SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 11, 397–414 (1981)

Edison Programs

PER BRINCH HANSEN

Computer Science Department, University of Southern California, Los Angeles, California 90007, U.S.A.

SUMMARY

This paper describes three sample programs written in the programming language Edison. These programs illustrate the practical use of modules, concurrent statements, and input/output operations. The paper concludes with a brief overview of the Emono operating system and the Edison compiler both of which are written entirely in Edison.

KEY WORDS Edison programs Modular programs Concurrent programs Input/output

CONTENTS

1. INTRODUCTION

This paper describes three sample programs written in the programming language Edison.[1] These programs illustrate the practical use of modules, concurrent statements, and input/output operations.

The first program handles the operating system task of copying the contents of a disk onto magnetic tape. The second program enables the user of a personal computer to measure his typing speed. The third program illustrates some of the programming techniques used in compilation. It prints an Edison program text and underlines the word symbols of the language. These programs have all been tested under the Emono operating system on a PDP-11/55 computer.

If these programs had been developed directly for a particular operating system such as Emono) they would have used the available procedures of that system to perform input/output operations. But since one of their purposes is to illustrate how Edison can be used to control peripheral devices the programs have been made largely self-contained.

The programming language Edison has been available on the PDP-11 computers since 1 July 1980. A single-user operating system, called Emono, has been written in Edison. It includes an Edison compiler, a PDP-11 assembler, a file system, a screen editor, and a variety of utility programs, all written in Edison. This report concludes with a brief overview of the Emono operating system and the Edison compiler.

0038–0644/81/040397–18$01.80 *Received 15 October 1980*

2. THE BACKUP PROGRAM

The purpose of the backup program is to write a copy of the entire contents of a disk onto magnetic tape. The disk is divided into 200 cylinders of 12K bytes each. The cylinders are to be written onto the tape as separate blocks starting at the beginning of the tape. The last block must be followed by an end of file mark on the tape.

The backup program has the form of a procedure declaration preceded by constant and type declarations

```
const nl = char(10); linelength = 132

array line [1:linelength] (char)

proc backup(
  proc writetext(text: line))

  ...
```

The program uses a procedure writetext to report disk or tape failure to the operator. This procedure is supplied by the operating system that invokes the execution of the backup program. The declarations which precede the program heading describe the parameter type of the procedure writetext as well as the control character new line (nl).

The body of the program (represented by three dots above) consists of a type declaration of disk cylinders, a disk module, a tape module, and a statement part describing the backup operation. A cylinder is declared as an array of 6K words

```
const cylinder_length = 6144 "words"

array cylinder [1:cylinder_length] (int)
```

The disk transfers are controlled by a module which exports two procedures called read disk and more disk (Algorithm 1).

When the procedure read disk is called repeatedly it returns the values of successive disk cylinders. The current cylinder number is given by a local variable which is automatically initialized to zero by the module.

The boolean function more disk compares the current cylinder number to the disk size to determine whether more cylinders remain to be read from the disk.

The disk is controlled by four device registers. The byte addresses of these registers are given by octal numerals named status, count, address, and search. The following takes place when a cylinder value is read from the disk and assigned to a variable named block:

1. The store address of the variable is placed in the address register of the disk. The store address is computed by calling a standard function named addr.
2. The (negative) value of the cylinder length (in words) is placed in the count register of the disk.
3. The cylinder number (multiplied by a constant) is placed in the search register of the disk.
4. A read command is placed in the status register of the disk.
5. The process that initiates the disk transfer is delayed until the status register shows that the disk is ready to perform another operation.
6. If the status register indicates that an error was detected during the disk transfer then an error message is displayed for the operator and the program execution halts after resetting the disk (to remove the error indication).

```
module "disk"
  const status = #177404; count = #177406;
    address = #177410; search = #177412; reset = #1;
    read = #5; ready = #200; error = #100000;
    disk_size = 200 "cylinders"

  var cylinder_no: int

  proc disk_error
  begin writetext(line('disk error', nl, '#'));
    place(status, reset);
    when sense(status, ready) do skip end;
    halt
  end

  *proc more_disk: bool
  begin val more_disk : = cylinder_no < disk_size end

  *proc read_disk(var block: cylinder)
  begin place(address, addr(block));
    place(count, -cylinder_length);
    place(search, cylinder_no * 32);
    place(status, read);
    when sense(status, ready) do skip end;
    if sense(status, error) do disk_error end;
    cylinder_no : = cylinder_no+1
  end

begin cylinder_no : = 0 end
```

Algorithm 1

7. If the disk transfer succeeded the cylinder number is increased by one.

The disk module describes all the possible disk operations performed by the backup program. It illustrates the use of the standard procedures addr, place and sense to control a non-trivial device directly in the Edison-11 language.

Since there is no attempt to schedule the use of the disk among competing processes it is assumed that the disk module will be used only by a single process.

The Edison-11 implementation uses neither clock interrupts nor peripheral interrupts to control processor multiplexing. Concurrent processes are executed one at a time in cyclical order. Each process runs until it either terminates or delays itself by means of a when statement. The disk module illustrates the use of a when statement to delay a process until an input/output operation has been completed.

The magnetic tape is controlled by another module (Algorithm 2), which exports procedures for rewinding the tape and for writing a cylinder value or a file mark on it.

The tape is controlled by four device registers with addresses named status, command, count and address. Input/output commands are represented by constants named rewind, write, and write eof. The relevant states of the tape unit are given by constants named online, ready, error and eof.

The initial operation of the tape module checks that the power of the tape unit is turned on, and that a tape reel is mounted and ready to be used by the computer (as indicated by the online state).

Although the details of the disk and tape procedures differ the programming techniques used are very similar for the two devices.

```
module "tape"

  const status = #172520; command = #172522;
    count = #172524; address = #172526; rewind = #60017;
    write = #60005; write_eof = #60007; online = #100;
    ready = #200; error = #100000; eof = #40000

  proc tape_error
  begin writetext(line('tape error', nl, '#')); halt end

  *proc rewind_tape
  begin place(command, rewind);
    when sense(command, ready) do skip end
  end

  *proc write_tape(var block: cylinder)
  begin place(address, addr(block));
    place(count, -2 * cylinder_length);
    place(command, write);
    when sense(command, ready) do skip end;
    if sense(command, error) do tape_error end
  end

  *proc mark_tape
  begin place(command, write_eof);
    when sense(command, ready) do skip end;
    if not sense(status, eof) do tape_error end
  end

begin if not sense(status, online) do tape_error end end
```

Algorithm 2

When the backup program is executed the initial statements of the modules are executed one at a time in the order written. Following this the statement part of the program is executed. It uses a variable x to hold a pair of cylinder values and uses another variable i as an index to select one of the two cylinder values.

```
array cylinder_set [false:true] (cylinder)

var x: cylinder_set; i: bool
```

The statement part itself is shown below:

```
begin
  rewind_tape; i : = false; read_disk(x[i]);
  while more_disk do
    i : = not i;
    cobegin 1 do write_tape(x[not i])
    also 2 do read_disk(x[i]) end
  end;
  write_tape(x[i]); mark_tape; rewind_tape
end
```

This algorithm is initially executed as a single, sequential process which rewinds the tape and reads the first cylinder value from the disk.

If there are more cylinder values to be read from the disk the execution continues as two concurrent processes: while one process writes the previous cylinder value onto the tape another process reads the next cylinder value from the disk. When both data transfers have been completed successfully the two processes terminate and the execution continues as a single process which repeats the above if necessary.

When the last cylinder value has been read from the disk, while at the same time the second last cylinder value has been written on the tape, the loop terminates.

The execution now becomes sequential again and the last cylinder value is written on the tape. Following this the tape is marked with an end of file mark and is then rewound.

The reasoning about this concurrent program is quite simple because the two processes always operate on disjoint, indexed variables x[i] and x[**not** i]. These variables are disjoint because their indices i and **not** i only can assume 'opposite' values (false and true).

Since the indices are changed only when the execution is purely sequential, no synchronization of the processes is needed during the execution of the concurrent statement. The simplicity then is achieved by using a concurrent statement within a while statement so that the processes are recreated for every cylinder transfer.

For a PDP-11/55 minicomputer with a mixture of bipolar store and core store the creation and termination of two processes takes only about 0.2 ms. When this is repeated 200 times the overhead of process creation is still only 40 ms for the whole backup operation. So it seems that the 'simplistic' form of concurrency used in Edison makes the underlying implementation so trivial that an 'unrealistic' style of programming now becomes quite practical in some cases.

The backup program copies the whole disk to tape in 74 s (not including the final rewinding of the tape). This corresponds to a transfer rate of 33200 bytes/s which is 92 per cent of the top speed of the slowest device (the tape unit). If the program is rewritten to run completely sequentially the execution time is increased by 29 per cent to 96 s.

This program shows how Edison can be used to implement the basic input/output operations of an operating system.

3. THE TYPING PROGRAM

The purpose of the typing program is to measure the typing speed of a person. The timing begins when the user types the first character. The characters typed at the keyboard are shown on a display. When the user types the character # the typing speed (in words/min) is displayed and the program terminates. (A word is defined as an uninterrupted sequence of letters typed).

The typing program is shown in its entirety (Algorithm 3). It uses three peripheral devices: a display, a keyboard, and a line frequency clock. Each device is controlled by a separate module.

The display module exports procedures for writing a single character, a textstring, and an integer value. The character new line (nl) is output as a carriage return (cr) followed by a line feed (lf).

The module uses a local procedure to display a single character. The character is placed in a buffer register when a status register indicates that the device is ready.

```
proc typing
const nl = char(10); cr = char(13); linelength = 132
array line [1:linelength] (char)
module "display"
  const status = #177564; buffer  = #177566;
    ready = #200; lf = nl
  proc display(c: char)
  begin
    when sense(status, ready) do
      place(buffer, int(c))
    end
  end

  *proc write(c: char)
  begin if c = lf do display(cr) end;
    display(c)
  end

  *proc writetext(text: line)
  var i: int; c: char
  begin i : = 1; c : = text[1];
    while c 〈 〉 '#' do
      write(c); i : = i+1; c : = text[i]
    end
  end
  *proc writeint(value: int)
  array numeral [1:5] (char)
  var no: numeral; n: int; more: bool
  begin "value 〉 = 0"
    n : = 0; more : = true;
    while more do
      n : = n+1;
      no[n] : = char(value mod 10+int('0'));
      value : = value div 10;
      more : = value 〉 0
    end;
    while n 〉 0 do
      write(no[n]); n : = n−1
    end
  end

begin skip end "display"
module "keyboard"
  const status = #177560; buffer = #177562;
    ready = #200; last = '#'
```

Algorithm 3

```
  set charset (char)

  var ch: char; letters: charset

  proc read
  begin
    when sense(status, ready) do
      obtain(buffer, ch:int)
    end;
    ch : = char(int(ch) mod 128);
    if ch = cr do ch : = nl end;
    write(ch)
  end

  *proc typist(var words: int)
  begin words : = 0;
    while ch in letters do
      read;
      while ch in letters do read end;
      words : = words+1
    else ch <> last do read end
  end

begin
  letters : = charset('ABCDEFGHIJKLMNOPQRSTUVWXYZ')
    +charset('abcdefghijklmnopqrstuvwxyz');
  writetext(line('start typing', nl, '#'));
  read
end "keyboard"

module "clock"

  const status = #177546; ready = #200;
    interval = 167 "units of 0.1 msec";
    second = 10000 "units"

  var running: bool

  *proc clock(var sec: int)
  var elapsed: int
  begin elapsed : = 0; sec : = 0;
    while running do
      when sense(status, ready) do
        place(status, 0)
      end;
      sec : = sec+(elapsed+interval) div second;
      elapsed : = (elapsed+interval) mod second
    end;
    sec : = sec+2 * elapsed div second
  end
```

Algorithm 3 (continued)

```
    *proc stop
    begin running : = false end

  begin running : = true; place(status, 0) end "clock"

  var words, seconds: int
  begin
    cobegin 1 do typist(words); stop
    also 2 do clock(seconds) end;
    if seconds > 0 do
      write(nl); writeint(60 * words div seconds);
      writetext(line(' words per minute', nl, '#'))
    end
  end "typing"
```

Algorithm 3 (continued)

The keyboard module exports a procedure which describes a typist process. This process reads one character at a time from the keyboard until the last character (#) has been typed and records the number of words typed.

The module uses a local procedure to read a single character from the keyboard. The character is obtained from a buffer register when the device is ready. For some keyboards the character value must be converted from teletype code to ASCII code by the modulo operation shown. A carriage return (cr) is converted to the character new line (nl). Finally, the character is echoed on the display by a write operation.

The initial operation of the module defines the set of letters of which words are composed, displays the message 'start typing' and waits until the user has typed the first character (which is input by a read operation). This delay prevents the clock module from being initialized until the typing begins.

The clock module exports a procedure which describes a clock process and another procedure which stops the clock process. The real-time is measured by means of a line frequency device which sets a bit in a status register every 16·7 ms. The bit is reset by the program when it has been recognized.

The initial operation of the module describes the clock as running and resets the device status to start the first clock cycle.

The clock process repeats a loop in which it waits until the status register has been changed before starting another clock cycle. The clock process measures the real-time in whole seconds using a parameter named sec and measures the current fraction of a second in units of 0·1 ms using a local variable named elapsed. When the variable named running assumes the value false the clock process rounds off the last fraction of a second and terminates after recording the total number of seconds elapsed.

When the procedure stop is called by another process the variable running is assigned the value false to stop the clock process.

The statement part of the program describes two concurrent processes: a typist process which keeps pace with the user, and a clock process which keeps track of time. When the typing stops the number of words typed is known and the clock is stopped. At this point the number of seconds used is also known. After termination of the processes the typing speed is displayed.

The backup program described earlier uses concurrency only to speed up a time-consuming operation limited by the computer peripherals. Since the typing process is limited by the user (and not by the computer) the use of concurrency does not speed things up in the typing program. But it serves as a convenient way of describing two independent activities (the typing and the progress of time) by separate program pieces instead of using a sequential program that alternates between operations on the keyboard and the clock.

The typing program demonstrates the additional freedom of design which concurrent programming can add to personal computing if it is supported by an abstract programming language.

4. THE PRINT PROGRAM

The purpose of the print program is to print the text of an Edison program and underline the word symbols of the language. The source text to be printed is a sequence of characters with the following structure

```
Source text:
  [ Source line ]* End medium character
Source line:
  [ Graphic character ]* New line character
```

A new line character (nl) marks the end of each source line. An end medium character (em) marks the end of the source text. All other control characters in the source text must be ignored by the print program.

The source text is a correct Edison program written in small letters without underlined word symbols. In the printed text all word symbols must be underlined. The text must be printed with at most 50 lines per page, and every page must begin with 2 blank lines.

The procedure which is used to read the source text character by character is a parameter of the print program

```
proc print(
  proc read(var ch: char))
```

The program heading is followed by declarations of control characters and two data types called line and character set

```
const lf = char(10); nl = lf; ff = char(12);
  cr = char(13); em = char(25); sp = ' ';
  linelength = 127

array line [1:linelength] (char)

set charset (char)
```

The program describes two concurrent processes called the scanner and the printer. The scanner inputs the source text and supplements it with underscores, while the printer breaks the text into pages and outputs it with underscoring. These processes exchange characters through a common buffer module. In addition, the scanner uses three modules: (1) A source module inputs the source text while skipping invisible characters; (2) A symbol module looks up words to determine whether they should be underlined or not; (3) An underscore module sends characters from the scanner through the buffer and supplements them with underscores.

The final text received by the printer process has the following structure

```
Final text:
  [ Final line ]* End medium character
Final line:
  [ Graphic character ]* Carriage return character
  [ Extra character ]* Line feed character
Extra character:
  '_' # Space
```

Each final line consists of the graphic characters of the corresponding source line followed by a carriage return character (cr). If the line includes one or more word symbols the carriage return is followed by extra characters, one for each graphic character of the line. The extra characters are underscores where the corresponding graphic characters should be underlined and are spaces where they should not be underlined. Each line ends with a line feed character (lf).

The buffer module exports procedures for sending and receiving a single character (Algorithm 4). The characters are stored in a cyclical buffer of type line. The current sequence of characters stored in the buffer is described by the indices of the first and last characters in the buffer and by the length of the sequence. The processes that use the buffer to communicate are synchronized by means of when statements which delay the sending (or receiving) of characters until the buffer is nonfull (or nonempty). Initially the buffer is empty.

```
module "buffer"

  var buffer: line; first, last, length: int

  *proc send(ch: char)
  begin
    when length < linelength do
      buffer[last] := ch;
      last := last mod linelength+1;
      length := length+1
    end
  end

  *proc receive(var ch: char)
  begin
    when length > 0 do
      ch := buffer[first];
      first := first mod linelength+1;
      length := length−1
    end
  end

begin first := 1; last := 1; length := 0 end
```

Algorithm 4

The source text module exports a procedure nextchar which inputs the next visible character of the source text (Algorithm 5). The invisible characters consist of all the control characters with the exception of new line and end medium.

The word symbol module exports a boolean function which determines whether or not a given word is a word symbol (Algorithm 6). This is done by a linear search in a

```
module "source text"

  const nul = char(0); us = char(31); del = char(127)

  var invisible: charset; x: char

  *proc nextchar(var ch: char)
  begin read(ch);
    while ch in invisible do read(ch) end
  end

begin invisible : = charset(del); x : = nul;
  while x < = us do
    invisible : = invisible + charset(x);
    x : = char(int(x) + 1)
  end;
  invisible : = invisible - charset(nl, em)
end
```

Algorithm 5

```
module "word symbols"

  const maxsym = 28
  array symboltable [1:maxsym] (line)
  var symbol: symboltable

  *proc word_symbol(word: line): bool
  var i, j: int
  begin i : = 1; j : = maxsym;
    while i < j do
      if word <> symbol[i] do i : = i+1
      else true do j : = i end
    end
    val word_symbol : = word = symbol[i]
  end

begin
  symbol : = symboltable(line('also'), line('and'),
    line('array'), line ('begin'), line ('cobegin'),
    line('const'), line('div'), line('do'), line ('else'),
    line('end'), line('enum'), line('if'), line('in'),
    line('lib'), line('mod'), line('module'),
    line('not'), line('or'), line('post'), line('pre'),
    line('proc'), line('record'), line('set'),
    line('skip'), line('val'), line('var'), line('when'),
    line('while'))
end
```

Algorithm 6

table that contains all the word symbols of the language. The table is initialized by evaluating a single constructor of type symbol table.

The underscore module exports four procedures named roman, italic, new line, and end medium (Algorithm 7). These procedures are used by the scanner to transmit the text one character at a time through the buffer. The module uses an integer variable to count the length of the current line and uses a set variable named underscore to remember the positions of the characters (if any) that should be underlined on that line. Initially, the first line is of length zero and the set of characters to be underlined is empty.

The roman operation sends a given character through the buffer and increases the line length by one. The character will not be underlined.

The italic operation sends a given character through the buffer and increases the line length by one. The character is also included in the set of characters to be underlined.

```
module "underscore"

  set intset (int)
  var underscore: intset; length: int

  *proc roman(ch: char)
  begin send(ch); length : = length+1 end

  *proc italic(ch: char)
  begin roman(ch);
    underscore : = underscore+intset(length)
  end

  *proc newline
  var i: int
  begin send(cr);
    if underscore <> intset do
      i : = 0;
      while i < length do
        i : = i+1;
        if i in underscore do send('_')
        else true do send(sp) end
      end;
      underscore : = intset
    end;
    send(lf); length : = 0
  end

  *proc endmedium
  begin send(em) end

begin underscore : = intset; length : = 0 end
```

Algorithm 7

The new line operation sends a carriage return through the buffer and examines the underscore set. If it is nonempty an extra character is transmitted for each character position of the line. The extra character is an underscore, if the position is included in the set, and is a space, if it is not in the set. Finally, a line feed character is transmitted and the length of the next line is set to zero with an empty set of underscores.

The end medium operation sends an end medium character through the buffer.

The scanner process is described by a procedure which examines the source text character by character. (Algorithm 8). A letter is the beginning of a word which is assembled in a local variable of type line. When the word is complete it is looked up to determine whether it is a word symbol in the Edison language. In that case, it is sent to the printer as a sequence of italic characters; otherwise, it is transmitted as roman characters. Any other graphic characters are copied directly as roman characters. The control characters new line and end medium are treated as described earlier.

```
proc scanner
var ch: char; letters, alphanum: charset;
  word: line; length, i: int; mark: bool
begin
  letters : = charset('abcdefghijklmnopqrstuvwxyz');
  alphanum : = letters + charset ('0123456789_');
  nextchar(ch);
  while ch in letters do
    word : = line(sp); length : = 0;
    while ch in alphanum do
      length : = length + 1; word[length] : = ch; nextchar(ch)
    end;
    mark : = word_symbol(word); i : = 0;
    while i < length do
      i : = i + 1;
      if mark do italic(word[i])
      else true do roman(word[i]) end
    end
  else ch = nl do newline; nextchar(ch)
  else ch <> em do roman(ch); nextchar(ch) end;
  endmedium
end
```

Algorithm 8

A printer module exports a single procedure which describes the printer process (Algorithm 9). It uses a local procedure to write a single character on a line printer. The printer process counts the number of lines printed in a local variable. At the beginning and after every 50 lines printed it outputs a form feed (ff) followed by two new line characters (nl). It then receives lines character by character from the buffer and prints them. This continues until an end medium character (em) has been received.

```
module "printer"
  const status = #177514; buffer = #177516; ready = #200

  proc write(ch: char)
  begin
    when sense(status, ready) do
      place(buffer, int(ch))
    end
  end

  *proc printer
  const pagelimit = 50 "lines"
  var lineno: int; ch: char
  begin lineno := 1; receive(ch);
    while ch <> em do
      if lineno mod pagelimit = 1 do
        write(ff); write(nl); write(nl)
      end;
      while ch <> lf do
        write(ch); receive(ch)
      end;
      write(ch); receive(ch);
      lineno := lineno+1
    end
  end

begin skip end
```

Algorithm 9

The statement part of the print program consists of a concurrent statement which describes the scanning and printing as simultaneous operations

```
begin
  cobegin 1 do scanner
  also 2 do printer end
end
```

In its present form, the print program underlines word symbols wherever they occur in the source text. By extending the scanner with more cases one can prevent underlining within comments and character strings.

This program demonstrates the use of message buffers to smooth temporary fluctuations of processing speeds. It also illustrates in a simplified form well-known programming techniques which are used in compilers to recognize and classify text symbols.

5. THE EMONO SYSTEM

The Emono system enables a single user to develop and execute Edison programs on a PDP-11 computer with 28K words of store, a disk of 1M words, a display terminal, a magnetic tape unit, and a line printer.

The system consists of an operating system and a set of standard programs, all written in Edison. Emono was derived from a Concurrent Pascal system called Mono by rewriting all the programs in Edison.[2] By adopting the same filing system as Mono it was possible to develop Emono on a Mono disk using a preliminary Edison compiler written in Sequential Pascal. The Emono system has been in operation since 1 July 1980.

The following is a brief overview of the system to illustrate the capabilities of the Edison language for non-trivial software development.

The programs of a user are stored as text and code files on a removable disk pack. The standard files are described in a single system catalog, while those files that are still being developed are described in one or more user catalogs.

At the beginning of a session, the user mounts his own disk pack and starts the Emono system. The user then types a command that gives him access to the files described in a given user catalog as well as those described in the system catalog. The user can now input, compile, edit, and test Edison programs. When a program is finished, the user can move its description from the given user catalog to any other catalog (including the system catalog).

The user can make copies of text files on the line printer. But the system makes the display and editing at the terminal so convenient that the need for printed listings is reduced considerably.

It is possible to copy the files of a single catalog (or the whole disk) onto magnetic tape and use it to reestablish disk files after hardware or software failure.

To begin with the operating system executes a cyclical Edison program called do which accepts commands from the terminal. Each command specifies the execution of an Edison program with a set of arguments. The given program is loaded from the disk and executed as a library procedure called by the do program. Any Edison program can call any other Edison program stored on the disk by means of a procedure called run implemented by the operating system.

The interface between the Emono system and any of its user programs is a set of procedures that are implemented by the operating system and are passed as arguments to the user programs before they are executed. These procedures give each user program simultaneous access to at most four sequential files. Other procedures enable programs to use the terminal, to create and delete files on the disk, and to call other programs stored on the disk.

The Emono system uses the same structure of catalogs and files as the Mono system from which it is derived. To avoid occasional, but time consuming, relocation of data on the disk, the pages allocated to a single file are addressed indirectly through a page map—a single disk page which defines the addresses of the data pages of the file. The page map allows the operating system to place the data pages anywhere on the disk and let them remain there until the file is deleted.

The system catalog is a file that starts at a fixed disk address and which describes the name and page map address of each standard file (including itself). Each user catalog is described as a file in the system catalog. At the beginning of a terminal session, all file names are looked up in the system catalog. The user can now select a given user catalog by name. Following this, all file names are first looked up in the given user catalog, and (if that fails) they are then looked up in the system catalog. A file is opened by looking it up in the current set of catalogs used and bringing its page map into the main store.

Since each user has his own removable disk pack, files need only be protected against accidental overwriting and deletion. All files are initially unprotected. The user protects a file by calling a standard program that sets the protection attribute of the file to true in the catalog that describes it.

The Emono operating system itself and most of its standard programs are sequential Edison programs. Some of the standard programs and user programs do, however, include concurrent statements. The usefulness of concurrency in application programs has already been demonstrated by the three sample programs.

The Emono system consists of about 30 Edison programs of the following lengths:

Emono program	900 lines
Do program	300 lines
File program	700 lines
Catalog programs	3000 lines
Device programs	1100 lines
Edit program	1000 lines
Edison compiler	4300 lines
Assembler	1900 lines
Other programs	1100 lines
Total system	14300 lines

By comparison the Mono system written in Concurrent and Sequential Pascal consists of 25600 lines of program text.

The Emono system requires a main store of 28K words which is used as follows

Edison kernel	2K words
Operating system	8K words
User programs and variables	18K words
Main store	28K words

With this amount of store the Edison compiler can recompile its largest pass and still leave 2K words unused. The Mono system written in Concurrent and Sequential Pascal requires 34K words of store.

The system kernel is the only program written in assembler language. Its function is to load and execute abstract Edison code. It occupies 2K words of store

Operator communication	400 words
Program loading	500 words
Code interpreter	1100 words
Edison kernel	2000 words

I developed the Edison compiler, the assembler, and the kernel over a period of 6 months. The rest of the Mono programs were rewritten in Edison by Habib Maghami and Peter Lyngbaek in two months.

It would be a simple task to transfer the Emono system to an LSI microcomputer with a dual floppy disk. However, since Emono was copied directly from a system that was originally written in Concurrent Pascal it does not fully take advantage of the greater flexibility of the Edison language. It would also be cumbersome to move the Emono system to other microcomputers since the Edison programs include machine-dependent input/output operations. The main purpose of the Emono system is to serve as an early program development system for future Edison systems.

6. THE EDISON COMPILER

The Edison compiler is the largest program written so far in Edison. The following is a brief overview of the compiler which will be described in detail elsewhere.

The compiler is divided into four passes, which are called one at a time by a control program named Edison. In general, each pass makes a single scan of the program text and outputs intermediate code on the disk. This becomes the input to the next pass.

The function of the passes is listed below.

Pass 1: Symbol analysis
Pass 2: Syntax and scope analysis
Pass 3: Syntax and semantic analysis
Pass 4: Code generation

Pass 1 scans the program text character by character and converts symbols to integer values. This pass does not distinguish between different uses of the same name in different blocks.

Pass 2 checks the program text by means of recursive descent using one procedure for each syntactic form of the language and skipping over invalid sentences. Different uses of the same name in different blocks are replaced by unique name indices. Apart from this, scope analysis is only concerned about whether a name can be used in a given context, but does not worry about what kind of entity it refers to.

Pass 3 checks that operands and operators are compatible and computes the lengths of types and the addresses of variables. To make this (the most complicated pass) understandable, the method of recursive descent is again used to recognize syntactic forms.

Pass 4 scans its input twice and builds a table of program labels and procedure stack requirements during the first scan. During the second scan it outputs abstract code in which program labels are replaced by relative addresses.

The compiler was first written in Edison to make sure that the language was convenient for compiler implementation. The compiler was then rewritten in Sequential Pascal using a programming style corresponding to the available data types and statements of Edison. Finally, the compiler was completely rewritten in Edison and compiled by means of the Pascal version of the compiler.

The final compiler consists of 4300 lines of Edison text

Edison	500 lines
Pass 1	300 lines
Pass 2	1000 lines
Pass 3	1500 lines
Pass 4	1000 lines
Compiler	4300 lines

The Edison compiler is only half the size of the Concurrent Pascal compiler, and compiles a language that is more powerful than the combination of Sequential and Concurrent Pascal.

On a PDP-11/55 minicomputer, the Edison compiler uses a storage area of 16K words to recompile its largest pass. After an initial time of 9 s it compiles about 14 lines/s. The compilation of pass 3 (1500 lines) takes about 2 min. The compiler generates about 4 words of abstract code per line of program text. The size of the Edison code is about 75 per cent of the corresponding code for Sequential Pascal.

7. EXECUTION TIMES

The execution times of Edison-11 programs shown below are measured in units known as average operation times. For a PDP-11/55 computer with 16K words of bipolar store and 12K words of core store the average operation time is about 6 μs. For the LSI-11 microcomputer it is approximately 30 μs.

	elementary operands	set operands (n members)	record or array operands (n words)
constant c	1	$4+2\ n$	$1+0{\cdot}4\ n$
whole variable v	1	4	$2+0{\cdot}4\ n$
:=	1	4	$1+0{\cdot}4\ n$
= <>	2	5	$1+0{\cdot}4\ n$
< <= > >=	1		
in		3	
and or not	1		
+ −	1	4	
*	2	6	
div mod	3		

Other execution times are shown below:

field variable v.f	v+2
indexed variable v[e]	v+e+4
procedure call (no parameters)	7
if B **do** S **end**	B+S+1
while B **do** S **end**	(B+S+2) n
when B **do** S **end**	(B+6) n+S
cobegin 1 **do** S1 ... **also** n **do** Sn **end**	S1+...+Sn +8+13 n

A conditional statement of the form

v = c **do** S

where v is a whole variable and c is a constant of the same elementary type has the execution time S+1.

REFERENCES

1. P. Brinch Hansen, 'Edison—a multiprocessor language', *Software—Practice and Experience,* **11**, 325–361 (1981).
2. P. Brinch Hansen and J. Fellows, 'The Trio operating system', *Software—Practice and Experience,* **10**, 943–948 (1980).