, endfactor)
else mode in prockinds do
  function_call(typ, endfactor)
else true do
  kinderror2(x, typ); nextsym
end
```

If the order in which the expressions are evaluated is unknown (as it is for guarded commands) then the final expression in this example must be changed from *true* to the following monstrosity

```
(mode <> constant) and
not (mode in typekinds) and
not (mode in varkinds) and
not (mode in prockinds)
```

In Edison the execution of an if statement has no effect if all the boolean expressions yield the value false. For guarded commands, Dijkstra assumed that this would cause a program failure. If a programmer wishes to provoke such a failure it can be done by ending an if statement as follows

```
... else true do halt end
```

where halt is a procedure that causes program failure when executed. (The language Edison-11 for the PDP 11 computers includes a standard procedure halt.)

Dijkstra has convincingly demonstrated the use of the while statement that describes a repeated choice among several conditional statements. A beautiful example of its use is found in the Edison compiler in a procedure that recognizes the syntax of a variable symbol consisting of a variable name possibly followed by one or more field names, index expressions, or type names (indicating a temporary retyping of the variable). When the variable name has been recognized the compilation proceeds as follows

```
while sym = period1 do
  field_selector(endvar)
else sym = lbracket1 do
  indexed_selector(endvar)
else sym = colon1 do
  type_transfer(endvar)
end
```

The if statements of Pascal are special cases of the ones in Edison and can be expressed as follows

if B **do** S **end**

if B_1 **do** S_1
else true **do** S_2 **end**

In an attempt to eliminate empty options, I have resisted the temptation to introduce an abbreviation for the clause

else true **do**

Needless to say, the compiler does not emit any code for this clause.

The while statements of Pascal and Modula are also special cases of the one used in Edison and can be expressed as follows

while B **do** SL **end**

The conditional critical region proposed by Hoare

when B **do** SL **end**

is a special case of the more general *when statement* of Edison which has the same syntactic structure as the if and while statements

when B_1 **do** SL_1
else B_2 **do** SL_2
...
else B_n **do** SL_n **end**

In an earlier paper, Staunstrup and I introduced this language concept, which we called a guarded region, and illustrated it by examples (Brinch Hansen 1978). The following example describes a process that continuously polls three boolean variables and sounds an alarm if any one of them yields the value true.

```
proc poll(var on1, on2, on3: bool)
begin on1 := false; on2 := false; on3 := false;
  while true do
    when on1 do alarm(1); on1 := false
    else on2 do alarm(2); on2 := false
    else on3 do alarm(3); on3 := false end
  end
end
```

As Wirth has pointed out what is omitted from a programming language is as important as what is included.

The *repeat statement* of Pascal and Modula

repeat SL **until** B

can be represented in Edison either by the statements

SL; **while not** B **do** SL **end**

or by the following program piece

```
var again: bool
begin again := true;
  while again do
    SL; again := not B
  end
end
```

The *for statement* of Pascal

for $i := 1$ **to** n **do** S

can be written either as

$i := 0;$
while $i < n$ **do** $i := i + 1; S$ **end**

or as

$i := 1;$
while $i <= n$ **do** $S; i := i + 1$ **end**

in Edison. It may seem that the two representations of the for statement are not equivalent since the final value of the control variable i will be n in the first version and $n + 1$ in the second. Although the Pascal report is silent on this issue the more informal user report states that "the final value of the control variable is left undefined upon normal exit from the for statement." This rule is introduced precisely to give the language implementor the freedom to choose the most efficient implementation for a given machine.

The decreasing variant of the for statement in Pascal

for $i := n$ **downto** 1 **do** S

can be represented similarly in Edison.

The *case statement* of Pascal and Modula

case *expression* **of**
  $constant_1$: S_1;
  $constant_2$: S_2;
    ...
  $constant_n$: S_n
end

was originally part of Edison as well, but was later removed. The experience of writing the Edison compiler showed that a case statement often is used to describe actions on symbols that are grouped together as in

case ch **of**
  'a', 'b', ..., 'z': name;
  '0', '1', ..., '9': numeral;
    ...
end

The same clarity and efficiency can be achieved by using a combination of if statements and sets representing the necessary symbol classes, for example

```
if ch in letters do name
else ch in digits do numeral
  ...
end
```

This grouping of symbols reduces the number of conditional statements to eight in the lexical analysis of the compiler.

The syntactic and semantic analysis of the Edison compiler use recursive descent and include one procedure for each syntactic form of the language. Since a given procedure only is prepared to recognize a small number of symbols (corresponding to the syntactic form it represents), an if statement combined with sets is again quite efficient. Several examples described earlier illustrate this point.

The code generator is the only part of the compiler that inputs one symbol at a time and immediately uses it to branch to one of about sixty procedures. In that one case, I had to resort to an awkward construct of the form

```
if op <= construct2 do
  if op = add2 do add
  else op = also2 do alsox(a, b)
  else op = and2 do andx
  else op = assign2 do assign(a)
  else op = blank2 do blank(a)
  else op = cobegin2 do cobeginx(a, b, c)
  else op = constant2 do constant(a)
  else op = construct2 do construct(a) end
else op <= endproc2 do
  if op = difference2 do difference
    ...
```

to obtain fast compilation. But that was only 64 lines out of a compiler of 4300 lines and hardly worth the addition of another kind of statement to the language.

The *with statement* of Pascal and Modula appears to be used mostly to assign values to all the fields of a record variable as in the following statement

```
with names[nameno] do
begin
  kind := mode;
  minlevel := scope;
  maxlevel := origin;
  originalname := x
end
```

In Edison this is expressed more concisely by means of an assignment statement and a record constructor

```
names[nameno] := nameattr(mode, scope, origin, x)
```

The *goto statement* of Pascal was already eliminated in Concurrent Pascal and was never missed during the five years of experience with this language.

Finally, I agree with Dijkstra that the *empty action* (like the number zero) should be denoted by a distinct symbol

skip

and not be implied by the absence of one.

Although the use of distinct terminators, such as *endif, endwhile, endwhen, coend, endproc*, and *endmodule* would have made error recovery more successful in the compiler, I have used the symbol *end* in all these cases to make the typographical image of correct programs more pleasing (for they are the only ones I publish). This is obviously a matter of personal preference.

8 INPUT/OUTPUT

In Concurrent Pascal, all input/output is handled by standard procedure calls of the form

```
io(data, operation, device)
```

The calling process is delayed until the operation is completed. In the meantime, other processes can continue to use the rest of the computer. The advantage of this approach is that a data transfer is just another sequential operation that takes a finite time and produces a reproducible result. Another benefit of making input/output an indivisible operation for a single

process is that peripheral interrupts become irrelevant to the programmer. They are handled completely at the machine level as part of the implementation of the io operations.

The disadvantage of using a single standard procedure for low-level input/output is that the system kernel for the language must contain a separate piece of code for each kind of peripheral device. This means that industrial programmers must be prepared to extend the kernel (which is a non-trivial task since peripheral interrupts interact with the processor multiplexing).

In Modula, Niklaus Wirth offered a more practical solution in the form of *device modules* combined with device processes and device registers. The whole concept was tailored to the PDP 11 computers to enable programmers to write device procedures in the language itself. Algorithm 11 shows a device module that outputs one character at a time on a screen. The module consists of a device process, named driver, which is connected to an exported procedure, named display, through a set of variables used as a character buffer.

The standard procedure *doio* delays the device process until the screen produces an interrupt with the octal number 64B. Since the doio operation only can be performed by a device process and since a device process must be hidden within a device module the above appears to be the simplest possible way of displaying a character on the screen. It does not seem to be possible to eliminate a device process and let a user process control the peripheral directly. This example shows that Modula (like Concurrent Pascal) attempts to support and enforce a particular programming style by means of very specialized language constructs.

The Edison language implemented for the PDP 11 computers is called Edison-11. Since the language is designed to support inexpensive microprocessors I decided to anticipate this use and ignore interrupts completely on the PDP 11 (even at the machine level).

In Edison-11, input/output is controlled by standard procedure calls of the form

```
place(device address, value)
obtain(device address, variable)
sense(device address, value)
```

The operation *place(x,y)* assigns the integer value y to the device register with the byte address x. The operation *obtain(x,y)* assigns the value of the device register with the byte address x to the integer variable y. The operation *sense(x,y)* compares the integer value y to the value of the device

```
device module screen [4];
  define display;
  var slot: char; full: boolean;
    nonempty, nonfull: signal;

  procedure display(c: char);
  begin
    if full do wait(nonfull) end;
    slot := c; full := true;
    send(nonempty)
  end put;

  process driver [64B];
  var status [177564B]: bits;
    buffer [177566B]: char;
  begin
    loop
      if not full do wait(nonempty) end;
      buffer := slot; full := false;
      status[6] := true; doio; status[6] := false;
      send(nonfull)
    end
  end driver;

begin full := false; driver end screen
```

Algorithm 11

register with the byte address x. If some of the corresponding bits of the two values both are equal to one then the operation yields the value true; otherwise, it yields the value false.

Edison-11 also includes octal numerals of the form

#177564

to denote device addresses. This option is necessary because the computer manufacturer does not use the decimal system in the computer manuals. Why I do not know.

A process described by a concurrent statement in Edison-11 can output a character directly on the screen by calling the following procedure

```
proc display(c: char)
const status = #177564; buffer = #177566;
  ready = #200
begin
  when sense(status, ready) do
    place(buffer, int(c))
  end
end
```

The effect of executing the when statement is to delay the output until the status register shows that the device is ready.

A PDP 11 computer executes one process at a time. This continues until the process either terminates or attempts to execute a when statement in which the boolean expressions yield the value false. The waiting processes are executed in cyclical order in Edison-11. The simplicity of process scheduling makes the overhead of process switching five times shorter than in Concurrent Pascal (which is interrupt driven). The Edison-11 implementation ignores all interrupts.

The input or output of non-elementary data types requires the use of a standard function call

addr(y)

which yields the byte address of a variable y of any type. The operation *place(x,addr(y))* assigns the address of the variable y to the device register with the address x. This operation is necessary to transfer data between the variable y and a block-oriented device, such as a disk or magnetic tape.

9 LANGUAGE DESCRIPTION

In 1967 the designer of the programming language Pascal, Niklaus Wirth, wrote that "the definition of a language, comprising its syntax specifying the set of well-formed sentences, and its semantics defining the meaning of these sentences, should not extend over more than 50 pages." The Edison report is comparable in size to the reports which define its predecessors: Algol 60 (43 pages), Pascal (38 pages), Concurrent Pascal (34 pages), and Modula (29 pages). The Edison report is 34 pages long. (The sizes of these reports are measured in pages of 50 lines each.)

The shortness of a language report is, of course, of no help unless it is written with complete clarity. As Wirth put it: "In programming, we are dealing with complicated issues, and the more complicated the issue is,

the simpler the language must be to describe it. Sloppy use of language—be it English, German, Fortran or PL/1—is an unmistakable symptom of inadequacy."

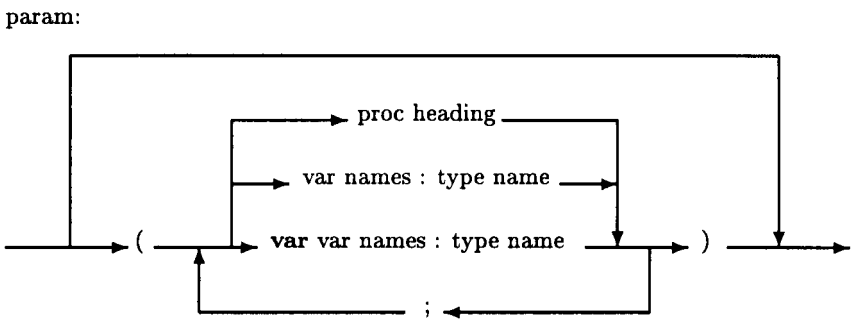
The only language report that has been widely recognized as a model of clarity is the Algol 60 report written by Peter Naur. In 1967 Donald Knuth wrote that "the most notable feature of the Algol 60 report was the new standard it set for language definition." Unfortunately, as Tony Hoare said, the Algol 60 report was a considerable improvement over its successors.

Even though the Pascal language was far more successful than Algol 60, Nico Habermann severely criticized Wirth for the imprecision of the Pascal report and pointed out that it was hiding some conceptual inconsistencies in the definition of data types (Habermann 1973). Peter Naur pointed out that the Concurrent Pascal report, which I wrote, suffered from similar problems.

The task of writing a language report that explains a programming language with complete clarity to its implementors and users may look deceptively easy to someone who hasn't done it before. But in reality it is one of the most difficult intellectual tasks in the field of programming.

In writing the Edison report I have benefited greatly from the constructive criticism of Peter Naur. Naur made almost no comments about my choice and design of language features. His main concern was the clarity of the report. I would write a complete draft of the language report and Naur would then point out what the weaknesses were and suggest broadly how they might be removed in my next draft. The following describes the stages of development which the Edison report went through over a period of two years.

The first Edison report of January 1979 used *syntax graphs* of the form shown below with an explanation such as the following:



"Each parameter declaration introduces a name to represent the parameter and specifies its type or procedure heading."

"A value parameter is a local variable which is assigned the value of an expression before the procedure is executed.

A variable parameter denotes a variable which is bound to the procedure before it is executed. The symbol *var* distinguishes a variable parameter from a value parameter.

A procedure parameter denotes another procedure which is bound to the given procedure before the latter is executed."

About this report, Naur wrote the following:

"The weaknesses that will be described in most cases are such that may cause only little difficulty to a reader who is familiar with, for example, reports on the language Pascal, and who is generous in his willingness to rely on his own ability to fill in open holes by guesses and to remove inconsistencies. With the view taken all such holes and inconsistencies are unacceptable and should be removed, as far as possible."

"There is a poor connection between the formal syntactic description and the prose explanation."

I had used syntax graphs because they enable a programmer to see quite complex structures at a glance. Unfortunately, when a complicated concept is presented under a single name, the substructures of the graph have no names and cannot easily be referred to in the text. So, although the text refers to the concept of a "parameter declaration," the graph does not reveal what it is. The reader must also guess what the syntax of value, variable, and procedure parameters looks like. Note also that rather than explaining the meaning of certain concepts (such as variable parameters) the first report would use suggestive (but undefined) terms such as "binding."

In the second Edison report of July 1979 a parameter list was defined as follows (now using an extended *Backus-Naur form* instead of syntax graphs)

Parameter list:
  Parameter group [';' Parameter group]∗
Parameter group:
  Value parameter group # Variable parameter group
Value parameter group:
  Parameter name [',' Parameter name]∗ ':' Type name
Variable parameter group:
  '**var**' Parameter name [',' Parameter name]∗ ':' Type name
Parameter name:

Name

(In this version of the language I had tentatively excluded the use of procedures as parameters of other procedures.)

These syntactic forms were now explained as follows:

"A parameter list consists of a sequence of parameter groups. Each parameter group is either a value parameter group or a variable parameter group.

A value parameter group introduces one or more names, called parameter names, to denote value parameters. Each value parameter is a local variable that is assigned the value of an argument before the procedure block is executed.

A variable parameter group introduces one or more names to denote variable parameters. Each variable parameter denotes a variable argument which is selected before the procedure block is executed. During the execution of the procedure block all operations performed on the variable parameter stand for the same operations performed on the variable argument. The variable parameter is said to be bound to the variable argument during the execution of the procedure block."

The concept of a parameter list is now presented as the composition of four simpler concepts. Each concept is described by a syntactic form that introduces a name for the concept and defines its syntax. The prose explanation carefully follows the syntactic structures and explains their meaning using the same terminology. As a minor detail the meaning of variable binding is now explained.

The idea of using an abundance of simple syntactic forms to introduce names for most concepts, so that the text can refer directly to these definitions, had not occurred to me previously. But, once you realize that syntactic forms can be used to define terminology (and not just syntax), the job of writing a language report becomes far more well-defined. When the syntactic forms become very trivial the syntax graphs lose their visual appeal and appear bulky and clumsy. So I returned to the BNF notation, which has the great advantage that it can be included in program texts as comments.

Later, when I wrote the Edison compiler, I found that the syntactic forms used in the report were so short that each of them conveniently could be represented by a single procedure with the same name as the syntactic form. Each procedure is now preceded by the corresponding syntactic form written as a comment in BNF notation.

About the second report, Naur wrote this:

"The report is a vast improvement over the previous version in clarity, consistency, and completeness. The remaining weaknesses, described below in detail, are to a large extent concerned merely with finer matters of conceptual clarity."

After this pleasant introduction, Naur goes on to enumerate 79 conceptual problems. The first of these is that "a number of weaknesses of the description can be traced to a general disinclination to accept abstract notions, not uniquely associated with syntactic entities of the language, as well-defined useful constituents of the language and its description."

The conceptual difficulties of trying to describe all concepts purely in terms of their notation are both philosophical and practical. If, for example, the notion of a data value is purely notational then it is quite difficult to explain why the relation

$$\text{'A'} = \text{char}(65)$$

is true for the ASCII character set. But the matter becomes quite simple if both symbols are viewed as different notations for the same abstract concept—the first capital letter of the alphabet.

In the third report of January 1980 the distinction between fixed values and the notations used to represent them in the language was made as follows: "A constant is a fixed value of a fixed type," and "A constant symbol denotes a constant."

The distinction between an *abstract concept* and the *symbol* that denotes the concept is immediately obscured when a syntactic form is given the same name as the abstract concept. In the second Edison report, a variable used as an operand was described by a syntactic form named variable

```
Variable:
  Whole variable # Subvariable
```

The report explained that "a variable denotes either a whole variable or a subvariable." For a purist this is already unsatisfactory. Does the term "variable" in that sentence refer to the abstract concept of a variable that exists during program execution, or does it refer to one of the symbols in the program text described by the syntactic form named variable? For the writer of the language report it becomes downright awkward to describe what action a processor takes when a variable is used as an operand in a program. If we stick strictly to the rule of using the name introduced by the syntactic form, the result is the following cryptic sentence: "The

evaluation of a variable causes the processor to locate the variable in the current context."

In the third report, the matter was settled by making a distinction between a variable (which is a named entity that may assume any of the values of a known type) and a variable symbol (which denotes a variable used as an operand). It was now easy to explain that "the evaluation of a variable symbol involves locating the corresponding variable in the current context."

The failure to make this conceptual distinction pervades most language reports. In the Pascal, Concurrent Pascal, and Modula reports one finds syntactic forms named constant, number, type, and variable instead of the more appropriate terms constant symbol, numeral, type description, and variable symbol (or similar ones).

After reading both the first and second drafts of the Edison report, Naur felt that "the central concept of types is not properly introduced, while the many different aspects of it are scattered around in an unsystematic and confusing manner." In the third version of the report I tried to clarify the *type concept* and its relation to the abstract notions of *value* and *operation*. I did not, however, succeed in organizing the report into chapters each describing all the aspects of a single type in one place.

The problem is that the type concept pervades the whole language. To explain even a simple type, such as boolean, in all its aspects, one must discuss constants, ranges, variables, expressions, functions, and statements as they relate to boolean values. Now that is practically the whole language right there. So, even to begin this chapter, one has to assume that the reader already understands all these concepts in general. And then one has to do it all over again for integers, characters, enumerations, arrays, records, and sets. It ends up being a very repetitive report in which the same story is told seven times. And that is, of course, because the language is designed in such a way that all the types have very similar properties which are described by very similar syntax.

After spending about 25 full working days trying to write a report along these lines I gave up and wrote another one instead (the third version). As a compromise, I now described several facets of each kind of type in the same chapter, namely the introduction of a new data type by means of a type declaration, the representation of values of the type by constants or constructors, and the meaning of the operators that apply to these values. The chapters on data types were followed by chapters describing the general properties of variables, expressions, statements, and procedures.

The concept of *scope* is explained quite concisely in the Algol 60 report:

"Every block automatically introduces a new level of nomenclature. This is realized as follows: Any identifier occurring within the block may through a suitable declaration be specified to be local to the block in question. This means (a) that the entity represented by this identifier inside the block has no existence outside it and (b) that any entity represented by this identifier outside the block is completely inaccessible inside the block.

Identifiers occuring within a block and not being declared to this block will be non-local to it, i.e. will represent the same entity inside the block and in the level immediately outside it."

"Since a statement of a block may itself be a block the concepts local and non-local must be understood recursively."

"No identifier may be declared more than once in any one block head."

One may complain about the complexity of block structure but not, I think, about the precision with which it is explained in the Algol 60 report. In the Pascal report written ten years later the explanation of the same concept is much more incomplete:

"All identifiers introduced in the formal parameter part, the constant definition part, the type definition part, the variable, procedure or function declaration parts are *local* to the procedure declaration which is called the *scope* of these identifiers. They are not known outside their scope."

This explanation fails to distinguish between a name and the entity it denotes. Since a name can be used with different meanings in different blocks, it is not the name itself that is unknown outside a given block but the use of the name to denote a given entity declared inside the block. In short, it is named entities (rather than names) which have scopes. I too failed to make that distinction in the earlier Edison reports.

More importantly, the above explanation from the Pascal report does not explain which entity a name denotes when it is declared with different meanings in nested blocks. To paraphrase an earlier remark, such weaknesses of the Pascal report have (fortunately) caused little difficulty for readers who are already familiar with the Algol 60 report.

The Concurrent Pascal and Modula reports also do not succeed in explaining scope rules precisely. This is much more unfortunate since these languages introduce new forms of modularity which primarily serve to change the traditional scope rules of block structured languages. My present attempts to explain these rules are contained in the chapter on Named Entities in the Edison report.

Having deprived myself of the opportunity to gloss over ill-understood issues in the Edison report it was quite embarrassing to explain the effect of executing processes described by *concurrent statements*. Although a programmer may have no difficulty understanding a particular concurrent statement, the language designer faces the much harder task of explaining in a few paragraphs the meaning of every conceivable use of this idea.

Unfortunately, one can only make a simple statement about the most trivial (and uninteresting) use of concurrent processes, which is the following: "If concurrent processes only operate on private variables and disjoint, common variables then the effect of executing a concurrent statement is the same as the effect of executing the process statements one at a time in any order."

As soon as processes begin to communicate, one needs the whole apparatus of the Gries–Owicki theory to reason about the effects of using semaphores, buffers, monitors, and so on (Owicki 1976). And, even so, processes must satisfy very rigid assumptions for the theory to apply. Although one can design language features (such as monitors) which enable a compiler to check that some of these assumptions are satisfied one cannot expect a programming language to capture all aspects of well-behaved concurrent programs. And, even if one could, it would not be very helpful to repeat the entire theory behind these concepts in a language report.

The difficulty of explaining what concurrent processes really do will, of course, arise in any report that describes a concurrent programming language. In yet another appeal to the generous reader the Concurrent Pascal and Modula reports evade the issue by suggestive use of undefined phrases to the effect that "all processes are executed concurrently." In writing the Edison report I decided to make the difficulty explicit by the following statement:

"If concurrent processes operate on intersecting, common variables then the effect of executing a concurrent statement is generally unpredictable since nothing is known about the order in which the processes operate on these variables. However by restricting the operations on intersecting variables to well-defined disciplines under the control of modules, procedures, and when statements it is possible to formulate concurrent statements that make predictable use of such variables. The theories used to reason about concurrent statements, when statements, and common variables are beyond the scope of this report."

In effect, the report says that one can write meaningless Edison programs

that will produce unpredictable results. But it does not prescribe how to avoid doing this. This is surely a startling property of a programming tool. The report does, however, say that when this happens even the most basic programming concepts can lose their meaning:

"The meaning of variable retrieval and assignment defined earlier is valid only if these operations are performed one at a time on a given variable. Concurrent processes can ensure that this assumption is satisfied by performing all operations on intersecting, common variables within when statements."

The execution of recursive procedures and concurrent statements will generally create multiple instances of named entities, such as variables. Furthermore, procedures used as parameters of other procedures can have side-effects on some of these variable instances (and not necessarily the most recent ones!). In such cases, the language report must make it possible to decide which variable instance a process refers to at any given time.

In the fourth version of the Edison report (September 1980) the dynamic meaning of names is explained in terms of the so-called context of a process. This concept is introduced at the beginning of the report as follows:

"When a process executes a statement it operates on known entities selected from a set of entities called the current context of a process."

(A language implementor might say more loosely that a context is an abstract description of that part of a stack that is currently accessible to a process by following the chain of static links.)

Subsequent sections of the report then explain how the initial context of an executed program is established and how it is changed by the execution of procedure calls and concurrent statements.

A key idea is to associate contexts with procedures as well as processes. When a procedure body is executed by a process the current context of the process is attached as an attribute to all the local procedures declared in the body. If one of these procedures is called directly by the process its execution will take place in the context associated with the procedure. And if the procedure and its current context are bound to a parameter of another procedure the above rule still applies: when a procedure parameter is called it is executed in the context currently associated with the procedure parameter.

So the effect of any procedure call is now explained by a single rule.

This concludes the discussion of the linguistic dificulties which a language designer is faced with during the writing of a language report. The writing of the Edison report was far more difficult and time-consuming than the

selection of the language features and the design of the first compiler.

References

Brinch Hansen, P. 1970. The nucleus of a multiprogramming system. *Communications of the ACM 13*, 4 (April), 238–250. *Article 2.*

Brinch Hansen, P. 1972. Structured multiprogramming. *Communications of the ACM 15*, 7 (July), 574–578. *Article 4.*

Brinch Hansen, P. 1973. *Operating System Principles.* Prentice-Hall, Englewood Cliffs, NJ (July).

Brinch Hansen, P. 1975. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering 1*, 2 (June), 199–207. *Article 7.*

Brinch Hansen, P. 1977. *The Architecture of Concurrent Programs.* Prentice-Hall, Englewood Cliffs, NJ, (July).

Brinch Hansen, P., and Staunstrup, J. 1978. Specification and implementation of mutual exclusion. *IEEE Transactions on Software Engineering 4*, 5 (September), 365–370.

Brinch Hansen, P. 1981. Edison—A multiprocessor language. *Software—Practice and Experience 11*, 4 (April), 325–361.

Dahl, O.-J., Dijkstra, E.W., and Hoare, C.A.R. 1972. *Structured Programming.* Academic Press, New York.

Dijkstra, E.W. 1968. Cooperating sequential processes. In *Programming Languages*, F. Genyus, Ed. Academic Press, New York.

Dijkstra, E.W. 1975. Guarded commands, nondeterminacy and formal derivation. *Communications of the ACM 18*, 8 (August), 453–457.

Habermann, A.N. 1973. Critical comments on the programming language Pascal. *Acta Informatica 3*, 47–57.

Hartmann, A.C. 1977. A Concurrent Pascal compiler for minicomputers. *Lecture Notes in Computer Science 50*, Springer-Verlag, New York.

Hoare, C.A.R. 1972. Towards a theory of parallel programming. In *Operating Systems Techniques*, C.A.R. Hoare and R.H. Perrott, Eds. Academic Press, New York, 61–71.

Hoare, C.A.R. 1974. Monitors: An operating system structuring concept. *Communications of the ACM 17*, 10 (October), 549–557.

Jensen, K., and Wirth, N. 1974. Pascal—User manual and report. *Lecture Notes in Computer Science 18*, New York.

Knuth, D.E. 1967. The remaining trouble spots in Algol 60. *Communications of the ACM 10*, 10 (October), 611–618.

Naur, P., Ed. 1962. *Revised Report on the Algorithmic Language Algol 60.* Regnecentralen, Copenhagen, Denmark.

Owicki, S., and Gries, D. 1976. An axiomatic proof technique for parallel programs. *Acta Informatica 6*, 319–340.

Welsh, J., and McKeag, M. 1980. *Structured System Programming.* Prentice-Hall, Englewood Cliffs, NJ.

Welsh, J., Sneeringer, W.J., and Hoare, C.A.R. 1977. Ambiguities and insecurities in Pascal. *Software—Practice and Experience 7*, (November–December), 685–696.

Wirth, N. 1971. The programming language Pascal. *Acta Informatica 1*, 35–63.

Wirth, N. 1977. Modula: A language for modular multiprogramming. *Software—Practice and Experience 7*, 1 (January–February), 3–35.