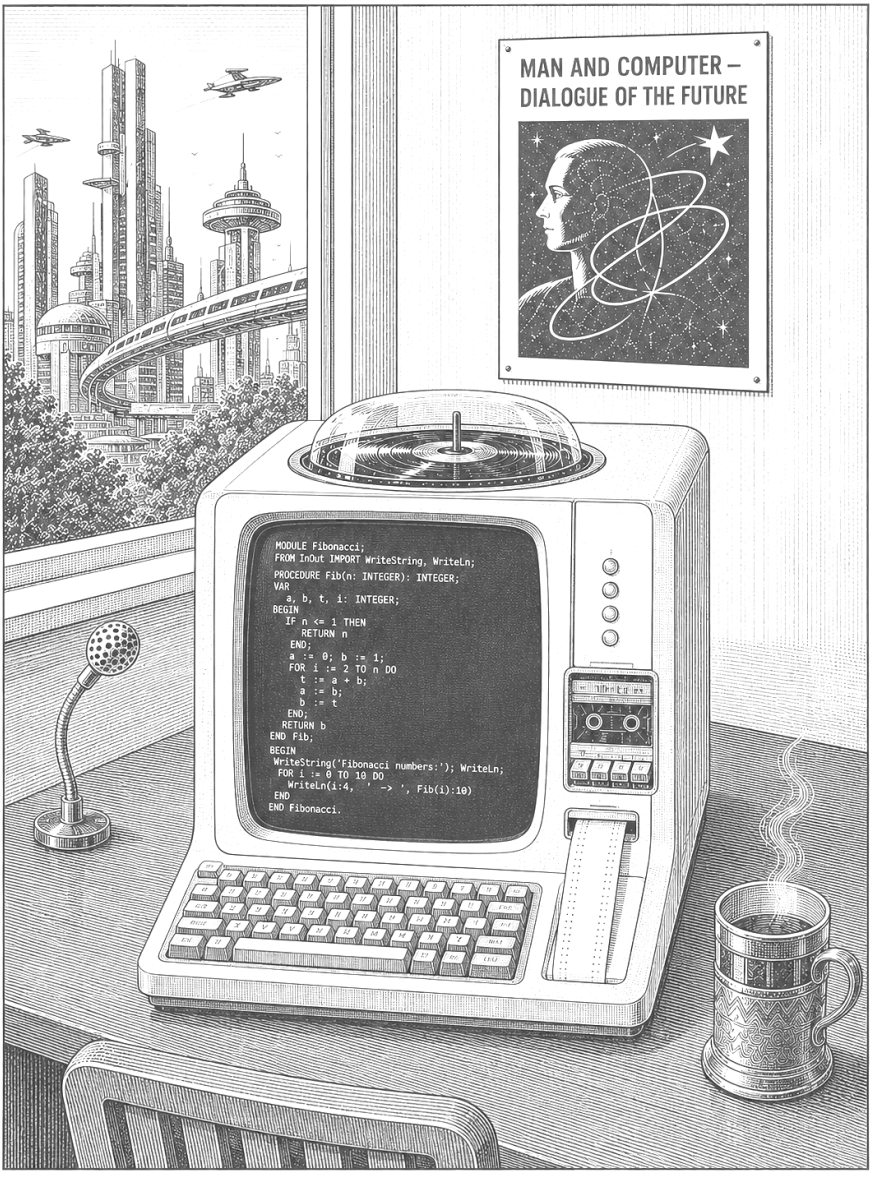
MAN AND COMPUTER –
DIALOGUE OF THE FUTURE
MODULE Fibonacci;
FROM InOut IMPORT WriteString, WriteLn;
PROCEDURE Fib(n: INTEGER): INTEGER;
VAR
a, b, t, i: INTEGER;
BEGIN
IF n <= 1 THEN
RETURN n
END;
a := 0; b := 1;
FOR i := 2 TO n DO
t := a + b;
a := b;
b := t
END;
RETURN b
END Fib;
BEGIN
WriteString('Fibonacci numbers:'); WriteLn;
FOR i := 0 TO 10 DO
WriteLn(i:4, ' -> ', Fib(i):10)
END
END Fibonacci.

questions of cybernetics.

issue 001

contents

beginnings

ortfero

the present issue opens "questions of cybernetics", the bulletin of a temporary labor collective formed for the development of a general-purpose computer, the aldan-4. the work is conducted in deliberately limited scope. it addresses a circumstance now plainly visible: the systems on which contemporary computing rests have grown to a complexity that exceeds the grasp of any individual engineer.

the measure we have chosen is surveyability. complexity has silted up layer on layer, and cannot be reduced by further addition; surveyability is approached only by fresh construction, with the benefit of accumulated experience.

over several months the collective has designed the machine, written its emulator, and built upon it the operating system aldos, the procedural language almac, and the tooling that surrounds them. as this issue goes out, the system has reached working self-sufficiency in its local configuration: on the machine itself the full cycle of work is now possible with no recourse to any external means. it was this that moved us to begin publishing.

the bulletin will report the collective's work systematically; it will set out our design decisions and it will take up questions of a general theoretical character bearing on the system. we hope it becomes a place of useful exchange.

on the risc-v processor

pin47

abstract

the architecture of the 64-bit processor risc-v is considered. its base instruction set — the integer core with multiply-and-divide — is set out in summary tables and illustrated by a euclid subroutine.

statement of the question

the choice of a base instruction-set architecture for the project is determined by three considerations: the regularity and small extent of the system of instructions, the openness of the specification, and the absence of accumulated historical compatibility.

below, the risc-v architecture is considered.

organization of the risc-v processor (rv64)

the base (integer) architecture rv64i has the following features.

- word length. the word length of data and of the address is 64 bits (xlen = 64). operations on the byte (8 bits), the half-word (16), the word (32) and the double word (64) are supported.
- register file. the processor contains 32 general-purpose registers of 64 bits each, designated

x0...x31. register x0 is fixed at zero: reading yields zero, writing is ignored. this lets several operations (transfer, comparison with zero, unconditional transfer of control) be expressed uniformly without separate instructions. the address of the next instruction is held in a separate program counter, not counted among the general-purpose registers.

in the hardware, registers x1...x31 are interchangeable; the control unit assigns them no special purpose. their roles are fixed not by the hardware but by an adopted calling convention, common to system and applied software. observance of this convention ensures the interoperability of subroutines and the portability of programs between samples of the computer. the standard mnemonics and roles are given in table 1.

table 1. convention on the use of the general-purpose registers

hardware	**mnemonic**	**purpose**	**preserved across a call**
x0	zero	constant zero	—
x1	ra	subroutine return address	no (caller's concern)
x2	sp	stack pointer	yes (callee's concern)
x3	gp	pointer to the global data area	— (unchanged)

hardware	mnemonic	purpose	preserved across a call
x4	tp	pointer to task (thread) data	— (unchanged)
x5	t0	temporary (also auxiliary link address)	no
x6-x7	t1-t2.	temporary registers	no
x8	s0/fp	saved; stack frame pointer.	yes
x9	s1	saved register	yes
x10-x11	a0-a1	subroutine arguments and returned values	no
x12-x17	a2-a7	subroutine arguments	no
x18-x27	s2-s11	saved registers	yes
x28-x31	t3-t6	temporary registers	no

— zero (x0) is the source of constant zero and the receiver of discarded results. ra (x1) holds the return address, written by the transfer-with-return instructions. sp (x2) addresses the top of the stack, on which lie the local data and saved registers of subroutines; fp (x8), where required, marks the start of the current stack frame. gp and tp (x3, x4) point to global program data and to task

(thread) data; they are not altered during execution.

- a0...a7 (x10...x17) pass arguments to a subroutine; the result is returned in a0, a1. s0...s11 are saved registers: a subroutine that uses them must restore the prior contents before returning, so the caller may rely on their invariance. t0...t6 are temporary: not preserved across a call, the caller must save them where needed. the division into saved and temporary registers reduces the number of memory accesses on subroutine calls.
- the "load-store" principle. the architecture is of the "load-store" type: memory is accessed only by load and store instructions; all arithmetic and logical instructions operate on register contents. this gives a regular pipeline and a small number of instruction formats.
- instruction format. the instruction length is fixed at 32 bits (4 bytes). a small number of encoding formats differ only in the placement of register numbers and of the immediate constant: register-register (r), immediate (i), store (s), conditional branch (b), upper-bits immediate (u), unconditional transfer (j). byte order in memory is little-endian.
- execution modes. machine, supervisor and user modes are provided. this separates system and applied software and supports protection under multitask operation.
- extensibility.the integer core (i) may be supplemented by standard extensions: integer multiply and divide (m), atomic operations (a),

floating-point (f, d), and others. for the present project, the core with the multiply-and-divide extension — rv64im — is sufficient.

base instruction set (rv64im)

the base instruction set is orthogonal: operations, operands, and addressing modes combine uniformly. the principal classes are set out below.

- arithmetic-logical register-register instructions. the result of an operation on two general-purpose registers is placed in a third: addition add, subtraction sub, bitwise "and" and, "or" or, "exclusive or" xor, the logical and arithmetic shifts sll/srl/sra, comparisons with the setting of a flag slt/sltu.
- instructions with an immediate operand. the same operations with one operand as a constant encoded in the instruction: addi, andi, ori, xori, slti/sltiu, shifts by a constant slli/srli/srai.
- instructions for the formation of the upper bits. lui places a 20-bit constant into the upper bits of a register; auipc forms an address relative to the program counter. together they provide loading of any 64-bit constant and position-independent addressing.
- "word" instructions. for 32-bit operations there are instructions with the suffix "w" (addw, subw, sllw, srlw, sraw, addiw, and so on), operating on the lower 32 bits with sign-extension of the result to 64 bits.

- instructions of memory access. loading from memory into a register: lb, lh, lw, ld (with sign-extension), lbu, lhu, lwu (with zero-extension); storing from a register into memory: sb, sh, sw, sd.
- instructions of the transfer of control. the unconditional transfer with preservation of the return address jal and the transfer by register jalr; conditional branches on the comparison of two registers: beq, bne, blt, bge, bltu, bgeu.
- system and synchronizing instructions. the supervisor call ecall, the breakpoint ebreak, the ordering of memory accesses fence.
- the multiplication and division extension (m). multiplication mul (the lower bits of the product), mulh/mulhsu/mulhu (the upper bits, for signed and unsigned operands), division div/divu, remainder rem/remu, and their "word" forms mulw, divw, divuw, remw, remuw.

in the summary below, the operand composition is indicated for each instruction in place of the formal class. notation: rd — destination register; rs1, rs2 — source registers; imm — immediate operand (a constant encoded in the instruction); shamt — shift amount; off(rs1) — a memory address formed as rs1 + off; label — target address of a transfer.

table 2. summary of the base instructions of rv64im (mnemonic — operands — purpose)

mnemonic	operands	purpose
add, sub, and, or, xor, sll, srl, sra, slt, sltu	rd, rs1, rs2	register-register arithmetic, logic, shifts, comparison
addw, subw, sllw, srlw, sraw	rd, rs1, rs2	the same upon the lower 32 bits ("word" operations)
mul, mulh, mulhsu, mulhu, div, divu, rem, remu	rd, rs1, rs2	multiplication, division, remainder (m extension)
addi, andi, ori, xori, slti, sltiu	rd, rs1, imm	the same operations with an immediate operand
slli, srli, srai	rd, rs1, shamt	shift by a constant amount
lui, auipc	rd, imm	formation of the upper bits of a constant / of an address
lb, lh, lw, ld, lbu, lhu, lwu	rd, off(rs1)	loading from memory into a register
sb, sh, sw, sd	rs2, off(rs1)	storing from a register into memory
beq, bne, blt, bge, bltu, bgeu	rs1, rs2, label	conditional branch upon a comparison of registers

mnemonic	operands	purpose
jal	rd, label	transfer keeping the return address
jalr	rd, off(rs1)	transfer by register keeping the return address
ecall, ebreak	(none)	call to the supervisor; breakpoint
fence	pred, succ	ordering of the preceding and succeeding memory accesses

the register x0, fixed at zero, lets a number of common operations be expressed without a separate opcode: register transfer (addi with a zero constant), sign inversion (sub from x0), unconditional transfer (jal writing the address into x0), and so on. this economises the control unit while keeping the instruction set complete.

example of a program

as an example, consider a subroutine computing the greatest common divisor (gcd) of two non-negative integers by euclid's algorithm. it uses the conditional branch, the remainder instruction of the m extension, register transfers, the transfer-with-return instructions, and the calling convention (arguments in a0, a1; result in a0).

the algorithm replaces the larger number by the remainder of the division until the second is zero; the gcd is the last non-zero value. the listing:

```
-- subroutine gcd (greatest common divisor)
-- of two non-negative integers by the algorithm of euclid.
-- entry:  a0 = m, a1 = n
-- exit:   a0 = gcd(m, n)
-- working register: t0
fn gcd s64 (m s64, n s64) asm:
0:  beq   a1, zero, 1f   -- if n = 0 then return
    remu  t0, a0, a1     -- t0 := remainder of m / n
    addi  a0, a1, 0      -- m := n
    addi  a1, t0, 0      -- n := t0
    jal   zero, 0b       -- repeat the cycle
1:  ;
```

the subroutine uses only the base core and the m extension. the register zero (x0) lets several common actions be expressed without separate opcodes: register transfer by addi rd, rs, 0, unconditional transfer by jal zero, gcd (the return address discarded into zero), subroutine return by jalr zero, 0(ra). in practice, the pseudo-instructions mv, j, ret are accepted for these forms and translated by the assembler into the corresponding base instructions.

as an illustration, consider gcd(48, 36). the sequence of states of a0, a1 is given in table 3.

table 3. course of the computation of gcd(48, 36)

step	a0 (m)	a1 (n)	remainder t0 = m mod n
initial	48	36	—
1	36	12	12
2	12	0	0

step	a0 (m)	a1 (n)	remainder t0 = m mod n
return	12	—	—

on exit, a0 contains 12, the gcd. the subroutine accesses no memory and uses no saved registers, and may therefore be called from any context with no overhead for state preservation.

substantiation of the choice

taken together, the properties set out above make rv64im a rational base for the central processor of the project.

- regularity and small extent. the small number of instructions and the uniformity of their encoding give a compact, regular control unit and a system that one developer can wholly grasp.
- predictability of execution. most instructions execute in a fixed number of clock cycles; there are no complex microprogrammed operations. timings are predictable, and the analysis and verification of programs are simpler.
- rv64i/m is an open standard, not controlled by any one company. the project is therefore not tied to the policy of a single architecture owner.
- portability of the software. the unified 64-bit architecture, with direct addressing of large memory volumes, allows programs to be developed in a single high-level language and translated without binding to the features of a particular machine sample.

— feasibility of emulation. the small size of the base instruction set allows full functional emulation in software by the present collective. this supports both program development before the hardware is available and in-service verification of the equipment.

conclusion

the 64-bit reduced-instruction-set processor rv64im — by regularity, predictability, openness, and portability — answers the requirements for the base architecture of an open computing complex. the instruction set, small as it is, is functionally complete for the principal classes of computing tasks.

aldos from a bird's eye view

ortfero

abstract

the operating system aldos for the aldan-4 machine is considered. its composition is set out — the kernel, the operating-system interface, the standing supplies, the user applications and the procedural language almac — together with the sequence of the bringing-up of the machine.

composition of the system

the system is divided into five layers, distinguishable by function and by the address at which each resides in operative memory.

the kernel is the lowest layer. resident at a fixed kernel address in operative memory, it brings up the machine from the moment of the transfer of control from the firmware, registers the operating-system interface, initialises the task manager, prepares the first task (the boot task), and starts the cooperative scheduler.

the operating-system interface is a single table of named operations covering memory, libraries, processes, files, devices and logging, through which every resident service and every native program addresses the kernel. it is published at a fixed memory slot. the standing libraries are loaded by the init manager from `/a/init/a`. three are shipped at present; any further

library placed in that directory is loaded the same way:

- opcon — the operator console: drives the text framebuffer, the keyboard and the cursor;
- diskman — the volume and filesystem library: path handling, directory limits, and the mounting of drives `a`..`z`;
- bagpipe — the sound library: playback of pcm samples through two alternating buffers.

almac, the procedural language and its environment, is launched by the init manager as a native executable; from that moment it is the principal interface to the machine. at its prompt programs are composed, compiled and executed; modules saved on the boot volume can be imported in any later session. almac programs are not processes of the system — they are modules of the almac environment.

three applications are shipped with the present issue — the text editor `jot`, the file pager `gaze` and the game `snake`. all are written in almac as modules of this kind, residing under `/a/tools/` and `/a/games/` on the boot volume, and are imported at the start of every session by the operator's standing start-up script. once imported, they are invoked from the almac prompt by calling their entry functions:

```
jot.in '/a/codex/markoff.moff    -- open the file in the text editor
gaze.at '/a/codex/markoff.moff   -- page through the file
snake.play                       -- play the game
```

imported modules share the address space, libraries and standing supplies of the almac session.

the boot sequence: from power-on to the almac prompt

the machine is brought up from power-on to the almac prompt in seven stages, each holdable in mind. the addresses below are those of the fixed memory map of the aldan-4.

- power-on. the processor begins execution at the reset vector at `0x00000000`, where the firmware `bootmgr` resides. it identifies itself by an eight-byte header (version, edition, date). machine-mode supervisor calls (`ecall`) are serviced by opensbi, included in the firmware.
- the firmware brings the machine to a usable state. it sets up a 16-kilobyte system stack and initialises the video device, the framebuffer and the trap table. the sub-sequence is given below.
- the firmware loads the kernel from disk. it reads the root directory of the boot volume, finds the kernel image by name, reads its inode and body into `0x00040000` (a 128-kilobyte region), and transfers execution there. on failure, a diagnostic is written directly into the framebuffer and the processor halts.
- the kernel establishes its runtime. on receiving control it zeroes its working data and runs the static initialisers of its units in turn.
- the kernel brings up the subsystems, publishes the operating-system interface, prepares the first task and dispatches the scheduler. the sub-sequence is given below.

- the boot task mounts the volumes and launches the init manager. it starts the disk server, mounts the boot volume as drive `a`, optionally registers the host filesystem as drive `b` (a non-fatal step: its absence yields a warning, not a halt), and launches the init manager. a failure at any fatal step is signalled to the operator console as a panic, and the system is halted.
- the init manager loads the standing libraries and launches almac. it traverses the standing library directory of the boot volume, loads every standing library there, and launches the compiler. almac builds its working image, optionally runs the operator's start-up script, and enters the read-compile-execute cycle. the operator may terminate the emulator at any moment by pressing `f12`.

the bringing of the machine to a usable state by the firmware, given as one step above, proceeds in this sequence:

- initialisation of the video display processor: the base addresses of the framebuffer and the glyph atlas are written into the video device's registers;
- clearing of the framebuffer;
- installation of the vectored trap table and enabling of the machine-mode interrupts.

the bringing-up of the subsystems by the kernel, given as one step above, proceeds in this sequence:

- bringing-up of the video display processor, keyboard, disk and sound device;

- publication of the operating-system interface at the fixed slot `0x0003FFF8`;
- initialisation of the task manager;
- construction of the first task (the boot task);
- dispatching of the scheduler, which yields control to the boot task.

the fixed memory map of the aldan-4, which the operator does well to hold in mind through the whole bringing-up, is given below.

address	contents
`0x00000000`	firmware `bootmgr` (boot rom, opensbi handler)
`0x0003FFF8`	operating-system interface slot
`0x00040000`	kernel image (128 KB)
`0x00060000`	operating memory available to programs (1664 KB)

conclusion

the article has set out the composition of aldos — the kernel, the operating-system interface, the standing libraries of the init manager, the almac environment and the user applications — and the seven-stage bringing-up of the machine from power-on to the prompt of almac. this arrangement answers the principle of surveyability set out in the opening article.

benchmarking with eratosthenes' sieve

```
--
-- sieve -- cpu benchmark
-- usage:
--      sieve.run
-- authors:
--      ortfero <ortfero@gmail.com>, 2026
-- license: cc-by-nc-sa 4.0
--

import "clock", "task"

fault wrong_prime_count

-- sieve `numbers` in place and return the count of primes found.
fn find s64 (numbers& []bool)

-- benchmark `find` and return the maximum iterations-per-second.
fn run s64 () raises

implementation

const size         = 8190
const primes_count = 1899
const nof_samples  = 20
const window_ms    = 50

-- only odd numbers are tracked: flags[i] stands for
-- the odd number 2*i + 3.
var flags [size]bool

-- sieve `numbers` in place and return the count of primes found.
fn find s64 (numbers& []bool)
    i s64, count s64, k s64, prime s64, n s64:
    n = length numbers;
    -- reset the sieve
    i = 0; for i != n; i += 1:
        numbers[i] = true;;
    count = 0;
    i = 0; for i != n; i += 1:
```

```
        if not numbers[i] then
            continue;
        -- numbers[i] is prime p = 2*i + 3; cross out its
        -- odd multiples. the index stride is `prime` because
        -- each step advances 2*prime in value space, skipping
        -- the even multiples we never tracked.
        prime = i + i + 3;
        k = i + prime; for k < n; k += prime:
            numbers[k] = false;;
        count += 1;;
    count;;

-- benchmark `find` and return the maximum iterations-per-second.
fn run s64 () raises
    s s64, n s64, c s64,
    started clock.tick, actual_ms s64,
    ips s64, best_ips s64:
    c = find flags&;
    -- sanity check: `find` must agree with the known prime count.
    if c != primes_count then
        raise wrong_prime_count;
    best_ips = 0;
    s = 0; for s != nof_samples; s += 1:
        started = clock.now;
        -- count iterations
        n = 0; for (clock.elapsed_ms started) < window_ms; n += 1:
            c = find flags&;;
        actual_ms = clock.elapsed_ms started;
        ips = (n * 1000) / actual_ms;
        if ips > best_ips then
            best_ips = ips;
        task.yield;;
    best_ips;;
```

trifles

print current date

jukki

add to `/a/almac/rc.alm`:

```
fn date string ()
    buffer [32]u8:
    say (time.format buffer&, time.now);;
```

or

```
var ans [256]u8

fn date string ():
    say (time.format ans&, time.now);;
```

what a function equals

jukki

the first almac function i read looked unfinished — the word that announces the result was nowhere on the page. it was whole all the same, and i wondered how it managed without. so: how a function delivers its value, and why return comes up less often than you might expect.

the value of a function is its last expression

here is the main thing, and most of the rest follows from it. the body of a function is what sits between the colon and the closing `;`. the value of the function is the value of the last expression in its body. no keyword, no marker — it is enough that the right expression is evaluated last.

look at adding two integers:

```
fn add s64 (a s64, b s64):
    a + b;;
```

there is no `return` here, and still the function returns the sum. `a + b` is the last expression in the body (here it is also the only one), so its value is what goes back to the caller. if this feels strange at first, try reading the body as a claim: "compute `a + b` — that is the answer."

then the type is not arbitrary

if the last expression is the result, its type cannot be just anything: it must match the type you promised in the header. otherwise the promise is broken, and the translator says so at once.

a typical slip — a function declared to return a byte, ending in a string:

```
fn first_byte u8 (s string):
    s;;            -- the result came out string,
                   -- but u8 was promised
```

the program never runs; the mismatch is caught during translation. and this is good: the error is found early, while it is still cheap.

when `return` is needed

while the computation runs straight down, top to bottom, no special word is required. but sometimes you want to leave a function early — say, to throw out a degenerate case without going into the rest. that is what `return` is for: it stops the function immediately and hands back the value given with it.

```
fn list_head_value s64 (head ^node):
    if head == none then
        return 0;
    head^.value;;
```

if the reference is empty, `return 0` fires and that is the end of it. if not, control reaches `head^.value` quietly, and it comes back by the rule we already know.

so the two ways of returning live together and divide the work by meaning: you take `return` for an early exit, and you leave the ordinary "bottom" result as the last expression. writing `return` at the very end, where leaving the expression last would do, is superfluous.

one small thing to remember, since we will want it shortly: `return` must always be followed by an expression. a bare `return;`, with nothing after it, the language does not allow.

functions that return nothing

very many functions are called not for an answer but for an act — to print something, write somewhere, change some state. these have type `none`. moreover: if no return type is given in the header at all, it is taken to be `none`. so these two lines

```
fn greet none (who string): ...
fn greet      (who string): ...
```

mean the same thing, and in practice one writes the shorter second form.

for such functions the type rule can be relaxed. nothing leaves the function anyway, so the type of the last expression no longer matters: it can be of any type, and its value is simply discarded.

```
fn greet (who string)
    n s64:
    n = length who;
    say "hello, ", who;;
```

here the last expression, `say "hello, ", who`, is evaluated for its output on the screen. whatever its type turns out to be, it cannot leak out — the function returns nothing.

leaving such a function early

now suppose you want to exit a `none` function early too. the hand reaches for an empty return — but we just agreed that `return` without an expression is not allowed. the literal `none` helps:

```
fn announce (x s64):
    if x <= 0 then
        return none;
    say "positive:", x;;
```

`return none` reads as "leave, handing nothing back." there is an expression after `return`, so the form is satisfied; the type of `none` is the same as the function's result; and what comes back is exactly "nothing." this is the familiar early exit from a function that delivers nothing.

a question to think over

set two of our observations side by side and a small gap opens.

on one hand, the last expression in a none function may be of any type: it is discarded anyway. on the other, the early exit we wrote carefully as return none, matching it to the result type on purpose. so: what does the translator do if, in a none function, you

write return with an expression of plainly the wrong type?

```
fn announce (x s64):
    return x + 1;        -- ?
    say "we never reach here";;
```

put another way — does the leniency granted to the last expression reach as far as return? or must return in a none function carry only none?

work it out for yourself. one hint: an expression that merely falls last is silence; one you hand to return is a claim. the rest follows.

conway's game of life

```
--
-- golife -- conway's game of life
-- usage:
--      golife.play
-- authors:
--      ortfero <ortfero@gmail.com>, 2026
-- license: cc-by-sa 4.0
--

fn play() raises

implementation

const board_x   = 2
const board_y   = 3
const board_w   = 130
const board_h   = 40
const grid_size = 5200 -- board_w * board_h
const tick_ms   = 100
const live_pct  = 30

var cells        [grid_size]bool
var next_cells   [grid_size]bool
var rng          random.state
var running      bool
var paused       bool
var generation   s64
var population   s64


fn count_neighbors s64 (x s64, y s64)
    n s64, above s64, here s64, below s64, l s64, r s64:
    n = 0;
    l = x - 1;
    r = x + 1;
    above = (y - 1) * board_w;
    here  = y * board_w;
    below = (y + 1) * board_w;
    if and (l >= 0), (y > 0), cells[above + l] then
```

```
        n += 1;
    if and (y > 0), cells[above + x] then
        n += 1;
    if and (r < board_w), (y > 0), cells[above + r] then
        n += 1;
    if and (l >= 0), cells[here + l] then
        n += 1;
    if and (r < board_w), cells[here + r] then
        n += 1;
    if and (l >= 0), (y < board_h - 1), cells[below + l] then
        n += 1;
    if and (y < board_h - 1), cells[below + x] then
        n += 1;
    if and (r < board_w), (y < board_h - 1), cells[below + r] then
        n += 1;
    n;;


fn step ()
    x s64, y s64, n s64, alive bool, born bool, i s64, p s64:
    p = 0;
    y = 0; for y < board_h; y += 1:
        x = 0; for x < board_w; x += 1:
            i = y * board_w + x;
            n = count_neighbors x, y;
            alive = cells[i];
            born = if alive then (or (n == 2), (n == 3)) else n == 3;
            next_cells[i] = born;
            if born then
                p += 1;
            if born != alive then
                screen.poke (board_x + x), (board_y + y), (if born then
    i = 0; for i < grid_size; i += 1:
        cells[i] = next_cells[i];;
    population = p;
    generation += 1;;


fn randomize () raises
    i s64, p s64:
    p = 0;
    i = 0; for i < grid_size; i += 1:
        cells[i] = (random.range rng&, 0, 100) < live_pct;
```

```
        if cells[i] then
            p += 1;;
    population = p;
    generation = 0;;


fn clear_cells ()
    i s64:
    i = 0; for i < grid_size; i += 1:
        cells[i] = false;;
    population = 0;
    generation = 0;;


fn draw_title ():
    screen.at board_x, (board_y - 3);
    sayin "golife -- space: pause, r: reset, c: clear, ctrl+q: quit";;


fn draw_border ()
    i s64:
    i = 0; for i < board_w; i += 1:
        screen.poke (board_x + i), (board_y - 1), '-';
        screen.poke (board_x + i), (board_y + board_h), '-';;
    i = 0; for i < board_h; i += 1:
        screen.poke (board_x - 1), (board_y + i), '|';
        screen.poke (board_x + board_w), (board_y + i), '|';;
    screen.poke (board_x - 1), (board_y - 1), '+';
    screen.poke (board_x + board_w), (board_y - 1), '+';
    screen.poke (board_x - 1), (board_y + board_h), '+';
    screen.poke (board_x + board_w), (board_y + board_h), '+';;


fn draw_cells ()
    x s64, y s64, i s64:
    y = 0; for y < board_h; y += 1:
        x = 0; for x < board_w; x += 1:
            i = y * board_w + x;
            screen.poke (board_x + x), (board_y + y),
                (if cells[i] then '#' else ' ');
        ;;;
```

```
fn draw_status ():
    screen.at board_x, (board_y + board_h + 1);
    sayin "gen: ", generation, "  pop: ", population, "   ",
        (if paused then "[paused]" else "        ");;


fn handle_input () raises
    k screen.key, c u8:
    for:
        k = screen.poll_key;
        if k == 0 then
            return none;
        if not (screen.is_key_down k) then
            continue;
        c = screen.key_code k;
        -- ctrl+q exits.
        if and (c == 'q'), (screen.is_ctrl_pressed k) then:
            running = false;
            return none;;
        if not (screen.is_control_key k) then:
            if c == ' ' then:
                paused = not paused;
                draw_status;;
            if c == 'r' then:
                randomize;
                draw_cells;
                draw_status;;
            if c == 'c' then:
                clear_cells;
                draw_cells;
                draw_status;
            ;;;;


fn play () raises
    last clock.tick:
    screen.save;
    defer screen.restore;
    screen.clear;
    draw_title;
    draw_border;
    random.seed rng&, clock.now;
    randomize;
```

```
draw_cells;
draw_status;
running = true;
paused = false;
last = clock.now;
for running:
    handle_input;
    if and (not paused), (clock.elapsed_ms last >= tick_ms) then:
        last = clock.now;
        step;
        draw_status;
    ;;;
```

from programming language design to computer construction

niklaus wirth

reprinted from the acm turing award lectures

niklaus wirth, "from programming language design to computer construction," 1984 acm a.m. turing award lecture. in: acm turing award lectures, association for computing machinery, new york, ny, usa, 2007. isbn 978-1-4503-1049-9. https://doi.org/10.1145/1283920.1283941

included here by permission.

niklaus wirth of the swiss federal institute of technology (eth) was presented the 1984 acm a. m. turing award at the association's annual conference in san francisco in october in recognition of his outstanding work in developing a sequence of innovative computer languages: euler, algol-w, modula, and pascal. pascal, in particular, has become significant pedagogically and has established a foundation for future research in the areas of computer language, systems, and architecture. the hallmarks of a wirth language are its simplicity, economy of design, and high-quality engineering, which result in a language whose notation appears to be a natural extension of algorithmic thinking rather than an extraneous formalism.

wirth's ability in language design is complemented by a masterful writing ability. in the april 1971 issue of communications of the acm, wirth published a seminal paper on structured programming ("program development by stepwise refinement") that recommended top-down structuring of programs (i.e., successively refining program stubs until the program is fully elaborated). the resulting elegant and powerful method of exposition remains interesting reading today even after the furor over structured programming has subsided. two

later papers, "toward a discipline of real-time programming" and "what can we do about the unnecessary diversity of notation" (published in cacm in august and november 1974, respectively), speak to wirth's consistent and dedicated search for an adequate language formalism.

the turing award, the association's highest recognition of technical contributions to the computing community, honors alan m. turing, the english mathematician who defined the computer prototype turing machine and helped break german ciphers during world war ii.

wirth received his ph.d. from the university of california at berkeley in 1963 and was assistant professor at stanford university until 1967. he has been professor at the eth zurich since 1968; from 1982 until 1984 he was chairman of the division of computer science (informatik) at eth. wirth's recent work includes the design and development of the personal computer lilith in conjunction with the modula-2 language. in his lecture, wirth presents a short history of his major projects, drawing conclusions and highlighting the principles that have guided his work.

from neliac (via algol 60) to euler and algol w, to pascal and modula-2, and ultimately lilith, wirth's search for an appropriate

formalism for systems programming yields intriguing insights and surprising results.

it is a great pleasure to receive the turing award, and both gratifying and encouraging to receive appreciation for work done over so many years. i wish to thank acm for bestowing upon me this prestigious award. it is particularly fitting that i receive it in san francisco, where my professional career began.

soon after i received notice of the award, my feeling of joy was tempered somewhat by the awareness of having to deliver the turing lecture. for someone who is an engineer rather than an orator or preacher, this obligation causes some noticeable anxiety. foremost among the questions it poses is the following: what do people expect from such a lecture? some will wish to gain technical insight about one's work, or expect an assessment of its relevance or impact. others will wish to hear how the ideas behind it emerged. still others expect a statement from the expert about future trends, events, and products. and some hope for a frank assessment of the present engulfing us, either glorifying the monumental advance of our technology or lamenting its cancerous side effects and exaggerations.

in a period of indecision, i consulted some previous turing lectures and saw that a condensed report about the history of one's work would be quite acceptable. in order to be not just entertaining, i shall try to summarize what i believe i have learned from the past. this choice, frankly, suits me quite well, because neither do i pretend to know more about the future than most others, nor do i like to be proven wrong

afterwards. also, the art of preaching about current achievements and misdeeds is not my primary strength. this does not imply that i observe the present computing scene without concern, particularly its tumultuous hassle with commercialism.

certainly, when i entered the computing field in 1960, it was neither so much in the commercial limelight nor in academic curricula. during my studies at the swiss federal institute of technology (eth), the only mention i heard of computers was in an elective course given by ambros p. speiser, who later became the president of ifip. the computer ermeth developed by him was hardly accessible to ordinary students, and so my initiation to the computing field was delayed until i took a course in numerical analysis at laval university in canada. but alas, the alvac iii e machinery was out of order most of the time, and exercises in programming remained on paper in the form of untested sequences of hexadecimal codes.

my next attempt was somewhat more successful: at berkeley, I was confronted with harry huskey's pet machine, the bendix g-15 computer. although the bendix g-15 provided some feeling of success by producing results, the gist of the programming art appeared to be the clever allocation of instructions on the drum. if you ignored the art, your programs could well run slower by a factor of one hundred. but the educational benefit was clear: you could not afford to ignore the least little detail. there was no way to cover up deficiencies in your design by simply buying more memory. in retrospect, the most attractive feature was that every detail of the machine was visible and could

be understood. nothing was hidden in complex circuitry, silicon, or a magic operating system.

on the other hand, it was obvious that computers of the future had to be more effectively programmable. i therefore gave up the idea of studying how to design hardware in favor of studying how to use it more elegantly. it was my luck to join a research group that was engaged in the development — or perhaps rather improvement — of a compiler and its use on an ibm 704. the language was called neliac, a dialect of algol 58. the benefits of such a "language" were quickly obvious, and the task of automatically translating programs into machine code posed challenging problems. this is precisely what one is looking for when engaged in the pursuit of a doctorate. the compiler, itself written in neliac, was a most intricate mess. the subject seemed to consist of 1 percent science and 99 percent sorcery, and this tilt had to be changed. evidently, programs should be designed according to the same principles as electronic circuits, that is, clearly subdivided into parts with only a few wires going across the boundaries. only by understanding one part at a time would there be hope of finally understanding the whole.

this attempt received a vigorous starting impulse from the appearance of the report on algol 60. algol 60 was the first language defined with clarity; its syntax was even specified in a rigorous formalism. the lesson was that a clear specification is a necessary but not sufficient condition for a reliable and effective implementation. contact with aadrian van wijngaarden, one of algol's codesigners, brought out the central

theme more distinctly: could algol's principles be condensed and crystallized even further?

thus began my adventures in programming languages. the first experiment led to a dissertation and the language euler — a trip with the bush knife through the jungle of language features and facilities. the result was academic elegance, but not much of practical utility — almost an antithesis of the later data-typed and structured programming languages. but it did create a basis for the systematic design of compilers that, so was the hope, could be extended without loss of clarity to accommodate further facilities.

euler caught the attention of the ifip working group that was engaged in planning the future of algol. The language algol 60, designed by and for numerical mathematicians, had a systematic structure and a concise definition that were appreciated by mathematically trained people but lacked compilers and support by industry. to gain acceptance, its range of application had to be widened. the working group assumed the task of proposing a successor and soon split into two camps. on one side were the ambitious who wanted to erect another milestone in language design, and, on the other, those who felt that time was pressing and that an adequately extended algol 60 would be a productive endeavor. i belonged to this second party and submitted a proposal that lost the election. thereafter, the proposal was improved with contributions from tony hoare (a member of the same group) and implemented on stanford university's first ibm 360. the language later became known as algol w and was used in several universities for teaching purposes.

a small interlude in this sizable implementation effort is worth mentioning. the new ibm 360 offered only assembler code and, of course, fortran. neither particularly were loved, either by me or my graduate students, as a tool for designing a compiler. hence, i mustered the courage to define yet another language in which the algol compiler could be described: A compromise between algol and the facilities offered by the assembler, it would be a machine language with algol-like statement structures and declarations. notably, the language was defined in a couple of weeks; i wrote the cross compiler on the burroughs b-5000 computer within four months, and a diligent student transported it to the ibm 360 within an equal period of time. this preparative interlude helped speed up the algol effort considerably. although envisaged as serving our own immediate needs and to be discarded thereafter, it quickly acquired its own momentum. pl360 became an effective tool in many places and inspired similar developments for other machines.

ironically, the success of pl360 was also an indication of algol-w's failure. algol's range of application had been widened, but as a tool for systems programming, it still had evident deficiencies. the difficulty of resolving many demands with a single language had emerged, and the goal itself became questionable. pl/1, released around this time, provided further evidence to support this contention. the swiss army knife idea has its merits, but if driven to excess, the knife becomes a millstone. also, the size of the algol-w compiler grew beyond the limits within which one could rest comfortably with the feeling of

having a grasp, a mental understanding, of the whole program. the desire for a more concise yet more appropriate formalism for systems programming had not been fulfilled. systems programming requires an efficient compiler generating efficient code that operates without a fixed, hidden, and large so-called run-time package. this goal had been missed by both algol-w and pl/1, both because the languages were complex and the target computers inadequate.

in the fall of 1967, i returned to switzerland. a year later, i was able to establish a team with three assistants to implement the language that later became known as pascal. freed from the constraints of obtaining a committee consensus, i was able to concentrate on including the features i myself deemed essential and excluding those whose implementation effort i judged to be incommensurate with the ultimate benefit. the constraint of severely limited manpower is sometimes an advantage.

occasionally, it has been claimed that pascal was designed as a language for teaching. although this is correct, its use in teaching was not the only goal. in fact, i do not believe in using tools and formalisms in teaching that are inadequate for any practical task. by today's standards, pascal has obvious deficiencies for programming large systems, but 15 years ago it represented a sensible compromise between what was desirable and what was effective. At eth, we introduced pascal in programming classes in 1972, in fact against considerable opposition. it turned out to be a success because it allowed the teacher to concentrate more heavily on structures and concepts than features and

peculiarities, that is, on principles rather than techniques.

our first pascal compiler was implemented for the cdc 6000 computer family. it was written in pascal itself. no pl6000 was necessary, and i considered this a substantial step forward. nonetheless, the code generated was definitely inferior to that generated by fortran compilers for corresponding programs. speed is an essential and easily measurable criterion, and we believed the validity of the high-level language concept would be accepted in industry only if the performance penalty were to vanish or at least diminish. with this in mind, a second effort — essentially a one-man effort — was launched to produce a high-quality compiler. the goal was achieved in 1974 by urs ammann, and the compiler was thereafter widely distributed and is being used today in many universities and industries. yet the price was high; the effort to generate good (i.e., not even optimal) code is proportional to the mismatch between language and machine, and the cdc 6000 had certainly not been designed with high-level languages in mind.

ironically again, the principal benefit turned up where we had least expected it. after the existence of pascal became known, several people asked us for assistance in implementing pascal on various other machines, emphasizing that they intended to use it for teaching and that speed was not of overwhelming importance. thereupon, we decided to provide a compiler version that would generate code for a machine of our own design. this code later became known as p-code. the p-code version was very easy to construct because the

new compiler was developed as a substantial exercise in structured programming by stepwise refinement and therefore the first few refinement steps could be adopted unchanged. pascal-p proved enormously successful in spreading the language among many users. had we possessed the wisdom to foresee the dimensions of this movement, we would have put more effort and care into designing and documenting p-code. as it was, it remained a side effort to honor the requests in one concentrated stride. this shows that even with the best intentions one may choose one's goals wrongly.

but pascal gained truly widespread recognition only after ken bowles in san diego recognized that the p-system could well be implemented on the novel microcomputers. his efforts to develop a suitable environment with integrated compiler, filer, editor, and debugger caused a breakthrough: pascal became available to thousands of new computer users 'who were not burdened with acquired habits or stifled by the urge to stay compatible with software of the past.

in the meantime, i terminated work on pascal and decided to investigate the enticing new subject of multiprogramming, where hoare had laid respectable foundations and brinch hansen had led the way with his concurrent pascal. the attempt to distill concrete rules for a multiprogramming discipline quickly led me to formulate them in terms of a small set of programming facilities. in order to put the rules to a genuine test, i embedded them in a fragmentary language, whose name was coined after my principal aim: modularity in program systems. the module later turned out to be the principal asset of this language; it gave

the abstract concept of information hiding a concrete form and incorporated a method as significant in uniprogramming as in multiprogramming. also, modula contained facilities to express concurrent processes and their synchronization.

by 1976, i had become somewhat weary of programming languages and the frustrating task of constructing good compilers for existing computers that were designed for old-fashioned "by-hand" coding. fortunately, i was given the opportunity to spend a sabbatical year at the research laboratory of xerox corporation in palo alto, where the concept of the powerful personal workstation had not only originated but was also put into practice. instead of sharing a large, monolithic computer with many others and fighting for a share via a wire with a 3-kHz bandwidth, I now used my own computer placed under my desk over a 15-MHz channel. the influence of a 5000-fold increase in anything is not foreseeable; it is overwhelming. the most elating sensation was that after 16 years of working for computers, the computer now seemed to work for me. for the first time, i did my daily correspondence and report writing with the aid of a computer, instead of planning new languages, compilers, and programs for others to use. the other revelation was that a compiler for the language mesa whose complexity was far beyond that of pascal, could be implemented on such a workstation. these new working conditions were so many orders of magnitude above what i had experienced at home that i decided to try to establish such an environment there as well.

i finally decided to dig into hardware design. this decision was reinforced by my old disgust with existing

computer architectures that made life miserable for a compiler designer with a bent toward systematic simplicity. the idea of designing and building an entire computer system consisting of hardware, microcode, compiler, operating system, and program utilities quickly took shape in my imagination — a design that would be free from any constraint to be compatible with a pdp-11 or an ibm 360, or fortran, pascal, unix, or whatever other current fad or committee standard there might be.

but a sensation of liberation is not enough to succeed in a technical project. hard work, determination, a sensitive feeling of what is essential and what ephemeral, and a portion of luck are indispensable. the first lucky accident was a telephone call from a hardware designer enquiring about the possibility of coming to our university to learn about software techniques and acquire a ph.d. why not teach him about software and let him teach us about hardware? it didn't take long before the two of us became a functioning team, and richard ohran soon became so excited about the new design that he almost totally forgot both software and ph.d. that didn't disturb me too much, for i was amply occupied with the design of hardware parts; with specifying the micro- and macrocodes, and by programming the latter's interpreter; with planning the overall software system; and in particular with programming a text editor and a diagram editor, both making use of the new high-resolution bit-mapped display and the small miracle called mouse as a pointing device. this exercise in programming highly interactive utility programs

required the study and application of techniques quite foreign to conventional compiler and operating system design.

the total project was so diversified and complex that it seemed irresponsible to start it, particularly in view of the small number of part-time assistants available to us, who averaged around seven. the major threat was that it would take too long to keep the enthusiastic two of us persisting and to let the others, who had not yet experienced the power of the workstation idea, become equally enthusiastic. to keep the project within reasonable dimensions, i stuck to three dogmas: aim for a single-processor computer to be operated by a single user and programmed in a single language. notably, these cornerstones were diametrically opposed to the trends of the time, which favored research in multiprocessor configurations, time-sharing multiuser operating systems, and as many languages as you could muster.

under the constraints of a single language, i faced a difficult choice whose effects would be wide ranging, namely, that of selecting a language. of existing languages, none seemed attractive. neither could they satisfy all the requirements, nor were they particularly appealing to the compiler designer who knows the task has to be accomplished in a reasonable time span. in particular, the language had to accommodate all our wishes with regard to structuring facilities, based on 10 years' experience with pascal, and it had to cater to problems so far only handled by coding with an assembler. to cut a long story short, the choice was to design an offspring of both proven

pascal and experimental modula, that is, modula-2. the module is the key to bringing under one hat the contradictory requirements of high-level abstraction for security through redundancy checking and low-level facilities that allow access to individual features of a particular computer. it lets the programmer encapsulate the use of low-level facilities in a few small parts of the system, thus protecting him from falling into their traps in unexpected places.

the lilith project proved that it is not only possible but advantageous to design a single-language system. everything from device drivers to text and graphics editors is written in the same language. there is no distinction between modules belonging to the operating system and those belonging to the user's program. in fact, that distinction almost vanishes and with it the burden of a monolithic, bulky resident block of code, which no one wants but everyone has to accept. moreover, the lilith project proved the benefits of a well-matched hardware/software design. these benefits can be measured in terms of speed: comparisons of execution times of modula programs revealed that lilith is often superior to a vax 750 whose complexity and cost are a multiple of those of lilith. they can also be measured in terms of space: the code of modula programs for lilith is shorter than the code for pdp-11, vax, or 68000 by factors of 2 to 3, and shorter than that of the ns 32000 by a factor of 1.5 to 2. in addition, the code-generating parts of compilers for these microprocessors are considerably more intricate than they are in lilith due to their ill-matched instruction sets. this length factor has to

be multiplied by the inferior density factor, which casts a dark shadow over the much advertised high-level language suitability of modern microprocessors and reveals these claims to be exaggerated. the prospect that these designs will be reproduced millions of times is rather depressing, for by their mere number they become our standard building blocks. unfortunately, advances in semiconductor technology have been so rapid that architectural advances are overshadowed and have become seemingly less relevant. competition forces manufacturers to freeze new designs into silicon long before they have proved their effectiveness. and whereas bulky software can at least be modified and at best be replaced, nowadays complexity has descended into the very chips. and there is little hope that we have a better mastery of complexity when we apply it to hardware rather than software.

on both sides of this fence, complexity has and will maintain a strong fascination for many people. it is true that we live in a complex world and strive to solve inherently complex problems, which often do require complex mechanisms. however, this should not diminish our desire for elegant solutions, which convince by their clarity and effectiveness. simple, elegant solutions are more effective, but they are harder to find than complex ones, and they require more time, which we too often believe to be unaffordable.

before closing, let me try to distill some of the common characteristics of the projects that were mentioned. a very important technique that is seldom used as effectively as in computing is the bootstrap. we used it in virtually every project. when developing

a tool, be it a programming language, a compiler, or a computer, i designed it in such a way that it was beneficial in the very next step: pl360 was developed to implement algol w; pascal to implement pascal; modula-2 to implement the whole workstation software; and lilith to provide a suitable environment for all our future work, ranging from programming to circuit documentation and development, from report preparation to font design. bootstrapping is the most effective way of profiting from one's own efforts as well as suffering from one's mistakes. this makes it mandatory to distinguish early between what is essential and what ephemeral. i have always tried to identify and focus in on what is essential and yields unquestionable benefits. for example, the inclusion of a coherent and consistent scheme of data type declarations in a programming language i consider essential, whereas the details of varieties of for-statements, or whether the compiler distinguishes between upper- and lowercase letters, are ephemeral questions. in computer design, i consider the choice of addressing modes and the provision of complete and consistent sets of {signed and unsigned} arithmetic instructions including proper traps on overflow to be crucial; in contrast, the details of a multichannel prioritized interrupt mechanism are rather peripheral. even more important is ensuring that the ephemeral never impinge on the systematic, structured design of the central facilities. rather, the ephemeral must be added fittingly to the existing, well-structured framework.

rejecting pressures to include all kinds of facilities that "might also be nice to have" is

sometimes hard. the danger that one's desire to please will interfere with the goal of consistent design is very real. i have always tried to weigh the gains against the cost. for example, when considering the inclusion of either a language feature or the compiler's special treatment of a reasonably frequent construct, one must weigh the benefits against the added cost of its implementation and its mere presence, which results in a larger system. language designers often fail in this respect. i gladly admit that certain features of ada that have no counterparts in modula-2 may be nice to have occasionally, but at the same time, i question whether they are worth the price. the price is considerable: first, although the design of both languages started in 1977, ada compilers have only now begun to emerge, whereas we have been using modula since 1979. second, ada compilers are rumored to be gigantic programs consisting of several hundred thousand lines of code, whereas our newest modula compiler measures some five thousand lines only. i confess secretly that this modula compiler is already at the limits of comprehensible complexity, and i would feel utterly incapable of constructing a good compiler for ada. but even if the effort of building unnecessarily large systems and the cost of memory to contain their code could be ignored, the real cost is hidden in the unseen efforts of the innumerable programmers trying desperately to understand them and use them effectively.

another common characteristic of the projects sketched was the choice of tools. it is my belief that a tool should be commensurate with the product; it must

be as simple as possible, but no simpler. a tool is in fact counterproductive when a large part of the entire project is taken up by mastering the tool. within the euler, algol w, and pl360 projects, much consideration was given to the development of table-driven, bottom-up syntax analysis techniques. later, i switched back to the simple recursive-descent, top-down method, which is easily comprehensible and unquestionably sufficiently powerful, if the syntax of the language is wisely chosen. on the development of the lilith hardware, we restricted ourselves to a good oscilloscope; only rarely was a logic state analyzer needed. this was possible due to a relatively systematic, trick-free concept for the processor.

every single project was primarily a learning experiment. one learns best when inventing. only by actually doing a development project can i gain enough familiarity with the intrinsic difficulties and enough confidence that the inherent details can be mastered. i never could separate the design of a language from its implementation, for a rigid definition without the feedback from the construction of its compiler would seem to me presumptuous and unprofessional. thus, i participated in the construction of compilers, circuitry, and text and graphics editors, and this entailed microprogramming, much high-level programming, circuit design, board layout, and even wire wrapping. this may seem odd, but i simply like hands-on experience much better than team management. i have also learned that researchers accept leadership from a factual, in-touch team member much more readily than from an organization expert, be he a manager in

industry or a university professor. i try to keep in mind that teaching by setting a good example is often the most effective method and sometimes the only one available.

lastly, each of these projects was carried through by the enthusiasm and the desire to succeed in the knowledge that the endeavor was worthwhile. this is perhaps the most essential but also the most elusive and subtle prerequisite. i was lucky to have team members who let themselves be infected with enthusiasm, and here is my chance to thank all of them for their valuable contributions. my sincere thanks go to all who participated, be it in the direct form of working in a team, or in the indirect forms of testing our results and providing feedback, of contributing ideas through criticism or encouragement, or of forming user societies. without them, neither algol w, nor pascal, nor modula-2, nor lilith would have become what they are. this turing award also honors their contributions.

on process isolation

to the editor

progzmaker

i held off writing for some time, because the observations i wish to share have long been considered platitudes in our trade, and platitudes are not interesting to repeat. but reading the recent polemic on the design of operating systems, i noticed that these platitudes provoke doubts in some of your readers, and i decided after all to set them out as i understand them, from my own vantage point. my vantage point is fifteen years in a large computing center, where systems run not for elegance but because without them quite a lot else does not run either.

the subject is the separation of address spaces between processes — that same isolation that mature systems install without discussion. i want to say, positively, why it is installed, without arguing with anyone in particular.

the first and simplest point. programs make mistakes. not in theory but every day, and not from the author's irresponsibility, but because a person who has written three thousand lines cannot hold all their interactions in his head. a wild pointer in one program, left without obstruction, corrupts the memory of another program or of the kernel, and the result is a system in an undefined state. isolation turns this into "this particular program has crashed," and the

culprit points at itself. from here begins the possibility of debugging, and the possibility of continuing to work without rebooting everything. this is not even about security; this is about operation.

second — about security in the simplest sense. on any working machine there is data that one program is trusted with and another is not. the accounting program need not read the correspondence; the correspondence program need not poke into signing keys; the display program need not know passwords. without a barrier all of this is one common muddle in which every process is a potential observer of everything. isolation here is not censorship but elementary decency: to each their own, and no one looks into someone else's affairs without invitation.

third, a point rarely spoken of because there is no drama in it. process isolation enormously simplifies programming. every program is written as if it were alone in the machine and owned all memory from zero to infinity. it need not know where others live, need not negotiate placement, need not watch that it does not step on someone else's ground. this is a great unburdening of the author. secondarily, the system gains the freedom to move programs around in physical memory, swap them to disk, share common pieces between them — and the program need know nothing of it. a convenience for which we long ago stopped being grateful, because we got used to it.

fourth — about the data a program receives from outside. any nontrivial program reads something: a file, a message, a picture, a stream. and at the moment of reading it is forced to trust the contents precisely

insofar as its parser is well written. parsers are written by people; people make mistakes; mistakes in parsers turn foreign data into the execution of foreign intent inside the program. without isolation, this foreign intent executes with the rights of the whole system. with isolation, with the rights of a single process which by design has nothing extra. this is the difference between "we have lost a page" and "we have lost everything."

fifth. a barrier is not a guarantee, and no one who works with it claims otherwise. a barrier can be breached, and sometimes is. but exploitation is an economic occupation: the attacker spends time, talent and money on each layer that gets in his way. a barrier raises the cost of attack by factors, sometimes by orders of magnitude, and cuts off almost everyone who has nothing to pay with. a defense that cuts off ninety-nine percent of attackers and raises the remaining one percent's work to unaffordable sums is no trifle, even if it is not absolute. absolute defenses do not exist in our craft at all.

sixth, a finer point. between the moment a bug enters a program and the moment it becomes known, time passes — on serious systems we are accustomed to count it in years, and there is no exaggeration here. through those years the bug sits quietly in working code. if there is no defense, then anyone who has found that bug before the rest of the community owns everything those systems do for the entire length of the window. isolation is insurance for exactly that length. not against "it will never happen," but against "when it happens, let it not catch us all at once."

seventh. isolation is useful not only between different programs but inside a single program, if it is large enough. the part that parses incoming data should be separated from the part that stores the result; the part that talks to the network from the part that talks to the disk. every such internal barrier limits what an attacker obtains if he manages to subvert the logic of one component. on mature projects this has long been simply good form — to separate privileges within one's own application without waiting for the kernel to separate them for you.

eighth — an observation, not a theory. the most independent practitioners, those who build their own systems end to end, with no eye to the industry, for the sake of their own freedom and their own understanding, arrive at isolation in exactly the same way as we do. not because they have been told to, but because, having lived with their own machine long enough, they begin to see in it something of their own rather than something foreign. this is not an argument from authority; it is an argument from convergence: different people with different motives, starting from different places, arrive at the same thought, and that means something.

and last, the most personal. i install barriers not because i distrust my own code, but because i trust it exactly as much as i trust myself. and i have known myself for twenty-some years and can say with confidence: i make mistakes, and i will make mistakes, and on a friday evening i will make them especially. a barrier is the instrument by which i reduce the

consequences of my own inevitable fallibility in advance, while my head is still clear and i can still think about it. this is not self-deprecation, and not insurance against the vendor. it is professional maturity: the longer you are in the craft, the more calmly you admit that the safety harness is needed, and the more willingly you put it on.

there is no new knowledge in this letter. everything i have set out has long been known to anyone who has lived through a few production cycles and seen how simplifications dictated by aesthetics end. but platitudes are sometimes worth saying aloud, especially in a journal that is read by the young. if even one of them, having read this letter, pauses before tearing down the isolation in his next project, then consider that my shift today was not in vain.

to the progzmaker

ortfero

we read progzmaker's letter with close attention, and we wish to reply — because, as a small research project without significant resources, we are granted an unusual privilege in this profession: to re-examine settled claims and norms.

let us state plainly: we hold none of progzmaker's arguments for isolation to be false. programs do err. hostile input is dangerous. a barrier does raise the cost of attack. the window between the appearance of a defect and its discovery does indeed run to years. the personal maturity that prompts one to fasten one's own

safety harness is indeed an admirable thing. to dispute these points is pointless, and we shall not.

we wish only to note that isolation, like any engineering decision, has another side, at which long custom has trained us not to look. not in order to abolish it, but so that, at least once, we may see it as a choice rather than a law of nature.

first — the historical point. the separation of address spaces between processes was born not from concern for the user, but from the necessity of dividing an expensive machine among mutually-distrusting parties. it was the era of one computer to a hundred people, in which one part of the machine's population was, by office, not to see what another part was doing. upon this primary meaning later meanings were accreted: protection from one's own errors, fault localisation, security against hostile input. all are legitimate. but the primary meaning — the separation of people — is simply unnecessary on the personal machine of a single owner; there is no one to distrust. that we continue, by inertia, to build machines as though for many, is an inheritance worth noticing at least once.

second — on where the barrier runs. on a contemporary personal machine the most valuable thing is not the system files but the home directory: keys, correspondence, documents, sessions. the classical uid-based separation does not protect this, because all of a user's processes run under the same uid and read one another through the filesystem. that is, the isolation for whose sake the model was built is, in our reality, drawn not where the attacked thing actually lies. this is no reproach to isolation as an idea; it is an

observation: the boundary is drawn not along the line on which the real risk runs.

third — on alternative implementation. that barriers between programs are useful is not in question for us; we ask where to set them, and by what means. hardware separation of pages is one implementation, and it has a price: in silicon, in context switches, in kernel complexity, in the mitigations one must layer on after each new finding in speculative execution. language safety is another implementation, and it sets the barriers not in the hardware but in the types: the modules of a program cannot write into one another's memory not because the mmu forbids it, but because the compiler will not let them. this is not the abolition of the idea of isolation; it is its transfer to a floor on which it is cheaper and more exact. in our system the work of isolation is done — only not by the mechanisms to which the large computing centre has grown accustomed.

fourth — on an inconvenient property of hardware separation, on which we would wish our respected colleague to think as well. it is a structurally asymmetric mechanism. the processor's privileged mode — the very thing that affords isolation — is, by construction, designed so that one party may dictate to another what is permitted. on the operator's machine, where the key to privileged mode is in the owner's hands, it is an instrument of his sovereignty. but by the same mechanism, once the key has moved into the vendor's hands, the owner is forbidden to install and run on his own machine what he wishes. one and the same transistor serves two opposing policies. the machine is

defended from user code — and the vendor is defended from the owner — by one and the same primitive. it seems to us this is worth thinking of when one debates whether the primitive is necessary.

fifth — on complexity, and on why we have received it. our respected colleague works in a system whose count runs in millions of lines, and in such a system isolation does indeed become the last reasonable insurance, because to read all of it and hold it in one's head is physically impossible. but this complexity we have received not from the laws of nature. it was produced by the combination of industrial development, staff turnover, "efficiency" metrics, and standards layered over decades under the pressure of various interests pulling in their own directions. a small system, in which the author lives with his code for years and treats it as his own work, is a different regime, and the quality of code within it is qualitatively different. this does not render our colleague's arguments false for his context, but it does mean that part of the weight of his arguments rests upon conditions which, in our context, simply are not reproduced. the small system does not require all the safeguards of the large, precisely because it is small.

sixth — on new code which has not yet been tried in service. our respected colleague is right that such code cannot be allowed everything at once. but in our system there is a natural mechanism for this: the emulator upon which the system itself is built. an untried program is launched in an external instance of the virtual machine with a curtailed virtual world —

without access to valuable data, with a fake network, with its own virtual disk. when we have watched it long enough, we resettle it into the main machine, or leave it to live forever in its sandbox. this is not a renunciation of isolation; it is its transfer from the boundary of the process to the boundary of the virtual computer. the boundary turns out to be larger, and it can be drawn precisely along the line on which actual trust runs, rather than along the one drawn for us in the seventies.

and a last thing, of which we wish to speak honestly. we do not claim that our model will defeat our respected colleague's model. we are not, in fact, certain that it will defeat anything beyond our own curiosity. but in an industry in which small research groups do not permit themselves to re-examine foundations, the large computing centres receive nothing upon which, in the next generation, they might stand. do not, please, deprive us of the right to this question; and if it does not arise for you, consider that we ask it on your behalf.

with respect, and with thanks for raising the conversation.

feedback

"questions of cybernetics" is the bulletin of a small working group, and it lives on what comes back. if something in this issue is wrong, unclear, or worth arguing with, we want to hear it. if you have tried the system and found where it breaks, where it surprises you, or where it fails to be as surveyable as we claim, write to us. corrections are welcome and will be printed. so are disagreements, especially the well-reasoned kind.

we are also interested in programs, listings, notes from your own work, accounts of teaching the machine to someone, and short pieces of fiction or reflection for the reading room. if you have built something on the aldan stack, or ported something to it, or written something about it, send it along. we would rather print a rough piece by a working hand than a polished one by a distant observer.

and if any of this resonates more deeply —- if you find yourself wanting not only to write in but to take part in the work —- the aldan collective is open to new members. the condition is simple: an interest in building small, comprehensible systems. write to us about yourself and what you would like to do.

aldanteam@protonmail.com. we read everything that arrives.

the guy from hell. chapter 1

arkady strugatsky, boris strugatsky

what a village! i've never seen a village like this and didn't even know such villages existed. the houses are round, brown, windowless, standing on stilts like watchtowers, and underneath them, there's all sorts of junk piled up — huge pots, troughs, rusty cauldrons, wooden rakes, shovels... the ground between the houses is clay, so burnt and trampled that it even shines. and everywhere you look, there are nets. dry nets. i don't know what they catch with these nets here: swamp on the right, swamp on the left, smells like a dump... a dreadful hole. they've been rotting here for a thousand years and, if not for the duke, would have rotted for another thousand. the north. wilderness. and, of course, no residents in sight. either they ran away, were taken, or are hiding.

in the square near the trading post, a field kitchen, taken off its wheels, was smoking. a huge porcupine — as wide as he was tall — in a dirty white apron over his dirty gray uniform, was stirring a pot with a long-handled ladle. i think it was mainly this pot that was causing the village to stink.

we approached, and cheetah, stopping, asked where the commander was. this creature didn't even turn around — he muttered something into his stew and pointed with the ladle somewhere down the street. i gave him a kick in the backside with the toe of my

boot, and he quickly turned around, saw our uniforms, and immediately stood at attention. his face matched its ham-like body, and he hadn’t been shaved for a whole week, this porcupine.

"so where’s your commander?" cheetah asked again, pressing his cane into the porcupine’s fat neck under his double chin.

the porcupine rolled his eyes, smacked his lips, and rasped:

"sorry, sir senior instructor... the staff major is at the positions... please, down this street... right at the outskirts... accept my apologies, sir senior instructor..."

he kept rasping and gurgling something, and from around the corner of the trading post, two more porcupines emerged — even scarier than this one, complete scarecrows without weapons or headgear. they saw us and froze at attention. cheetah just looked at them, sighed, and then walked on, tapping his cane against his boot.

yes, we got here just in time. these porcupines, they would have made a mess of things! i’ve only seen three of them so far, but i’m already sick of them, and it’s clear to me that this, excuse the expression, unit, cobbled together from rear-line pests, hastily and haphazardly, with all these regimental bakers, brigade cobblers, clerks, quartermasters, idiots, lazybones, half-blind fools, and funeral detail eagles — they’re all walking fertilizer, lubricant for a bayonet. the imperial armored vehicles would have rolled through them without even noticing anyone was there. easily.

someone called out to us. to the left, between two houses, there was a camouflage tent set up and a white-green cloth hanging on a pole. a medical post. two more porcupines were slowly rummaging through green packs of medical supplies, and on mats thrown directly on the ground lay the wounded. there were three wounded in total; one, with his head bandaged, was propped up on his elbow, looking at us. when we turned around, he called out again:

"sir, instructor! just a moment, please!..."

we approached. cheetah squatted down, and i remained standing behind him. the wounded man had no visible insignia; he was wearing a torn, burnt camouflage jumpsuit, unbuttoned on his bare hairy chest. but from his face, from his wild eyes with singed eyelashes, i immediately knew that this one wasn't a porcupine, no, this one was the real deal. and i was right.

"brigadier-jaeger baron tragg," he introduced himself. it sounded like tank treads clanking. "commander of the independent 18th forest jaeger unit."

"senior instructor digga," cheetah said. "i'm listening, brave brother."

"a cigarette..." the baron asked in a suddenly hoarse voice.

while cheetah was taking out his cigarette case, he hurriedly continued:

"got hit by a flamethrower, scorched like a pig... thank god, the swamp was nearby, i submerged up to my eyebrows... but my cigarettes — turned to mush... thanks..."

he took a drag, closing his eyes, and immediately started coughing harshly, turning blue, convulsing. A

drop of blood oozed from under the bandage on his cheek and congealed, like resin. without turning, cheetah extended his hand over his shoulder and snapped his fingers. i pulled the flask from my belt and handed it to him. the baron took a few gulps, and he seemed to feel a bit better. the other two wounded lay motionless — either they were asleep or already gone. the medics glanced at us fearfully. not really looking, just sneaking glances.

"good..." baron tragg said, returning the flask. "how many men do you have?"

"four tens," cheetah replied. "keep the flask... keep it for yourself."

"forty... forty fighting cats..."

"kittens," cheetah said. "unfortunately... but we'll do everything we can."

the baron looked at him from under his burnt eyebrows. there was agony in his eyes.

"listen, brave brother," he said. "i have no one left. i've been retreating from the pass for three days. constant battles. the rat-eaters are coming on armored vehicles. i've burned about twenty of them. the last two — yesterday... right here, on the outskirts... you'll see. this staff major... an idiot and a coward... an old wreck... i wanted to shoot him, but i had no bullets left. can you imagine? not a single bullet! he was hiding in the village with his porcupines, watching as we were burned out one by one... what was i saying? yes! where is gagrid's brigade? the radio is shattered... the last message was: "hold on, gagrid's brigade is on the way..."

listen, a cigarette... and inform headquarters that the 18th independent is no more."

the baron was already delirious. his wild eyes were clouded, tongue barely moved. he fell onto his back, still talking, talking, mumbling, rasping while his gnarled fingers restlessly groped around, clutching at either the edges of the mat or his jumpsuit. then he suddenly fell silent mid-sentence, and cheetah stood up.

he slowly pulled out a cigarette, not taking his eyes off the upturned face, flicked the lighter, then leaned down and placed the cigarette case along with the lighter next to the blackened fingers, which greedily clutched the cigarette case and gripped it tightly. without a word, cheetah turned, and we moved on.

i thought that it was probably merciful — the brigadier had lost consciousness just in time. otherwise, he would have had to hear that gagrid's brigade was no more either. they were hit overnight by a carpet bombing on the supply route — we spent two hours clearing the highway of wreckage and the already cooling meat of bodies, fending off the mad ones trying to crawl under the trucks to hide. all we found of gagrid was his general's cap, crusted with blood... a chill ran through me when i remembered all this, and i involuntarily glanced at the sky and felt relieved at how low, gray, and bleak it was.

the first thing we saw as we stepped beyond the outskirts was an imperial armored vehicle that had veered off the road and crashed nose-first into the village well. it had already cooled down, and the grass

around it was coated in thick soot. a dead rat-eater lay face down under the open side hatch — everything on him was burned, except for his ginger boots with triple soles. the rat-eaters have good boots! their boots are good, their armored vehicles, and maybe their bombers too. but as soldiers, everyone knows they are useless. jackals.

“how do you like this position, gug?” asked cheetah.

i looked around. what a position! i couldn't believe my eyes. the porcupines had dug themselves trenches on both sides of the road, right in the middle of the clearing between the village outskirts and the jungle. the jungle stood like a wall in front of the trenches, about fifty paces away, no more. you could amass a regiment, a brigade, whatever you wanted there, and the trenches wouldn’t know a thing about it. and when they did find out, it would be too late to do anything. the trenches on the left flank had a swamp behind them. the trenches on the right flank had a flat field behind them, which had something planted before, now all burnt. yeah…

"i don't like this position," i said.

"me neither,” said cheetah.

of course! it wasn’t just the position here. there were also the porcupines. there were about a hundred of them, at least, wandering around their position like it was a marketplace. some had gathered in circles, lighting fires. others just stood there with their hands in their sleeves. last of them were wandering around.

rifles were lying around near the trenches, machine guns were standing aimlessly with their barrels pointed

up at the low sky. in the middle of the road, stuck in the mud up to the hubs, completely out of place, was a rocket launcher. an elderly porcupine sat on the carriage — maybe he was a sentry or just resting after wandering around. either way, he was harmless: he just sat there, picking his ear with a twig.

the whole scene made me feel sour. if it were up to me, i would have mowed down this entire marketplace with a machine gun... i looked at cheetah hopefully, but he remained silent, just moving his hooked nose left to right and right to left.

angry voices sounded behind us, and i turned around. under the stairs of the last house, two porcupines were arguing. they were fighting over a wooden trough – each one pulling it towards themselves, each spewing vile curses. these two, i would have mowed down with particular pleasure. cheetah said to me:

"bring them."

i quickly jumped over to these idiots, smacked one on the hands with my rifle barrel, then the other. when they looked at me, dropping their trough, i jerked my head towards cheetah. they didn't even squeak. it hit them both suddenly, like in a steam bath. wiping their faces with their sleeves as they went, they jogged over to cheetah and stood two steps in front of him, looking like untidy, sweaty piles. cheetah slowly raised his cane, aimed it like he was playing billiards, and whacked them – once across the face of one and then the other. Then he looked at them, the brutes, and simply said:

"bring the commander to me. quickly."

no, guys. cheetah clearly hadn’t expected things to be this bad here. of course, we weren’t expecting anything good. when they send the fighting cats to plug a breach, it’s obvious things are dire. but this... even the tip of cheetah's nose had turned white.

finally, their commander appeared. a tall, sleepy-looking stick of a man with gray sideburns emerged from behind the houses, buttoning his tunic as he came. he was at least fifty years old. his nose was red and veiny, and he wore smudged pince-nez like the staff officers did in that war. wet crumbs of chewing tobacco clung to his long chin. he introduced himself as a staff major and tried to switch to a casual tone with cheetah.

no way! cheetah gave him such a frosty reception that he seemed to shrink in stature. he had been half a head taller at first, but after a minute, snake's milk! he was already looking up at cheetah, a little gray old man of average height.

so, it turned out like this. the staff major didn't know where the enemy was or how many of them there were. his task was to hold the village until reinforcements arrived. his fighting force consisted of 116 soldiers with eight machine guns and two rocket launchers; almost all the soldiers were only partially fit for duty, and after yesterday's forced march, twenty-seven of them were lying in those houses over there — some with chafing, some with hernias, and various other issues.

"listen," cheetah suddenly said. "what's going on over there?"

the staff major stopped mid-sentence and looked where the polished cane was pointing. my goodness, what eyes our cheetah has! only now did i notice: in the biggest circle around one of the fires, amidst the gray jackets of our porcupines, were the disgusting striped jumpsuits of imperial armored infantry. snake's milk! one, two, three... four rat-eaters by our fire, and these pigs are practically hugging them. smoking. and even laughing about something...

"over there?" the staff major said, looking at cheetah with his rabbit-like eyes. "are you referring to the prisoners, sir?"

cheetah didn’t respond. the staff porcupine put his pince-nez back on and launched into an explanation. these, you see, are prisoners, but they have nothing to do with us, you see. they were captured in yesterday's battle by the jaegers. lacking transportation and sufficient personnel for proper guarding...

"gug" cheetah said. "take them to tick. just make him interrogate first..."

i snapped the bolt and walked over to the fire. the bastards were smoking and sipping something from mugs. their faces were content and shiny. disgusting... and that blond one, patting a porcupine on the back, and the porcupine, that brainless log, was grinning and nodding his head. are they drunk or something?

i walked right up to them. the porcupines noticed me from a distance, went silent, and started quietly scattering in all directions. some of them, apparently too scared to move, just sat there, wide-eyed and gaping. the striped ones, though — they turned pale gray. rat-eaters knew our emblem, had heard of us!

i ordered them to stand up. they stood, reluctantly. i ordered them to line up. they did, having no other choice. the blond one started babbling something in our language — i poked him in the ribs with the barrel, and he fell silent. so, i marched them off, single file, heads down, hands behind their backs. rats. and they even smelled ratlike... two were strong men, broad-shouldered, and two looked like they were from the latest conscription, scrawny gomers, maybe just a bit older than me.

i hate prisoners. what kind of slime are they, going to war and ending up captured? i get it, they’re rat-eaters, but still, it’s disgusting, no matter how you look at it... well, here we go: one of the scrawny ones doubled over and started vomiting. move it, move it, snake's milk! the second one started too. ugh! how these rats sense impending death, just like real rats. And now, they'd do anything to save themselves — betray, sell out, become slaves...

"double-time, march!" i barked in their language.

they started running. slowly, poorly. that blond one was limping. severely injured, i suppose; twisted his ankle in the latrine. no matter, you'll hobble along.

we reached the edge of the village, where the trucks were parked—the guys saw us and started shouting and whistling. i found a large puddle, shoved the prisoners face-first into the mud, and headed to the front truck where tick was. and there he was, jumping out to meet me — his face cheerful, the little mustache under his nose bristling, and a bone cigarette holder clenched in his teeth, just like the senior class fashion.

"well, what do you have to say, brother-doomsayer?" he says to me.

i reported to him: such and such, this is the situation, and the prisoners must be interrogated first. and then i added on my own:

"don't forget about me, tick," i said. "after all, i brought them here..."

"you're talking about the collar?" he asks absently, while looking around.

"exactly! who brought them here, after all?"

"i just don't see what we can use. we're not taking them all the way to the forest..."

"how about to use the stilts?"

"it's possible, of course, to use the stilts... but why?" he looked at me. "what if we do it without the stilts? would you take it on?"

well, there it is. just as i thought. i'm always unlucky. is it my fault that my wingman was kept at the headquarters? and how can i do it alone? i won't have the strength. i'll be struggling until evening and then scrubbing myself clean all night.

"you know," i said to tick. "i don't have a wingman."

"how about alone?" he asks. "got your cord with you?"

i got carried away with excitement here.

"will you hold them?" i ask.

he looked at me, and my heart sank immediately.

"kitten…" he says. "you're going to have fun here while cheetah is alone there? take three pairs and get to cheetah! quickly!"

there was nothing to be done. it wasn't meant to be, just unlucky. i took one last look at my striped ones, slung my rifle over my shoulder, and shouted at the top of my lungs:

"first, second, third pairs — to me!"

the kittens tumbled off the truck like peas: hare with rooster, nosy with crocodile, sniper with that one... what's his name... i wasn't used to him yet, he had just been transferred to us from the piggan school — he killed the wrong person there, so they sent him to us.

i had noticed it a long time ago but never mentioned it to anyone: if cat happened to off a civilian in a fit of anger, an order would immediately go out. so-and-so with such-and-such a nickname is to be executed for committing a criminal offense. they would drag him out to the parade ground, stand him before his best friends, fire a volley at him, toss his body into a truck for a dishonorable burial, and then you'd hear that the guys had seen him either on an operation or in another unit... and i think that's the right way to do it.

well, i commanded "double-time," and we hurried back to cheetah. and xheetah wasn't wasting any time there. i saw that pole of a man, the staff major, trotting towards us, and behind him was a column of about fifty porcupines with shovels and mattocks, their boots thudding, sweaty with steam rising off them. this meant cheetah had them digging a new position, a real one, for us. under the house opposite the medical unit, i saw shovels flashing, and there was a rocket launcher set up, and the whole village was bustling like the

main street on a name day. the porcupines were scurrying about, and not a single one was empty-handed: either carrying weapons, but those were few, and most were dragging crates of ammunition and machine gun mounts.

cheetah saw us and expressed his satisfaction. he immediately sent hare’s and sniper’s pairs into the jungle for advanced scouting, kept nosy and crocodile with him for communication, and said to me:

"gug. you're the best rocket launcher operator in the unit, and i’m counting on you. see those cockroaches? they’re yours. set up the rocket launcher on that edge, choose a position roughly where our trucks are now. camouflage well, and open fire when i set the village ablaze. move out, cat.”

when i heard all that, i didn't just run, i practically flew to my cockroaches. my cockroaches, along with the rocket launcher, were stuck in a muddy pothole in the middle of the road and looked like they intended to spend the entire war there. they were barely moving their paws, those lazybones. so, i gave one a smack, kicked another, and hit the third with the butt of my rifle between the shoulder blades, shouting so loudly that it rang in my own ears — my cockroaches started working properly, almost like humans. they lifted the rocket launcher out of the pothole by hand and — march, march — rolled it down the road, with wheels screeching and mud flying, and — straight into another pothole. this time, i had to pitch in myself. you see, guys, you can make porcupines work too, you just need to know how.

so, my situation was like this. i had already chosen a position – i remembered some thick ginger bushes near the trucks and a flat lowland behind them where we could easily dig in, hidden from any devils coming from the jungle. from there, i would see everything: the road up to the jungle, the whole village outskirts if they pushed through the houses, and the swamp on the left if the armored infantry tried to come through there... i also thought i should remember to ask tick for a few pairs to cover that side. i had twenty rockets in the trays, provided those clerks hadn't thrown any out to lighten their load on the way here... well, we'd check that out soon enough, and in any case, once we dug in, i'd have to send the cockroaches for more supplies. i hate having to ration during a fight. that's not a fight, i don't know what that is... we had enough time until dusk, and when they attacked at twilight, that wild village would light up, and i'd have them all in my sights – shoot at will. you won't regret counting on me, cheetah!

i finished this last thought almost automatically while lying on my back, and in the gray sky above me, burning fragments were flying like strange birds. i hadn't heard a shot or an explosion, and now i couldn't hear anything at all. i was deaf. i don't know how much time had passed, and then i sat up.

from the jungle, armored vehicles were crawling out four abreast, spitting fire and spreading out into a semicircle battle formation, with another four following behind them. the village was burning. smoke hung over the trenches ahead, and there wasn't a soul in sight. the field kitchen next to the trading post

was overturned, with its contents spilled out in a brown mess, steaming. my rocket launcher was also overturned, and the cockroaches were lying in a heap in the ditch. in short, i had taken up a perfect position, snake's milk!

then we were hit by a second barrage. it knocked me into the ditch, flipping me head over heels, filling my mouth with clay and my eyes with dirt. i had just gotten to my feet when the third barrage came. and it kept coming, and coming...

we managed to get the rocket launcher back on its wheels, rolled it into the ditch, and i burned one of the armored vehicles. there were only two cockroaches left now; where the third one went, i had no idea.

then, suddenly and without transition, i found myself on the road. ahead of me was a whole bunch of striped ones — close, very close, right next to me. the fire reflected blood-red off their blades. a machine gun was roaring deafeningly next to my ear, i had a knife in my hand, and someone was twitching at my feet, knocking against my knees...

then, meticulously, as if on the training ground, i aimed the rocket launcher at the steel shield advancing on me through the smoke. i could almost hear the instructor's command: "at the armored infantry... with armor-piercing rounds..." but I couldn't press the trigger because i had a knife in my hand again...

then suddenly, there was a lull. it was already twilight. it turned out that my rocket launcher was intact, and so was i. around me gathered a bunch of porcupines, about ten people. they were all smoking, and someone handed me a flask. who? hare? i don't

know... i remember that against the backdrop of a burning house about thirty steps away, there was a strange figure: everyone else was sitting or lying down, but this one was standing, and it looked like he was black and naked... he wasn't wearing any clothes — no coat, no jacket. or was he not naked after all? "hare, who is that standing there?" – "i don't know, i'm not hare." – "where's hare?" – "i don't know, just drink, drink..."

then we were digging, hurrying with all our might. this was already a different place. the village was now not to the side but ahead of us. well, there wasn't much of a village left — just piles of charred wood, but there were burning armored vehicles on the road. a lot of them. several. the swampy ground squelched underfoot... "i commend you, well done, cat..." — "sorry, cheetah, i'm not thinking straight. where are all our guys? why are there only porcupines here?" -– "everything's fine, gug, keep working, keep working, brave brother, everyone is safe, everyone is impressed with you..."

...aha! got it! right in the blunt snout. it's backing up, settling on its rear, throwing a shower of sparks into the black sky. they're running, running! "cat, on the right! on the right! now!" i don't see anything on the right, and i'm not looking. i turn the barrel that way, and suddenly a shower of liquid fire comes straight at me from the black-red haze. everything ignites at once — the corpses, the ground, the rocket launcher. and some bushes. and me. it hurts. an infernal pain. like baron tragg...

puddle, i need puddle! there was a puddle here! they were lying in it! i put them there, snake's milk, but i should have put them in the fire, in the fire! no puddle... the ground was burning, the ground was smoking, and suddenly someone with inhuman strength knocked it out from under my feet...