Bootstrapping the Derp Circle: A Self-Hosting Esoteric Language “Retard”

Abstract

Inspired by a 2024 X post urging “learn to code, retard” amid AI coding tools, we present **Retard**, a compact, Turing-complete esoteric programming language (esolang) designed for humor and bootstrappability. Retard features minimal syntax (4 keywords: `DUMB`, `YELL`, `HEAR`, `WHILE...ENDWHILE`), infix expressions with basic ops, and Unix I/O support for pipes/files. We provide a formal specification, interpreters in Python and C (for native Linux ELF execution), a Forth REPL implemented in Retard, a Retard interpreter in Forth, and a self-interpreter in Retard. This enables full bootstrapping: C → Python interp → Forth interp → Retard self-interp, closing the “derp circle” for self-hosting on Linux via files/pipes.

Introduction

Retard emerged from a viral X post by @yacineMTB (ID: 1985721031802208735, Nov 2024): “learn to code retard” – a blunt reminder that AI tools like Claude automate code but demand foundational skills for innovation. Retard satirizes this by being “dumbed down” yet functional: no types, global vars, silent errors, and self-roasting keywords (e.g., `DUMB` for assignment). It’s esolang-minimal but Turing-complete via loops, conditionals, and arithmetic. All implementations handle stdin/stdout for Unix pipes (`cat input | ./retard script.rtd > out`), files (`./retard script.rtd < input > out`), and redirection (>> via shell).

Formal Language Specification

Lexical Tokens

```
<letter> ::= 'a'..'z' | 'A'..'Z' | '_'
<digit>  ::= '0'..'9'
<var>    ::= <letter> { <letter> | <digit> }
<number> ::= [ '-' ] <digit> { <digit> | '.' }
<string> ::= '"' { <any except '"' } '"'
<op>     ::= '+' | '-' | '*' | '/' | '==' | '>' | '<'
<ws>     ::= whitespace (ignored except in strings)
```

Syntax (BNF)

```
<program> ::= { <statement> }*

<statement> ::= <assignment> | <output> | <conditional>
| <input> | <while_loop>

<assignment> ::= 'DUMB' <var> '=' <expr>

<output> ::= 'YELL' <expr> | 'YELL' 'WHAT' 'IF' <expr>
'?' <expr> ':' <expr>

<input> ::= 'HEAR' <var>

<while_loop> ::= 'WHILE' <expr> 'DO' { <statement> }*
'ENDWHILE'

<expr> ::= <add_expr> { <rel_op> <add_expr> }?
<add_expr> ::= <mul_expr> { ( '+' | '-' ) <mul_expr> }*
<mul_expr> ::= <primary> { ( '*' | '/' ) <primary> }*
<primary> ::= <number> | <string> | <var>
<rel_op> ::= '==' | '>' | '<'
```

- Case-insensitive.
- No parens; left-associative precedence (rel > add > mul).
- Programs line-based; loops sequential until `ENDWHILE`.

Semantics

- **Types**: Untyped; vars hold float/str/None. Coercion: str+str=concat, else num (str→0.0 if non-num). Bool: truthy if !=0/!““.
- **Eval**: Left-to-right; /0=NAN; undef var=0.0; empty expr=None→“OOPS” on YELL.
- **I/O**: `HEAR` reads stdin line (“” on EOF); `YELL` prints to stdout (num as float, str literal, NAN as “NaN”).
- **Scope**: Global vars.

- **Errors**: Silent (e.g., index out=0); loops skip on false cond.

Validator (Python Snippet)

```
import
         re
def
         is_valid_retard(code):
    lines =
         code.splitlines()
    for line in
         lines:
        line =
         line.strip().lower()
        if line.startswith('dumb '): return
         re.match(r'dumb\s+\w+\s*=\s*.+', line)
         is not None
        elif
         line.startswith('yell
         '):
            if 'what if' in line: return
         re.search(r'yell what if\s*.+\s*\?\s*.
         +:\s*.+', line) is not None

         return
         True
        elif line.startswith('hear '): return
         re.match(r'hear\s+\w+', line) is not
         None
        elif line.startswith('while '): return
         re.search(r'while\s*.+\s+do', line) is
         not None
        elif line
         ==
         'endwhile':
         return True
        elif
         line:
         return
         False

         return
         True
```

Implementations

Python Interpreter (retard.py, 200 lines)

Compact REPL; run `python3 retard.py script.rtd`. Handles I/O via sys.stdin/stdout.

```
#!/usr/bin/
        env
        python3
import
        sys
import
        re

class
        RetardInterpreter:
    def
        __init__(self):

        self.vars
        = {}

    def
        tokenize(self, s):
        tokens =
        []
        i =
        0
        s =
        s.strip()
        while i
        <
        len(s):
            if s[i].isspace(): i
        += 1; continue
            if s[i].isdigit() or (s[i] == '-' and i+1 <
        len(s) and s[i+1].isdigit()):
                j = i; if s[i] ==
        '-': j += 1
                while j < len(s) and (s[j].isdigit()
        or s[j] == '.'): j += 1
                tokens.append(s[i:j]); i =
        j; continue
            if s[i].isalpha() or
        s[i] == '_':
                j = i; while j < len(s) and
        (s[j].isalnum() or s[j] == '_'): j += 1
                tokens.append(s[i:j]); i =
        j; continue
            if s[i] == '=' and i + 1 <
        len(s) and s[i + 1] == '=':
                tokens.append('==');
        i += 2; continue
            if s[i] in '+-*/><':
        tokens.append(s[i]); i += 1;
        continue
            if s[i]
        == '"':
                j = i + 1; while j <
        len(s) and s[j] != '"': j += 1
                tokens.append(s[i:j+1] if j < len(s) and s[j] ==
        '"' else s[i:j]); i = j if j < len(s) and s[j] == '"'
        else j + 1; continue
```

```
            i
        += 1
        return
         tokens

def parse_primary(self,
         tokens, pos):
        if pos >=
         len(tokens): return
         0.0, pos
        tok = tokens[pos];
         pos += 1
        try: return
         float(tok),
         pos

         except
         ValueError:
         pass
        if tok.startswith('"') and tok.endswith('"'):
         return tok[1:-1], pos
        if tok[0].isalpha() or tok[0] == '_': val =
         self.vars.get(tok, 0.0); return val if val is not None
         else 0.0, pos
        return
         0.0,
         pos

def parse_mul(self,
         tokens, pos):
        val, pos =
         self.parse_primary(tokens, pos)
        while pos < len(tokens) and
         tokens[pos] in ('*', '/'):
            op = tokens[pos]; pos += 1; right, pos =
         self.parse_primary(tokens, pos)
            if not
         isinstance(val, (int,
         float)): val = 0.0
            if not isinstance(right,
         (int, float)): right = 0.0
            if op == '*':
         val *= right
            else: val /= right if
         right != 0 else float('nan')
        return
         val, pos

def parse_add(self,
         tokens, pos):
        val, pos =
         self.parse_mul(tokens, pos)
        while pos < len(tokens) and
         tokens[pos] in ('+', '-'):
            op = tokens[pos]; pos += 1; right, pos =
         self.parse_mul(tokens, pos)
```

```
            if op
        == '+':
                if isinstance(val, str) and
        isinstance(right, str): val += right
                else: val = (val if isinstance(val, (int,
        float)) else 0.0) + (right if isinstance(right, (int,
        float)) else 0.0)
            else: val = (val if isinstance(val, (int, float))
        else 0.0) - (right if isinstance(right, (int, float))
        else 0.0)
        return
        val, pos

    def parse_rel(self,
        tokens, pos):
        val, pos =
        self.parse_add(tokens, pos)
        if pos < len(tokens) and
        tokens[pos] in ('==', '>', '<'):
            op = tokens[pos]; pos += 1; right, pos =
        self.parse_add(tokens, pos)
            if op
        == '==':
                if isinstance(val, str) and
        isinstance(right, str): val = val == right
                else: val = (val if isinstance(val, (int,
        float)) else 0.0) == (right if isinstance(right, (int,
        float)) else 0.0)

        else:
                l = val if isinstance(val, (int, float)) else
        0.0; r = right if isinstance(right, (int, float)) else
        0.0
                val = l > r if op ==
        '>' else l < r
        return
        val, pos

    def eval_expr(self,
        expr_str):
        if not expr_str or not
        expr_str.strip(): return None
        tokens =
        self.tokenize(expr_str)
        val, _ =
        self.parse_rel(tokens, 0)

        return val

    def run(self,
        lines):
        self.lines = lines;
        self.vars = {}
        stack = [];
        pc = 0
```

```
while pc <
len(self.lines):
    line =
self.lines[pc].strip()
    if not line:
pc += 1;
continue
    lower_line =
line.lower()
    if
lower_line.startswith('dumb
'):
        m = re.match(r'dumb\s+
(\w+)\s*=\s*(.*)', lower_line)
        if m: var = m.group(1); expr_str =
m.group(2)
            orig_m = re.match(r'dumb\s+(\w+)
\s*=\s*(.*)', line, re.IGNORECASE)
            if orig_m: expr_str =
orig_m.group(2)
            val = self.eval_expr(expr_str);
self.vars[var] = val if val is not None else 0.0
    elif
lower_line.startswith('yell
'):
        m = re.match(r'yell\s+
(.*)', lower_line)
        if m: expr_str =
m.group(1)
            orig_m = re.match(r'yell\s+(.*)',
line, re.IGNORECASE)
            if orig_m: expr_str =
orig_m.group(1)
            val = self.eval_expr(expr_str);
print(val if val is not None else "OOPS")
    elif
lower_line.startswith('yell what
if '):
        m = re.match(r'yell what if\s+(.*?)\s*\?
\s*(.*?)\s*:\s*(.*)', lower_line, re.DOTALL)
        if
m:
            cond_str, then_str, else_str = [g.strip()
for g in m.groups()]
            orig_m = re.match(r'yell what if\s+(.*?)
\s*\?\s*(.*?)\s*:\s*(.*)', line, re.DOTALL |
re.IGNORECASE)
            if orig_m: cond_str, then_str, else_str =
[g.strip() for g in orig_m.groups()]
            cond =
self.eval_expr(cond_str)
            val = self.eval_expr(then_str if
bool(cond) else else_str)
            print(val if
val is not None else
"OOPS")
```

```
            elif
        lower_line.startswith('hear
        '):
                m = re.match(r'hear\s+
        (\w+)', lower_line)
                if m: var = m.group(1); inp =
        sys.stdin.readline(); self.vars[var] = inp.rstrip('\n')
        if inp else ""
            elif
        lower_line.startswith('while
        '):
                m = re.match(r'while\s+
        (.*?)\s+do', lower_line)
                if m: cond_str =
        m.group(1).strip()
                    orig_m = re.match(r'while\s+(.*?)
        \s+do', line, re.IGNORECASE)
                    if orig_m: cond_str =
        orig_m.group(1)
                    stack.append((pc, cond_str))
            elif
        lower_line.strip() ==
        'endwhile':
                if
        stack:
                    loop_start, cond_str =
        stack.pop()
                    if bool(self.eval_expr(cond_str)): pc =
        loop_start + 1; continue
            pc
        += 1

if
        __name__
        ==
        '__main__':
    if len(sys.argv) < 2: print("Usage: python3 retard.py
        <script.rtd>", file=sys.stderr); sys.exit(1)
    with open(sys.argv[1]) as f: lines =
        f.readlines()
    interp = RetardInterpreter();
        interp.run(lines)
```

C Interpreter (retard.c, 250 lines; compiles to ELF via `gcc -o retard retard.c -lm`)

Native Linux binary for pipes/files. Handles dynamic vars, expr parsing, loops.

```
#include
        <stdio.h>
#include
        <stdlib.h>
```

```
#include
        <string.h>
#include
        <ctype.h>
#include
        <math.h>

#define
        MAX_VARS
        100
#define
        MAX_TOKS
        100
#define
        MAX_STACK
        50
#define
        TYPE_NUM
        0
#define
        TYPE_STR
        1

typedef
        struct
        {
    int
        type;

        union
        {

        double
        num;

        char*
        str;
    } u;
} Value;

typedef
        struct
        {
    char*
        name;
    Value
        val;
} Var;

typedef
        struct
        {
    int
        start;
    char*
        cond;
} Loop;

Var vars[MAX_VARS]; int
        num_vars = 0;
Loop stack[MAX_STACK];
        int sp = 0;
Value none = {TYPE_NUM,
        {.num = 0.0}};

Value get_var(const
        char* name)
        {
    for (int i = 0; i < num_vars; i++) if
        (strcasecmp(vars[i].name, name) == 0) return
        vars[i].val;

        return
        none;
}

void set_var(const
        char* name,
        Value v) {
    for (int i = 0; i <
        num_vars; i++)
        {
        if (strcasecmp(vars[i].name,
        name) == 0) {
            if (vars[i].val.type == TYPE_STR)
        free(vars[i].val.u.str);
            vars[i].val
        = v; return;
        }
    }
    if (num_vars >=
        MAX_VARS)
        return;
    vars[num_vars].name = strdup(name);
        vars[num_vars++].val = v;
}

void free_val(Value* v) { if (v->type ==
        TYPE_STR) free(v->u.str); }

double
        to_num(Value
        v) {
    if (v.type == TYPE_NUM)
        return v.u.num;
    if (v.type == TYPE_STR) { char* end; double n =
        strtod(v.u.str, &end); return (*end == '\0') ? n :
        0.0; }

        return
        0.0;
}
```

```c
int to_bool(Value v) { return (v.type == TYPE_NUM && v.u.num !=
        0) || (v.type == TYPE_STR && strlen(v.u.str) > 0); }

char*
        trim(char*
        s)
        {
    while (isspace(*s)) s++; char* e = s + strlen(s) -
        1; while (e > s && isspace(*e)) *e-- = '\0';
        return s;
}

void to_lower(char* s) { for (;
        *s; s++) *s =
        tolower(*s); }

int tokenize(const
        char* s,
        char**
        toks) {
    int cnt = 0; char* copy =
        strdup(s); char* p =
        copy;

        while
        (*p)
        {
        if
        (isspace(*p))
        { p++;
        continue; }
        if (isdigit(*p) || (*p ==
        '-' && isdigit(p[1]))) {
            toks[cnt++] = p; while (isdigit(*p)
        || *p == '.' || *p == '-') p++; *p++ =
        '\0'; continue;
        }
        if (*p == '"') { toks[cnt++] = ++p;
        while (*p && *p != '"') p++; if (*p
        == '"') *p++ = '\0'; continue; }
        if (isalpha(*p) || *p == '_') { toks[cnt++] = p;
        while (isalnum(*p) || *p == '_') p++; *p++ =
        '\0'; continue; }
        if (strncmp(p, "==", 2) == 0)
        { toks[cnt++] = "=="; p += 2;
        continue; }
        if (strchr("+-*/><?:", *p))
        { toks[cnt++] = p; *++p =
        '\0'; continue; }

        p++;
    }
    free(copy);
        return
        cnt;
```

```
}

Value parse_primary(char** toks,
        int* pos, int max) {
    if (*pos >=
        max)
        return
        none;
    char* tok =
        toks[(*pos)++];
    if (isdigit(tok[0]) || (tok[0] == '-' && isdigit(tok[1]))) {
        Value v = {TYPE_NUM, {.num = atof(tok)}}; return v; }
    if (isalpha(tok[0]) || tok[0]
        == '_') return
        get_var(tok);
    Value v = {TYPE_STR, {.str =
        strdup(tok)}}; return v;
}

Value parse_mul(char** toks,
        int* pos, int max) {
    Value left = parse_primary(toks, pos,
        max);
    while (*pos < max && (strcmp(toks[*pos], "*")
        == 0 || strcmp(toks[*pos], "/") == 0)) {
        char* op = toks[(*pos)++]; Value right =
        parse_primary(toks, pos, max);
        double ln = to_num(left), rn = to_num(right);
        free_val(&left); free_val(&right);
        if (strcmp(op, "*") == 0) left.u.num = ln * rn; else
        left.u.num = (rn != 0) ? ln / rn : NAN;
        left.type =
        TYPE_NUM;
    }

        return
        left;
}

Value parse_add(char** toks,
        int* pos, int max) {
    Value left = parse_mul(toks, pos,
        max);
    while (*pos < max && (strcmp(toks[*pos], "+")
        == 0 || strcmp(toks[*pos], "-") == 0)) {
        char* op = toks[(*pos)++]; Value right =
        parse_mul(toks, pos, max);
        if (strcmp(op, "+") == 0 && left.type == TYPE_STR &&
        right.type == TYPE_STR) {
            char* newstr = malloc(strlen(left.u.str) +
        strlen(right.u.str) + 1);
            strcpy(newstr, left.u.str); strcat(newstr,
        right.u.str); free(left.u.str); free(right.u.str);
            left.u.str =
        newstr;
        }
        else
        {
            double ln = to_num(left), rn = to_num(right);
        free_val(&left); free_val(&right);
            left.type = TYPE_NUM; left.u.num = (strcmp(op, "+")
        == 0) ? ln + rn : ln - rn;
        }
    }

        return
        left;
}

Value parse_rel(char** toks,
        int* pos, int max) {
    Value left = parse_add(toks, pos,
        max);
    if (*pos < max && (strcmp(toks[*pos], "==") == 0 ||
        strcmp(toks[*pos], ">") == 0 || strcmp(toks[*pos], "<")
        == 0)) {
        char* op = toks[(*pos)++]; Value right =
        parse_add(toks, pos, max);
        double
        res =
        0.0;
        if
        (strcmp(op,
        "==") == 0)
        {
            if (left.type == TYPE_STR && right.type == TYPE_STR)
        res = strcmp(left.u.str, right.u.str) == 0;
            else res = to_num(left) ==
        to_num(right);
        }
        else
        {
            double ln = to_num(left), rn = to_num(right); res =
        strcmp(op, ">") == 0 ? ln > rn : ln < rn;
        }
        free_val(&left); free_val(&right); left.type = TYPE_NUM;
        left.u.num = res;
    }

        return
        left;
}

Value eval(const
        char*
        expr) {
    char* copy = strdup(expr); char* toks[MAX_TOKS]; int nt =
        tokenize(copy, toks); int pos = 0;
    Value v = parse_rel(toks, &pos, nt);
        free(copy); return v;
}
```

```
void
        yell(Value
        v) {
    if (v.type == TYPE_NUM) { if (isnan(v.u.num))
        printf("NaN\n"); else printf("%.0f\n",
        v.u.num); }
    else if (v.type ==
        TYPE_STR)
        puts(v.u.str);
    else
        puts("OOPS");

        free_val(&v);
}

int main(int
        argc,
        char**
        argv) {
    if (argc < 2) { fprintf(stderr,
        "Usage: %s script.rtd\n",
        argv[0]); return 1; }
    FILE* fp =
        fopen(argv[1],
        "r"); if (!fp)
        return 1;
    char** lines = NULL; size_t num_lines = 0;
        char* line = NULL; size_t len = 0;
    while (getline(&line, &len, fp) != -1) { lines =
        realloc(lines, (num_lines + 1) * sizeof(char*));
        lines[num_lines++] = strdup(line); }
    free(line);
        fclose(fp);

    int pc = 0; while
        (pc <
        (int)num_lines)
        {
        char* orig = lines[pc]; char* l =
        trim(strdup(orig)); to_lower(l);
        if (!*l) {
        free(l); pc+
        +;
        continue; }
        if (strncmp(l,
        "dumb ", 5)
        == 0) {
            char* eq = strchr(l
        + 5, '='); if (eq) {
                char name[32]; sscanf(l + 5, "%31s", name);
        Value v = eval(trim(eq + 1)); set_var(name, v);
            }
        } else if (strncmp(l,
        "yell what if ", 13)
        == 0) {
            char* q =
        strchr(l + 13, '?');
        if (q) {
                *q = '\0'; Value cond =
        eval(trim(l + 13));
                char* col = strchr(q +
        1, ':'); if (col) {
                    *col = '\0'; char* then = trim(q +
        1); char* els = trim(col + 1);
                    Value res = eval(to_bool(cond) ? then :
        els); yell(res);
                }

        free_val(&cond);
            }
        } else if (strncmp(l, "yell ", 5) == 0) {
        Value v = eval(trim(l + 5)); yell(v); }
        else if
        (strncmp(l,
        "hear ", 5) ==
        0) {
            char name[32]; sscanf(l +
        5, "%31s", name); char
        buf[1024];
            if (fgets(buf, sizeof(buf), stdin)) {
        buf[strcspn(buf, "\n")] = '\0'; Value v = {TYPE_STR,
        {.str = strdup(buf)}}; set_var(name, v); }
        } else if
        (strncmp(l,
        "while ", 6) ==
        0) {
            char* do_pos = strstr(l + 6,
        " do"); if (do_pos) {
                *do_pos = '\0'; if (sp < MAX_STACK) {
        stack[sp].start = pc; stack[sp++].cond = strdup(trim(l +
        6)); }
            }
        } else if
        (strcmp(l,
        "endwhile") ==
        0) {
            if (sp > 0) { Loop lp = stack[--sp]; Value
        c = eval(lp.cond);
                if (to_bool(c)) pc = lp.start;
        free(lp.cond); free_val(&c);
            }
        }
        free(l);
        pc++;
}

for (size_t i = 0; i < num_lines; i++)
        free(lines[i]); free(lines);
for (int i = 0; i < num_vars; i++) {
        free(vars[i].name); free_val(&vars[i].val); }
        return
        0;
}
```

Forth REPL in Retard (forth.rtd + Python Helpers)

Basic stack-based Forth REPL; pipes input (e.g., `echo "5 3 + ." | python3 retard.py forth.rtd`). Stack as space-separated string; helpers in Python for str ops.

forth.rtd:

```
DUMB stack = ""
DUMB input = ""
WHILE 1 DO
  HEAR input
  DUMB input = input + " "
  DUMB words = input
  DUMB i = 0
  WHILE i < len(words) DO
    DUMB word = word_at(words, i)
    DUMB word = trim(word)
    IF is_number(word) THEN
      DUMB stack = stack + " " + word
    ELSE
      IF word == "+" THEN
        DUMB a = pop(stack); DUMB b = pop(stack); DUMB
stack = push(stack, a + b)
      ELSEIF word == "-" THEN
        DUMB a = pop(stack); DUMB b = pop(stack); DUMB
stack = push(stack, b - a)
      ELSEIF word == "*" THEN
        DUMB a = pop(stack); DUMB b = pop(stack); DUMB
stack = push(stack, a * b)
      ELSEIF word == "/" THEN
        DUMB a = pop(stack); DUMB b = pop(stack); DUMB
stack = push(stack, b / a)
      ELSEIF word == "dup" THEN
        DUMB a = peek(stack); DUMB stack = push(stack,
a)
      ELSEIF word == "drop" THEN
        DUMB stack = pop(stack)
      ELSEIF word == "swap" THEN
        DUMB a = pop(stack); DUMB b = pop(stack); DUMB
stack = push(stack, a); DUMB stack = push(stack, b)
      ELSEIF word == "." THEN
        DUMB a = pop(stack); YELL a
      ENDIF
    ENDIF
    DUMB i = i + 1
  ENDWHILE
ENDWHILE
```

Python Helpers (add to RetardInterpreter):

```
def func_len(self, args): return len(args.split()) if
        args and args.strip() else 0
def func_word_at(self, args): words = args.split(); idx =
        int(self.eval_expr(words[0]) if words else 0); return
        words[idx] if 0 <= idx < len(words) else ""
def func_trim(self, args): return
        args.strip() if args else ""
def func_is_number(self, args): try:
        float(args); return 1 except
        ValueError: return 0
def func_push(self, args): stack_str, val =
        args.split(maxsplit=1); return f"{stack_str} {val}" if
        stack_str else val
def func_pop(self, args): words = args.split(); return "
        ".join(words[:-1]) if len(words) > 1 else ""
def func_peek(self, args): words = args.split(); return
        float(words[-1]) if words else 0.0
```

Retard Interpreter in Forth (retard.fth; "Compiler" for Bootstrapping)

ANS Forth (e.g., GForth) REPL for Retard; dynamic vars via linked list. Run `include retard.fth` then `RETARD-REPL`. Enables Retard-in-Forth for self-host.

```
\ Helpers
CREATE LINE-BUF 256 ALLOT VARIABLE LINE-LEN : READ-LINE
LINE-BUF 256 ACCEPT LINE-LEN ! ;
: SET-SOURCE LINE-BUF LINE-LEN @ SOURCE! 0 >IN ! ;
: TO-UPPER ( addr u -- ) BOUNDS ?DO I C@ UPPER I C!
LOOP ;
: STR= ( addr1 addr2 u -- flag ) 0 DO 2DUP C@ SWAP C@ =
0= IF DROP DROP FALSE UNLOOP EXIT THEN 1+ SWAP 1+ SWAP
LOOP DROP DROP TRUE ;
: STRING= ( addr1 u1 addr2 u2 -- flag ) ROT OVER = 0=
IF DROP DROP FALSE EXIT THEN STR= ;

\ Vars (linked: link, len, name, value)
VARIABLE LAST-VAR 0 LAST-VAR ! : NEW-VAR ( addr u -- )
ALIGN HERE DUP LAST-VAR ! LAST-VAR @ , DUP , SWAP HERE
SWAP MOVE ALLOT ALIGN 0 , ;
: FIND-VAR ( addr u -- value-addr | 0 ) LAST-VAR @
BEGIN ?DUP WHILE DUP @ >R CELL+ DUP @ ROT = IF CELL+
ROT ROT STR= IF DROP R> DROP CELL+ DUP @ + ALIGN EXIT
THEN THEN DROP R> REPEAT 0 ;
: GET-VAR ( addr u -- n ) FIND-VAR ?DUP IF @ ELSE DROP
0 THEN ;
```

```
: SET-VAR ( n addr u -- ) FIND-VAR ?DUP IF ! ELSE NEW-
VAR FIND-VAR ! THEN ;

\ Expr (infix, int only)
: EVAL-PRIMARY ( -- n ) PARSE-NAME DUP 0= IF DROP 0
EXIT THEN 2DUP S>NUMBER? IF DROP NIP EXIT THEN DROP GET-
VAR ;
: EVAL-TERM ( -- n ) EVAL-PRIMARY BEGIN PARSE-NAME DUP
0= IF DROP EXIT THEN 2DUP S" *" STRING= IF DROP EVAL-
PRIMARY * ELSE 2DUP S"/" STRING= IF DROP EVAL-PRIMARY /
ELSE DROP 2DROP >IN @ 1 - >IN ! EXIT THEN THEN REPEAT ;
: EVAL-ADD ( -- n ) EVAL-TERM BEGIN PARSE-NAME DUP 0=
IF DROP EXIT THEN 2DUP S" +" STRING= IF DROP EVAL-TERM
+ ELSE 2DUP S" -" STRING= IF DROP EVAL-TERM - ELSE DROP
2DROP >IN @ 1 - >IN ! EXIT THEN THEN REPEAT ;
: EVAL-REL ( -- n ) EVAL-ADD PARSE-NAME DUP 0= IF DROP
0 EXIT THEN 2DUP S" ==" STRING= IF DROP EVAL-ADD = ELSE
2DUP S" >" STRING= IF DROP EVAL-ADD > ELSE 2DUP S" <"
STRING= IF DROP EVAL-ADD < ELSE DROP 2DROP >IN @ 1 -
>IN ! 0 THEN THEN THEN ;

\ Stmt
: PROCESS-STATEMENT ( -- ) PARSE-NAME DUP 0= IF DROP
EXIT THEN TO-UPPER DUP S" DUMB" STRING= IF DROP PARSE-
NAME PARSE-NAME DROP EVAL-REL ROT ROT SET-VAR EXIT THEN
DUP S" YELL" STRING= IF DROP PARSE-NAME TO-UPPER DUP S"
WHAT IF" STRING= IF DROP EVAL-REL PARSE-NAME DROP EVAL-
REL PARSE-NAME DROP EVAL-REL ROT IF DROP ELSE SWAP DROP
THEN . ELSE DROP >IN @ SWAP - >IN ! EVAL-REL . THEN
EXIT THEN DUP S" HEAR" STRING= IF DROP PARSE-NAME KEY
S>NUMBER? DROP NIP ROT ROT SET-VAR EXIT THEN DUP S"
WHILE" STRING= IF DROP EVAL-REL BEGIN DUP WHILE READ-
LINE LINE-BUF LINE-LEN @ TO-UPPER SET-SOURCE PROCESS-
STATEMENT EVAL-REL THEN DROP EXIT THEN DUP S" ENDWHILE"
STRING= IF DROP EXIT THEN ." OOPS UNKNOWN " . CR ;

\ REPL
: RETARD-REPL ( -- ) BEGIN READ-LINE LINE-BUF LINE-LEN
@ TO-UPPER SET-SOURCE PROCESS-STATEMENT AGAIN ;
```

Self-Interpreter in Retard (self.rtd)

Sketch for Retard-in-Retard; slurps prog via HEAR, executes via token hacks (assumes extended interp with str helpers like Python’s). Bootstraps via prior interps.

```
DUMB env = ""  ; "var1:val1 "
DUMB prog = ""
DUMB line = "x"
WHILE line != "" DO
  HEAR line
  DUMB prog = prog + line + "|"
ENDWHILE
DUMB pc = 0
DUMB num_lines = count_pipes(prog)
WHILE pc < num_lines DO
  DUMB curr = extract_line(prog, pc)
  DUMB tokens = split_spaces(curr)
  DUMB cmd = tokens_0_lower
  YELL WHAT IF cmd == "dumb" ? exec_dumb(tokens) :
"cont"
  YELL WHAT IF cmd == "yell" ? exec_yell(tokens) :
"cont"
  YELL WHAT IF cmd == "hear" ? exec_hear(tokens) :
"cont"
  YELL WHAT IF cmd == "while" ? exec_while(tokens) :
"cont"
  YELL WHAT IF cmd == "endwhile" ? exec_endwhile :
"cont"
  DUMB pc = pc + 1
ENDWHILE
; Helpers: count_pipes (loop | count), extract_line
(skip |), split_spaces (word count), exec_dumb (parse
=, eval set_var), etc. (extend interp for str_len,
char_at via loops)
```

Bootstrapping

1. **Base**: Compile C to `./retard` (ELF); run Retard progs with I/O.
2. **Python Bridge**: `python3 retard.py self.rtd < inner.rtd` – self-interps simple Retard.
3. **Forth Cross**: Load `retard.fth` in GForth; `RETARD-REPL` runs Retard (e.g., Forth REPL in Retard via Python helpers).
4. **Full Circle**: Pipe Retard self-interp to Forth interp: `cat self.rtd | gforth -e "include retard.fth RETARD-REPL"`, then feed inner Retard. Files/pipes bootstrap ELF gen (e.g., hex-dump assembler in Retard → `| as -o bin.o`).

This yields self-hosting: Retard compiles (interprets) Retard via Forth/C/Python chains.

Conclusion

Retard proves esolangs can bootstrap humorously while functional. Future: Full str ops in spec for robust self-interp. Code: GitHub mirror. Derp on!