Introduction

↻ Reset Interpreter 339 code blocks • state persists between runs • use ▶▶ to auto-run dependencies

> **Download PDF**: The BioLang Programming Language (PDF)

BioLang is a domain-specific language built for bioinformatics. It is not a general-purpose language that happens to have biology libraries — every feature, from its type system to its operators, exists because bioinformatics workflows demand it.

Why a DSL?

Bioinformaticians spend most of their time gluing tools together: reading FASTA files, filtering variants, piping data through transformations, querying biological databases, and building reproducible pipelines. General-purpose languages require dozens of imports, boilerplate, and framework knowledge before you can write your first useful script.

BioLang eliminates that friction. A complete variant filtering workflow:

#1 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
read_vcf("data/variants.vcf")
    |> filter(|v| v.qual > 30)
    |> filter(|v| is_snp(v))
    |> filter(|v| is_transition(v))
    |> map(|v| {chrom: v.chrom, pos: v.pos, ref: v.ref_allele, alt:
v.alt_allele, qual: v.qual})
    |> write_tsv("filtered_transitions.csv")
```

No imports. No boilerplate. No framework. DNA, RNA, proteins, intervals, variants, and quality scores are first-class types — not strings pretending to be biology.

Design Principles

Pipe-first. The `|>` operator is the backbone of BioLang. Bioinformatics is inherently a series of transformations: raw reads become aligned reads become variant calls become annotated reports. Pipes make this natural:

#2 ▶ Run ▶▶ Run All Above + This

```
samples
    |> par_map(|s| read_fastq(s.path))
    |> map(|reads| filter(reads, |r| mean(r.quality) > 20))
    |> map(|reads| gc_content(reads))
    |> summarize(|key, vals| {mean_gc: mean(vals)})
```

Biology is built in. DNA, RNA, protein, and quality score literals are part of the syntax. Genomic intervals and variants have dedicated types with domain-specific methods. You do not need to install a package to reverse-complement a sequence:

#3 ▶ Run ▶▶ Run All Above + This

```
let seq = dna"ATCGATCGATCG"
let rc = reverse_complement(seq)    # dna"CGATCGATCGAT"
let rna = transcribe(seq)           # rna"AUCGAUCGAUCG"
let protein = translate(rna)        # protein"IDRS"
```

Tables are native. Bioinformatics lives in tables — sample sheets, count matrices, variant lists, BED files. BioLang tables support column operations, grouping, summarization, and joins without any library:

#4 ▶ Run ▶▶ Run All Above + This

```
let counts = csv("gene_counts.csv")
counts
    |> mutate("log_count", |row| log2(row.count + 1))
    |> group_by("sample")
    |> summarize(|sample, rows| {mean_expr: mean(col(rows, "log_count")),
n_genes: len(rows)})
```

Fifteen bio databases at your fingertips. NCBI, Ensembl, UniProt, UCSC, KEGG, STRING, PDB, Reactome, Gene Ontology, COSMIC, BioMart, NCBI Datasets, nf-core, BioContainers, and Galaxy ToolShed are built-in — no API keys required for most (NCBI key optional for higher rate limits, COSMIC key required):

#5 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# requires: internet connection
# requires: NCBI_API_KEY (optional, increases rate limit)
let gene = ncbi_gene("TP53")
let pathways = reactome_pathways("TP53")
let interactions = string_network(["TP53"], 9606)
```

Pipelines are natural. Data flows through pipes just like a bioinformatics workflow. No separate workflow engine needed — compose operations with `|>` and use `group_by`, `count_by`, and `filter_by` for efficient data processing:

#6 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# Complete QC pipeline in 5 lines
let reads = read_fastq("data/reads.fastq") |> collect()
let passed = reads |> filter(|r| mean_phred(r.quality) >= 25)
let gc_vals = passed |> map(|r| gc_content(r.seq))
println(f"Passed: {len(passed)}/{len(reads)}")
println(f"Mean GC: {mean(gc_vals)}")
```

Fast without trying. BioLang is implemented in Rust with native I/O. On benchmarks against Python (BioPython) and R (Bioconductor) across 30 tasks, BioLang is up to **7.1x** faster on ENCODE peak overlap, **7.0x** on protein k-mers, **6.7x** on FASTA parsing, **3.2x** on k-mer counting — while using 50–70% fewer lines of code. Python wins on text-heavy VCF/CSV parsing. See Benchmarks & Correctness for full results, methodology, and three-way correctness validation.

Streams handle scale. File readers return lazy streams. Process a 100 GB FASTQ without loading it into memory. Parallel maps distribute work across cores:

#7 ▶ Run ▶▶ Run All Above + This

```
read_fastq("data/reads.fastq")
    |> filter(|r| mean(r.quality) > 25)
    |> par_map(|r| gc_content(r.seq))
    |> summarize(|key, vals| {mean_gc: mean(vals), n_reads: len(vals)})
```

Who This Book Is For

This book is for bioinformaticians, computational biologists, and genomics researchers who want to write analysis scripts faster and more clearly. You do not need deep programming experience — if you have used R, Python, or shell scripts for bioinformatics, you will feel at home.

Every example in this book uses real biological data and real analysis tasks. There are no toy examples. By the end, you will be able to:

- Process sequencing data (FASTA, FASTQ, BAM, VCF, BED, GFF)
- Query biological databases directly from your scripts
- Build reproducible analysis pipelines

- Perform statistical analysis on expression and variant data
- Scale to large datasets using streams and parallel operations
- Extend BioLang with plugins in Python, R, or TypeScript

Running the Examples

Install BioLang:

```
cargo install biolang
```

Or build from source:

```
git clone https://github.com/oriclabs/biolang
cd biolang
cargo build --release
```

Run any example from this book:

```
bl run example.bl
```

Or use the interactive REPL:

```
bl repl
```

Every block labeled `biolang` or `bio` is syntax-checked against the current BioLang parser. Conceptual workflows and deliberate error demonstrations are labeled as text rather than executable BioLang.

Sample Data

Many examples in this book reference files like `contigs.fa`, `reads.fq`, `calls.vcf`, and `gene_counts.csv`. BioLang ships with sample data for all of these in the `examples/sample-data/` directory:

```
examples/sample-data/
  contigs.fa          4 assembled contigs (FASTA)
  reads.fq            8 sequencing reads with quality scores (FASTQ)
  calls.vcf           12 variants across 5 chromosomes (VCF)
  promoters.bed       10 promoter regions (BED)
  chip_peaks.bed      7 ChIP-seq peaks (BED)
  annotations.gff     16 gene features for 6 genes (GFF3)
  samples.csv         6-sample sheet with conditions and batches (CSV)
  gene_counts.csv     15 cancer genes × 6 samples expression matrix (CSV)
  counts.tsv          8 genomic region read counts (TSV)
  data.sam            6 SAM alignment records with headers (SAM)
```

To run the quickstart script that exercises all sample data:

```
bl run examples/quickstart.bl
```

When a book example references a bare filename like `read_fasta("data/sequences.fasta")`, you can substitute the sample data path:

#8 ▶ Run ▶▶ Run All Above + This

```
let sequences = read_fasta("data/sequences.fasta")
```

Chapter 1: Getting Started

BioLang is a pipe-first domain-specific language built for bioinformatics workflows. This chapter walks you through installation, the interactive REPL, running scripts, and writing your first real analysis.

Installation

From crates.io

```
cargo install biolang
```

This installs the `bl` binary, which provides both the REPL and the script runner.

From source

```
git clone https://github.com/oriclabs/biolang.git
cd biolang
cargo build --release
cp target/release/bl ~/.local/bin/
```

Verify installation

```
bl --version
```

Updating

BioLang has built-in update checking. Run `bl version` to see the current version and check if a newer release is available:

```
bl version
# BioLang v0.3.0
# Checking for updates... up to date.
```

To upgrade to the latest release:

```
bl upgrade
```

This downloads the correct binary for your platform from GitHub Releases and replaces the current `bl` executable.

BioLang also checks for updates automatically in the background when you run `bl run` or `bl repl`. If a newer version is available, a one-line notice appears on stderr. This check runs at most once per 24 hours and never blocks startup. Disable it with:

```
export BIOLANG_NO_UPDATE_CHECK=1
```

The REPL

Launch the interactive REPL:

```
bl
```

You will see the BioLang prompt:

```
BioLang v0.3.0
Type :help for commands, :quit to exit.
bl>
```

Try evaluating a bio literal directly:

```
bl> dna"ATCGATCG" |> gc_content()
0.5
```

REPL Commands

The REPL supports several meta-commands, all prefixed with `:`.

`:env` – Inspect current bindings

```
bl> let ref_genome = "GRCh38"
bl> let min_mapq = 30
bl> :env
ref_genome : Str = "GRCh38"
min_mapq   : Int = 30
```

`:reset` – Clear all bindings

```
bl> :reset
Environment cleared.
```

`:load` and `:save` – Session persistence

Load a script into the current session, executing every statement:

```
bl> :load preprocessing.bl
Loaded 42 bindings from preprocessing.bl
```

Save the current session bindings to a file:

```
bl> :save my_session.bl
Saved 12 bindings to my_session.bl
```

`:time` – Benchmark an expression

```
bl> :time read_fastq("data/reads.fastq") |> filter(|r| mean(r.quality) >= 30)
|> len()
Result: 1847293
Elapsed: 4.38s
```

`:type` – Check the type of an expression

```
bl> :type dna"ATCG"
DNA
bl> :type {chrom: "chr1", start: 100, end: 200}
Record{chrom: Str, start: Int, end: Int}
```

`:plugins` – List available plugins

```
bl> :plugins
fastq      read_fastq, write_fastq
fasta      read_fasta, write_fasta
sam        read_sam, read_bam
vcf        read_vcf, write_vcf
bed        read_bed, write_bed
table      csv, tsv, write_tsv
```

`:profile` – Profile an expression

```
bl> :profile read_fasta("data/sequences.fasta") |> filter(|r| seq_len(r.seq)
> 1000) |> len()
Total:     2.14s
  read:    1.87s (87.4%)
  filter:  0.26s (12.1%)
  len:     0.01s (0.5%)
Result: 24891
```

Running Scripts

BioLang scripts use the `.bl` extension. Run a script with:

```
bl run gc_analysis.bl
```

Pass arguments to a script:

```
bl run qc_report.bl -- --input sample.fastq.gz --min-quality 20
```

Arguments are available inside the script via the `args` record:

#9 ▶ Run ▶▶ Run All Above + This

```
# qc_report.bl
let input_file = args.input
let min_qual = into(args.min_quality ?? "20", "Int")

let reads = read_fastq(input_file)
  |> filter(|r| mean_phred(r.quality) >= min_qual)

print(f"Passing reads: {len(reads)}")
print(f"Mean quality: {reads |> map(|r| mean_phred(r.quality)) |> mean()}")
```

Your First Script: FASTA GC Content Analyzer

BioLang includes sample data in `examples/sample-data/` — see the Introduction for the full list. The script below uses `examples/sample-data/contigs.fa`.

Create a file called `gc_scan.bl`:

#10 ▶ Run ▶▶ Run All Above + This

```
# gc_scan.bl
# Read a FASTA file, compute per-sequence GC content, report statistics.

let sequences = read_fasta("examples/sample-data/contigs.fa")

# Compute GC content for each sequence
let gc_table = sequences
  |> map(|seq| {
    name: seq.id,
    length: seq_len(seq.seq),
    gc: gc_content(seq.seq)
  })
  |> table()

# Summary statistics
let gc_vals = col(gc_table, "gc")
let mean_gc = mean(gc_vals)
let std_gc = stdev(gc_vals)
let min_gc = min(gc_vals)
let max_gc = max(gc_vals)
let n_seqs = len(gc_vals)

print(f"Analyzed {n_seqs} sequences")
print(f"GC content: {mean_gc:.3f} (range: {min_gc:.3f} - {max_gc:.3f})")
print(f"Standard deviation: {std_gc:.4f}")

# Flag outlier contigs (GC > 2 std devs from mean)
# |> into binds the pipe result to a variable (like let, but reads left-to-
right)
gc_table
  |> filter(|row| abs(row.gc - mean_gc) > 2.0 * std_gc)
  |> sort_by(|row| -row.gc)
  |> into outliers

print(f"\nOutlier contigs ({len(outliers)}):")
outliers |> each(|row| print(f"  {row.name}: GC={row.gc:.3f}, length=
{row.length}"))
```

Run it:

```
bl run gc_scan.bl
```

Example output:

```
Analyzed 847 sequences
GC content: 0.412 (range: 0.198 - 0.687)
Standard deviation: 0.0531

Outlier contigs (12):
  contig_441: GC=0.687, length=3421
  contig_002: GC=0.621, length=15789
  ...
```

Project Structure

Initialize a BioLang project:

```
mkdir my-rnaseq-pipeline
cd my-rnaseq-pipeline
bl init --name my-rnaseq-pipeline
```

This creates the following structure:

```
my-rnaseq-pipeline/
  biolang.toml       # package metadata and dependencies
  main.bl            # entry point
```

Create project-specific directories and modules when the analysis needs them:

```
my-rnaseq-pipeline/
  biolang.toml
  main.bl
  src/
    paths.bl
    qc.bl
  data/
  results/
```

The manifest records package metadata and optional path or Git dependencies:

```
[package]
name = "my-rnaseq-pipeline"
version = "0.1.0"

[dependencies]
shared-qc = { path = "../shared-qc" }
variants = { git = "https://github.com/example/variants.git", branch = "main"
}
```

Run `bl install` in the project directory to install the declared dependencies.

Multi-file projects

Use `import` to split your pipeline across files:

#11 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# src/main.bl
import "src/qc.bl" as qc
import "src/alignment.bl" as align
import "src/variant_calling.bl" as vc

let samples = csv("data/sample_sheet.csv")

samples |> each(|sample| {
  let cleaned = qc.run(sample.fastq_r1, sample.fastq_r2)
  let bam = align.run(cleaned.r1, cleaned.r2, sample.reference)
  vc.run(bam, sample.reference, sample.sample_id)
})
```

#12 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# src/qc.bl
let run = |r1, r2| {
  let filt_r1 = read_fastq(r1) |> filter(|r| mean(r.quality) >= 30) |>
write_fastq(f"{r1}.filtered.fq.gz")
  let filt_r2 = read_fastq(r2) |> filter(|r| mean(r.quality) >= 30) |>
write_fastq(f"{r2}.filtered.fq.gz")
  {r1: filt_r1, r2: filt_r2}
}
```

BIOLANG_PATH

The `BIOLANG_PATH` environment variable controls where BioLang searches for imported modules and plugins. It accepts a colon-separated (or semicolon on Windows) list of directories:

```
export BIOLANG_PATH="/home/user/biolang-libs:/shared/team-modules"
```

Resolution order for `import "module.bl"`:

1. Relative to the importing file
2. The current working directory
3. Each directory in `BIOLANG_PATH`
4. `~/.biolang/stdlib/`
5. `~/.biolang/packages/`

This is useful for sharing utility modules across projects:

#13 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# This resolves via BIOLANG_PATH if not found locally
import "genomics_utils.bl" as gutils

let kmers = dna"ATCGATCGATCG" |> gutils.kmer_frequencies(k: 3)
print(kmers)
```

What's Next

You now have BioLang installed, know how to use the REPL for interactive exploration, and can write and run scripts. In the next chapter, we will explore bio literals – the first-class DNA, RNA, protein, and quality score types that make BioLang unique for bioinformatics work.

Chapter 2: Bio Literals

BioLang has first-class types for biological sequences. These are not strings – they carry domain semantics, validate their contents, and support biological operations directly. This chapter covers DNA, RNA, Protein, and Quality literals, along with the operations that make them powerful.

DNA Literals

A DNA literal is created with the `dna` prefix:

#14 ▶ Run ▶▶ Run All Above + This

```
let seq = dna"ATCGATCGATCG"
```

DNA literals accept only valid IUPAC nucleotide codes: `A`, `T`, `C`, `G`, and ambiguity codes `N`, `R`, `Y`, `S`, `W`, `K`, `M`, `B`, `D`, `H`, `V`.

#15 ▶ Run ▶▶ Run All Above + This

```
# Ambiguous positions are valid
let probe = dna"ATCNNGATCG"

# Case is normalized to uppercase
let lower = dna"atcgatcg"
print(lower)   # => dna"ATCGATCG"
```

Invalid bases cause a compile-time error:

```
# Conceptual or diagnostic example; not directly executable.
# This is a compile error -- U is not a DNA base
let bad = dna"AUCG"
# Error: invalid DNA base 'U' at position 1
```

RNA Literals

RNA uses the `rna` prefix and contains `A`, `U`, `C`, `G`:

#16 ▶ Run ▶▶ Run All Above + This

```
let mrna = rna"AUGCUUAAGGCUAG"
```

Protein Literals

Protein sequences use standard single-letter amino acid codes:

#17 ▶ Run ▶▶ Run All Above + This

```
let p53_fragment = protein"MEEPQSDPSVEPPLSQETFSDLWKLLPENNVLSPLPS"
```

Protein literals validate against the 20 standard amino acids plus `X` (unknown), `*` (stop), `U` (selenocysteine), and `O` (pyrrolysine):

#18 ▶ Run ▶▶ Run All Above + This

```
let with_stop = protein"MVLSPADKTNVK*"
```

Quality Score Literals

Phred+33 quality scores are first-class values:

#19 ▶ Run ▶▶ Run All Above + This

```
let quals = qual"IIIIIHHHGGFFEEDCBA"
```

Each character maps to a Phred quality score. You can work with them numerically:

#20 ▶ Run ▶▶ Run All Above + This

```
let q = qual"IIIIIHHHGG"
print(mean_phred(q)) # => 39.2
print(min_phred(q))  # => 38  (G = 38)
print(max_phred(q))  # => 40  (I = 40)

# Filter reads by mean quality
let reads = read_fastq("data/reads.fastq")
let hq_reads = reads |> filter(|r| mean_phred(r.quality) >= 30)
```

Sequence Operations

Complement and Reverse Complement

#21 ▶ Run ▶▶ Run All Above + This

```
let forward = dna"ATCGATCG"

let comp = forward |> complement()
print(comp)    # => dna"TAGCTAGC"

let rc = forward |> reverse_complement()
print(rc)      # => dna"CGATCGAT"
```

Reverse complement is the workhorse of strand-aware bioinformatics:

#22 ▶ Run ▶▶ Run All Above + This

```
# Check if a primer binds to either strand
let template = dna"ATCGATCGAATTCGCTAGC"
let primer = dna"GCTAGC"

let fwd_match = template |> find_motif(primer)
let rev_match = template |> find_motif(primer |> reverse_complement())

print(f"Forward hits: {len(fwd_match)}, Reverse hits: {len(rev_match)}")
```

Transcription and Translation

#23 ▶ Run ▶▶ Run All Above + This

```
let gene = dna"ATGAAAGCTTTTCGATAG"

# Transcribe DNA to RNA (T -> U)
let mrna = gene |> transcribe()
print(mrna)    # => rna"AUGAAAGCUUUUCGAUAG"

# Translate RNA (or DNA) to protein
let protein_seq = gene |> translate()
print(protein_seq)    # => protein"MKAFR*"

let mito_protein = gene |> translate()
print(mito_protein)  # standard genetic code (table 1)
```

GC Content

#24 ▶ Run ▶▶ Run All Above + This

```
let contig = dna"ATCGATCGGGCCCATATATGCGCGC"

let gc = contig |> gc_content()
print(f"GC: {gc:.2%}")    # => GC: 56.00%

# Sliding window GC content for detecting isochores
let gc_windows = contig |> window(10) |> map(|w| gc_content(w))
print(gc_windows)
```

Subsequences with `slice` and `len`

#25 ▶ Run ▶▶ Run All Above + This

```
let genome_fragment = dna"ATCGATCGATCGATCGATCG"

print(seq_len(genome_fragment))           # => 20
print(genome_fragment |> slice(0, 6)) # => dna"ATCGAT"

# Extract a coding region by coordinates
let cds_start = 3
let cds_end = 15
let cds = genome_fragment |> slice(cds_start, cds_end)
```

Motif Searching

`find_motif` returns a list of match positions:

#26 ▶ Run ▶▶ Run All Above + This

```
let seq = dna"ATCGAATTCGATCGAATTCG"

# Find EcoRI recognition site
let ecori_sites = seq |> find_motif(dna"GAATTC")
print(ecori_sites)   # => [3, 13]

# Regex search on sequences (returns match records)
let matches = seq |> regex_find("GA[AT]TTC")
matches |> each(|m| print(f"Match at {m.start}: {m.text}"))
```

Example: Find All ORFs in a DNA Sequence

An open reading frame (ORF) starts with ATG and ends at the first in-frame stop codon (TAA, TAG, TGA). This script finds all ORFs in all six reading frames.

```
# orf_finder.bl
# Find all open reading frames in a FASTA sequence.

let find_orfs = |seq, min_length| {
  let orfs = []
  let codons = seq |> chunk(3)
  let in_orf = false
  let orf_start = 0

  codons |> enumerate() |> each(|i, codon| {
    if !in_orf && codon == dna"ATG" then {
      in_orf = true
      orf_start = i * 3
    }
    if in_orf && (codon == dna"TAA" || codon == dna"TAG" || codon ==
dna"TGA") then {
      let orf_end = (i + 1) * 3
      let orf_len = orf_end - orf_start
      if orf_len >= min_length then {
        orfs = orfs ++ [{start: orf_start, end: orf_end, length: orf_len,
seq: seq |> slice(orf_start, orf_end)}]
      }
      in_orf = false
    }
  })
  orfs
}

let sequences = read_fasta("data/sequences.fasta")

# Search all 6 reading frames
let all_orfs = sequences |> flat_map(|entry| {
  let fwd = entry.seq
  let rev = entry.seq |> reverse_complement()

  0..3 |> flat_map(|frame| {
    let fwd_orfs = fwd |> slice(frame, seq_len(fwd)) |> find_orfs(300)
      |> map(|orf| {
        ...orf,
        seq_id: entry.id,
        strand: "+",
        frame: frame,
        start: orf.start + frame
      })
    let rev_orfs = rev |> slice(frame, seq_len(rev)) |> find_orfs(300)
      |> map(|orf| {
        ...orf,
        seq_id: entry.id,
        strand: "-",
        frame: frame
      })
    fwd_orfs ++ rev_orfs
  })
})

print(f"Found {len(all_orfs)} ORFs (>= 300 bp)")
all_orfs
  |> sort_by(|orf| -orf.length)
  |> take(20)
  |> each(|orf| print(f"  {orf.seq_id} {orf.strand}:{orf.start}-{orf.end}
({orf.length} bp)"))
```

Example: Codon Usage Table

Given a coding sequence, calculate the frequency of each codon and display the codon usage bias.

#28 ▶ Run ▶▶ Run All Above + This

```
# codon_usage.bl
# Build a codon usage table from a coding sequence.

let cds_records = read_fasta("data/sequences.fasta")

# Aggregate codons across all coding sequences
let codon_list = cds_records
  |> flat_map(|rec| rec.seq |> chunk(3))
  |> filter(|codon| seq_len(codon) == 3)
  |> map(|codon| {codon: to_string(codon)})

let codon_counts = table(codon_list)
  |> group_by("codon")
  |> summarize(|key, group| {codon: key, count: len(group)})

let total = codon_counts |> map(|c| c.count) |> sum()

let usage_table = codon_counts
  |> map(|c| {
    ...c,
    amino_acid: into(c.codon, "DNA") |> translate() |> to_string(),
    frequency: c.count / total,
    per_thousand: (c.count / total) * 1000.0
  })
  |> sort_by(|c| c.amino_acid)

print(f"Codon usage from {len(cds_records)} sequences ({total} codons)\n")
print("Codon  AA  Count     Freq   /1000")
print("-----  --  --------  -----  -----")
usage_table |> each(|row|
  print(f"{row.codon}    {row.amino_acid}   {row.count:>8}
{row.frequency:.4f}  {row.per_thousand:.1f}")
)

# Identify rare codons (< 10 per thousand)
let rare = usage_table |> filter(|c| c.per_thousand < 10.0)
print(f"\nRare codons ({len(rare)}):")
rare |> each(|c| print(f"  {c.codon} ({c.amino_acid}):
{c.per_thousand:.1f}/1000"))
```

Example: Restriction Enzyme Site Finder

Find all restriction enzyme recognition sites in a sequence and predict fragment sizes for a virtual digest.

#29 ▶ Run ▶▶ Run All Above + This

```
# digest.bl
# Virtual restriction digest of a DNA sequence.

# Common restriction enzymes: name -> recognition site
let enzymes = [
  {name: "EcoRI",   site: dna"GAATTC",  cut_offset: 1},
  {name: "BamHI",   site: dna"GGATCC",  cut_offset: 1},
  {name: "HindIII", site: dna"AAGCTT",  cut_offset: 1},
  {name: "NotI",    site: dna"GCGGCCGC", cut_offset: 2},
  {name: "XhoI",    site: dna"CTCGAG",  cut_offset: 1},
  {name: "SalI",    site: dna"GTCGAC",  cut_offset: 1},
  {name: "NcoI",    site: dna"CCATGG",  cut_offset: 1},
  {name: "PstI",    site: dna"CTGCAG",  cut_offset: 5}
]

let entry = read_fasta("data/sequences.fasta") |> first()
let seq = entry.seq
print(f"Sequence length: {seq_len(seq)} bp\n")

# Find all cut sites for each enzyme
let results = enzymes |> map(|enzyme| {
  let sites = seq |> find_motif(enzyme.site)
  let cut_positions = sites |> map(|pos| pos + enzyme.cut_offset)
  {name: enzyme.name, site: to_string(enzyme.site), n_cuts: len(sites),
positions: cut_positions}
})

# Report
results |> each(|r| {
  if r.n_cuts > 0 then {
    print(f"{r.name} ({r.site}): {r.n_cuts} site(s) at {r.positions}")
  } else {
    print(f"{r.name} ({r.site}): no sites")
  }
})

# Virtual double digest with EcoRI + BamHI
let ecori_result = results |> find(|r| r.name == "EcoRI")
let ecori_cuts = ecori_result.positions
let bamhi_result = results |> find(|r| r.name == "BamHI")
let bamhi_cuts = bamhi_result.positions
let all_cuts = (ecori_cuts ++ bamhi_cuts) |> sort() |> unique()

# Calculate fragment sizes (circular plasmid)
let fragments = if len(all_cuts) == 0 then [seq_len(seq)]
  else {
    let sorted_cuts = all_cuts |> sort()
    let frags = sorted_cuts
      |> window(2)
      |> map(|pair| pair[1] - pair[0])
    # Add the wrap-around fragment for circular DNA
    let wrap = seq_len(seq) - last(sorted_cuts) + first(sorted_cuts)
    frags ++ [wrap]
  }

print(f"\nEcoRI + BamHI double digest: {len(fragments)} fragments")
fragments
  |> sort() |> reverse()
  |> enumerate()
  |> each(|i, size| print(f"  Fragment {i + 1}: {size} bp"))
```

Summary

Bio literals are the foundation of BioLang. They carry biological meaning, enforce validity at parse time, and chain naturally with biological operations through pipes. Instead of calling string manipulation functions on raw text, you work directly with typed sequences that know about complements, codons, reading frames, and motifs.

In the next chapter, we will cover the full type system – variables, records, type coercion, and nil handling – which builds on these bio types to represent complex biological data structures.

Chapter 3: Variables and Types

BioLang has a rich type system designed around bioinformatics data. This chapter covers variable bindings, the full set of types, records, type checking and coercion, and nil handling.

`let` Bindings

Variables are introduced with `let` and are immutable by default:

#30 ▶ Run ▶▶ Run All Above + This

```
let sample_id = "TCGA-A1-A0SP"
let min_depth = 30
let reference = dna"ATCGATCGATCG"
```

Attempting to rebind a `let` variable in the same scope is an error:

#31 ▶ Run ▶▶ Run All Above + This

```
let threshold = 0.05
let threshold = 0.01   # Error: 'threshold' is already bound in this scope
```

Reassignment

Use `=` without `let` to reassign an existing variable:

#32 ▶ Run ▶▶ Run All Above + This

```
let total_reads = 0
total_reads = total_reads + 1847293

let status = "pending"
status = "complete"
```

Primitive Types

Type	Examples	Description
`Int`	`42`, `-1`, `1_000_000`	64-bit integer
`Float`	`3.14`, `1e-10`, `0.05`	64-bit float
`Str`	`"hello"`, `f"GC: {gc}"`, `r"C:\data"`	UTF-8 string
`Bool`	`true`, `false`	Boolean
`Nil`	`nil`	Absence of value

#33 ▶ Run ▶▶ Run All Above + This

```
let num_samples = 48
let p_value = 3.2e-8
let experiment = "RNA-seq time course"
let is_paired_end = true
let adapter_seq = nil   # not yet determined

# String variants
let name = "sample_01"                  # regular string (\n = newline, \t =
tab)
let msg = f"Found {num_samples} reads"  # f-string with interpolation
let path = r"C:\Users\data\reads.fq"    # raw string (backslashes literal)
```

Bio Types

These types carry domain semantics beyond raw data:

Type	Literal	Description
`DNA`	`dna"ATCG"`	DNA sequence (IUPAC)
`RNA`	`rna"AUGC"`	RNA sequence
`Protein`	`protein"MVLK"`	Amino acid sequence
`Quality`	`qual"IIIHH"`	Phred+33 quality scores
`Interval`	`interval("chr1", 1000, 2000)`	Genomic interval
`Variant`	`variant("chr17", 7674220, "C", "T")`	Sequence variant

#34 ▶ Run ▶▶ Run All Above + This

```
let primer_fwd = dna"TGTAAAACGACGGCCAGT"
let stop_codon = rna"UAG"
let signal_peptide = protein"MKWVTFISLLFLFSSAYS"
let read_quals = qual"IIIIIIIIHHHHHGGGGFFFF"

# Genomic coordinates
let tp53_exon1 = interval("chr17", 7668421, 7669690)
let brca1_snp = variant("chr17", 43094464, "G", "A")
```

Interval Type

Intervals represent genomic regions with chromosome, start, and end:

#35 ▶ Run ▶▶ Run All Above + This

```
let region = interval("chr1", 1000000, 2000000)
print(region.chrom)    # => "chr1"
print(region.start)    # => 1000000
print(region.end)      # => 2000000
print(region.length)   # => 1000000

# Interval arithmetic
let exon1 = interval("chr1", 1000, 1500)
let exon2 = interval("chr1", 1400, 2000)
print(exon1.end > exon2.start && exon2.end > exon1.start)  # => true
print(intersect(exon1, exon2))      # => interval("chr1", 1400, 1500)
```

Variant Type

Variants represent sequence changes at specific positions:

#36 ▶ Run ▶▶ Run All Above + This

```
let snv = variant("chr7", 55249071, "C", "T")
print(snv.chrom)     # => "chr7"
print(snv.pos)       # => 55249071
print(snv.ref_allele)  # => "C"
print(snv.alt_allele)  # => "T"

# Classify variant type
print(variant_type(snv))    # => "SNV"

let indel = variant("chr7", 55249071, "CT", "C")
print(variant_type(indel)) # => "DEL"
```

Records

Records are BioLang's primary structured data type. They map naturally to the tabular, metadata-rich world of bioinformatics:

#37 ▶ Run ▶▶ Run All Above + This

```
let sample = {
  sample_id: "S001",
  patient: "P-4821",
  tissue: "tumor",
  reads: 45_000_000,
  mean_coverage: 32.5,
  contamination: 0.02
}

print(sample.sample_id)       # => "S001"
print(sample.mean_coverage)   # => 32.5
```

String Keys

Record keys can be identifiers or strings:

```
# Conceptual or diagnostic example; not directly executable.
let annotation = {
  "gene_name": "EGFR",
  "Gene Ontology": "GO:0005524",
  chrom: "chr7"
}

print(annotation."Gene Ontology")   # => "GO:0005524"
```

Record Spread

The spread operator `...` merges one record into another. Later keys override earlier ones:

#38 ▶ Run ▶▶ Run All Above + This

```
let defaults = {
  aligner: "bwa-mem2",
  min_mapq: 30,
  mark_duplicates: true,
  reference: "GRCh38"
}

let sample_config = {
  ...defaults,
  sample_id: "S001",
  min_mapq: 20   # override the default
}

print(sample_config.aligner)   # => "bwa-mem2"
print(sample_config.min_mapq)  # => 20
```

This pattern is essential for bioinformatics pipelines where you have standard parameters with per-sample overrides.

Nested Records

Records can nest to arbitrary depth:

#39 ▶ Run ▶▶ Run All Above + This

```
let variant_annotation = {
  variant: variant("chr17", 7674220, "C", "T"),
  gene: {
    name: "TP53",
    transcript: "NM_000546.6",
    exon: 7,
    consequence: "missense_variant"
  },
  population: {
    gnomad_af: 0.00003,
    clinvar: "Pathogenic",
    cosmic_count: 4521
  },
  prediction: {
    sift: {score: 0.001, label: "deleterious"},
    polyphen: {score: 0.998, label: "probably_damaging"}
  }
}

print(variant_annotation.gene.consequence)
print(variant_annotation.prediction.sift.score)
```

Type Checking

BioLang provides introspection functions for runtime type checking:

#40 ▶ Run ▶▶ Run All Above + This

```
let seq = dna"ATCG"
print(type(seq))       # => "DNA"
print(is_dna(seq))     # => true
print(is_rna(seq))     # => false

let rec = {name: "BRCA1", chrom: "chr17"}
print(type(rec))       # => "Record"
print(is_record(rec))  # => true
print(is_str(rec))     # => false
```

Available type check functions:

#41 ▶ Run ▶▶ Run All Above + This

```
let val = dna"ATCG"
is_int(val)
is_float(val)
is_str(val)
is_bool(val)
is_nil(val)
is_dna(val)
is_rna(val)
is_protein(val)
is_quality(val)
is_list(val)
is_set(val)
is_record(val)
is_table(val)
is_interval(val)
is_variant(val)
```

Type Coercion with `into`

`into(value, "TargetType")` converts between compatible types:

#42 ▶ Run ▶▶ Run All Above + This

```
# String to DNA
let seq = into("ATCGATCG", "DNA")

# DNA to RNA (transcription)
let mrna = into(dna"ATCGATCG", "RNA")

# Int to Float
let ratio = into(42, "Float")

# String to Int
let depth = into("30", "Int")

# List of records to Table
let rows = [{gene: "TP53", pval: 0.001}, {gene: "BRCA1", pval: 0.05}]
let tbl = into(rows, "Table")

# Variant to Interval (single-base or span)
let region = into(variant("chr1", 1000, "A", "T"), "Interval")
```

Type Aliases

Define aliases to make code self-documenting:

```
# Conceptual or diagnostic example; not directly executable.
type Locus = Record
type SampleMeta = Record
type QCMetrics = Record

let build_locus = |chrom, start, end, gene| {
  chrom: chrom, start: start, end: end, gene: gene
}

let tp53 = build_locus("chr17", 7668421, 7687490, "TP53")
```

Nil Handling

Bioinformatics data is full of missing values – absent annotations, failed QC metrics, unmapped reads. BioLang has two operators for nil handling.

`??` Null Coalesce

Returns the left side if non-nil, otherwise the right:

#43 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
let depth = sample.coverage ?? 0
let gene_name = annotation.symbol ?? annotation.id ?? "unknown"

# Useful for setting defaults in pipeline parameters
let min_qual = args.min_quality ?? 30
let threads = args.threads ?? 4
```

`?.` Optional Chain

Safely accesses nested fields, returning nil if any intermediate value is nil:

#44 ▶ Run ▶▶ Run All Above + This

```
let variant_record = {
  annotation: {clinvar: {significance: "Pathogenic"}}
}
let clinvar_status = variant_record?.annotation?.clinvar?.significance
# Returns nil if annotation, clinvar, or significance is missing

# Combine with ??
let pathogenicity = variant_record?.annotation?.clinvar?.significance ??
"Unknown"
```

Example: Building a Sample Metadata Registry

#45 ▶ Run ▶▶ Run All Above + This

```
# sample_registry.bl
# Parse a sample sheet and build a typed metadata registry.

let defaults = {
  platform: "Illumina",
  chemistry: "NovaSeq 6000",
  read_length: 150,
  paired: true,
  reference: "GRCh38"
}

let raw_sheet = csv("sample_sheet.csv")

let registry = raw_sheet |> map(|row| {
  ...defaults,
  sample_id: row.Sample_ID,
  patient_id: row.Patient ?? "unknown",
  tissue: row.Tissue,
  condition: row.Condition ?? "normal",
  fastq_r1: row.FASTQ_R1,
  fastq_r2: row.FASTQ_R2 ?? nil,
  paired: !is_nil(row.FASTQ_R2),
  library_prep: row.Library ?? "WGS",
  date: row.Date
})

# Validate: every sample must have R1
let invalid = registry |> filter(|s| is_nil(s.fastq_r1))
if len(invalid) > 0 then {
  print(f"ERROR: {len(invalid)} samples missing FASTQ R1:")
  invalid |> each(|s| print(f"  {s.sample_id}"))
  exit(1)
}

# Group by condition for downstream analysis
let by_condition = registry |> group_by("condition")
by_condition |> each(|group| {
  print(f"{group.key}: {len(group.values)} samples")
  group.values |> each(|s| print(f"  {s.sample_id} ({s.tissue})"))
})

# Summary
let tumor_count = registry |> filter(|s| s.condition == "tumor") |> len()
let normal_count = registry |> filter(|s| s.condition == "normal") |> len()
print(f"\nRegistry: {len(registry)} samples ({tumor_count} tumor,
{normal_count} normal)")
```

Example: Parsing and Validating Variant Records

#46 ▶ Run ▶▶ Run All Above + This

```
# validate_variants.bl
# Read a VCF file, parse variants into typed records, validate, classify.

let raw_variants = read_vcf("data/variants.vcf")

# Build typed variant records with full annotation
let typed_variants = raw_variants |> map(|v| {
  var: variant(v.chrom, v.pos, v.ref, v.alt),
  qual: into(v.qual ?? "0", "Float"),
  depth: into(v.info?.DP ?? "0", "Int"),
  allele_freq: into(v.info?.AF ?? "0.0", "Float"),
  genotype: v.samples?.GT ?? "./.",
  filter_status: v.filter
})

# Validate variant quality
let validate = |v| {
  let issues = []
  if v.qual < 30.0 then issues = issues ++ ["low_qual"]
  if v.depth < 10 then issues = issues ++ ["low_depth"]
  if v.allele_freq < 0.05 then issues = issues ++ ["low_af"]
  if v.genotype == "./." then issues = issues ++ ["missing_gt"]
  {...v, issues: issues, is_valid: len(issues) == 0}
}

let validated = typed_variants |> map(validate)

# Classify variants
let classify = |v| {
  let vtype = variant_type(v.var)
  let size = if vtype == "SNV" then 1
    else abs(len(v.var.alt_allele) - len(v.var.ref_allele))
  {...v, variant_type: vtype, size: size}
}

let classified = validated |> map(classify)

# Report summary
let total = len(classified)
let valid = classified |> filter(|v| v.is_valid) |> len()
let by_type = classified |> group_by("variant_type")

print(f"Total variants: {total}")
print(f"Passing filters: {valid} ({valid / total * 100.0:.1f}%)")
print(f"\nBy type:")
by_type |> each(|g| print(f"  {g.key}: {len(g.values)}"))

# Flag high-impact variants
let high_impact = classified
  |> filter(|v| v.is_valid && v.variant_type != "SNV" && v.size > 50)
  |> sort_by(|v| -v.size)

print(f"\nHigh-impact structural variants (>50bp): {len(high_impact)}")
high_impact |> take(10) |> each(|v| {
  print(f"  {v.var.chrom}:{v.var.pos} {v.variant_type} size={v.size} AF=
{v.allele_freq:.3f}")
})
```

Summary

BioLang's type system combines standard programming types with biological data types and record-based data modeling. Records with spread make it natural to build layered configurations and annotated data structures. Nil handling with `??` and `?.` keeps pipelines resilient to the missing data that pervades genomics workflows.

The next chapter covers collections and tables – the workhorses for processing the large tabular datasets that dominate bioinformatics.

Chapter 4: Collections and Tables

Bioinformatics data is inherently tabular and set-oriented. BioLang provides Lists, Sets, Tables, and Ranges as first-class collection types, with rich operations for filtering, grouping, and summarizing biological datasets.

Lists

Lists are ordered, heterogeneous sequences:

#47 ▶ Run ▶▶ Run All Above + This

```
let chromosomes = ["chr1", "chr2", "chr3", "chrX", "chrY"]
let read_lengths = [150, 151, 149, 150, 148, 150]
let mixed = ["TP53", 42, true, dna"ATCG"]
```

Lists support standard access and manipulation:

#48 ▶ Run ▶▶ Run All Above + This

```
let genes = ["BRCA1", "TP53", "EGFR", "KRAS", "BRAF"]

print(genes[0])        # => "BRCA1"
print(genes[-1])       # => "BRAF"
print(len(genes))      # => 5

# Concatenation
let more = genes ++ ["PIK3CA", "PTEN"]

# Nested lists (e.g., paired-end read pairs)
let pairs = [
  ["sample1_R1.fq.gz", "sample1_R2.fq.gz"],
  ["sample2_R1.fq.gz", "sample2_R2.fq.gz"]
]
print(pairs[0][1])   # => "sample1_R2.fq.gz"
```

Sets

Sets are unordered collections of unique elements. They are the right tool for gene lists, sample IDs, and any membership-testing workload:

#49 ▶ Run ▶▶ Run All Above + This

```
let panel_genes = set(["BRCA1", "BRCA2", "TP53", "EGFR", "KRAS", "BRAF"])
let mutated_genes = set(["TP53", "KRAS", "APC", "PIK3CA"])

# Set operations
let overlap = intersection(panel_genes, mutated_genes)
print(overlap)   # => set(["TP53", "KRAS"])

let all_genes = union(panel_genes, mutated_genes)
let panel_only = difference(panel_genes, mutated_genes)
let mutated_only = difference(mutated_genes, panel_genes)

print(f"On panel and mutated: {len(overlap)}")
print(f"Mutated, not on panel: {len(mutated_only)}")

# Membership
print(contains(panel_genes, "TP53"))   # => true
```

A real use case – finding shared variants between tumor and normal:

#50 ▶ Run ▶▶ Run All Above + This

```
let tumor_variants = read_vcf("data/variants.vcf")
  |> map(|v| f"{v.chrom}:{v.pos}:{v.ref}>{v.alt}")
  |> set()

let normal_variants = read_vcf("data/variants.vcf")
  |> map(|v| f"{v.chrom}:{v.pos}:{v.ref}>{v.alt}")
  |> set()

let somatic = difference(tumor_variants, normal_variants)
let germline = intersection(tumor_variants, normal_variants)
print(f"Somatic: {len(somatic)}, Germline: {len(germline)}")
```

Tables

Tables are the central data structure for tabular bioinformatics data. They are created from lists of records:

#51 ▶ Run ▶▶ Run All Above + This

```
let samples = table([
  {sample_id: "S001", tissue: "tumor",  reads: 42_000_000, coverage: 35.2},
  {sample_id: "S002", tissue: "normal", reads: 38_000_000, coverage: 31.4},
  {sample_id: "S003", tissue: "tumor",  reads: 51_000_000, coverage: 42.1},
  {sample_id: "S004", tissue: "normal", reads: 44_000_000, coverage: 36.8}
])
```

Column Access

#52 ▶ Run ▶▶ Run All Above + This

```
let coverages = col(samples, "coverage")
print(coverages)   # => [35.2, 31.4, 42.1, 36.8]

# Multiple columns
let subset = samples |> select("sample_id", "coverage")
```

Row Iteration

#53 ▶ Run ▶▶ Run All Above + This

```
samples |> each(|row| {
  print(f"{row.sample_id}: {row.coverage}x ({row.tissue})")
})
```

Table Operations

`select` – Choose Columns

#54 ▶ Run ▶▶ Run All Above + This

```
let qc_summary = samples |> select("sample_id", "reads", "coverage")
```

`mutate` – Add or Transform Columns

#55 ▶ Run ▶▶ Run All Above + This

```
let enriched = samples
  |> mutate("reads_millions", |row| row.reads / 1_000_000.0)
  |> mutate("is_high_coverage", |row| row.coverage >= 30.0)
  |> mutate("label", |row| f"{row.sample_id}_{row.tissue}")
```

`summarize` – Aggregate Statistics

#56 ▶ Run ▶▶ Run All Above + This

```
let coverages = col(samples, "coverage")
let all_reads = col(samples, "reads")

let stats = {
  n_samples: len(samples),
  total_reads: sum(all_reads),
  mean_coverage: mean(coverages),
  min_coverage: min(coverages),
  max_coverage: max(coverages),
}

print(f"Samples: {stats.n_samples}, Mean coverage:
{stats.mean_coverage:.1f}x")
```

`group_by` – Split-Apply-Combine

#57 ▶ Run ▶▶ Run All Above + This

```
let by_tissue = samples
  |> group_by("tissue")
  |> summarize(|tissue, group| {
    tissue: tissue,
    n: len(group),
    mean_cov: mean(col(group, "coverage")),
    mean_reads: mean(col(group, "reads")),
  })

by_tissue |> each(|row|
  print(f"{row.tissue}: n={row.n}, cov={row.mean_cov:.1f}x, reads=
{row.mean_reads:.0f}")
)
```

`sort` – Order Rows

#58 ▶ Run ▶▶ Run All Above + This

```
# Sort by coverage descending
let ranked = samples |> sort_by(|row| -row.coverage)

# Sort by a field ascending
let ordered = samples |> sort_by(|row| row.tissue)
```

`filter` – Select Rows by Condition

#59 ▶ Run ▶▶ Run All Above + This

```
let high_cov = samples |> filter(|row| row.coverage >= 30.0)
let tumor_only = samples |> filter(|row| row.tissue == "tumor")

# Chained filters
let good_tumor = samples
  |> filter(|row| row.tissue == "tumor")
  |> filter(|row| row.coverage >= 30.0)
  |> filter(|row| row.reads >= 40_000_000)
```

Joins

Use `inner_join` when only matched rows should remain, or `left_join` when all rows from the first table must be preserved:

#60 ▶ Run ▶▶ Run All Above + This

```
let annotations = table([
  {sample_id: "S001", patient: "P101", stage: "III"},
  {sample_id: "S002", patient: "P101", stage: "III"},
  {sample_id: "S003", patient: "P202", stage: "II"}
])

let joined = left_join(samples, annotations, "sample_id")
```

Ranges

Ranges generate sequences of integers, useful for genomic coordinate windows:

#61 ▶ Run ▶▶ Run All Above + This

```
let indices = range(0, 10)          # [0, 1, 2, ..., 9]
let inclusive = range(0, 11)        # [0, 1, 2, ..., 10]
let stepped = range(0, 100, 10)     # [0, 10, 20, ..., 90]

# Chromosome positions in 1 Mb windows
let chrom_length = 248_956_422   # chr1
let windows = range(0, chrom_length, 1_000_000)
  |> map(|start| {
    chrom: "chr1",
    start: start,
    end: min(start + 1_000_000, chrom_length)
  })

print(f"Generated {len(windows)} windows for chr1")
```

Example: Sample QC Summary Table from FASTQ Stats

Read multiple FASTQ files, compute QC metrics, and build a summary table.

#62 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# fastq_qc_summary.bl
# Build a QC summary table from raw FASTQ files.

let sample_sheet = csv("samples.csv")

let read_stats = |reads| {
  let quals = reads |> map(|r| mean(r.quality))
  let lengths = reads |> map(|r| seq_len(r.seq))
  let gc_vals = reads |> map(|r| gc_content(r.seq))
  {
    total_reads: len(reads),
    total_bases: sum(lengths),
    mean_length: mean(lengths),
    mean_quality: mean(quals),
    q30_pct: reads |> filter(|r| mean(r.quality) >= 30) |> len() / len(reads)
* 100.0,
    gc_pct: mean(gc_vals) * 100.0
  }
}

let qc_results = sample_sheet |> map(|row| {
  let r1 = read_fastq(row.fastq_r1)
  let r1_stats = read_stats(r1)
  let r2_stats = if !is_nil(row.fastq_r2) then
    read_stats(read_fastq(row.fastq_r2))
  else nil

  {
    sample_id: row.sample_id,
    total_reads: r1_stats.total_reads + (r2_stats?.total_reads ?? 0),
    total_bases: r1_stats.total_bases + (r2_stats?.total_bases ?? 0),
    mean_length: r1_stats.mean_length,
    mean_quality: r1_stats.mean_quality,
    q30_pct: r1_stats.q30_pct,
    gc_pct: r1_stats.gc_pct
  }
})

let qc_table = table(qc_results)

# Add pass/fail column
let qc_flagged = qc_table |> mutate("qc_pass", |row| row.mean_quality >= 30.0
&& row.q30_pct >= 80.0 && row.total_reads >= 10_000_000)

# Summary by pass/fail
let summary = qc_flagged |> group_by("qc_pass") |> summarize(|pass, group| {
  qc_pass: pass,
  count: len(group),
  mean_reads: mean(group |> select("total_reads")),
  mean_q: mean(group |> select("mean_quality")),
})

print("QC Summary:")
print("=" * 60)
qc_flagged |> each(|row| {
  let flag = if row.qc_pass then "PASS" else "FAIL"
  print(f"  [{flag}] {row.sample_id}: {row.total_reads / 1e6:.1f}M reads, Q=
{row.mean_quality:.1f}, Q30={row.q30_pct:.1f}%")
})

let pass_count = qc_flagged |> filter(|r| r.qc_pass) |> len()
let fail_count = qc_flagged |> filter(|r| !r.qc_pass) |> len()
print(f"\n{pass_count} passed, {fail_count} failed out of {len(qc_flagged)}
samples")
```

```
# Write results
qc_flagged |> write_tsv("qc_summary.csv")
```

Example: Gene Expression Matrix – Filter and Normalize

Load a counts matrix, filter low-count genes, and apply TPM normalization.

```
# Conceptual or diagnostic example; not directly executable.
# expression_normalize.bl
# Filter low-count genes and compute TPM from a raw counts matrix.

let counts = csv("gene_counts.csv")   # rows = genes, columns = samples
let gene_lengths = csv("gene_lengths.csv")  # gene_id, length

# Build a length lookup
let length_map = gene_lengths |> group_by("gene_id")

# Convert to table with gene metadata
let expr_table = table(counts)
let sample_cols = columns(expr_table) |> filter(|c| c != "gene_id")

# Filter: keep genes with >= 10 counts in at least 3 samples
let filtered = expr_table |> filter(|row| {
  let expressed = sample_cols |> filter(|col| row[col] >= 10) |> len()
  expressed >= 3
})

print(f"Genes before filter: {len(expr_table)}")
print(f"Genes after filter: {len(filtered)}")

# RPK: reads per kilobase — transform each row
let rpk_table = filtered |> map(|gene_row| {
  let gene_group = length_map[gene_row.gene_id]
  let gene_len = if !is_nil(gene_group) then first(gene_group) else nil
  let len_kb = (gene_len?.length ?? 1000) / 1000.0
  let updated = sample_cols |> reduce({...gene_row}, |acc, col| {
    ...acc, [col]: gene_row[col] / len_kb
  })
  updated
}) |> table()

# TPM: normalize RPK to sum to 1 million per sample
let rpk_sums = sample_cols |> map(|col| {
  col: col,
  total: rpk_table |> select(col) |> sum()
})

let tpm_table = rpk_table |> map(|row| {
  sample_cols |> reduce({...row}, |acc, col| {
    let col_sum = rpk_sums |> find(|s| s.col == col)
    {...acc, [col]: (row[col] / col_sum.total) * 1e6}
  })
}) |> table()

# Summary statistics per gene
let gene_stats = tpm_table
  |> mutate("mean_tpm", |row| sample_cols |> map(|c| row[c]) |> mean())
  |> mutate("max_tpm", |row| sample_cols |> map(|c| row[c]) |> max())
  |> mutate("cv", |row| {
    let vals = sample_cols |> map(|c| row[c])
    stdev(vals) / (mean(vals) + 1e-10)
  })

# Top variable genes
let top_variable = gene_stats
  |> sort_by(|g| -g.cv)
  |> take(20)

print("\nTop 20 most variable genes (by CV):")
top_variable |> each(|g|
  print(f"  {g.gene_id}: mean TPM={g.mean_tpm:.1f}, CV={g.cv:.2f}")
)
```

```
tpm_table |> write_tsv("tpm_normalized.csv")
```

Example: Merge Variant Calls from Multiple Samples

Combine VCF calls from multiple samples into a unified genotype matrix.

```
# Conceptual or diagnostic example; not directly executable.
# merge_variants.bl
# Merge variant calls from multiple samples into a unified table.

let sample_vcfs = [
  {sample: "tumor_A",  vcf: "tumor_A.vcf.gz"},
  {sample: "tumor_B",  vcf: "tumor_B.vcf.gz"},
  {sample: "normal_A", vcf: "normal_A.vcf.gz"},
  {sample: "normal_B", vcf: "normal_B.vcf.gz"}
]

# Read all variants with sample tag
let all_calls = sample_vcfs |> flat_map(|s| {
  read_vcf(s.vcf) |> map(|v| {
    key: f"{v.chrom}:{v.pos}:{v.ref}>{v.alt}",
    chrom: v.chrom,
    pos: v.pos,
    ref_allele: v.ref,
    alt_allele: v.alt,
    sample: s.sample,
    qual: into(v.qual ?? "0", "Float"),
    depth: into(v.info?.DP ?? "0", "Int"),
    af: into(v.info?.AF ?? "0.0", "Float"),
    genotype: v.samples?.GT ?? "./."
  })
})

# Collect unique variant sites
let unique_sites = all_calls
  |> map(|v| v.key)
  |> unique()
  |> sort()

print(f"Total calls: {len(all_calls)}")
print(f"Unique variant sites: {len(unique_sites)}")

# Build genotype matrix: one row per variant site, one column per sample
let sample_names = sample_vcfs |> map(|s| s.sample)
let calls_by_key = all_calls |> group_by("key")

let merged = unique_sites |> map(|site_key| {
  let site_vals = calls_by_key[site_key]
  let first_call = first(site_vals)

  let base = {
    chrom: first_call.chrom,
    pos: first_call.pos,
    ref_allele: first_call.ref_allele,
    alt_allele: first_call.alt_allele,
    n_samples: len(site_vals)
  }

  # Add per-sample genotype columns
  let gt_fields = sample_names |> reduce({}, |acc, sname| {
    let call = site_vals |> find(|v| v.sample == sname)
    {...acc, [f"{sname}_GT"]: call?.genotype ?? "./.", [f"{sname}_AF"]:
call?.af ?? 0.0}
  })

  {...base, ...gt_fields}
})

let merged_table = table(merged)

# Find variants present in tumor but absent in normal (somatic candidates)
```

```
let somatic = merged_table |> filter(|row| {
  let tumor_present = row.tumor_A_GT != "./." || row.tumor_B_GT != "./."
  let normal_absent = row.normal_A_GT == "./." && row.normal_B_GT == "./."
  tumor_present && normal_absent
})

let shared_tumor = merged_table |> filter(|row|
  row.tumor_A_GT != "./." && row.tumor_B_GT != "./."
)

print(f"\nSomatic candidates: {len(somatic)}")
print(f"Shared across tumors: {len(shared_tumor)}")
print(f"\nTop somatic candidates:")
somatic
  |> sort_by(|v| -v.n_samples)
  |> take(10)
  |> each(|v| print(f"  {v.chrom}:{v.pos} {v.ref_allele}>{v.alt_allele} (in
{v.n_samples} sample(s))"))

merged_table |> write_tsv("merged_genotypes.csv")
```

Summary

Collections and tables give you the tools to handle the data shapes that dominate bioinformatics: sample sheets, QC matrices, expression counts, and multi-sample variant calls. Combined with `group_by`, `summarize`, and `mutate`, you can express complex data wrangling pipelines concisely.

The next chapter introduces pipes and transforms – the core of BioLang's workflow composition model.

Chapter 5: Pipes and Transforms

The pipe operator is the defining feature of BioLang. Every bioinformatics workflow is a chain of transformations: read data, filter, transform, summarize, output. Pipes make these chains explicit, readable, and composable.

The Pipe Operator `|>`

The pipe `a |> f(b)` desugars to `f(a, b)`. The left-hand value becomes the first argument of the right-hand function:

#63 ▶ Run ▶▶ Run All Above + This

```
# These are equivalent:
gc_content(dna"ATCGATCGATCG")
dna"ATCGATCGATCG" |> gc_content()

# And these:
let reads = read_fastq("data/reads.fastq")
filter(reads, |r| mean_phred(r.quality) >= 30)
reads |> filter(|r| mean_phred(r.quality) >= 30)
```

Why Pipe-First Matters for Bioinformatics

Without pipes, nested function calls read inside-out:

#64 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# Hard to follow -- you read from the innermost call outward
write_tsv(
  sort_by(
    summarize(
      group_by(
        filter(read_vcf("data/variants.vcf"), |v| v.qual >= 30),
        "chrom"
      ),
      |chrom, rows| {n: len(rows), mean_qual: mean(col(rows, "qual"))}
    ),
    |row| -row.n
  ),
  "chrom_summary.csv"
)
```

With pipes, the same workflow reads top-to-bottom, matching how you think about the analysis:

#65 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
read_vcf("data/variants.vcf")
  |> filter(|v| v.qual >= 30)
  |> group_by("chrom")
  |> summarize(|chrom, rows| {n: len(rows), mean_qual: mean(col(rows,
"qual"))})
  |> sort_by(|row| -row.n)
  |> write_tsv("chrom_summary.csv")
```

Each line is one transformation step. You can read the pipeline like a protocol.

Chaining Transforms

The real power emerges when you chain multiple operations. Here is a complete read-to-report workflow:

#66 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# Read a BAM file, compute per-chromosome alignment statistics
read_bam("sample.sorted.bam")
  |> filter(|r| r.mapq >= 30 && !r.is_duplicate && !r.is_unmapped)
  |> group_by("chrom")
  |> summarize(|chrom, rows| {
    mapped_reads: len(rows),
    mean_mapq: mean(col(rows, "mapq")),
    mean_insert: mean(col(rows, "insert_size"))
  })
  |> sort_by(|row| -row.mapped_reads)
  |> mutate("pct", |row| row.mapped_reads / sum(col(rows, "mapped_reads")) *
100.0)
  |> write_tsv("alignment_stats.csv")
```

The Tap Pipe `|>>`

Sometimes you need a side effect in the middle of a chain – logging, writing an intermediate file, printing a progress message – without disrupting the data flow. The tap pipe `|>>` evaluates the right side for its effect but passes the left side through unchanged:

#67 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
read_fastq("data/reads.fastq")
  |>> |reads| print(f"Raw reads: {len(reads)}")
  |> filter(|r| mean(r.quality) >= 25)
  |>> |reads| print(f"After quality filter: {len(reads)}")
  |> filter(|r| seq_len(r.seq) >= 50)
  |>> |reads| print(f"After length filter: {len(reads)}")
  |> write_fastq("filtered_R1.fastq.gz")
```

Output:

```
Raw reads: 2847291
After quality filter: 2541083
After length filter: 2539847
```

The tap pipe is invaluable for debugging pipelines:

#68 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
read_vcf("data/variants.vcf")
  |>> |vs| print(f"Step 0 - Raw: {len(vs)} variants")
  |> filter(|v| v.filter == "PASS")
  |>> |vs| print(f"Step 1 - PASS only: {len(vs)}")
  |> filter(|v| v.qual >= 30)
  |>> |vs| print(f"Step 2 - Qual >= 30: {len(vs)}")
  |> filter(|v| into(v.info?.DP ?? "0", "Int") >= 10)
  |>> |vs| print(f"Step 3 - Depth >= 10: {len(vs)}")
  |> write_vcf("filtered_calls.vcf.gz")
```

You can also use tap to write intermediate checkpoints:

#69 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
read_fasta("data/sequences.fasta")
  |> filter(|s| seq_len(s.seq) >= 1000)
  |>> |seqs| write_fasta(seqs, "contigs_gt1kb.fa")
  |> filter(|s| gc_content(s.seq) >= 0.3 && gc_content(s.seq) <= 0.7)
  |>> |seqs| write_fasta(seqs, "contigs_normal_gc.fa")
  |> sort_by(|s| -seq_len(s.seq))
  |> take(100)
  |> write_fasta("top100_contigs.fa")
```

Higher-Order Functions in Pipe Chains

`map` – Transform Each Element

#70 ▶ Run ▶▶ Run All Above + This

```
# Extract GC content per contig
let gc_profile = read_fasta("data/sequences.fasta")
  |> map(|seq| {id: seq.id, gc: gc_content(seq.seq), length:
seq_len(seq.seq)})
```

`filter` – Select Elements by Predicate

#71 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# Keep only high-quality mapped reads
let good_reads = read_bam("aligned.bam")
  |> filter(|r| r.mapq >= 30)
  |> filter(|r| !r.is_duplicate)
  |> filter(|r| r.is_proper_pair)
```

`reduce` – Accumulate a Single Value

#72 ▶ Run ▶▶ Run All Above + This

```
# Total bases across all chromosomes
let total_bases = read_fasta("data/sequences.fasta")
  |> map(|s| seq_len(s.seq))
  |> reduce(0, |acc, n| acc + n)

print(f"Reference genome size: {total_bases / 1e9:.2f} Gb")
```

`sort` – Order Elements

#73 ▶ Run ▶▶ Run All Above + This

```
# Rank genes by expression level
let top_genes = csv("tpm.csv")
  |> sort_by(|row| -row.tpm)
  |> take(50)
  |> map(|row| row.gene_name)
```

`flat_map` – Map and Flatten

Essential when each input produces multiple outputs:

#74 ▶ Run ▶▶ Run All Above + This

```
# Extract all exons from a gene annotation
let all_exons = read_gff("data/annotations.gff")
  |> filter(|f| f.type == "gene")
  |> flat_map(|gene| gene.children |> filter(|c| c.type == "exon"))

# Count k-mers across multiple sequences (auto-spills to disk for large data)
let top_kmers = fasta("contigs.fa")
  |> kmer_count(21)        # Table sorted by count descending
  |> head(20)

# For bounded memory, use the top-N parameter:
# fasta("contigs.fa") |> kmer_count(21, 100)  # only top 100
```

`take_while` – Stream Until Condition Fails

#75 ▶ Run ▶▶ Run All Above + This

```
# Read sorted variants until we pass a genomic position
let nearby_variants = read_vcf("data/variants.vcf")
  |> filter(|v| v.chrom == "chr17")
  |> take_while(|v| v.pos <= 7700000)
  |> filter(|v| v.pos >= 7660000)
```

`scan` – Running Accumulation

Like `reduce`, but emits every intermediate value:

#76 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# Cumulative read count across sorted BAM regions
let cumulative = read_bam("sorted.bam")
  |> group_by("chrom")
  |> map(|g| {chrom: g.key, count: len(g.values)})
  |> sort("chrom")
  |> scan(0, |acc, row| acc + row.count)
```

`window` and `chunk` – Sliding and Fixed Blocks

#77 ▶ Run ▶▶ Run All Above + This

```
# Sliding window GC content
let gc_track = dna"ATCGATCGATCGATCGCCCCGGGG"
  |> window(10)
  |> map(|w| gc_content(w))

# Chunk reads into batches for parallel processing
let batches = read_fastq("data/reads.fastq")
  |> chunk(100_000)
  |> map(|batch| {n_reads: len(batch), mean_gc: batch |> map(|r|
gc_content(r.seq)) |> mean()})
```

`zip` – Pair Two Collections

#78 ▶ Run ▶▶ Run All Above + This

```
# Pair forward and reverse reads
let r1 = read_fastq("data/reads.fastq")
let r2 = read_fastq("data/reads.fastq")

let paired = zip(r1, r2) |> map(|pair| {
  id: pair[0].id,
  r1_qual: mean_phred(pair[0].quality),
  r2_qual: mean_phred(pair[1].quality),
  combined_length: seq_len(pair[0].seq) + seq_len(pair[1].seq)
})
```

`enumerate` – Track Position

#79 ▶ Run ▶▶ Run All Above + This

```
# Number contigs by size rank
read_fasta("data/sequences.fasta")
  |> sort_by(|s| -seq_len(s.seq))
  |> enumerate()
  |> map(|pair| {rank: pair[0] + 1, id: pair[1].id, length:
seq_len(pair[1].seq)})
  |> take(10)
  |> each(|row| print(f"#{row.rank}: {row.id} ({row.length} bp)"))
```

Pipe Binding with `|> into`

When you want to capture an intermediate result from a pipe chain, the usual approach is `let name = expr`. But this reads right-to-left, breaking the flow. The `|> into` operator binds the result of a pipe chain to a name while preserving left-to-right reading order:

#80 ▶ Run ▶▶ Run All Above + This

```
# Traditional style — reads right-to-left:
let variants = read_vcf("data/variants.vcf")
let passed = variants |> filter(|v| v.qual >= 30)

# With |> into — reads left-to-right:
variants |> filter(|v| v.qual >= 30) |> into passed_again
```

The syntax `expr |> into name` is equivalent to `let name = expr`. It evaluates the expression, binds the result to `name` (creating or shadowing), and returns the value.

This is especially useful in multi-step workflows where you need to reference intermediate results:

#81 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# Multi-step pipeline with into
read_fastq("data/reads.fastq")
  |> filter(|r| mean_phred(r.quality) >= 30)
  |> into high_quality

high_quality
  |> map(|r| {...r, trim_end: trim_quality(r.quality, 20)})
  |> into trimmed

print(f"Kept {len(high_quality)} reads, trimmed to {len(trimmed)}")
```

Each `|> into` step saves the result under a name so it can be reused later, without wrapping the whole chain in `let x = (...)`. The pipeline reads top-to-bottom, matching the conceptual flow of the analysis.

Example: FASTQ Quality Filtering Pipeline

A complete quality filtering workflow with logging at each step.

#82 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# quality_filter.bl
# Multi-step FASTQ quality filtering pipeline.

let input_r1 = "raw_data/sample_R1.fastq.gz"
let input_r2 = "raw_data/sample_R2.fastq.gz"

# Process R1 and R2 in parallel-style
let process_reads = |input_path, label| {
  read_fastq(input_path)
    |>> |reads| print(f"[{label}] Input: {len(reads)} reads")

    # Step 1: Remove short reads
    |> filter(|r| seq_len(r.seq) >= 50)
    |>> |reads| print(f"[{label}] After length filter: {len(reads)}")

    # Step 2: Filter by mean quality
    |> filter(|r| mean(r.quality) >= 25)
    |>> |reads| print(f"[{label}] After quality filter: {len(reads)}")

    # Step 3: Remove low-complexity reads
    |> filter(|r| {
      let gc = gc_content(r.seq)
      gc > 0.1 && gc < 0.9
    })
    |>> |reads| print(f"[{label}] After complexity filter: {len(reads)}")
}

let filtered_r1 = process_reads(input_r1, "R1")
let filtered_r2 = process_reads(input_r2, "R2")

# Keep only concordant pairs (both mates survived)
let r1_ids = filtered_r1 |> map(|r| r.id) |> set()
let r2_ids = filtered_r2 |> map(|r| r.id) |> set()
let shared_ids = intersection(r1_ids, r2_ids)

let paired_r1 = filtered_r1 |> filter(|r| contains(shared_ids, r.id))
let paired_r2 = filtered_r2 |> filter(|r| contains(shared_ids, r.id))

print(f"\nFinal paired reads: {len(paired_r1)} pairs")

paired_r1 |> write_fastq("filtered/sample_R1.fastq.gz")
paired_r2 |> write_fastq("filtered/sample_R2.fastq.gz")

# Orphan reads (mate was filtered out)
let orphan_r1 = filtered_r1 |> filter(|r| !contains(shared_ids, r.id))
let orphan_r2 = filtered_r2 |> filter(|r| !contains(shared_ids, r.id))
let orphans = orphan_r1 ++ orphan_r2
print(f"Orphan reads: {len(orphans)}")
orphans |> write_fastq("filtered/sample_orphans.fastq.gz")
```

Example: Variant Annotation Pipeline

Read VCF, classify variants, annotate against known databases, filter, and report.

#83 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# annotate_variants.bl
# Variant annotation and classification pipeline.

let known_pathogenic = csv("clinvar_pathogenic.csv")
  |> map(|r| f"{r.chrom}:{r.pos}:{r.ref}>{r.alt}")
  |> set()

let gnomad_af = csv("gnomad_af.csv")
  |> map(|r| {key: f"{r.chrom}:{r.pos}:{r.ref}>{r.alt}", af: into(r.af,
"Float")})
  |> group_by("key")
  |> map(|g| {key: g.key, af: first(g.values).af})

read_vcf("data/variants.vcf")

  # Step 1: Basic quality filter
  |> filter(|v| v.qual >= 30 && v.filter == "PASS")
  |>> |vs| print(f"After quality filter: {len(vs)}")

  # Step 2: Classify variant type
  |> map(|v| {
    ...v,
    key: f"{v.chrom}:{v.pos}:{v.ref}>{v.alt}",
    vtype: variant_type(variant(v.chrom, v.pos, v.ref, v.alt))
  })

  # Step 3: Annotate with population frequency
  |> map(|v| {
    let gnomad = gnomad_af |> find(|g| g.key == v.key)
    {...v, gnomad_af: gnomad?.af ?? 0.0}
  })

  # Step 4: Annotate with ClinVar pathogenicity
  |> map(|v| {
    ...v,
    is_known_pathogenic: contains(known_pathogenic, v.key)
  })

  # Step 5: Assign priority tier
  |> map(|v| {
    let tier = if v.is_known_pathogenic then "Tier1_Known"
      else if v.gnomad_af < 0.001 && v.vtype != "SNV" then "Tier2_RareIndel"
      else if v.gnomad_af < 0.01 then "Tier3_Rare"
      else "Tier4_Common"
    {...v, tier: tier}
  })
  |>> |vs| {
    let tier_counts = vs |> group_by("tier") |> map(|g| f"{g.key}:
{len(g.values)}")
    print(f"Tier distribution: {tier_counts}")
  }

  # Step 6: Filter to actionable variants
  |> filter(|v| v.tier == "Tier1_Known" || v.tier == "Tier2_RareIndel")
  |>> |vs| print(f"Actionable variants: {len(vs)}")

  # Step 7: Sort by priority and position
  |> sort_by(|v| v.pos)
  |> sort_by(|v| v.chrom)
  |> sort_by(|v| v.tier)

  # Step 8: Generate report
  |> map(|v| {
    chrom: v.chrom,
    pos: v.pos,
```

```
    ref_allele: v.ref,
    alt_allele: v.alt,
    type: v.vtype,
    qual: v.qual,
    gnomad_af: v.gnomad_af,
    tier: v.tier,
    clinvar: if v.is_known_pathogenic then "Pathogenic" else "."
  })
  |> write_tsv("annotated_actionable.csv")
```

Example: Multi-Sample Comparison with Zip and Reduce

Compare variant calls across matched tumor-normal pairs and compute concordance.

```
# multi_sample_compare.bl
# Compare variant calls across tumor-normal pairs.

let pairs = [
  {tumor: "patient1_tumor.vcf.gz", normal: "patient1_normal.vcf.gz", patient:
"P001"},
  {tumor: "patient2_tumor.vcf.gz", normal: "patient2_normal.vcf.gz", patient:
"P002"},
  {tumor: "patient3_tumor.vcf.gz", normal: "patient3_normal.vcf.gz", patient:
"P003"}
]

# Process each pair
let pair_results = pairs |> map(|pair| {
  let tumor_vars = read_vcf(pair.tumor)
    |> filter(|v| v.qual >= 30)
    |> map(|v| f"{v.chrom}:{v.pos}:{v.ref}>{v.alt}")
    |> set()

  let normal_vars = read_vcf(pair.normal)
    |> filter(|v| v.qual >= 30)
    |> map(|v| f"{v.chrom}:{v.pos}:{v.ref}>{v.alt}")
    |> set()

  let somatic = difference(tumor_vars, normal_vars)
  let germline = intersection(tumor_vars, normal_vars)
  let normal_only = difference(normal_vars, tumor_vars)

  {
    patient: pair.patient,
    tumor_total: len(tumor_vars),
    normal_total: len(normal_vars),
    somatic: len(somatic),
    germline: len(germline),
    normal_only: len(normal_only),
    somatic_variants: somatic
  }
})

# Summary table
let summary = table(pair_results |> map(|r| {
  patient: r.patient,
  tumor_total: r.tumor_total,
  normal_total: r.normal_total,
  somatic: r.somatic,
  germline: r.germline,
  somatic_rate: r.somatic / r.tumor_total * 100.0
}))

print("Patient Variant Comparison")
print("=" * 70)
summary |> each(|row|
  print(f"  {row.patient}: tumor={row.tumor_total}, somatic={row.somatic}
({row.somatic_rate:.1f}%), germline={row.germline}")
)

# Find recurrent somatic variants (present in 2+ patients)
let all_somatic = pair_results
  |> flat_map(|r| r.somatic_variants |> collect() |> map(|v| {variant: v,
patient: r.patient}))
  |> group_by("variant")
  |> filter(|g| len(g.values) >= 2)
  |> map(|g| {
    variant: g.key,
    n_patients: len(g.values),
```

```
    patients: g.values |> map(|v| v.patient)
  })
  |> sort_by(|row| -row.n_patients)

print(f"\nRecurrent somatic variants (in >= 2 patients): {len(all_somatic)}")
all_somatic |> take(20) |> each(|v|
  print(f"  {v.variant} -- found in {v.n_patients} patients: {v.patients}")
)

# Cross-patient aggregates via reduce
let totals = pair_results |> reduce(
  {total_somatic: 0, total_germline: 0, patients: 0},
  |acc, r| {
    total_somatic: acc.total_somatic + r.somatic,
    total_germline: acc.total_germline + r.germline,
    patients: acc.patients + 1
  }
)

print(f"\nCohort totals across {totals.patients} patients:")
print(f"  Mean somatic variants: {totals.total_somatic /
totals.patients:.0f}")
print(f"  Mean germline variants: {totals.total_germline /
totals.patients:.0f}")

# Zip tumor-normal read depths for concordance check
let p1_tumor_depths = read_vcf(pairs[0].tumor) |> map(|v| into(v.info?.DP ??
"0", "Int"))
let p1_normal_depths = read_vcf(pairs[0].normal) |> map(|v| into(v.info?.DP
?? "0", "Int"))

let depth_comparison = zip(p1_tumor_depths, p1_normal_depths)
  |> map(|pair| {tumor_dp: pair[0], normal_dp: pair[1], ratio: pair[0] /
(pair[1] + 1.0)})
  |> filter(|d| d.ratio > 3.0)   # flag positions with 3x tumor vs normal
depth

print(f"\nPatient 1 depth outliers (tumor/normal > 3x):
{len(depth_comparison)}")
```

Summary

Pipes and transforms are the backbone of BioLang. The `|>` operator turns nested function calls into linear, readable workflows. The tap pipe `|>>` lets you observe and debug without disrupting the data flow. The `|> into` operator captures intermediate results without breaking left-to-right reading order. Higher-order functions like `map`, `filter`, `reduce`, `flat_map`, `zip`, and `window` cover the full range of data transformation patterns you encounter in bioinformatics pipelines.

Every example in this chapter represents a real analysis pattern: quality filtering, variant annotation, multi-sample comparison. As your pipelines grow more complex, pipes keep them readable and composable.

Functions and Closures

Functions are the primary unit of reuse in BioLang. Whether you are writing a normalisation routine for RNA-seq counts, a scoring function for sequence alignment, or a filter predicate for variant calls, functions let you name a computation and invoke it wherever it is needed.

Defining Functions

A function is introduced with the `fn` keyword, followed by a name, a parameter list, and a body enclosed in braces.

#85 ▶ Run ▶▶ Run All Above + This

```
fn gc_content(seq) {
    let bases = range(0, seq_len(seq)) |> map(|i| char_at(str(seq), i))
    let gc = bases |> filter(|b| b == "G" || b == "C") |> len()
    gc / seq_len(seq)
}

let ratio = gc_content(dna"ATGCGCTA")
# ratio => 0.5
```

The last expression in the body is the implicit return value. There is no need for a `return` keyword in the common case.

Explicit Return

When you need to exit early – for example after detecting an invalid input – use `return`.

#86 ▶ Run ▶▶ Run All Above + This

```
fn median_quality(quals) {
    if len(quals) == 0 then {
        return 0.0
    }
    let sorted = quals |> sort()
    let mid = len(sorted) / 2
    if len(sorted) % 2 == 0 then {
        (sorted[mid - 1] + sorted[mid]) / 2.0
    } else {
        sorted[mid]
    }
}
```

Default Parameters

Parameters can carry default values. Callers may omit them.

```
# Conceptual or diagnostic example; not directly executable.
fn trim_reads(records, min_qual: 20, min_len: 36) {
    records
        |> filter(|r| mean(r.quality) >= min_qual)
        |> filter(|r| seq_len(r.seq) >= min_len)
}

# Use defaults
let trimmed = trim_reads(raw_reads)

# Override quality threshold only
let strict = trim_reads(raw_reads, min_qual: 30)
```

Default parameters must appear after positional parameters.

Named Return Values

You can declare named return fields to make the return shape explicit.

```
fn count_variants(vcf_path) -> (snps: Int, indels: Int, other: Int) {
    let records = read_vcf(vcf_path)
    let snps = 0
    let indels = 0
    let other = 0
    records |> each(|v| {
        let vt = variant_type(v)
        if vt == "Snp" then snps = snps + 1
        else if vt == "Indel" then indels = indels + 1
        else other = other + 1
    })
}
```

Closures and Lambdas

A closure is an anonymous function written with pipe delimiters. Short closures use a single expression; longer ones use `{ }` for a block body.

#88 ▶ Run ▶▶ Run All Above + This

```
# Single-expression closure
let is_high_qual = |v| v.qual >= 30.0

# Block closure
let normalise = |counts, size_factors| {
    let total = counts |> reduce(0, |a, b| a + b)
    counts |> map(|c| (c / total) * 1e6 / size_factors)
}
```

Closures capture variables from the surrounding scope, which makes them ideal for building specialised predicates on the fly.

#89 ▶ Run ▶▶ Run All Above + This

```
fn make_depth_filter(min_depth) {
    |record| record.info.DP >= min_depth
}

let deep_enough = make_depth_filter(30)
let variants = [
    {info: {DP: 12}},
    {info: {DP: 48}}
]
let filtered = variants |> filter(deep_enough)
```

Higher-Order Functions

A higher-order function accepts another function (or closure) as an argument, returns one, or both.

#90 ▶ Run ▶▶ Run All Above + This

```
fn apply_qc_chain(reads, filters) {
    filters |> reduce(reads, |current, f| current |> filter(f))
}

let filters = [
    |r| mean_phred(r.quality) >= 20,
    |r| seq_len(r.seq) >= 50,
    |r| gc_content(r.seq) < 0.8
]

let raw_reads = read_fastq("data/reads.fastq")
let passing = apply_qc_chain(raw_reads, filters)
```

Because pipe inserts as the first argument, you can chain higher-order functions fluently:

#91 ▶ Run ▶▶ Run All Above + This

```
let counts = [
    {gene_name: "TP53", biotype: "protein_coding", tpm: 72.4},
    {gene_name: "MALAT1", biotype: "lncRNA", tpm: 530.0},
    {gene_name: "BRCA1", biotype: "protein_coding", tpm: 18.7}
]
let top_genes = counts
    |> filter(|g| g.biotype == "protein_coding")
    |> sort_by(|g| -g.tpm)
    |> take(100)
    |> map(|g| g.gene_name)
```

Where Clauses

A `where` clause attaches a precondition to a function. If the predicate is false at call time, a runtime error is raised.

#92 ▶ Run ▶▶ Run All Above + This

```
fn normalise_counts(counts) where len(counts) > 0 {
    let total = counts |> reduce(0, |a, b| a + b)
    counts |> map(|c| c / total)
}

fn log2_fold_change(treated, control) where control > 0.0 {
    log2(treated / control)
}

fn call_consensus(reads) where len(reads) >= 3 {
    let bases = reads |> map(|r| {base: r.base})
    let counts = table(bases) |> group_by("base")
      |> summarize(|base, group| {base: base, n: len(group)})
    counts |> sort_by(|g| -g.n) |> first() |> |g| g.base
}
```

Where clauses serve as executable documentation: they state the contract and enforce it at the boundary.

The @memoize Decorator

Bioinformatics computations are often called repeatedly with the same inputs – the same gene queried in an annotation database, the same k-mer scored in a seed-extension aligner. The `@memoize` decorator caches results keyed by argument values.

#93 ▶ Run ▶▶ Run All Above + This

```
@memoize
fn gc_of_kmer(kmer) {
    gc_content(dna(kmer))
}

# First call computes and caches the result.
# Subsequent calls with the same k-mer return instantly.
let g1 = gc_of_kmer("ATCGGC")  # cache miss
let g2 = gc_of_kmer("ATCGGC")  # cache hit
```

Memoization is especially valuable when combined with recursion.

Decorator Notes

`@memoize` is currently the only built-in decorator. For cross-cutting concerns like timing or input validation, use wrapper functions or `where` clauses.

#94 ▶ Run ▶▶ Run All Above + This

```
# Use a wrapper function for timing
fn timed_align(fastq, ref_genome) {
    let t0 = timestamp()
    let result = align_reads(fastq, ref_genome)
    print("Alignment took " + str(timestamp() - t0) + "s")
    result
}

# Use where clauses for input validation
fn merge_bams(bam_paths) where len(bam_paths) > 0 {
    # ... merge logic ...
}
```

Recursion

Recursive functions call themselves. BioLang supports standard recursion, which is useful for tree-structured biological data such as phylogenies and ontology DAGs.

#95 ▶ Run ▶▶ Run All Above + This

```
fn tree_leaf_count(node) {
    if len(node.children) == 0 then {
        return 1
    }
    node.children |> map(tree_leaf_count) |> reduce(0, |a, b| a + b)
}
```

Combine `@memoize` with recursion for dynamic-programming-style algorithms:

```
# Conceptual or diagnostic example; not directly executable.
@memoize
fn needleman_wunsch_score(seq1, seq2, i, j, gap_penalty: -2) {
    if i == 0 then { return j * gap_penalty }
    if j == 0 then { return i * gap_penalty }

    let match_score = if seq1[i - 1] == seq2[j - 1] then 1 else -1

    let diag   = needleman_wunsch_score(seq1, seq2, i - 1, j - 1) +
match_score
    let up     = needleman_wunsch_score(seq1, seq2, i - 1, j) + gap_penalty
    let left   = needleman_wunsch_score(seq1, seq2, i, j - 1) + gap_penalty

    max(diag, max(up, left))
}

let score = needleman_wunsch_score("AGTACG", "ACATAG", 6, 6)
```

Example: Memoized K-mer Scoring Function

Counting k-mer matches across many sequences calls the same function millions of times. Caching the GC computation removes redundant work.

```
# Conceptual or diagnostic example; not directly executable.
@memoize
fn kmer_gc(kmer_str) {
    gc_content(dna(kmer_str))
}

fn score_sequence_kmers(seq, k: 21) {
    let kmer_list = seq |> kmers(k)
    let gc_scores = kmer_list |> map(|km| kmer_gc(to_string(km)))
    {
        n_kmers: len(gc_scores),
        mean_gc: mean(gc_scores),
        high_gc_count: gc_scores |> filter(|g| g > 0.6) |> len()
    }
}

let result = score_sequence_kmers(dna"ATCGATCGCCCCGGGGATATATATCGCGCGATCG")
print(f"K-mers: {result.n_kmers}, Mean GC: {result.mean_gc:.3f}")
```

Example: Reusable QC Filter Factory

A factory function returns a closure configured with experiment-specific thresholds. Different sequencing protocols produce different acceptable ranges.

```
# Conceptual or diagnostic example; not directly executable.
fn make_qc_filter(min_qual: 20, min_len: 50, max_n_frac: 0.05) {
    |read| {
        let avg_q = mean(read.quality)
        let n_frac = (read.seq |> filter(|b| b == "N") |> len()) /
seq_len(read.seq)
        avg_q >= min_qual && seq_len(read.seq) >= min_len && n_frac <=
max_n_frac
    }
}

# Whole-genome: permissive
let wgs_filter = make_qc_filter(min_qual: 15, min_len: 36)

# Amplicon: strict
let amp_filter = make_qc_filter(min_qual: 30, min_len: 100, max_n_frac: 0.01)

let wgs_reads = read_fastq("data/reads.fastq") |> filter(wgs_filter)
let amp_reads = read_fastq("data/reads.fastq") |> filter(amp_filter)

print("WGS passing: " + to_string(len(wgs_reads)))
print("Amplicon passing: " + to_string(len(amp_reads)))
```

Example: Recursive Phylogenetic Tree Traversal

Phylogenetic trees are naturally recursive. This example collects all leaf taxa whose branch length from the root exceeds a threshold – useful for detecting fast-evolving lineages.

#96 ▶ Run ▶▶ Run All Above + This

```
fn collect_distant_leaves(node, dist_so_far, threshold) {
    let current_dist = dist_so_far + (node.branch_length ?? 0.0)

    if len(node.children) == 0 then {
        # Leaf node
        if current_dist > threshold then {
            return [{name: node.name, distance: current_dist}]
        } else {
            return []
        }
    }

    # Internal node: recurse into children and concatenate results
    node.children
        |> flat_map(|child| collect_distant_leaves(child, current_dist,
threshold))
}

# Build a simple tree as nested records
let tree = {
    name: "root", branch_length: 0.0, children: [
        {name: "primates", branch_length: 0.08, children: [
            {name: "human", branch_length: 0.1, children: []},
            {name: "chimp", branch_length: 0.12, children: []}
        ]},
        {name: "rodents", branch_length: 0.20, children: [
            {name: "mouse", branch_length: 0.35, children: []},
            {name: "rat", branch_length: 0.33, children: []}
        ]}
    ]
}

let fast_evolving = collect_distant_leaves(tree, 0.0, 0.30)

fast_evolving |> each(|taxon|
    print(taxon.name + " => total branch length " +
to_string(taxon.distance))
)
# mouse => total branch length 0.55
# rat => total branch length 0.53
```

Summary

Feature	Syntax	Use Case
Function	`fn name(params) { }`	Named, reusable computation

Feature	Syntax	Use Case
Default params	`fn f(x, k: 10) { }`	Flexible call sites
Where clause	`fn f(x) where x > 0 { }`	Precondition contracts
Closure	`|x| expr`	Inline predicates, callbacks
Block closure	`|x| { stmts }`	Multi-step anonymous logic
@memoize	`@memoize fn f() { }`	Cache repeated computations
Recursion	Self-call in body	Trees, dynamic programming

Functions and closures are the building blocks that the rest of BioLang – pipelines, parallel maps, and the standard library – is built upon. Master them and every subsequent chapter becomes easier.

Control Flow

BioLang provides a rich set of control-flow constructs designed for the messy realities of biological data. Variants need classification, samples need multi-criteria gating, and searches over sequences must handle both the found and not-found cases gracefully.

if/else Expressions

`if/else` in BioLang is an expression – it returns a value. This lets you embed conditional logic directly in assignments and pipes.

#97 ▶ Run ▶▶ Run All Above + This

```
let seq = dna"ATGCGCATGC"
let label = if gc_content(seq) > 0.6 {
    "GC-rich"
} else if gc_content(seq) < 0.4 {
    "AT-rich"
} else {
    "balanced"
}
```

Because it is an expression, you can use it inline:

#98 ▶ Run ▶▶ Run All Above + This

```
let total_depth = 80
let alt_count = 24
let allele_freq = if total_depth > 0 { alt_count / total_depth } else { 0.0 }
```

match Expressions

`match` compares a value against a series of patterns. Like `if`, it is an expression and returns the result of the matching arm.

#99 ▶ Run ▶▶ Run All Above + This

```
let codon = "ATG"

let amino_acid = match codon {
    "ATG" => "Met"
    "TGG" => "Trp"
    "TAA" | "TAG" | "TGA" => "Stop"
    _ => codon_usage(codon)
}
```

Patterns can destructure records:

```
# Conceptual or diagnostic example; not directly executable.
fn describe_alignment(aln) {
    match aln {
        {mapped: true, mapq: q} if q >= 30 => "high-confidence"
        {mapped: true, mapq: q} if q >= 10 => "low-confidence"
        {mapped: true}                     => "very-low-quality"
        {mapped: false}                    => "unmapped"
    }
}
```

for Loops

The basic `for` loop iterates over any collection or stream.

#100 ▶ Run ▶▶ Run All Above + This

```
let reads = read_fastq("data/reads.fastq")
let total_bases = 0

for read in reads {
    total_bases = total_bases + len(read.seq)
}

print("Total bases: " + str(total_bases))
```

Tuple Destructuring

When iterating over paired data, destructure directly in the loop header.

#101 ▶ Run ▶▶ Run All Above + This

```
let sample_ids = ["S1", "S2", "S3"]
let fastq_paths = [
    "data/S1_R1.fastq.gz",
    "data/S2_R1.fastq.gz",
    "data/S3_R1.fastq.gz"
]

for (sample, path) in zip(sample_ids, fastq_paths) {
    let stats = read_fastq(path) |> read_stats()
    print(sample + ": " + str(stats.total_reads) + " reads")
}
```

Destructuring also works with `enumerate`:

#102 ▶ Run ▶▶ Run All Above + This

```
let exons = read_bed("data/exons.bed")

for (i, exon) in enumerate(exons) {
    print("Exon " + str(i + 1) + ": " + exon.chrom + ":" + str(exon.start) +
"-" + str(exon.end))
}
```

for/else

The `else` block executes only when the loop completes without hitting a `break`. This is perfect for search-and-report patterns.

#103 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# requires: internet connection
let target = dna"TATAAA"
let promoters = read_bed("data/regions.bed")

for region in promoters {
    let seq = ucsc_sequence("hg38", region.chrom, region.start, region.end)
    if contains(seq, target) {
        print("TATA box found in " + region.name)
        break
    }
} else {
    print("No TATA box found in any promoter region")
}
```

while Loops

Use `while` when the number of iterations is unknown ahead of time.

#104 ▶ Run ▶▶ Run All Above + This

```
fn find_orfs_manual(seq) {
    let i = 0
    let in_orf = false
    let orf_start = 0
    let orfs = []

    while i + 2 < len(seq) {
        let codon = slice(seq, i, i + 3)
        if codon == "ATG" && !in_orf {
            in_orf = true
            orf_start = i
        }
        if (codon == "TAA" || codon == "TAG" || codon == "TGA") && in_orf {
            orfs = orfs + [{start: orf_start, end: i + 3, length: i + 3 -
orf_start}]
            in_orf = false
        }
        i = i + 3
    }
    orfs
}

let orfs = find_orfs_manual("ATGAAACCCTAGATGTTTGAATAA")
# [{start: 0, end: 12, length: 12}, {start: 12, end: 24, length: 12}]
```

break and continue

`break` exits the innermost loop. `continue` skips to the next iteration.

#105 ▶ Run ▶▶ Run All Above + This

```
let variants = read_vcf("data/variants.vcf")
let first_pathogenic = nil

for v in variants {
    # Skip low-quality calls
    if v.qual < 30.0 {
        continue
    }

    let info = parse_info(v.info_str)
    if info?.CLNSIG == "Pathogenic" {
        first_pathogenic = v
        break
    }
}
```

given/otherwise

`given/otherwise` is a declarative chain of conditions. It reads top to bottom; the first true condition wins. `otherwise` is the fallback.

#106 ▶ Run ▶▶ Run All Above + This

```
fn classify_read_pair(r1_len, r2_len, insert_size) {
    given {
        insert_size < 0         => "invalid_pair"
        insert_size > 1000      => "structural_variant_candidate"
        r1_len < 36 || r2_len < 36 => "short_read"
        insert_size < 150       => "short_insert"
        otherwise               => "normal"
    }
}
```

`given` is an expression, so you can assign its result:

#107 ▶ Run ▶▶ Run All Above + This

```
let allele_freq = 0.32
let coverage = 45
let risk = given {
    allele_freq >= 0.5 && coverage >= 30 => "high_confidence_somatic"
    allele_freq >= 0.2                   => "moderate_evidence"
    allele_freq >= 0.05                  => "low_frequency"
    otherwise                            => "below_detection"
}
```

guard Clauses

`guard` asserts that a condition is true. If it is not, the `else` block executes – typically an early return or error.

#108 ▶ Run ▶▶ Run All Above + This

```
fn calculate_tmb(variants, exome_size_mb) {
    guard exome_size_mb > 0 else {
        return {error: "Exome size must be positive"}
    }
    guard len(variants) > 0 else {
        return {tmb: 0.0, classification: "low"}
    }

    let somatic = variants |> filter(|v| v.filter == "PASS")
    let tmb = len(somatic) / exome_size_mb

    let classification = given {
        tmb >= 20.0  => "high"
        tmb >= 10.0  => "intermediate"
        otherwise    => "low"
    }

    {tmb: tmb, classification: classification}
}
```

Guards keep the main logic at the top indentation level by pushing error handling to the margin. Use them liberally for input validation.

unless

`unless` is syntactic sugar for `if !condition`. It reads naturally for negative checks.

#109 ▶ Run ▶▶ Run All Above + This

```
fn process_bam(path) {
    let header = sam_header(path)

    unless contains(header.sort_order, "coordinate") {
        print("WARNING: BAM is not coordinate-sorted, sorting first")
        shell("samtools sort -o " + path + " " + path)
    }

    unless file_exists(path + ".bai") {
        print("Indexing BAM")
        shell("samtools index " + path)
    }

    # Proceed with analysis
    let stats = flagstat(path)
    stats
}
```

Example: Classify Variants by Type

Read a VCF and produce a summary table of variant types using `match`.

```
# Conceptual or diagnostic example; not directly executable.
let variants = read_vcf("data/variants.vcf")

let classified = variants |> map(|v| {
    let vtype = match true {
        is_snp(v) && is_transition(v)   => "transition"
        is_snp(v) && is_transversion(v) => "transversion"
        is_snp(v)                       => "snp_other"
        is_indel(v) && len(v.alt) > len(v.ref) => "insertion"
        is_indel(v)                     => "deletion"
        _                               => "complex"
    }
    {...v, classification: vtype}
})

let summary = classified
    |> group_by("classification")
    |> map(|group| {type: group.0, count: len(group.1)})
    |> sort(|a, b| b.count - a.count)

for row in summary {
    print(row.type + ": " + str(row.count))
}
# transition: 45231
# transversion: 22890
# deletion: 3412
# insertion: 2876
# complex: 134
```

Example: Search for Motif with for/else

Scan upstream regions for a transcription-factor binding motif. Report the first hit or state that none was found.

#110 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# requires: internet connection
let motif = dna"CANNTG"  # E-box motif (N = any base)
let genes = read_gff("data/annotations.gff")
    |> filter(|f| f.type == "gene" && f.biotype == "protein_coding")

for gene in genes {
    let upstream = interval(gene.chrom, gene.start - 2000, gene.start)
    let seq = ucsc_sequence("hg38", gene.chrom, gene.start - 2000,
gene.start)
    let hits = find_motif(seq, motif)

    if len(hits) > 0 {
        print("E-box found upstream of " + gene.name + " at offset " +
str(hits[0].position))
        break
    }
} else {
    print("No E-box motif found in any upstream region scanned")
}
```

Example: Multi-Criteria Sample QC with given/otherwise

Gate samples through a series of quality thresholds and assign a disposition.

#111 ▶ Run ▶▶ Run All Above + This

```
fn qc_disposition(stats) {
    given {
        stats.total_reads < 1_000_000 =>
            {pass: false, reason: "insufficient_reads", reads:
stats.total_reads}

        stats.pct_mapped < 70.0 =>
            {pass: false, reason: "low_mapping_rate", pct: stats.pct_mapped}

        stats.mean_depth < 10.0 =>
            {pass: false, reason: "low_depth", depth: stats.mean_depth}

        stats.pct_duplicate > 40.0 =>
            {pass: false, reason: "high_duplication", pct:
stats.pct_duplicate}

        stats.contamination > 0.03 =>
            {pass: false, reason: "contamination", frac: stats.contamination}

        otherwise =>
            {pass: true, reason: "all_checks_passed"}
    }
}

let samples = ["S001", "S002", "S003", "S004"]
let bam_dir = "data/aligned"

for sample in samples {
    let stats = flagstat(bam_dir + "/" + sample + ".bam")
    let result = qc_disposition(stats)

    if result.pass {
        print(sample + ": PASS")
    } else {
        print(sample + ": FAIL (" + result.reason + ")")
    }
}
```

Example: Input Validation with guard

A variant-calling wrapper that validates every precondition before running the expensive computation.

```
# Conceptual or diagnostic example; not directly executable.
fn call_variants(bam_path, ref_path, bed_path, min_depth: 10) {
    guard file_exists(bam_path) else {
        return {error: "BAM file not found: " + bam_path}
    }
    guard file_exists(ref_path) else {
        return {error: "Reference FASTA not found: " + ref_path}
    }
    guard file_exists(bed_path) else {
        return {error: "Target BED not found: " + bed_path}
    }

    let header = sam_header(bam_path)
    guard header.sort_order == "coordinate" else {
        return {error: "BAM must be coordinate-sorted"}
    }
    guard file_exists(bam_path + ".bai") else {
        return {error: "BAM index (.bai) not found"}
    }

    # All preconditions met -- run the caller
    let regions = read_bed(bed_path)
    let vcf_out = "calls.vcf"
    shell("bcftools mpileup -f " + ref_path + " -R " + bed_path + " " +
bam_path
        + " | bcftools call -mv -Ov -o " + vcf_out)

    let raw_calls = read_vcf(vcf_out)
    let filtered = raw_calls
        |> filter(|v| v.qual >= 30.0)

    {
        total_calls: len(raw_calls),
        passing_calls: len(filtered),
        variants: filtered
    }
}
```

Summary

Construct	Returns Value?	Best For
`if/else`	Yes	Binary or ternary decisions
`match`	Yes	Multi-arm pattern dispatch
`for`	No	Iteration over collections
`for/else`	No	Search with not-found fallback
`while`	No	Indeterminate iteration
`given/otherwise`	Yes	Declarative condition chains
`guard ... else`	No	Early-exit preconditions

Construct	Returns Value?	Best For
`unless`	No	Negative-condition readability

Choose the construct that best communicates intent. Use `guard` for validation at function boundaries, `given` for multi-criteria classification, `match` for type-driven dispatch, and `for/else` when a search must report failure.

Error Handling

Biological data is messy. FASTA files contain malformed headers, VCF records have missing INFO fields, and remote databases time out. BioLang provides layered error-handling mechanisms so you can write code that degrades gracefully instead of crashing mid-pipeline.

try/catch Blocks

Wrap any code that might fail in `try`. If an error occurs, control transfers to the `catch` block, where you can inspect the error and decide how to proceed.

#112 ▶ Run ▶▶ Run All Above + This

```
let records = try {
    read_vcf("data/variants.vcf")
} catch e {
    print("Failed to parse VCF: " + e.message)
    []
}
```

`try/catch` is an expression – it returns the value of whichever branch executes:

#113 ▶ Run ▶▶ Run All Above + This

```
let depth = try {
    let info = parse_info(variant.info_str)
    int(info["DP"])
} catch e {
    0  # default when DP is absent or unparseable
}
```

Catching Specific Errors

You can pattern-match on error types inside `catch`:

#114 ▶ Run ▶▶ Run All Above + This

```
# requires: internet connection
try {
    let result = fetch_uniprot(accession)
    process_protein(result)
} catch e {
    match e.kind {
        "network_error" => {
            print("Network unreachable, using cached data")
            load_cache("uniprot", accession)
        }
        "parse_error" => {
            print("Malformed response for " + accession + ", skipping")
            nil
        }
        _ => {
            print("Unexpected error: " + e.message)
            nil
        }
    }
}
```

Error Propagation

When a function cannot handle an error meaningfully, let it propagate to the caller.

#115 ▶ Run ▶▶ Run All Above + This

```
fn load_reference(path) {
    guard file_exists(path) else {
        error("Reference file not found: " + path)
    }
    let records = read_fasta(path)
    guard len(records) > 0 else {
        error("Reference file is empty: " + path)
    }
    records
}

# Caller decides how to handle it
try {
    let ref_seqs = load_reference("hg38.fa")
    run_alignment(reads, ref_seqs)
} catch e {
    print("Pipeline aborted: " + e.message)
}
```

The `error()` function raises an exception that unwinds to the nearest `catch`.

Retry Loops

Network calls to biological databases are unreliable. Use a loop with `try/catch` to retry a block up to a specified number of attempts.

#116 ▶ Run ▶▶ Run All Above + This

```
# requires: internet connection
# requires: NCBI_API_KEY (optional, increases rate limit)
let gene_info = nil
let attempts = 0
while attempts < 3 {
    try {
        gene_info = ncbi_gene("BRCA1")
        break
    } catch e {
        attempts = attempts + 1
        if attempts >= 3 { error("All attempts failed: " + e.message) }
        sleep(2000)
    }
}
```

If all attempts fail, the last error propagates. Combine with an outer `try/catch` for a fully defensive pattern:

#117 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# requires: internet connection
let annotation = try {
    let result = nil
    let attempts = 0
    while attempts < 5 {
        try {
            result = ensembl_gene("ENSG00000139618")
            break
        } catch e {
            attempts = attempts + 1
            if attempts >= 5 { error("All attempts failed: " + e.message) }
            sleep(1000)
        }
    }
    result
} catch e {
    print("Ensembl unavailable after 5 attempts: " + e.message)
    {gene_name: "BRCA2", source: "fallback_cache"}
}
```

Exponential Backoff

For high-traffic APIs, increase delay between retries using a helper:

```
# Conceptual or diagnostic example; not directly executable.
fn fetch_with_backoff(url, max_attempts: 5) {
    let attempt = 0
    let result = nil

    while attempt < max_attempts {
        try {
            result = http_get(url)
            break
        } catch e {
            attempt = attempt + 1
            if attempt >= max_attempts {
                error("All " + str(max_attempts) + " attempts failed: " +
e.message)
            }
            let wait = 1000 * pow(2, attempt - 1)  # 1s, 2s, 4s, 8s, 16s
            print("Attempt " + str(attempt) + " failed, retrying in " +
str(wait) + "ms")
            sleep(wait)
        }
    }
    result
}

let blast_result =
fetch_with_backoff("https://blast.ncbi.nlm.nih.gov/blast/Blast.cgi?RID=" +
rid)
```

Null Coalescing: ??

The `??` operator returns the left-hand side if it is not `nil`; otherwise it returns the right-hand side. This is indispensable for fields that may be absent.

#118 ▶ Run ▶▶ Run All Above + This

```
let depth = variant.info?.DP ?? 0
let gene_name = annotation?.gene_name ?? "unknown"
let af = variant.info?.AF ?? variant.info?.MAF ?? 0.0
```

`??` chains naturally. The first non-nil value wins:

#119 ▶ Run ▶▶ Run All Above + This

```
let record = {gene_id: "ENSG00000141510"}
let symbol = if has_key(record, "hugo_symbol") then record.hugo_symbol
    else if has_key(record, "gene_id") then record.gene_id
    else if has_key(record, "locus_tag") then record.locus_tag
    else "uncharacterised"
```

Optional Chaining: ?.

The `?.` operator short-circuits a field access chain when any intermediate value is `nil`. Instead of crashing, the entire expression evaluates to `nil`.

#120 ▶ Run ▶▶ Run All Above + This

```
# Without optional chaining -- crashes if info is nil or CLNSIG is absent
let sig = variant.info.CLNSIG

# With optional chaining -- returns nil safely
let sig = variant.info?.CLNSIG

# Deep chains
let protein_change = variant.annotation?.consequences?.protein?.hgvs
```

Combine with `??` for a complete default:

#121 ▶ Run ▶▶ Run All Above + This

```
let consequence = variant.annotation?.consequences?.most_severe ?? "unknown"
```

Defensive Patterns for Bio Data

Missing Fields in Records

Biological file formats are under-specified. A BED file might have 3 columns or 12. A VCF INFO field might omit half the keys. Build defensive accessors.

```
# Conceptual or diagnostic example; not directly executable.
fn safe_info(variant, key, default: nil) {
    try {
        let info = parse_info(variant.info_str)
        info[key] ?? default
    } catch e {
        default
    }
}

let dp = safe_info(v, "DP", default: 0)
let af = safe_info(v, "AF", default: 0.0)
let clnsig = safe_info(v, "CLNSIG", default: "not_reported")
```

Malformed Records

Skip bad records instead of aborting the whole file:

#122 ▶ Run ▶▶ Run All Above + This

```
fn robust_parse(lines, parser) {
    let good = []
    let bad = 0

    for (i, line) in enumerate(lines) {
        try {
            good = good + [parser(line)]
        } catch e {
            bad = bad + 1
            print("Skipped line " + str(i + 1) + ": " + e.message)
        }
    }

    if bad > 0 {
        print("Warning: " + str(bad) + " malformed records skipped")
    }
    good
}
```

Nil-safe Aggregation

When computing statistics over optional fields, filter out nil values first:

#123 ▶ Run ▶▶ Run All Above + This

```
let variants = read_vcf("data/variants.vcf")
let qualities = variants
    |> map(|v| v.qual)
    |> filter(|q| q != nil)

let mean_qual = if len(qualities) > 0 { mean(qualities) } else { 0.0 }
```

Example: Robust FASTA Parser with try/catch

Parse a FASTA file that may contain corrupted entries mixed with valid ones.

#124 ▶ Run ▶▶ Run All Above + This

```
fn parse_fasta_robust(path) {
    let raw = read_lines(path)
    let records = []
    let current_header = nil
    let current_seq = ""
    let errors = []

    for line in raw {
        if starts_with(line, ">") {
            # Flush previous record
            if current_header != nil {
                try {
                    guard len(current_seq) > 0 else {
                        error("Empty sequence for " + current_header)
                    }
                    guard is_dna(current_seq) else {
                        error("Invalid characters in " + current_header)
                    }
                    records = records + [{header: current_header, seq:
current_seq}]
                } catch e {
                    errors = errors + [{header: current_header, reason:
e.message}]
                }
            }
            current_header = slice(line, 1, len(line)) |> trim()
            current_seq = ""
        } else {
            current_seq = current_seq + trim(line)
        }
    }

    # Flush last record
    if current_header != nil {
        try {
            guard len(current_seq) > 0 else { error("Empty sequence") }
            guard is_dna(current_seq) else { error("Invalid characters") }
            records = records + [{header: current_header, seq: current_seq}]
        } catch e {
            errors = errors + [{header: current_header, reason: e.message}]
        }
    }

    print("Parsed " + str(len(records)) + " records, " + str(len(errors)) + "
errors")
    for err in errors {
        print("  SKIP: " + err.header + " (" + err.reason + ")")
    }
    records
}

let sequences = parse_fasta_robust("mixed_quality.fasta")
```

Example: Retrying API Calls to NCBI

Query NCBI's E-utilities for gene summaries, handling the rate limit and transient failures that are common on public APIs.

```
# Conceptual or diagnostic example; not directly executable.
# requires: internet connection
# requires: NCBI_API_KEY (optional, increases rate limit)
fn fetch_gene_summaries(gene_ids) {
    let results = []

    for id in gene_ids {
        let summary = try {
            let result = nil
            let attempts = 0
            while attempts < 3 {
                try {
                    let url =
"https://eutils.ncbi.nlm.nih.gov/entrez/eutils/esummary.fcgi"
                        + "?db=gene&id=" + str(id) + "&retmode=json"
                    let resp = http_get(url)
                    let data = parse_json(resp.body)
                    guard data?.result != nil else {
                        error("No result field in response")
                    }
                    result = data.result[str(id)]
                    break
                } catch e {
                    attempts = attempts + 1
                    if attempts >= 3 { error("All attempts failed: " +
e.message) }
                    sleep(1500)
                }
            }
            result
        } catch e {
            print("Failed to fetch gene " + str(id) + ": " + e.message)
            {gene_id: id, name: "unknown", description: "fetch_failed"}
        }

        results = results + [summary]

        # Respect NCBI rate limit: max 3 requests per second without API key
        sleep(400)
    }
    results
}

let gene_ids = [672, 675, 7157, 4609]  # BRCA1, BRCA2, TP53, MYC
let summaries = fetch_gene_summaries(gene_ids)

for s in summaries {
    print(str(s.gene_id ?? s.uid) + ": " + (s.name ?? "unknown") + " - " +
(s.description ?? "N/A"))
}
```

Example: Processing VCF with Missing INFO Fields

Real VCF files have inconsistent INFO columns. Some records carry `AF`, others do not. Some have `CLNSIG`, most do not. Use `?.` and `??` to process them uniformly.

```
# Conceptual or diagnostic example; not directly executable.
let variants = read_vcf("data/variants.vcf")

let annotated = variants |> map(|v| {
    let info = parse_info(v.info_str)

    let af = info?.AF ?? info?.CAF ?? 0.0
    let clinical_sig = info?.CLNSIG ?? "not_reported"
    let gene = split(info?.GENEINFO ?? "", ":") |> first() ?? "intergenic"
    let review_status = info?.CLNREVSTAT ?? "no_assertion"
    let origin = info?.ORIGIN ?? "unknown"

    {
        chrom: v.chrom,
        pos: v.pos,
        ref: v.ref,
        alt: v.alt,
        gene: gene,
        clinical_significance: clinical_sig,
        allele_frequency: af,
        review_status: review_status,
        origin: origin
    }
})

# Filter to pathogenic variants with at least two-star review
let pathogenic = annotated
    |> filter(|v| v.clinical_significance == "Pathogenic"
               || v.clinical_significance == "Likely_pathogenic")
    |> filter(|v| v.review_status != "no_assertion")

print("Pathogenic/likely pathogenic variants: " + str(len(pathogenic)))

for v in pathogenic |> take(10) {
    print(v.gene + " " + v.chrom + ":" + str(v.pos)
        + " " + v.ref + ">" + v.alt
        + " AF=" + str(v.allele_frequency))
}

# Write a clean report
write_tsv(pathogenic, "pathogenic_report.csv")
```

Summary

Mechanism	Syntax	Purpose
try/catch	`try { } catch e { }`	Graceful failure handling
error()	`error("msg")`	Raise an exception
retry loop	`while attempts < n { try { ... break } catch { ... sleep(ms) } }`	Transient-failure recovery
??	`val ?? default`	nil substitution

Mechanism	Syntax	Purpose
?.	`obj?.field`	Safe field access
guard	`guard cond else { }`	Precondition assertion

Layer these mechanisms: `guard` at function entry to reject invalid inputs, `?.` and `??` for data access, `try/catch` around I/O and parsing, and retry loops for network calls. Together they produce pipelines that finish with partial results instead of crashing with none.

File I/O

Bioinformatics begins and ends with files. Sequencers emit FASTQ, aligners produce BAM, callers output VCF, annotators read GFF. BioLang provides first-class readers and writers for every major format, with consistent interfaces and stream support for files that exceed available memory.

> **Sample data:** BioLang ships with sample files in `examples/sample-data/` covering FASTA, FASTQ, VCF, BED, GFF, CSV, TSV, and SAM formats. You can substitute these paths wherever the examples below use bare filenames. Run `bl run examples/quickstart.bl` to verify all files are present.

Reading FASTA

`read_fasta()` returns a table with `id`, `description`, `seq`, and `length` columns.

#125 ▶ Run ▶▶ Run All Above + This

```
let contigs = read_fasta("data/sequences.fasta")

for c in contigs {
    print(c.id + ": " + str(c.length) + " bp")
}

# Find the longest contig
contigs |> sort_by(|c| -c.length) |> first() |> into longest
print("Longest: " + longest.id + " (" + str(longest.length) + " bp)")
```

FASTA records carry the raw sequence as a string. Use bio literals for compile-time validated sequences, and `read_fasta` for runtime file data.

Reading FASTQ

`read_fastq()` returns a table with `id`, `description`, `seq`, `length`, and `quality` columns. The `quality` value is the FASTQ ASCII quality string; use `mean_phred`, `min_phred`, or `max_phred` to decode it.

#126 ▶ Run ▶▶ Run All Above + This

```
let reads = read_fastq("data/reads.fastq")

reads |> map(|r| gc_content(r.seq)) |> mean() |> into gc_mean
let summary = {
    total: len(reads),
    mean_length: reads |> map(|r| len(r.seq)) |> mean(),
    mean_qual: reads |> map(|r| mean_phred(r.quality)) |> mean(),
    gc_pct: gc_mean * 100
}

print("Reads: " + str(summary.total))
print("Mean length: " + str(round(summary.mean_length, 1)))
print("Mean quality: " + str(round(summary.mean_qual, 1)))
print("GC%: " + str(round(summary.gc_pct, 1)))
```

Compressed files (`.gz`) are decompressed transparently.

Reading VCF

`read_vcf()` returns variant records with fields `chrom`, `pos`, `id`, `ref`, `alt`, `qual`, `filter`, `info_str`, and `genotypes`.

#127 ▶ Run ▶▶ Run All Above + This

```
let variants = read_vcf("data/variants.vcf")

let passing = variants |> filter(|v| v.filter == "PASS")
let snps = passing |> filter(|v| is_snp(v))
let indels = passing |> filter(|v| is_indel(v))

print("PASS variants: " + str(len(passing)))
print("  SNPs: " + str(len(snps)))
print("  Indels: " + str(len(indels)))
```

Reading BED

`read_bed()` returns interval records with `chrom`, `start`, `end`, and optional `name`, `score`, `strand` fields depending on the number of columns.

#128 ▶ Run ▶▶ Run All Above + This

```
let targets = read_bed("data/exons.bed")

let total_bp = targets |> map(|t| t.end - t.start) |> reduce(0, |a, b| a + b)
print("Total target region: " + str(total_bp) + " bp")
print("Number of targets: " + str(len(targets)))

# Group by chromosome
let by_chrom = targets
    |> group_by("chrom")
    |> summarize(|chrom, regions| {
        chrom: chrom,
        targets: len(regions),
        bp: regions |> map(|r| r.end - r.start) |> reduce(0, |a, b| a + b)
    })
by_chrom |> each(|row|
    print(row.chrom + ": " + str(row.targets) + " targets, " + str(row.bp) +
" bp")
)
```

Reading GFF/GTF

`read_gff()` parses GFF3 into a table with `seqid`, `source`, `type`, `start`, `end`, `score`, `strand`, `phase`, and the raw `attributes` text. Use `read_gtf()` when you need parsed GTF attributes as a record.

#129 ▶ Run ▶▶ Run All Above + This

```
let features = read_gff("data/annotations.gff")

let genes = features |> filter(|f| f.type == "gene")
let protein_coding = genes |> filter(|g| contains(g.attributes,
"protein_coding"))

print("Total genes: " + str(len(genes)))
print("Protein-coding: " + str(len(protein_coding)))

# Gene length distribution
let lengths = protein_coding |> map(|g| g.end - g.start)
print("Median gene length: " + str(median(lengths)) + " bp")
print("Mean gene length: " + str(round(mean(lengths), 0)) + " bp")
```

Reading CSV and TSV

`csv()` and `tsv()` return tables where column headers become field names.

#130 ▶ Run ▶▶ Run All Above + This

```
let metadata = csv("sample_metadata.csv")

# Group samples by condition
let groups = metadata |> group_by("condition")

for (condition, samples) in groups {
    let ids = samples |> map(|s| s.sample_id) |> join(", ")
    print(condition + ": " + ids)
}

# DESeq2-style count matrix
let counts = tsv("gene_counts.tsv")
let gene_names = counts |> map(|row| row.gene_id)
print("Genes in count matrix: " + str(len(gene_names)))
```

Writing Files

Writer functions take a list of records and an output path. The format is determined by the function name.

#131 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# Write FASTA
let filtered = contigs |> filter(|c| len(c.seq) >= 500)
write_fasta(filtered, "assembly_filtered.fasta")

# Write FASTQ
let trimmed = reads |> map(|r| trim_quality(r, 20, 0))
write_fastq(trimmed, "trimmed_R1.fastq.gz")

# Write CSV
let stats_table = samples |> map(|s| {
    sample: s.id,
    total_reads: s.read_count,
    pct_mapped: round(s.mapped / s.read_count * 100, 2),
    mean_depth: round(s.depth, 1)
})
write_tsv(stats_table, "qc_summary.csv")
```

The `.gz` extension triggers gzip compression automatically.

Stream-Based Reading

For files too large to fit in memory – a 300 GB whole-genome BAM, a multi-sample VCF with millions of records – use streaming. Stream readers return a lazy sequence that is consumed once, record by record.

#132 ▶ Run ▶▶ Run All Above + This

```
# Stream a large FASTQ without loading it all
let stream = read_fastq("data/reads.fastq")

# Streams work with all HOFs -- processing is lazy
let high_qual_count = stream
    |> filter(|r| mean_phred(r.quality) >= 30)
    |> filter(|r| len(r.seq) >= 100)
    |> len()

print("High-quality long reads: " + str(high_qual_count))
```

Streams are consumed once. If you need multiple passes, collect into a list or read the file again:

#133 ▶ Run ▶▶ Run All Above + This

```
# Two-pass: first pass counts, second pass filters
let total = read_fastq("data/reads.fastq") |> len()
let passing = read_fastq("data/reads.fastq")
    |> filter(|r| mean_phred(r.quality) >= 20)
    |> len()

print("Pass rate: " + str(round(passing / total * 100, 2)) + "%")
```

Example: Convert FASTQ to FASTA

Strip quality scores from a FASTQ file to produce a FASTA for downstream assembly or BLAST.

#134 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
let reads = read_fastq("data/reads.fastq")

let fasta_records = reads |> map(|r| {
    header: r.id + " " + (r.description ?? ""),
    seq: r.seq
})

write_fasta(fasta_records, "nanopore_reads.fasta")
print("Converted " + str(len(fasta_records)) + " reads to FASTA")
```

For large files, this streams through without holding everything in memory because `map` on a stream produces a stream.

Example: Extract Coding Sequences from GFF + FASTA

Combine a genome FASTA with GFF annotation to extract CDS sequences for each protein-coding gene.

```
# Conceptual or diagnostic example; not directly executable.
let genome = read_fasta("data/sequences.fasta")
let features = read_gff("data/annotations.gff")

# Build a lookup from chromosome name to sequence
let chrom_seqs = {}
for contig in genome {
    let name = split(contig.header, " ") |> first()
    chrom_seqs = {...chrom_seqs, [name]: contig.seq}
}

# Collect CDS exons grouped by transcript
let cds_features = features
    |> filter(|f| f.type == "CDS")
    |> group_by("transcript_id")

let cds_records = []

for (tx_id, exons) in cds_features {
    let sorted_exons = exons |> sort(|a, b| a.start - b.start)
    let chrom = sorted_exons[0].chrom
    let strand = sorted_exons[0].strand

    guard chrom_seqs[chrom] != nil else { continue }

    let seq = sorted_exons
        |> map(|e| slice(chrom_seqs[chrom], e.start - 1, e.end))
        |> join("")

    let final_seq = if strand == "-" { reverse_complement(seq) } else { seq }

    cds_records = cds_records + [{
        header: tx_id + " " + chrom + " " + strand,
        seq: final_seq
    }]
}

write_fasta(cds_records, "coding_sequences.fasta")
print("Extracted " + str(len(cds_records)) + " coding sequences")
```

Example: Filter VCF by Quality and Write Passing Variants

Apply multiple quality filters to a VCF and write both passing and failing records to separate files.

```
# Conceptual or diagnostic example; not directly executable.
let variants = read_vcf("data/variants.vcf")

let results = variants |> map(|v| {
    let info = parse_info(v.info_str)
    let dp = info?.DP ?? 0
    let mq = info?.MQ ?? 0.0
    let af = info?.AF ?? 0.0

    let pass = v.qual >= 30.0
        && dp >= 10
        && mq >= 40.0
        && v.filter != "LowQual"

    {...v, qc_pass: pass, depth: dp, map_qual: mq, allele_freq: af}
})

let passing = results |> filter(|v| v.qc_pass)
let failing = results |> filter(|v| !v.qc_pass)

write_vcf(passing, "filtered_PASS.vcf")
write_vcf(failing, "filtered_FAIL.vcf")

print("Total: " + str(len(results)))
print("PASS: " + str(len(passing)))
print("FAIL: " + str(len(failing)))

# Summarise failure reasons
let low_qual = failing |> filter(|v| v.qual < 30.0) |> len()
let low_depth = failing |> filter(|v| v.depth < 10) |> len()
let low_mq = failing |> filter(|v| v.map_qual < 40.0) |> len()

print("Reasons (not mutually exclusive):")
print("  Low QUAL (<30): " + str(low_qual))
print("  Low depth (<10): " + str(low_depth))
print("  Low MQ (<40): " + str(low_mq))
```

Example: Merge BED Files and Compute Coverage

Load target regions from multiple BED files, merge overlapping intervals, and compute total coverage.

```
# Conceptual or diagnostic example; not directly executable.
let bed_files = [
    "targets_panel_v1.bed",
    "targets_panel_v2.bed",
    "custom_hotspots.bed"
]

# Read and concatenate all regions
let all_regions = bed_files
    |> flat_map(|path| {
        let regions = read_bed(path)
        print("Loaded " + str(len(regions)) + " regions from " + path)
        regions
    })

print("Total regions before merge: " + str(len(all_regions)))

# Sort by chromosome and start position
let sorted = all_regions
    |> sort(|a, b| {
        if a.chrom != b.chrom {
            compare(a.chrom, b.chrom)
        } else {
            a.start - b.start
        }
    })

# Merge overlapping intervals
fn merge_intervals(intervals) {
    guard len(intervals) > 0 else { return [] }

    let merged = [intervals[0]]

    for region in intervals |> drop(1) {
        let last_region = merged |> last()
        if region.chrom == last_region.chrom && region.start <=
last_region.end {
            # Overlapping -- extend the current interval
            let new_end = max(last_region.end, region.end)
            merged = merged |> take(len(merged) - 1)
            merged = merged + [{...last_region, end: new_end}]
        } else {
            merged = merged + [region]
        }
    }
    merged
}

let merged = merge_intervals(sorted)
print("Regions after merge: " + str(len(merged)))

let total_bp = merged |> map(|r| r.end - r.start) |> reduce(0, |a, b| a + b)
print("Total coverage: " + str(total_bp) + " bp (" + str(round(total_bp /
1e6, 2)) + " Mb)")

# Per-chromosome summary
let by_chrom = merged |> group_by("chrom")
for (chrom, regions) in by_chrom |> sort(|a, b| compare(a.0, b.0)) {
    let bp = regions |> map(|r| r.end - r.start) |> reduce(0, |a, b| a + b)
    print("  " + chrom + ": " + str(len(regions)) + " regions, " + str(bp) +
" bp")
}
```

```
# Write merged BED
write_bed(merged, "merged_targets.bed")
```

Format Reference

Function	Format	Returns
`read_fasta(path)`	FASTA	`[{header, seq}]`
`read_fastq(path)`	FASTQ	`[{id, seq, quality, comment}]`
`read_vcf(path)`	VCF	`[{chrom, pos, id, ref, alt, qual, filter, info_str, genotypes}]`
`read_bed(path)`	BED	`[{chrom, start, end, name?, score?, strand?}]`
`read_gff(path)`	GFF3/GTF	`[{chrom, source, type, start, end, score, strand, phase, attributes}]`
`csv(path)`	CSV	`[{col1: val, col2: val, ...}]`
`tsv(path)`	TSV	`[{col1: val, col2: val, ...}]`
`write_fasta(records, path)`	FASTA	writes file
`write_fastq(records, path)`	FASTQ	writes file
`write_vcf(records, path)`	VCF	writes file
`write_bed(records, path)`	BED	writes file
`write_tsv(records, path)`	CSV	writes file

All readers accept `.gz` paths and decompress automatically. All writers compress when the output path ends in `.gz`.

Genomic Intervals and Variants

Two data types sit at the heart of genomic analysis: intervals (regions on a chromosome) and variants (differences from a reference). BioLang provides built-in types and operations for both, including interval trees for fast overlap queries and variant classification functions that handle the full spectrum from SNPs to complex structural events.

Genomic Intervals

An interval represents a contiguous region on a chromosome. Create one with the `interval()` constructor.

#135 ▶ Run ▶▶ Run All Above + This

```
let promoter = interval("chr1", 11869, 14409)
let exon = interval("chr1", 11869, 12227, strand: "+")
```

Interval Fields

Every interval has `chrom`, `start`, `end`, and `strand`. Attach names, scores, or other annotations to table rows before building an interval tree.

#136 ▶ Run ▶▶ Run All Above + This

```
let region = interval("chr7", 55181000, 55182000, "+")

print(region.chrom)   # chr7
print(region.start)   # 55181000
print(region.end)     # 55182000
print(region.strand)  # +

let width = region.end - region.start
print("Width: " + str(width) + " bp")
```

BioLang uses 0-based, half-open coordinates (BED convention) throughout.

Interval Arithmetic

Test relationships between intervals directly.

#137 ▶ Run ▶▶ Run All Above + This

```
let a = interval("chr1", 100, 200)
let b = interval("chr1", 150, 250)
let c = interval("chr1", 300, 400)

# Overlap test
print(a.start < b.end and b.start < a.end)   # true
print(a.start < c.end and c.start < a.end)   # false

# Containment
let outer = interval("chr1", 100, 500)
print(contains(outer, c))  # true

# Distance between non-overlapping intervals
let dist = if a.end <= c.start then c.start - a.end else a.start - c.end
print(dist)   # 100

# Merge overlapping intervals into one
let merged = merge_intervals([a, b])
# [interval("chr1", 100, 250)]

# Intersection
let common = intersect(a, b)
# interval("chr1", 150, 200)
```

Interval Trees

When you have thousands of intervals and need to query overlaps repeatedly, build an interval tree. Construction is O(n log n); each query is O(log n + k) where k is the number of hits.

#138 ▶ Run ▶▶ Run All Above + This

```
let exons = read_bed("data/exons.bed")

let tree = interval_tree(exons)
```

query_overlaps

Find all intervals in the tree that overlap a query region.

#139 ▶ Run ▶▶ Run All Above + This

```
let query = interval("chr17", 43044294, 43044295)  # single position in BRCA1
let hits = query_overlaps(tree, query.chrom, query.start, query.end)

for hit in hits {
    print("Overlaps: " + hit.chrom + ":" + str(hit.start) + "-" +
str(hit.end))
}
```

query_nearest

Find the closest interval to a query, even if there is no overlap.

```
# Conceptual or diagnostic example; not directly executable.
let snp_pos = interval("chr7", 55181378, 55181379)
let nearest = query_nearest(tree, snp_pos)

let d = if snp_pos.end <= nearest.start then nearest.start - snp_pos.end else
snp_pos.start - nearest.end
print("Nearest feature: " + nearest.name
    + " at distance " + str(d) + " bp")
```

coverage

Compute per-base coverage across a set of intervals – how many intervals cover each position.

#140 ▶ Run ▶▶ Run All Above + This

```
let aligned_reads = table([
    {chrom: "chr17", start: 43044000, end: 43044400},
    {chrom: "chr17", start: 43044200, end: 43044600},
    {chrom: "chr17", start: 43044800, end: 43045200}
])
let tree = interval_tree(aligned_reads)
let target = interval("chr17", 43044000, 43045000)

let cov = coverage(tree)
    |> filter(|row| row.chrom == target.chrom && row.start < target.end &&
row.end > target.start)
print("Mean segment depth over target: " + str(mean(col(cov, "depth"))))
print("Segments >= 2x: " + str(cov |> filter(|row| row.depth >= 2) |> len()))
```

Variants

A variant represents a difference from the reference genome. Create one with the `variant()` constructor.

#141 ▶ Run ▶▶ Run All Above + This

```
let snp = variant("chr7", 55181378, "T", "A")           # EGFR T790M
let del = variant("chr17", 43045684, "TCAA", "T")       # BRCA1 deletion
let ins = variant("chr2", 29416089, "A", "AGCTGCTG")    # ALK insertion
```

Variant Classification

BioLang provides functions to classify variants by their molecular type.

#142 ▶ Run ▶▶ Run All Above + This

```
let v = variant("chr7", 55181378, "T", "A")

print(is_snp(v))           # true
print(is_indel(v))         # false
print(is_transition(v))    # false  (T>A is transversion)
print(is_transversion(v))  # true
print(variant_type(v))     # "Snp"
```

The full classification set:

#143 ▶ Run ▶▶ Run All Above + This

```
let examples = [
    variant("chr1", 100, "A", "G"),      # transition (purine to purine)
    variant("chr1", 200, "A", "T"),      # transversion
    variant("chr1", 300, "ACG", "A"),    # deletion
    variant("chr1", 400, "A", "ATCG"),   # insertion
    variant("chr1", 500, "ACG", "TGA"),  # MNP (multi-nucleotide
polymorphism)
]

for v in examples {
    let vtype = variant_type(v)
    let detail = given {
        is_snp(v) && is_transition(v)   => "transition"
        is_snp(v) && is_transversion(v) => "transversion"
        is_indel(v)                     => "indel (" + str(abs(len(v.alt) -
len(v.ref))) + "bp)"
        otherwise                       => vtype
    }
    print(v.chrom + ":" + str(v.pos) + " " + v.ref + ">" + v.alt + " => " +
detail)
}
```

Genotype Queries

When a variant carries genotype information (from a VCF), query zygosity.

#144 ▶ Run ▶▶ Run All Above + This

```
let variants = read_vcf("data/variants.vcf")

let het_snps = variants
    |> filter(|v| is_snp(v) && is_het(v))

let hom_alt = variants
    |> filter(|v| is_hom_alt(v))

print("Heterozygous SNPs: " + str(len(het_snps)))
print("Homozygous alt calls: " + str(len(hom_alt)))

# Allele balance for heterozygous calls
let allele_balances = het_snps |> map(|v| {
    let info = parse_info(v.info_str)
    let ad = info?.AD ?? [0, 0]
    let total = ad |> reduce(0, |a, b| a + b)
    if total > 0 { ad[1] / total } else { 0.0 }
})

print("Median allele balance (het): " + str(round(median(allele_balances),
3)))
```

parse_vcf_info

The VCF INFO column packs key-value pairs into a semicolon-delimited string. `parse_vcf_info()` turns it into a record.

#145 ▶ Run ▶▶ Run All Above + This

```
let info_str = "DP=42;MQ=60.0;AF=0.48;CLNSIG=Pathogenic;GENEINFO=BRCA1:672"
let info = parse_info(info_str)

print(info.DP)       # 42
print(info.AF)       # 0.48
print(info.CLNSIG)   # "Pathogenic"
print(info.GENEINFO) # "BRCA1:672"
```

Example: Find Genes Overlapping with ChIP-seq Peaks

Identify which genes have promoter regions that overlap with transcription factor binding peaks from a ChIP-seq experiment.

```
# Conceptual or diagnostic example; not directly executable.
# Load gene annotations
let genes = read_gff("data/annotations.gff")
    |> filter(|f| f.type == "gene" && f.attributes.gene_type ==
"protein_coding")

# Define promoter regions: 2kb upstream of TSS
let promoters = genes |> map(|g| {
    let tss = if g.strand == "+" { g.start } else { g.end }
    let prom_start = if g.strand == "+" { tss - 2000 } else { tss }
    let prom_end = if g.strand == "+" { tss } else { tss + 2000 }
    interval(g.chrom, max(0, prom_start), prom_end,
             name: g.attributes.gene_name)
})

# Load ChIP-seq peaks
let peaks = read_bed("data/regions.bed")
    |> map(|p| interval(p.chrom, p.start, p.end, name: p.name ?? "peak"))

# Build interval tree from promoters for fast lookup
let promoter_tree = interval_tree(promoters)

# Query each peak against promoters
let genes_with_peaks = []

for peak in peaks {
    let hits = query_overlaps(promoter_tree, peak)
    for hit in hits {
        genes_with_peaks = genes_with_peaks + [{
            gene: hit.name,
            peak_chrom: peak.chrom,
            peak_start: peak.start,
            peak_end: peak.end,
            overlap_bp: let iv = intersect(peak, hit); iv.end - iv.start
        }]
    }
}

# Deduplicate by gene name
let unique_genes = genes_with_peaks
    |> map(|g| g.gene)
    |> unique()

print("ChIP-seq peaks: " + str(len(peaks)))
print("Genes with H3K27ac at promoter: " + str(len(unique_genes)))

# Top genes by peak overlap size
let top = genes_with_peaks
    |> sort(|a, b| b.overlap_bp - a.overlap_bp)
    |> take(20)

for entry in top {
    print(entry.gene + ": " + str(entry.overlap_bp) + " bp overlap at "
        + entry.peak_chrom + ":" + str(entry.peak_start) + "-" +
str(entry.peak_end))
}

write_tsv(genes_with_peaks, "genes_with_h3k27ac_peaks.csv")
```

Example: Classify Variants and Generate a Ti/Tv Report

Transition/transversion ratio (Ti/Tv) is a key quality metric for SNP calling. Whole-genome sequencing typically yields Ti/Tv around 2.0-2.1; exome around 2.8-3.0. Deviations suggest systematic errors.

```
# Conceptual or diagnostic example; not directly executable.
let variants = read_vcf("data/variants.vcf")

# Keep only PASS SNPs
let pass_snps = variants
    |> filter(|v| v.filter == "PASS" && is_snp(v))

# Classify each SNP
let classified = pass_snps |> map(|v| {
    let cls = if is_transition(v) { "Ti" } else { "Tv" }
    let change = v.ref + ">" + v.alt
    {chrom: v.chrom, pos: v.pos, change: change, class: cls, qual: v.qual}
})

classified |> filter(|v| v.class == "Ti") |> len() |> into ti_count
classified |> filter(|v| v.class == "Tv") |> len() |> into tv_count
let ratio = if tv_count > 0 { ti_count / tv_count } else { 0.0 }

print("Transitions: " + str(ti_count))
print("Transversions: " + str(tv_count))
print("Ti/Tv ratio: " + str(round(ratio, 3)))

# Per-chromosome Ti/Tv
let by_chrom = classified |> group_by("chrom")

let chrom_report = by_chrom |> map(|group| {
    let chrom = group.0
    let vars = group.1
    let ti = vars |> filter(|v| v.class == "Ti") |> len()
    let tv = vars |> filter(|v| v.class == "Tv") |> len()
    {
        chrom: chrom,
        ti: ti,
        tv: tv,
        ratio: if tv > 0 { round(ti / tv, 3) } else { 0.0 },
        total: ti + tv
    }
})

chrom_report |> sort(|a, b| compare(a.chrom, b.chrom)) |> into sorted_report

print("\nPer-chromosome Ti/Tv:")
print("Chrom\tTi\tTv\tRatio\tTotal")
for row in sorted_report {
    print(row.chrom + "\t" + str(row.ti) + "\t" + str(row.tv)
        + "\t" + str(row.ratio) + "\t" + str(row.total))
}

# Substitution spectrum: count each of the 12 possible changes
let spectrum = classified
    |> group_by("change")
    |> map(|g| {change: g.0, count: len(g.1)})
    |> sort(|a, b| b.count - a.count)

print("\nSubstitution spectrum:")
for entry in spectrum {
    let bar = str_repeat("*", entry.count / 100)
    print(entry.change + "\t" + str(entry.count) + "\t" + bar)
}

write_tsv(sorted_report, "titv_per_chromosome.csv")
write_tsv(spectrum, "substitution_spectrum.csv")
```

Example: Annotate Variants with Overlapping Regulatory Regions

Given a set of variants and a regulatory region BED file (e.g., ENCODE cCREs), annotate each variant with the regulatory elements it falls within.

```
# Conceptual or diagnostic example; not directly executable.
# Load regulatory regions from ENCODE
let regulatory = read_bed("data/regions.bed")
    |> map(|r| interval(r.chrom, r.start, r.end, name: r.name ?? "cCRE"))

let reg_tree = interval_tree(regulatory)

# Load variants
let variants = read_vcf("data/variants.vcf")

# Annotate each variant with overlapping regulatory regions
let annotated = variants |> map(|v| {
    let pos_interval = interval(v.chrom, v.pos - 1, v.pos + len(v.ref) - 1)
    let overlapping = query_overlaps(reg_tree, pos_interval)
    let nearest = query_nearest(reg_tree, pos_interval)

    let reg_names = overlapping |> map(|r| r.name) |> join(";")
    let nearest_name = nearest?.name ?? "none"
    let nearest_dist = if len(overlapping) > 0 {
        0
    } else {
        if pos_interval.end <= nearest.start then nearest.start -
pos_interval.end else pos_interval.start - nearest.end
    }

    {
        chrom: v.chrom,
        pos: v.pos,
        ref: v.ref,
        alt: v.alt,
        in_regulatory: len(overlapping) > 0,
        regulatory_elements: if len(reg_names) > 0 { reg_names } else {
"none" },
        num_overlapping: len(overlapping),
        nearest_element: nearest_name,
        distance_to_nearest: nearest_dist
    }
})

# Summary statistics
let in_reg = annotated |> filter(|v| v.in_regulatory) |> len()
let total = len(annotated)
print("Variants in regulatory regions: " + str(in_reg) + "/" + str(total)
    + " (" + str(round(in_reg / total * 100, 1)) + "%)")

# Distribution of distance to nearest regulatory element
let distances = annotated
    |> filter(|v| !v.in_regulatory)
    |> map(|v| v.distance_to_nearest)

print("Non-regulatory variants:")
print("  Median distance to nearest cCRE: " + str(median(distances)) + " bp")
print("  Within 1kb of cCRE: " + str(distances |> filter(|d| d <= 1000) |>
len()))
print("  Within 10kb of cCRE: " + str(distances |> filter(|d| d <= 10000) |>
len()))

# Variants overlapping multiple regulatory elements
let multi = annotated |> filter(|v| v.num_overlapping > 1)
print("\nVariants in multiple regulatory regions: " + str(len(multi)))
for v in multi |> take(10) {
    print("  " + v.chrom + ":" + str(v.pos) + " overlaps " +
str(v.num_overlapping)
        + " elements: " + v.regulatory_elements)
}
```

```
write_tsv(annotated, "variants_regulatory_annotation.csv")
```

Summary

Interval Operations

Function	Description
`interval(chrom, start, end)`	Create an interval
`a.start < b.end and b.start < a.end`	Test for overlap
`contains(a, b)`	Test containment
`if a.end <= b.start then b.start - a.end else a.start - b.end`	Gap between non-overlapping intervals
`intersect(a, b)`	Overlapping portion
`merge_intervals([a, b])`	Merge list of overlapping intervals
`interval_tree(list)`	Build tree for fast queries
`query_overlaps(tree, q)`	All intervals overlapping q
`query_nearest(tree, q)`	Closest interval to q
`coverage(tree, region)`	Per-base depth array

Variant Operations

Function	Description
`variant(chrom, pos, ref, alt)`	Create a variant
`variant_type(v)`	“Snp”, “Indel”, “Mnp”, “Other”
`is_snp(v)`	Single nucleotide change
`is_indel(v)`	Insertion or deletion
`is_transition(v)`	Purine-purine or pyrimidine-pyrimidine

Function	Description
`is_transversion(v)`	Purine-pyrimidine or vice versa
`is_het(v)`	Heterozygous genotype
`is_hom_ref(v)`	Homozygous reference
`is_hom_alt(v)`	Homozygous alternate
`parse_vcf_info(str)`	Parse INFO column to record

Intervals and variants are the coordinate system of genomics. Master these types and you can express peak-calling, variant annotation, coverage analysis, and regulatory overlap queries concisely and efficiently.

Chapter 11: Pipelines

Bioinformatics workflows are rarely a single step. A typical analysis chains quality control, alignment, variant calling, filtering, and annotation into a multi-stage workflow. BioLang's pipe-first design with `|>` makes these workflows natural to write. For larger workflows with named stages, BioLang also provides `pipeline` blocks with `stage` declarations.

Pipe Composition: The Primary Pipeline Mechanism

The pipe operator `|>` is BioLang's core tool for building pipelines. It inserts the left-hand value as the first argument to the right-hand function, creating readable chains of transformations.

#146 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# Single-step pipe
let reads = read_fastq("data/reads.fastq")

# Multi-step pipeline via pipe chaining
let result = read_fastq("data/reads.fastq")
  |> filter(|r| mean_phred(r.quality) >= 30)
  |> map(|r| {seq: r.sequence, gc: gc_content(r.sequence)})
  |> sort_by(|a, b| b.gc - a.gc)

println("Top GC: " + str(result[0].gc))
```

Each `|>` feeds its result forward. You can chain as many steps as you need with no special syntax. This is how most BioLang pipelines are written.

Multi-Step Workflows with Variable Bindings

For workflows where intermediate results need names, use sequential `let` bindings connected by pipes. Each step is explicit and inspectable.

#147 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# FASTQ QC pipeline
let raw = read_fastq("data/reads.fastq")

let stats = raw
  |> map(|r| mean_phred(r.quality))
let avg_quality = mean(stats)
let total_reads = len(raw)

println("Reads: " + str(total_reads) + ", Mean Q: " + str(round(avg_quality,
1)))

# Filter and write
let passed = read_fastq("data/reads.fastq")
  |> filter(|r| mean_phred(r.quality) >= 25)
  |> collect()

write_fastq(passed, "sample_R1.filtered.fastq.gz")
println("Kept " + str(len(passed)) + " of " + str(total_reads) + " reads")
```

This pattern gives you full control: name any intermediate, inspect it, branch on it, or feed it into multiple downstream steps.

Group, Summarize, and Count

BioLang's table operations work naturally in pipe chains for aggregation workflows.

#148 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# Variant summary pipeline
let summary = read_vcf("data/variants.vcf")
  |> filter(|v| v.qual >= 30)
  |> classify_variants()
  |> group_by("type")
  |> count_by("type")

println(summary)

# Per-chromosome depth analysis
let depths = read_bed("data/regions.bed")
  |> group_by("chrom")
  |> summarize(|chrom, rows| {chrom: chrom, mean_depth: mean(col(rows,
"score"))})
  |> arrange("chrom")

println(depths)
```

The `group_by`, `count_by`, `filter_by`, `summarize`, and `arrange` builtins all accept pipe input, so they compose seamlessly.

Pipeline Blocks

For larger workflows with named stages, BioLang provides `pipeline` blocks. A `pipeline` declares a named workflow. Inside the block you can use `stage` declarations to name intermediate results with the arrow (`->`) syntax.

#149 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
pipeline fastq_qc {
  # stage name -> expression
  stage raw_stats -> read_fastq("data/reads.fastq")
    |> map(|r| mean_phred(r.quality))
    |> mean()

  stage filtered -> read_fastq("data/reads.fastq")
    |> filter(|r| mean_phred(r.quality) >= 30)

  stage write_out -> write_fastq(filtered, "sample_R1.filtered.fastq.gz")

  stage report -> {
    mean_quality: raw_stats,
    kept: len(filtered),
  }
}

# fastq_qc is bound to the result of the last stage
println("Mean Q: " + str(fastq_qc.mean_quality))
```

When a `pipeline` block has no parameters, it executes immediately and binds its result to the pipeline name. Each `stage name -> expr` evaluates the expression and makes the result available by name to subsequent stages.

Passing Data Between Stages

Stage names become bindings that later stages can reference. This creates an implicit dependency chain.

#150 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
pipeline align_and_call {
  stage aligned -> shell("bwa-mem2 mem -t 16 GRCh38.fa sample_R1.fq.gz
sample_R2.fq.gz | samtools sort -@ 8 -o aligned.sorted.bam")

  stage indexed -> shell("samtools index " + aligned)

  stage variants -> shell("bcftools mpileup -f GRCh38.fa " + aligned + " |
bcftools call -mv -Oz -o variants.vcf.gz")

  stage stats -> read_vcf("data/variants.vcf")
    |> filter(|v| v.qual >= 30)
    |> classify_variants()
    |> count_by("type")
}

println(align_and_call)
```

Each stage name resolves to whatever its expression evaluated to. The `aligned` stage returns a string (the shell output), which `indexed` and `variants` reference directly.

Parameterized Pipelines (Templates)

A pipeline can accept parameters, turning it into a reusable template. When you declare `pipeline name(params) { ... }`, BioLang defines a callable function instead of executing immediately.

```
# Conceptual or diagnostic example; not directly executable.
# Define a reusable alignment template
pipeline align_sample(sample_id, r1, r2, reference) {
  stage sorted -> shell("bwa-mem2 mem -t 16 " + reference + " " + r1 + " " +
r2
        + " | samtools sort -@ 8 -o " + sample_id + ".sorted.bam")

  stage indexed -> shell("samtools index " + sorted)

  # Pipeline returns its last stage value
  sorted
}

# Call with different inputs
let tumor_bam = align_sample("tumor", "tumor_R1.fq.gz", "tumor_R2.fq.gz",
"GRCh38.fa")
let normal_bam = align_sample("normal", "normal_R1.fq.gz", "normal_R2.fq.gz",
"GRCh38.fa")
println("Tumor BAM: " + tumor_bam)
println("Normal BAM: " + normal_bam)
```

Parameterized pipelines behave exactly like functions. You can loop over a sample list to process a whole cohort:

#151 ▶ Run ▶▶ Run All Above + This

```
let samples = [
  {id: "S1", r1: "S1_R1.fq.gz", r2: "S1_R2.fq.gz"},
  {id: "S2", r1: "S2_R1.fq.gz", r2: "S2_R2.fq.gz"},
  {id: "S3", r1: "S3_R1.fq.gz", r2: "S3_R2.fq.gz"},
]

let bams = samples |> map(|s| align_sample(s.id, s.r1, s.r2, "GRCh38.fa"))
println("Aligned " + str(len(bams)) + " samples")
```

Parallel Processing

par_map and par_filter

For data-parallel operations on lists, `par_map` and `par_filter` distribute work across available cores.

#152 ▶ Run ▶▶ Run All Above + This

```
let sequences = read_fasta("data/sequences.fasta")
  |> to_records()

# Compute GC content in parallel
let gc_values = sequences
  |> par_map(|seq| {name: seq.id, gc: gc_content(seq.seq)})

# Filter in parallel
let high_gc = gc_values
  |> par_filter(|s| s.gc > 0.6)

println("High GC sequences: " + str(len(high_gc)))
```

parallel for

For workflows that need to run a block of statements per item, `parallel for` fans out iterations. In the current tree-walking interpreter these run sequentially; a future bytecode backend will parallelize them.

```
# Conceptual or diagnostic example; not directly executable.
let samples = [
  {id: "tumor_01", r1: "t01_R1.fq.gz", r2: "t01_R2.fq.gz"},
  {id: "tumor_02", r1: "t02_R1.fq.gz", r2: "t02_R2.fq.gz"},
  {id: "normal_01", r1: "n01_R1.fq.gz", r2: "n01_R2.fq.gz"},
]

parallel for sample in samples {
  let bam = shell("bwa-mem2 mem -t 4 GRCh38.fa " + sample.r1 + " " +
sample.r2
                  + " | samtools sort -@ 4 -o " + sample.id + ".sorted.bam")
  shell("samtools index " + sample.id + ".sorted.bam")
  println("Done: " + sample.id)
}
```

The result of a `parallel for` is the value of the last iteration's block.

Shell Integration

The `shell()` builtin executes external commands and returns their stdout as a string. This is how BioLang integrates with existing bioinformatics tools.

#153 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# Run a single command
let flagstat = shell("samtools flagstat aligned.bam")
println(flagstat)

# Chain shell commands in a pipeline
let bam = shell("bwa-mem2 mem -t 16 ref.fa R1.fq.gz R2.fq.gz | samtools sort
-@ 8 -o out.bam")

# Use shell output in downstream BioLang processing
let depth_lines = shell("samtools depth aligned.bam")
let depths = depth_lines
  |> split("\n")
  |> filter(|line| len(line) > 0)
  |> map(|line| split(line, "\t"))
  |> map(|parts| float(parts[2]))

println("Mean depth: " + str(mean(depths)))
```

Defer for Cleanup

`defer` registers an expression that runs when the enclosing scope exits, whether it succeeds or fails. This keeps intermediate files from piling up.

#154 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
pipeline trimmed_alignment {
  stage trimmed -> shell("fastp -i sample_R1.fq.gz -I sample_R2.fq.gz -o
trimmed_R1.fq.gz -O trimmed_R2.fq.gz")

  defer shell("rm -f trimmed_R1.fq.gz trimmed_R2.fq.gz")

  stage aligned -> shell("bwa-mem2 mem -t 16 GRCh38.fa trimmed_R1.fq.gz
trimmed_R2.fq.gz | samtools sort -@ 8 -o aligned.bam")

  stage indexed -> shell("samtools index " + aligned)

  aligned
}
# When the pipeline finishes, the deferred rm command fires
```

Error Handling in Pipelines

Wrap pipeline steps in `try` / `catch` to handle failures gracefully without aborting the entire workflow.

#155 ▶ Run ▶▶ Run All Above + This

```
let samples = ["sample_A", "sample_B", "sample_C"]

let results = samples |> map(|name| {
  try {
    let bam = shell("bwa-mem2 mem -t 8 ref.fa " + name + "_R1.fq.gz " + name
+ "_R2.fq.gz | samtools sort -o " + name + ".bam")
    shell("samtools index " + name + ".bam")
    {id: name, status: "ok", bam: name + ".bam"}
  } catch e {
    println("WARN: " + name + " failed: " + str(e))
    {id: name, status: "failed", bam: nil}
  }
})

let passed = results |> filter(|r| r.status == "ok")
let failed = results |> filter(|r| r.status == "failed")
println(str(len(passed)) + " succeeded, " + str(len(failed)) + " failed")
```

You can also use `try` / `catch` around individual stages within a pipeline block to continue past non-critical failures.

Example: FASTQ QC Pipeline

A complete quality-control workflow using only pipe composition.

#156 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# Read and analyze
let reads = read_fastq("data/reads.fastq")
let qualities = reads |> map(|r| mean_phred(r.quality))

let qc = {
  total_reads: len(reads),
  mean_quality: round(mean(qualities), 2),
  median_quality: round(median(qualities), 2),
  q30_pct: round(len(filter(qualities, |q| q >= 30)) / len(qualities) * 100,
1),
  gc_mean: round(mean(reads |> map(|r| gc_content(r.sequence))), 4),
}

println("=== FASTQ QC Report ===")
println("Total reads:    " + str(qc.total_reads))
println("Mean quality:   " + str(qc.mean_quality))
println("Median quality: " + str(qc.median_quality))
println("Q30%:           " + str(qc.q30_pct) + "%")
println("Mean GC:        " + str(qc.gc_mean))

# Filter and write passing reads
let passed = reads |> filter(|r| mean_phred(r.quality) >= 25)
write_fastq(passed, "sample_R1.qc_passed.fastq.gz")
println("Wrote " + str(len(passed)) + " passing reads")

# Notify on completion
notify("FASTQ QC complete: " + str(qc.total_reads) + " reads, Q30=" +
str(qc.q30_pct) + "%")
```

Example: Variant Calling Pipeline

Germline variant calling from FASTQ to filtered VCF using pipeline blocks.

```
# Conceptual or diagnostic example; not directly executable.
pipeline germline_variants {
  stage aligned -> shell(
    "bwa-mem2 mem -t 16 -R '@RG\\tID:S1\\tSM:sample' GRCh38.fa "
    + "sample_R1.fq.gz sample_R2.fq.gz "
    + "| samtools sort -@ 8 -o sample.sorted.bam"
  )

  stage indexed -> shell("samtools index " + aligned)

  stage called -> shell(
    "gatk HaplotypeCaller -I " + aligned
    + " -R GRCh38.fa -L exome_targets.bed -O sample.g.vcf.gz -ERC GVCF"
  )

  stage genotyped -> shell(
    "gatk GenotypeGVCFs -V " + called + " -R GRCh38.fa -O sample.vcf.gz"
  )

  stage filtered -> shell(
    "gatk VariantFiltration -V " + genotyped
    + " --filter-name 'LowQD' --filter-expression 'QD < 2.0'"
    + " --filter-name 'HighFS' --filter-expression 'FS > 60.0'"
    + " -O sample.filtered.vcf.gz"
  )

  # Analyze the results in BioLang
  stage summary -> read_vcf("data/variants.vcf")
    |> filter(|v| v.filter == "PASS")
    |> classify_variants()
    |> count_by("type")
}

println(germline_variants)
```

Example: Multi-Sample Analysis

Processing a cohort with parameterized pipelines and aggregation.

```
# Conceptual or diagnostic example; not directly executable.
# Reusable per-sample pipeline
pipeline process_sample(sample_id, r1, r2) {
  stage bam -> shell(
    "bwa-mem2 mem -t 8 -R '@RG\\tID:" + sample_id + "\\tSM:" + sample_id
    + "' GRCh38.fa " + r1 + " " + r2
    + " | samtools sort -@ 4 -o " + sample_id + ".sorted.bam"
  )

  stage idx -> shell("samtools index " + bam)

  stage depth_info -> shell("samtools depth " + bam)
    |> split("\n")
    |> filter(|line| len(line) > 0)
    |> map(|line| float(split(line, "\t")[2]))
    |> mean()

  {id: sample_id, bam: bam, mean_depth: depth_info}
}

# Load sample manifest
let manifest = read_csv("data/sample_sheet.csv")

# Process all samples
let results = manifest
  |> map(|row| {
    try {
      process_sample(row.sample_id, row.fastq_r1, row.fastq_r2)
    } catch e {
      println("WARN: " + row.sample_id + " failed: " + str(e))
      {id: row.sample_id, bam: nil, mean_depth: 0.0}
    }
  })

# QC gate: check depth thresholds
let passed = results |> filter(|s| s.mean_depth >= 20.0)
let failed = results |> filter(|s| s.mean_depth < 20.0)

if len(failed) > 0 then
  println("WARNING: " + str(len(failed)) + " samples below 20x depth:")
each(failed, |s| println("  " + s.id + ": " + str(round(s.mean_depth, 1)) +
"x"))

println(str(len(passed)) + " of " + str(len(results)) + " samples passed QC")

# Write summary table
let summary = results
  |> map(|s| {sample: s.id, depth: round(s.mean_depth, 1), pass: s.mean_depth
>= 20.0})
write_tsv(summary, "cohort_qc_summary.csv")

# Notify the team
slack("Cohort processing complete: " + str(len(passed)) + "/" +
str(len(results)) + " passed QC")
```

Pipeline Composition

Pipelines are values. Parameterized pipelines are functions. You can call one from another to build layered workflows.

```
# Conceptual or diagnostic example; not directly executable.
pipeline align_one(sample) {
  stage sorted -> shell(
    "bwa-mem2 mem -t 8 GRCh38.fa " + sample.r1 + " " + sample.r2
    + " | samtools sort -@ 4 -o " + sample.id + ".sorted.bam"
  )
  stage idx -> shell("samtools index " + sorted)
  sorted
}

pipeline somatic_pair(tumor, normal) {
  stage tumor_bam -> align_one(tumor)
  stage normal_bam -> align_one(normal)

  stage called -> shell(
    "gatk Mutect2 -I " + tumor_bam + " -I " + normal_bam
    + " -R GRCh38.fa -O somatic.vcf.gz"
  )

  stage filtered -> shell(
    "gatk FilterMutectCalls -V " + called + " -R GRCh38.fa -O
somatic.filtered.vcf.gz"
  )

  let variants = read_vcf("data/variants.vcf")
    |> filter(|v| v.filter == "PASS")
  println("Found " + str(len(variants)) + " somatic variants")
  filtered
}

let result = somatic_pair(
  {id: "tumor", r1: "tumor_R1.fq.gz", r2: "tumor_R2.fq.gz"},
  {id: "normal", r1: "normal_R1.fq.gz", r2: "normal_R2.fq.gz"}
)
```

Summary

BioLang pipelines come in two flavors:

- **Pipe chains** (`|>`): The everyday tool. Chain any sequence of transformations into a readable, left-to-right flow. No special syntax needed.
- **Pipeline blocks** (`pipeline name { stage x -> expr }`): For larger workflows where you want named stages, parameterized templates, and stage-to-stage data passing.

Both approaches compose freely. Use `par_map` and `par_filter` for data-parallel operations, `parallel for` for per-item workflows, `shell()` to call external tools, `try`/`catch` for error resilience, and `defer` for cleanup. The result is a language where analysis pipelines are just ordinary code, with no framework overhead.

nf-core Integration

Why nf-core in a DSL

nf-core is the largest curated collection of bioinformatics pipelines, maintained by a global community and covering over 100 workflows – from bulk RNA-seq to rare-disease variant calling. Every pipeline follows strict standards: peer review, continuous integration, container packaging, and a machine-readable parameter schema.

BioLang exposes nf-core as a set of built-in functions. There is nothing to import and nothing to install. You can browse, search, inspect, and compare nf-core pipelines from the same scripts that process your FASTQ files and call variants. The five builtins – `nfcore_list`, `nfcore_search`, `nfcore_info`, `nfcore_releases`, and `nfcore_params` – turn the nf-core catalog into a queryable data source that fits naturally into pipes, filters, and record operations.

Browsing the Catalog

`nfcore_list()` returns every pipeline in the nf-core organization. Each entry is a record with fields like `name`, `description`, `stars`, and `topics`.

#157 ▶ Run ▶▶ Run All Above + This

```
# requires: internet connection (all nfcore_* functions query the nf-core
GitHub API)
# List all nf-core pipelines
nfcore_list()
```

The result is a list of records. You can pipe it through any BioLang operation:

#158 ▶ Run ▶▶ Run All Above + This

```
# Show names and star counts, sorted by popularity
nfcore_list(sort_by: "stars")
  |> each |p| { name: p.name, stars: p.stars }
```

To limit the result set, pass `limit`. This is useful in exploratory sessions where you want a quick overview rather than the full catalog:

#159 ▶ Run ▶▶ Run All Above + This

```
# Top 10 pipelines by most recent release date
nfcore_list(sort_by: "release", limit: 10)
  |> each |p| { name: p.name, updated: p.updated }
```

Sorting accepts three values:

- `"stars"` – community popularity (default)
- `"name"` – alphabetical
- `"release"` – most recently updated first

Searching Pipelines

When you know what kind of analysis you need but not which pipeline implements it, use `nfcore_search`. It matches against pipeline names, descriptions, and topic tags.

#160 ▶ Run ▶▶ Run All Above + This

```
# Find RNA-seq related pipelines
nfcore_search("rnaseq")
```

Search accepts an optional `limit` to cap results:

#161 ▶ Run ▶▶ Run All Above + This

```
# Find variant-calling pipelines, show top 5
nfcore_search("variant", 5)
```

The returned records have the same shape as `nfcore_list` entries, so you can chain the same downstream operations:

#162 ▶ Run ▶▶ Run All Above + This

```
# Search for methylation pipelines and extract their topics
nfcore_search("methylation")
  |> each |p| { name: p.name, topics: p.topics }
```

Pipeline Details

`nfcore_info` returns a single record with comprehensive metadata for a named pipeline. The returned record contains:

Field	Type	Description
`name`	String	Short pipeline name
`full_name`	String	GitHub-qualified name (nf-core/...)
`description`	String	One-line summary
`stars`	Int	GitHub star count
`url`	String	Repository URL
`license`	String	License identifier (usually MIT)
`topics`	List	GitHub topic tags
`open_issues`	Int	Current open issue count
`created`	String	Repository creation date
`updated`	String	Last push date

#163 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# Inspect the Sarek germline/somatic variant calling pipeline
let sarek = nfcore_info("sarek")
print("Pipeline: " + sarek.full_name)
print("License:  " + sarek.license)
print("Stars:    " + to_string(sarek.stars))
print("Topics:   " + join(sarek.topics, ", "))
```

You can use the metadata to make decisions in scripts. For example, checking whether a pipeline is actively maintained before committing to it:

#164 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# Only proceed if the pipeline has been updated recently
let info = nfcore_info("taxprofiler")
if info.open_issues < 50 then
  print(info.name + " looks actively maintained")
```

Releases and Versions

Production workflows should pin to a specific release rather than tracking the development branch. `nfcore_releases` returns the full release history as a list of records, each with `tag` and `published_at` fields.

#165 ▶ Run ▶▶ Run All Above + This

```
# List all rnaseq releases
nfcore_releases("rnaseq")
```

The list is ordered newest-first, so the head element is the latest stable version:

#166 ▶ Run ▶▶ Run All Above + This

```
# Get the latest stable release tag for rnaseq
let latest = nfcore_releases("rnaseq")
  |> first
print("Latest release: " + latest.tag + " (" + latest.published_at + ")")
```

You can also search for a specific version range or find how many releases a pipeline has had – a rough proxy for maturity:

#167 ▶ Run ▶▶ Run All Above + This

```
# Count total releases for sarek
let release_count = nfcore_releases("sarek") |> len
print("sarek has " + to_string(release_count) + " releases")
```

#168 ▶ Run ▶▶ Run All Above + This

```
# Find all 3.x releases of rnaseq
nfcore_releases("rnaseq")
  |> filter |r| starts_with(r.tag, "3.")
  |> each |r| r.tag
```

Parameter Schemas

Every nf-core pipeline publishes a JSON schema describing its configurable parameters – input paths, reference genomes, trimming options, resource limits, and more. `nfcore_params` fetches and parses this schema, returning a record

whose keys are parameter group names and whose values are records of individual parameters.

#169 ▶ Run ▶▶ Run All Above + This

```
# Fetch the full parameter schema for rnaseq
let params = nfcore_params("rnaseq")
```

The top-level keys are group names such as `"input_output_options"`, `"reference_genome_options"`, or `"trimming_options"`. Each group is a record of parameter entries:

#170 ▶ Run ▶▶ Run All Above + This

```
# List all parameter groups
nfcore_params("rnaseq")
  |> keys
```

#171 ▶ Run ▶▶ Run All Above + This

```
# Inspect the reference genome options
nfcore_params("rnaseq").reference_genome_options
  |> keys
```

This is particularly useful for validating that your configuration covers the required parameters before submitting a long-running pipeline:

#172 ▶ Run ▶▶ Run All Above + This

```
# Check whether a specific parameter exists
let params = nfcore_params("sarek")
let genome_opts = params.reference_genome_options
print(keys(genome_opts))
```

Example: Finding the Right Pipeline

A common task: you need to process single-cell RNA-seq data but are unsure which nf-core pipeline to use. Search the catalog, compare candidates, and pick the best fit.

#173 ▶ Run ▶▶ Run All Above + This

```
# Step 1: Search for single-cell pipelines
let candidates = nfcore_search("single-cell")

# Step 2: Show names, stars, and descriptions side by side
candidates
  |> each |p| {
    name: p.name,
    stars: p.stars,
    description: p.description
  }
  |> sort_by |a, b| b.stars - a.stars
  |>> each |p| print(p.name + " (" + to_string(p.stars) + " stars): " +
p.description)

# Step 3: Get detailed info on the top candidate
let top = candidates
  |> sort_by |a, b| b.stars - a.stars
  |> first

let info = nfcore_info(top.name)
print("Selected: " + info.full_name)
print("License:  " + info.license)
print("Topics:   " + join(info.topics, ", "))

# Step 4: Check the latest release
let latest = nfcore_releases(top.name) |> first
print("Latest release: " + latest.tag + " published " + latest.published_at)

# Step 5: Preview the parameter groups
nfcore_params(top.name)
  |> keys
  |>> each |group| print("  - " + group)
```

This entire exploration runs in a single script – no switching between a browser, the nf-core CLI tool, and your pipeline configuration files.

Example: Auditing Pipeline Parameters

Before launching a whole-genome sequencing analysis with Sarek, you want to confirm that your reference genome configuration covers every required parameter and that no deprecated options have crept into your config.

```
# Conceptual or diagnostic example; not directly executable.
# Fetch the Sarek parameter schema
let params = nfcore_params("sarek")

# Extract all reference genome parameters
let ref_params = params.reference_genome_options

# Your current configuration
let my_config = {
  genome: "GRCh38",
  igenomes_base: "s3://ngi-igenomes/igenomes",
  fasta: nil,
  fasta_fai: nil
}

# Check for parameters in the schema that you have not configured
let schema_keys = keys(ref_params)
let config_keys = keys(my_config)

let missing = schema_keys
  |> filter |k| not(contains(config_keys, k))

if len(missing) > 0 then
  print("WARNING: unset reference genome parameters:")
  missing |>> each |k| print("  - " + k)
else
  print("All reference genome parameters are covered.")

# Check for keys in your config that are not in the schema (possibly
deprecated)
let extra = config_keys
  |> filter |k| not(contains(schema_keys, k))

if len(extra) > 0 then
  print("WARNING: config keys not in current schema (possibly deprecated):")
  extra |>> each |k| print("  - " + k)
```

This catches configuration drift early – before you discover it four hours into a whole-genome analysis run.

Example: Building a Pipeline Registry

For a core facility managing dozens of active projects, it helps to maintain a catalog of available pipelines organized by topic. This script builds a topic index from the full nf-core catalog.

#174 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# Fetch all pipelines
let all_pipelines = nfcore_list(sort_by: "stars")

# Build a topic-to-pipeline mapping
# Flatten: for each pipeline, emit one record per topic tag
let entries = all_pipelines
  |> flat_map |p| p.topics |> each |t| { topic: t, name: p.name, stars:
p.stars }

# Group by topic
let topics = entries
  |> group_by |e| e.topic

# Print a summary: topics with 3 or more pipelines
keys(topics)
  |> filter |t| len(topics[t]) >= 3
  |> sort
  |>> each |t| {
    let pipelines = topics[t]
      |> sort_by |a, b| b.stars - a.stars
      |> each |p| p.name
    print(t + " (" + to_string(len(pipelines)) + " pipelines): " +
join(pipelines, ", "))
  }
```

You can extend this to generate a Markdown report, push to a shared wiki, or feed into a lab notebook:

#175 ▶ Run ▶▶ Run All Above + This

```
# Export the top 20 pipelines as a tab-separated table
let header = "name\tstars\ttopics\tlatest_release"
print(header)

nfcore_list(sort_by: "stars", limit: 20)
  |>> each |p| {
    let latest = nfcore_releases(p.name) |> first
    let tag = if latest != nil then latest.tag else "unreleased"
    let topic_str = join(p.topics, ";")
    print(p.name + "\t" + to_string(p.stars) + "\t" + topic_str + "\t" + tag)
  }
```

Redirect stdout to a file and you have a pipeline inventory that updates every time you run the script – no manual curation needed.

Parsing Nextflow Files

BioLang can parse Nextflow `.nf` files directly with `nf_parse()`. This function reads the file and extracts its structure without requiring a Nextflow runtime.

#176 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# Parse a Nextflow pipeline file
let parsed = nf_parse("main.nf")
```

The result is a record with five fields:

Field	Type	Content
`params`	Record	Parameter name-value pairs
`processes`	List[Record]	Process name, inputs, outputs, script, container, cpus, memory
`includes`	List[Record]	Include name, alias, source path
`workflow`	List[String]	Workflow block lines
`dsl`	String	DSL version ("DSL1" or "DSL2")

You can pipe the parsed structure through any BioLang operation:

#177 ▶ Run ▶▶ Run All Above + This

```
# List all processes and their containers
let parsed = nf_parse("variant_calling.nf")

parsed.processes
  |> each |p| {
    print(p.name + ": " + if p.container != nil then p.container else "no
container")
  }
```

#178 ▶ Run ▶▶ Run All Above + This

```
# Extract all parameters with their default values
let parsed = nf_parse("rnaseq.nf")

keys(parsed.params)
  |> sort
  |>> each |k| print(k + " = " + to_string(parsed.params[k]))
```

Generating BioLang Code

The `nf_to_bl` function takes a parsed Nextflow record and generates BioLang pipeline code as a string:

#179 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# Parse and convert a Nextflow pipeline
let parsed = nf_parse("rnaseq.nf")
let bl_code = nf_to_bl(parsed)

# Print the generated code
print(bl_code)

# Save to a file
write_file("rnaseq.bl", bl_code)
```

The generated code maps Nextflow constructs to BioLang equivalents:

- `params.*` become top-level variables
- `process` blocks become function definitions with `shell()` calls
- Container, CPU, and memory directives are preserved as comments
- Workflow call ordering maps to sequential pipe chains

This is a starting point – edit the generated skeleton to add BioLang-specific features like pipe chains, error handling, and parallel execution.

#180 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# Full workflow: parse, generate, and review
let parsed = nf_parse("sarek_main.nf")

# Show what was extracted
print("Found " + to_string(len(parsed.processes)) + " processes")
print("Found " + to_string(len(keys(parsed.params))) + " parameters")

# Generate BioLang code
let bl_code = nf_to_bl(parsed)
write_file("sarek.bl", bl_code)
print("Generated sarek.bl")
```

Chapter 13: BioContainers Integration

Reproducible bioinformatics demands pinned software versions. A variant calling pipeline that works today must produce identical results next year, on a different machine, with the same tool versions. The BioContainers project addresses this by packaging over 9,000 bioinformatics tools as container images hosted on registries like quay.io and Docker Hub.

BioLang gives you native access to the BioContainers registry. Four built-in functions let you search for tools, discover popular packages, inspect version histories, and retrieve exact container image URIs – all without leaving your script. No imports are needed.

Searching for Tools

`biocontainers_search` queries the BioContainers registry by name or keyword. It returns a list of records, each describing a matching tool.

#181 ▶ Run ▶▶ Run All Above + This

```
# requires: internet connection (all biocontainers_* functions query the
BioContainers API)
# Find all samtools-related containers
let results = biocontainers_search("samtools")
# results => [
#   {name: "samtools", description: "Tools for manipulating NGS
alignments...",
#    organization: "biocontainers", version_count: 42,
#    latest_version: "1.19--h50ea8bc_0",
#    latest_image: "quay.io/biocontainers/samtools:1.19--h50ea8bc_0"},
#   {name: "htslib", description: "C library for high-throughput
sequencing...", ...},
#   ...
# ]

results |> each(|tool| {
  print(tool.name + " (" + str(tool.version_count) + " versions)")
  print("  Latest: " + tool.latest_version)
})
```

The default limit is 25 results. Pass a second argument to narrow or widen the search.

#182 ▶ Run ▶▶ Run All Above + This

```
# Top 5 matches for short-read aligners
let aligners = biocontainers_search("bwa", 5)

aligners |> each(|a| print(a.name + " => " + a.latest_image))
```

Search terms are matched against tool names and descriptions, so broader queries work too.

#183 ▶ Run ▶▶ Run All Above + This

```
# Find tools related to RNA-seq quantification
let quant_tools = biocontainers_search("salmon rna-seq")

quant_tools
  |> filter(|t| t.version_count > 5)
  |> each(|t| print(t.name + ": " + t.description))
```

Popular Tools

`biocontainers_popular` returns the most-pulled tools in the registry. This is useful for discovering which tools the community relies on and for auditing whether your pipeline uses well-maintained software.

#184 ▶ Run ▶▶ Run All Above + This

```
let top20 = biocontainers_popular()

print("Top 20 BioContainers tools:")
top20 |> each(|t| print("  " + t.name + " - " + t.latest_version))
```

Pass a limit to retrieve more.

#185 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# Top 50 most popular bioinformatics containers
let top50 = biocontainers_popular(50)

# Which of our pipeline tools are in the top 50?
let our_tools = ["samtools", "bcftools", "bwa-mem2", "gatk4", "picard"]
let popular_names = top50 |> map(|t| t.name)

our_tools |> each(|tool| {
  let found = popular_names |> find(|n| n == tool)
  if found != nil then
    print(tool + " is in the top 50")
  else
    print(tool + " is NOT in the top 50")
})
```

Tool Details

`biocontainers_info` returns a detailed record for a single tool, including its full version history with per-version container images.

```
# Conceptual or diagnostic example; not directly executable.
let info = biocontainers_info("samtools")
# info => {
#   name: "samtools",
#   description: "Tools for manipulating NGS alignments...",
#   organization: "biocontainers",
#   aliases: ["samtools"],
#   versions: [
#     {version: "1.19--h50ea8bc_0",
#      images: [{registry: "quay.io", image:
"quay.io/biocontainers/samtools:1.19--h50ea8bc_0",
#                type: "Docker", size: 14250000}]},
#     {version: "1.18--h50ea8bc_0", images: [...]},
#     ...
#   ]
# }

print(info.name + " by " + info.organization)
print(str(len(info.versions)) + " versions available")

# List the 5 most recent versions
info.versions
  |> take(5)
  |> each(|v| {
    let image = v.images |> first()
    print("  " + v.version + " (" + image.registry + ", "
          + str(image.size / 1000000) + " MB)")
  })
```

The `images` list for each version may contain entries from multiple registries or image types (Docker, Singularity). Filter by registry or type if your infrastructure requires a specific format.

#186 ▶ Run ▶▶ Run All Above + This

```
# Find Singularity images for deepvariant
let dv = biocontainers_info("deepvariant")

dv.versions |> each(|v| {
  let singularity = v.images |> filter(|img| img.type == "Singularity")
  if len(singularity) > 0 then
    print(v.version + " has Singularity image")
})
```

Version Management

`biocontainers_versions` returns a flat list of all versions for a tool, each with a list of full image URI strings. This is the function to use when you need to pin a specific version in a pipeline manifest.

#187 ▶ Run ▶▶ Run All Above + This

```
let versions = biocontainers_versions("gatk4")
# versions => [
#   {version: "4.5.0.0--py310hdfd78af_0",
#    images: ["quay.io/biocontainers/gatk4:4.5.0.0--py310hdfd78af_0"]},
#   {version: "4.4.0.0--py310hdfd78af_0",
#    images: ["quay.io/biocontainers/gatk4:4.4.0.0--py310hdfd78af_0"]},
#   ...
# ]

# Find the latest GATK 4.4.x release
let gatk44 = versions
  |> filter(|v| starts_with(v.version, "4.4"))
  |> first()

print("Pinning GATK to: " + gatk44.images[0])
```

You can use this to check whether a specific version exists before committing to it in a pipeline definition.

#188 ▶ Run ▶▶ Run All Above + This

```
# Verify that bcftools 1.18 is available
let bc_versions = biocontainers_versions("bcftools")
let target = bc_versions |> find(|v| starts_with(v.version, "1.18"))

if target != nil then
  print("bcftools 1.18 available: " + target.images[0])
else
  print("bcftools 1.18 not found in BioContainers")
```

Example: Building a Reproducible Tool Manifest

A variant calling pipeline needs exact container images for every tool. Use the BioContainers builtins to resolve each tool to a pinned image URI and export the manifest.

#189 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# Tools required for a germline variant calling pipeline
let required = [
  {name: "bwa-mem2",  min_version: "2.2"},
  {name: "samtools",  min_version: "1.18"},
  {name: "gatk4",     min_version: "4.4"},
  {name: "bcftools",  min_version: "1.18"},
]

let manifest = required |> map(|req| {
  let versions = biocontainers_versions(req.name)

  # Find the newest version that satisfies the minimum
  let matching = versions
    |> filter(|v| starts_with(v.version, req.min_version))

  if len(matching) == 0 then {
    print("WARNING: no " + req.name + " >= " + req.min_version + " found")
    {tool: req.name, version: "MISSING", image: "MISSING"}
  } else {
    let chosen = matching |> first()
    {tool: req.name, version: chosen.version, image: chosen.images[0]}
  }
})

# Print the resolved manifest
print("Variant Calling Pipeline - Tool Manifest")
print("========================================")
manifest |> each(|m| {
  print(m.tool + ":")
  print("  version: " + m.version)
  print("  image:   " + m.image)
})

# Export as structured data
manifest |> write_json("pipeline_manifest.json")
```

This script produces a lockfile-style manifest that can be checked into version control alongside the pipeline definition.

Example: Tool Discovery for a New Analysis

When starting a new analysis type, you need to survey what tools are available. Here we explore the methylation analysis landscape.

```
# Conceptual or diagnostic example; not directly executable.
# What methylation tools exist in BioContainers?
let methyl_tools = biocontainers_search("methylation", 50)

print(str(len(methyl_tools)) + " methylation-related tools found")
print("")

# Group by version count to find well-maintained tools
let mature = methyl_tools
  |> filter(|t| t.version_count >= 5)
  |> sort_by(|t| -t.version_count)

let new_tools = methyl_tools
  |> filter(|t| t.version_count < 3)

print("Mature tools (" + str(len(mature)) + "):")
mature |> each(|t| {
  print("  " + t.name + " - " + str(t.version_count) + " versions"
        + " (latest: " + t.latest_version + ")")
  print("    " + t.description)
})

print("")
print("Newer tools (" + str(len(new_tools)) + "):")
new_tools |> take(10) |> each(|t| {
  print("  " + t.name + " - " + t.latest_version)
})

# Deep dive into the top candidate
let bismark = biocontainers_info("bismark")
print("")
print("Bismark detail:")
print("  " + bismark.description)
print("  " + str(len(bismark.versions)) + " releases")
print("  Aliases: " + join(bismark.aliases, ", "))

# Check image sizes across versions
bismark.versions |> take(5) |> each(|v| {
  let docker = v.images |> filter(|img| img.type == "Docker") |> first()
  if docker != nil then
    print("  " + v.version + ": " + str(docker.size / 1000000) + " MB")
})
```

Example: Container Image Audit

For an existing pipeline, verify that every tool has a valid BioContainers image and flag any that are outdated.

```
# Conceptual or diagnostic example; not directly executable.
# Current pipeline tools and their pinned versions
let pinned = [
  {tool: "bwa-mem2",  version: "2.2.1--hd03093a_2"},
  {tool: "samtools",  version: "1.17--h50ea8bc_0"},
  {tool: "gatk4",     version: "4.3.0.0--py310hdfd78af_0"},
  {tool: "bcftools",  version: "1.17--h3cc50cf_1"},
  {tool: "multiqc",   version: "1.14--pyhdfd78af_0"},
  {tool: "fastp",     version: "0.23.2--hb7a2d85_2"},
]

let audit = pinned |> map(|entry| {
  let info = biocontainers_info(entry.tool)
  let all_versions = info.versions |> map(|v| v.version)

  # Check if pinned version still exists
  let exists = all_versions |> find(|v| v == entry.version) != nil

  # Check if there is a newer version
  let latest = info.versions |> first()
  let is_latest = latest.version == entry.version

  # Count how many versions behind
  let versions_behind = if is_latest then
    0
  else {
    let idx = all_versions
      |> enumerate()
      |> find(|pair| pair.value == entry.version)
    if idx != nil then idx.index else -1
  }

  {
    tool: entry.tool,
    pinned: entry.version,
    latest: latest.version,
    exists: exists,
    is_latest: is_latest,
    versions_behind: versions_behind,
  }
})

# Report
print("Pipeline Container Audit")
print("========================")

let missing = audit |> filter(|a| not a.exists)
let outdated = audit |> filter(|a| a.exists and not a.is_latest)
let current = audit |> filter(|a| a.is_latest)

if len(missing) > 0 then {
  print("")
  print("MISSING (pinned version no longer in registry):")
  missing |> each(|a| print("  " + a.tool + " " + a.pinned
                            + " => latest: " + a.latest))
}

if len(outdated) > 0 then {
  print("")
  print("OUTDATED:")
  outdated |> each(|a| print("  " + a.tool + " " + a.pinned
                             + " => " + a.latest
                             + " (" + str(a.versions_behind) + " versions
behind)"))
}
```

```
if len(current) > 0 then {
  print("")
  print("CURRENT:")
  current |> each(|a| print("  " + a.tool + " " + a.pinned))
}

print("")
print(str(len(current)) + " current, "
      + str(len(outdated)) + " outdated, "
      + str(len(missing)) + " missing")

audit |> write_json("container_audit.json")
```

Summary

BioLang's four BioContainers builtins – `biocontainers_search`, `biocontainers_popular`, `biocontainers_info`, and `biocontainers_versions` – bring the full BioContainers registry into your scripts as native data. Use them to discover tools, pin container images for reproducibility, audit existing pipelines, and explore new analysis domains. Combined with BioLang's pipes and collection operations, a few lines of code replace manual registry browsing and ad hoc version tracking.

Galaxy ToolShed & Workflow Parsing

The Galaxy ToolShed is the app store for the Galaxy bioinformatics platform. It hosts thousands of community-contributed tools covering everything from read alignment and variant calling to phylogenetics and metabolomics. Each repository in the ToolShed carries metadata – owner, description, download counts, and revision history – making it a rich discovery resource even if you never run Galaxy itself.

BioLang provides read-only access to the Galaxy ToolShed through four built-in functions. There is nothing to import and nothing to install. You can search repositories, browse popular tools, list categories, and inspect individual tools from the same scripts that process your sequence data. A fifth builtin, `galaxy_to_bl`, takes a Galaxy workflow record and generates BioLang pipeline code from it.

Searching Repositories

`galaxy_search` queries the ToolShed by name or keyword. It returns a list of records, each describing a matching repository.

#190 ▶ Run ▶▶ Run All Above + This

```
# requires: internet connection (all galaxy_* functions query the Galaxy
ToolShed API)
# Find BWA-related tools in the Galaxy ToolShed
galaxy_search("bwa")
  |> each(|t| print(t.name + " by " + t.owner + " (" + str(t.downloads) + "
downloads)"))
```

Each record in the result contains:

Field	Type	Description
`name`	String	Repository name
`owner`	String	ToolShed username of the maintainer
`description`	String	Short summary
`downloads`	Int	Total download count
`url`	String	ToolShed repository URL

The default result set includes all matches. Pass a second argument to cap the number of results, which is useful in exploratory sessions where you want a quick overview.

#191 ▶ Run ▶▶ Run All Above + This

```
# Top 5 matches for short-read aligners
galaxy_search("bwa", 5)
  |> each(|t| print(t.name + " (" + t.owner + "): " + t.description))
```

Search terms are matched against repository names and descriptions, so broader queries work too.

#192 ▶ Run ▶▶ Run All Above + This

```
# Find tools related to RNA-seq
galaxy_search("rna-seq", 10)
  |> filter(|t| t.downloads > 1000)
  |> each(|t| print(t.name + ": " + str(t.downloads) + " downloads"))
```

Popular Tools

`galaxy_popular` returns the most-downloaded repositories in the ToolShed, sorted by download count descending. This is useful for discovering which tools the Galaxy community relies on most heavily.

#193 ▶ Run ▶▶ Run All Above + This

```
galaxy_popular(10)
  |> map(|t| { name: t.name, owner: t.owner, downloads: t.downloads })
```

Without an argument, `galaxy_popular()` returns a default set. Pass a limit to widen or narrow the list.

#194 ▶ Run ▶▶ Run All Above + This

```
# Top 25 Galaxy tools by download count
galaxy_popular(25)
  |> each(|t| print(t.name + " by " + t.owner + " - " + str(t.downloads) + "
downloads"))
```

Categories

The ToolShed organizes repositories into categories such as “Assembly”, “Variant Analysis”, “RNA”, and “Visualization”. `galaxy_categories` returns them all as a list of records.

#195 ▶ Run ▶▶ Run All Above + This

```
galaxy_categories()
  |> each(|c| print(c.name + ": " + c.description))
```

You can use this to explore what kinds of tools exist before drilling into a specific area.

#196 ▶ Run ▶▶ Run All Above + This

```
# Find categories related to sequencing
galaxy_categories()
  |> filter(|c| contains(lower(c.name), "sequenc") or
contains(lower(c.description), "sequenc"))
  |> each(|c| print(c.name))
```

Repository Details

`galaxy_tool` returns a detailed record for a single repository, identified by owner and name.

#197 ▶ Run ▶▶ Run All Above + This

```
let tool = galaxy_tool("devteam", "bwa")
print("Tool: " + tool.name)
print("Owner: " + tool.owner)
print("Downloads: " + str(tool.downloads))
print("Last updated: " + tool.last_updated)
print("Description: " + tool.description)
```

The returned record includes fields beyond what the search results provide, such as `last_updated` and the full repository URL. Use this to verify that a tool is actively maintained before building a workflow around it.

#198 ▶ Run ▶▶ Run All Above + This

```
# Check maintenance status of a tool before adopting it
let tool = galaxy_tool("iuc", "samtools_sort")
print(tool.name + " last updated: " + tool.last_updated)
print("Downloads: " + str(tool.downloads))
```

Cross-Registry Discovery

One of BioLang's strengths is that nf-core, BioContainers, and Galaxy ToolShed are all accessible from the same script. When evaluating a tool, you can check all three registries in a single pass to understand the full landscape – whether curated pipelines exist, whether container images are available, and whether Galaxy wrappers have been written.

#199 ▶ Run ▶▶ Run All Above + This

```
# Search for BWA across all three registries
let tool = "bwa"

let nf_results = nfcore_search(tool)
let bc_results = biocontainers_search(tool, 5)
let gx_results = galaxy_search(tool, 5)

print("=== Cross-registry search for: " + tool + " ===")
print("nf-core pipelines mentioning " + tool + ": " + str(len(nf_results)))
print("BioContainers: " + str(len(bc_results)))
print("Galaxy tools: " + str(len(gx_results)))
```

This pattern scales to any tool or keyword. Here is a more thorough version that prints details from each registry side by side.

```
# Conceptual or diagnostic example; not directly executable.
# Compare tool availability across registries
let tools = ["samtools", "bcftools", "hisat2", "salmon"]

tools |> each(|name| {
  print("--- " + name + " ---")

  let gx = galaxy_search(name, 3)
  let bc = biocontainers_search(name, 3)

  if len(gx) > 0 then
    print("  Galaxy: " + gx[0].name + " by " + gx[0].owner
          + " (" + str(gx[0].downloads) + " downloads)")
  else
    print("  Galaxy: not found")

  if len(bc) > 0 then
    print("  BioContainers: " + bc[0].name
          + " (latest: " + bc[0].latest_version + ")")
  else
    print("  BioContainers: not found")

  print("")
})
```

Workflow Code Generation

Galaxy workflows are JSON documents that describe a directed acyclic graph of tool invocations. BioLang can parse these workflows and generate equivalent BioLang pipeline code via the `galaxy_to_bl` builtin. This is useful when migrating from Galaxy to a script-based workflow, or when you want to use a Galaxy workflow as a starting point for a BioLang pipeline.

`galaxy_to_bl` takes a record that represents a Galaxy workflow and returns a string of BioLang code. The expected input format mirrors the structure of a Galaxy workflow export.

#200 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# Construct a Galaxy workflow record
let workflow = {
  name: "RNA-seq Analysis",
  annotation: "Basic RNA-seq pipeline",
  steps: [
    { name: "FastQC", tool_id: "fastqc/0.74", inputs: ["reads"], outputs:
["report"] },
    { name: "HISAT2", tool_id: "hisat2/2.2.1", inputs: ["reads"], outputs:
["aligned_bam"] },
    { name: "featureCounts", tool_id: "featurecounts/2.0.6", inputs:
["aligned_bam"], outputs: ["counts"] }
  ]
}

# Generate BioLang pipeline code
let bl_code = galaxy_to_bl(workflow)
print(bl_code)
```

The generated code maps each Galaxy step to a BioLang function call, preserving the data flow between steps. You can write it directly to a file and use it as a starting point for further customization.

#201 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# Save the generated code
let bl_code = galaxy_to_bl(workflow)
write_file("rnaseq.bl", bl_code)
print("Pipeline written to rnaseq.bl")
```

For more complex workflows with branching and multiple outputs, the generated code includes comments marking each step and its connections.

#202 ▶ Run ▶▶ Run All Above + This

```
# A branching QC + alignment workflow
let workflow = {
  name: "QC and Align",
  annotation: "Parallel QC with alignment",
  steps: [
    { name: "FastQC", tool_id: "fastqc/0.74", inputs: ["reads"], outputs:
["qc_report"] },
    { name: "Trimmomatic", tool_id: "trimmomatic/0.39", inputs: ["reads"],
outputs: ["trimmed"] },
    { name: "BWA-MEM2", tool_id: "bwa_mem2/2.2.1", inputs: ["trimmed"],
outputs: ["aligned"] },
    { name: "MultiQC", tool_id: "multiqc/1.14", inputs: ["qc_report"],
outputs: ["summary"] }
  ]
}

let bl_code = galaxy_to_bl(workflow)
print(bl_code)
```

Configuration

The Galaxy ToolShed URL defaults to the main public instance at `https://toolshed.g2.bx.psu.edu`. You can override it for private or institutional ToolShed installations.

Set the URL in `~/.biolang/apis.yaml`:

```
galaxy:
  toolshed_url: "https://toolshed.example.org"
```

Or via the environment variable `BIOLANG_GALAXY_TOOLSHED_URL`:

```
export BIOLANG_GALAXY_TOOLSHED_URL="https://toolshed.example.org"
```

The environment variable takes precedence over the config file. If neither is set, the public ToolShed is used.

Example: Building a Tool Inventory

A core facility managing Galaxy and non-Galaxy workflows needs to know which tools are available in the ToolShed and whether container images exist for running them outside Galaxy. This script searches Galaxy for tools matching a keyword, then cross-references each one with BioContainers to check container availability.

```
# Build a tool inventory for variant analysis
let keyword = "variant"

# Step 1: Search Galaxy for variant-related tools
let gx_tools = galaxy_search(keyword, 20)
print("Found " + str(len(gx_tools)) + " Galaxy tools for: " + keyword)
print("")

# Step 2: For each Galaxy tool, check BioContainers
let inventory = gx_tools |> map(|t| {
  # Search BioContainers using the tool name
  let bc = biocontainers_search(t.name, 1)

  let container_status = if len(bc) > 0 then
    "available (" + bc[0].latest_version + ")"
  else
    "not found"

  {
    name: t.name,
    owner: t.owner,
    galaxy_downloads: t.downloads,
    container: container_status
  }
})

# Step 3: Report
print("Tool Inventory: " + keyword)
print("======================================")
inventory
  |> sort_by(|a| -a.galaxy_downloads)
  |> each(|t| {
    print(t.name + " (" + t.owner + ")")
    print("  Galaxy downloads: " + str(t.galaxy_downloads))
    print("  BioContainers:    " + t.container)
  })

# Step 4: Summary statistics
let with_container = inventory |> filter(|t| not starts_with(t.container,
"not"))
let without_container = inventory |> filter(|t| starts_with(t.container,
"not"))

print("")
print("Summary:")
print("  Total tools:          " + str(len(inventory)))
print("  With container image: " + str(len(with_container)))
print("  Without container:    " + str(len(without_container)))
```

You can extend this pattern to cover additional registries or to export the inventory as a structured file.

#204 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# Export as JSON for downstream use
inventory |> write_json("variant_tool_inventory.json")
```

Example: Migrating a Galaxy Workflow

When moving a project from Galaxy to BioLang, you can combine the ToolShed lookup with code generation in a single script. Look up each tool to verify it exists, then generate the BioLang equivalent.

#205 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# Define the workflow steps from a Galaxy export
let workflow = {
  name: "Whole Genome Variant Calling",
  annotation: "BWA-MEM2 alignment with GATK HaplotypeCaller",
  steps: [
    { name: "FastQC", tool_id: "fastqc/0.74", inputs: ["reads_r1",
"reads_r2"], outputs: ["qc_report"] },
    { name: "BWA-MEM2", tool_id: "bwa_mem2/2.2.1", inputs: ["reads_r1",
"reads_r2"], outputs: ["aligned_bam"] },
    { name: "MarkDuplicates", tool_id: "picard_markduplicates/3.1.1", inputs:
["aligned_bam"], outputs: ["dedup_bam"] },
    { name: "HaplotypeCaller", tool_id: "gatk4_haplotypecaller/4.5", inputs:
["dedup_bam"], outputs: ["raw_vcf"] },
    { name: "FilterVcf", tool_id: "gatk4_filtermutectcalls/4.5", inputs:
["raw_vcf"], outputs: ["filtered_vcf"] }
  ]
}

# Verify each tool exists in the ToolShed
print("Verifying tools...")
workflow.steps |> each(|step| {
  # Extract base tool name from tool_id
  let parts = split(step.tool_id, "/")
  let tool_name = parts[0]
  let results = galaxy_search(tool_name, 1)
  if len(results) > 0 then
    print("  " + step.name + ": found (" + results[0].owner + ")")
  else
    print("  " + step.name + ": WARNING - not found in ToolShed")
})

# Generate the BioLang pipeline
print("")
let bl_code = galaxy_to_bl(workflow)
print("Generated pipeline:")
print(bl_code)

# Save to file
write_file("wgs_variant_calling.bl", bl_code)
print("Pipeline saved to wgs_variant_calling.bl")
```

Summary

BioLang's Galaxy ToolShed builtins – `galaxy_search`, `galaxy_popular`, `galaxy_categories`, `galaxy_tool`, and `galaxy_to_bl` – bring the Galaxy ecosystem into your scripts as native data. Use them to discover tools, evaluate

popularity and maintenance status, cross-reference with BioContainers and nf-core, and generate BioLang pipeline code from Galaxy workflows. The ToolShed becomes one more queryable registry alongside the others, all accessible from the same pipes, filters, and record operations you use for everything else.

Chapter 12: Statistics and Linear Algebra

Bioinformatics is quantitative at its core. Whether you are testing for differential expression, running principal component analysis on a transcriptome, or modeling drug kinetics, you need robust numerical tools. BioLang provides built-in statistics functions, matrix operations, and an ODE solver so these analyses live alongside your data processing code.

Descriptive Statistics

All descriptive stats operate on lists of numbers.

#206 ▶ Run ▶▶ Run All Above + This

```
let coverage = tsv("sample_depth.tsv") |> col("depth")

let summary = {
  mean: mean(coverage),
  median: median(coverage),
  sd: stdev(coverage),
  var: variance(coverage),
  q25: quantile(coverage, 0.25),
  q75: quantile(coverage, 0.75),
  iqr: quantile(coverage, 0.75) - quantile(coverage, 0.25),
}

print("Coverage: mean=" + str(summary.mean) + " median=" +
str(summary.median))
```

These functions handle missing values gracefully: `nil` entries are skipped with a warning.

Hypothesis Testing

t-test

Compare expression levels between two conditions.

#207 ▶ Run ▶▶ Run All Above + This

```
let tumor = tsv("expr.tsv") |> filter(|r| r.group == "tumor") |> col("BRCA1")
let normal = tsv("expr.tsv") |> filter(|r| r.group == "normal") |>
col("BRCA1")

let result = ttest(tumor, normal)
# result => {statistic: 4.32, pvalue: 0.00018, df: 28, mean_diff: 2.15}

if result.pvalue < 0.05 then
  print("BRCA1 significantly different (p=" + str(result.pvalue) + ")")
```

Wilcoxon Rank-Sum

For non-normally distributed data such as read counts.

#208 ▶ Run ▶▶ Run All Above + This

```
let treated = [1240, 890, 1560, 2100, 780, 1890, 1345]
let control = [450, 520, 380, 610, 490, 430, 550]

let result = wilcoxon(treated, control)
# result => {statistic: 49.0, pvalue: 0.0012}
```

Chi-squared Test

Test whether observed genotype frequencies match Hardy-Weinberg expectations.

#209 ▶ Run ▶▶ Run All Above + This

```
let observed = [210, 480, 310]  # AA, Aa, aa genotypes
let n = sum(observed)
let p = (2.0 * observed[0] + observed[1]) / (2.0 * n)
let q = 1.0 - p
let expected = [p * p * n, 2 * p * q * n, q * q * n]

let result = chi_square(observed, expected)
# result => {statistic: 3.21, pvalue: 0.073, df: 1}

if result.pvalue > 0.05 then
  print("Population is in Hardy-Weinberg equilibrium")
```

ANOVA

Compare expression across multiple tissue types.

#210 ▶ Run ▶▶ Run All Above + This

```
let brain = tsv("expr.tsv") |> filter(|r| r.tissue == "brain") |> col("TP53")
let liver = tsv("expr.tsv") |> filter(|r| r.tissue == "liver") |> col("TP53")
let lung = tsv("expr.tsv") |> filter(|r| r.tissue == "lung") |> col("TP53")
let kidney = tsv("expr.tsv") |> filter(|r| r.tissue == "kidney") |>
col("TP53")

let result = anova([brain, liver, lung, kidney])
# result => {f_statistic: 8.74, pvalue: 0.00003, df_between: 3, df_within:
76}
```

Fisher's Exact Test

Test enrichment of a mutation in cases versus controls.

#211 ▶ Run ▶▶ Run All Above + This

```
# Contingency table: mutation+/case, mutation-/case, mutation+/ctrl,
mutation-/ctrl
let result = fisher_exact(45, 155, 12, 188)
# result => {odds_ratio: 5.48, pvalue: 0.00001}
```

Multiple Testing Correction

When testing thousands of genes, you must correct for multiple comparisons. `p_adjust` supports Bonferroni, Benjamini-Hochberg (BH), and Benjamini-Yekutieli (BY) methods.

#212 ▶ Run ▶▶ Run All Above + This

```
let genes = tsv("gene_expression.tsv")
let gene_names = genes |> col("gene_name")
let tumor_cols = columns(genes) |> filter(|c| starts_with(c, "tumor_"))
let normal_cols = columns(genes) |> filter(|c| starts_with(c, "normal_"))

let pvalues = range(0, len(genes))
  |> map(|i| {
       let t = tumor_cols |> map(|c| col(genes, c)[i])
       let n = normal_cols |> map(|c| col(genes, c)[i])
       ttest(t, n).pvalue
     })

let adjusted_bh = p_adjust(pvalues, "BH")
let adjusted_bonf = p_adjust(pvalues, "bonferroni")

let significant = range(0, len(adjusted_bh))
  |> filter(|i| adjusted_bh[i] < 0.05)
  |> map(|i| {gene: gene_names[i], p: pvalues[i], q: adjusted_bh[i]})

print(str(len(significant)) + " genes significant at FDR < 0.05")
```

Correlation

Compute Pearson or Spearman correlation between expression profiles.

#213 ▶ Run ▶▶ Run All Above + This

```
let expr = tsv("tpm_matrix.tsv")
let gene_a = expr |> col("EGFR")
let gene_b = expr |> col("ERBB2")

let pearson = cor(gene_a, gene_b)
# pearson => {r: 0.72, pvalue: 0.00001}

let spearman = spearman(gene_a, gene_b)
# spearman => {rho: 0.68, pvalue: 0.00004}
```

Linear Models

Fit a linear model to predict expression from covariates.

#214 ▶ Run ▶▶ Run All Above + This

```
let data = tsv("sample_metadata.tsv")
let model = lm(~gene_expr ~ age + batch_id, data)
# model => {
#   coefficients: [5.2, 0.03, -1.1],
#   std_errors: [0.8, 0.01, 0.4],
#   p_values: [0.001, 0.02, 0.008],
#   r_squared: 0.45,
#   adj_r_squared: 0.42,
#   f_statistic: 12.3,
#   f_p_value: 0.00002
# }
```

Matrix Creation and Operations

Create matrices from data and perform standard operations.

#215 ▶ Run ▶▶ Run All Above + This

```
# Create a 3x3 count matrix (genes x samples)
let counts = matrix([
  [120, 340, 250],
  [890, 1200, 1050],
  [45, 30, 55]
])

# Transpose: samples x genes
let transposed = transpose(counts)

# Matrix multiplication
let product = mat_mul(transpose(counts), counts)  # 3x3 covariance-like

# Element-wise operations with mat_map
let log_counts = mat_map(counts, |x| log2(x + 1))

# Basic properties
print("Trace: " + str(trace(product)))
print("Rank: " + str(rank(counts)))
print("Frobenius norm: " + str(norm(counts)))
```

Linear Algebra

Determinant and Inverse

#216 ▶ Run ▶▶ Run All Above + This

```
let cov = matrix([
  [1.0, 0.8, 0.3],
  [0.8, 1.0, 0.5],
  [0.3, 0.5, 1.0]
])

let det = determinant(cov)
print("Determinant: " + str(det))  # non-zero => invertible

let precision = inverse(cov)  # precision matrix
```

Eigenvalues and Eigenvectors

#217 ▶ Run ▶▶ Run All Above + This

```
let eigen = eigenvalues(cov)
# eigen => {values: [2.1, 0.7, 0.2], vectors: [[...], [...], [...]]}

# Proportion of variance explained by each component
let total = sum(eigen.values)
let prop = eigen.values |> map(|v| v / total)
print("PC1 explains " + str(prop[0] * 100) + "% of variance")
```

Singular Value Decomposition

#218 ▶ Run ▶▶ Run All Above + This

```
let result = svd(counts)
# result => {u: matrix, s: [singular values], v: matrix}
```

Solving Linear Systems

#219 ▶ Run ▶▶ Run All Above + This

```
# Solve Ax = b
let a = matrix([[2, 1], [1, 3]])
let b = [5, 11]
let x = solve(a, b)
# x => [1.0, 3.0]
```

ODE Solving

`ode_solve` uses a 4th-order Runge-Kutta integrator. The signature is:

```
ode_solve(derivative_fn, initial_state, [t_start, t_end, dt])
```

It returns `{t: [...], y: [...]}` where `y` is a list of state vectors at each time point.

Example: Differential Expression Analysis

Run t-tests across all genes and correct for multiple testing.

#220 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
let expr = tsv("counts_normalized.tsv")
let metadata = tsv("sample_info.tsv")

let case_ids = metadata |> filter(|r| r.condition == "treated") |>
col("sample_id")
let ctrl_ids = metadata |> filter(|r| r.condition == "control") |>
col("sample_id")

let gene_names = expr |> col("gene_id")
let results = gene_names |> map(|gene| {
  let case_vals = case_ids |> map(|id| expr[id][index_of(gene_names, gene)])
  let ctrl_vals = ctrl_ids |> map(|id| expr[id][index_of(gene_names, gene)])
  let test = ttest(case_vals, ctrl_vals)
  let fc = mean(case_vals) / mean(ctrl_vals)
  {gene: gene, log2fc: log2(fc), pvalue: test.pvalue}
})

let p_vals = results |> map(|r| r.pvalue)
let q_vals = p_adjust(p_vals, "BH")

let de_genes = range(0, len(results))
  |> map(|i| {...results[i], q_value: q_vals[i]})
  |> filter(|r| r.q_value < 0.05 && abs(r.log2fc) > 1.0)
  |> sort_by(|r| r.q_value)

print(str(len(de_genes)) + " differentially expressed genes")
de_genes |> take(20) |> each(|g| print(g.gene + " log2FC=" + str(g.log2fc)))
de_genes |> write_tsv("de_results.tsv")
```

Example: PCA on Gene Expression

Use the covariance matrix eigendecomposition to project samples into PC space.

#221 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
let expr = tsv("tpm_matrix.tsv")
let sample_ids = expr |> columns() |> filter(|n| n != "gene_id")
let n_genes = len(expr)
let n_samples = len(sample_ids)

# Build samples x genes matrix
let data = sample_ids
  |> map(|id| expr |> col(id))
  |> matrix()

# Center each gene (column) to zero mean by rebuilding column-wise
let centered_data = range(0, n_samples) |> map(|i| {
  range(0, n_genes) |> map(|j| {
    let col_vals = mat_col(data, j)
    col_vals[i] - mean(col_vals)
  })
})
let centered = matrix(centered_data)

# Covariance matrix (samples x samples)
let cov = mat_mul(centered, transpose(centered))
  |> mat_map(|x| x / (n_genes - 1))

# Eigendecomposition
let eigen = eigenvalues(cov)
let total_var = sum(eigen.values)

# Project onto first 2 PCs
let pc1 = mat_col(eigen.vectors, 0)
let pc2 = mat_col(eigen.vectors, 1)

let pca_coords = range(0, n_samples)
  |> map(|i| {
       sample: sample_ids[i],
       pc1: pc1[i],
       pc2: pc2[i],
     })

print("PC1: " + str(eigen.values[0] / total_var * 100) + "% variance")
print("PC2: " + str(eigen.values[1] / total_var * 100) + "% variance")
pca_coords |> write_tsv("pca_coordinates.tsv")
```

Example: Pharmacokinetic Modeling

Model oral drug absorption and elimination using a two-compartment ODE system.

#222 ▶ Run ▶▶ Run All Above + This

```
# Parameters for a typical oral drug
let ka = 1.0      # absorption rate (1/hr)
let ke = 0.2      # elimination rate (1/hr)
let dose = 500.0  # mg

# State: [gut_amount, plasma_concentration]
# gut depletes by absorption, plasma gains from gut and loses by elimination
let pk_model = |t, y| [
  -ka * y[0],              # dA_gut/dt = -ka * A_gut
  ka * y[0] - ke * y[1],   # dC_plasma/dt = ka * A_gut - ke * C_plasma
]

let initial = [dose, 0.0]
let solution = ode_solve(pk_model, initial, [0.0, 24.0, 0.1])

# Find Cmax and Tmax
let plasma = solution.y |> map(|state| state[1])
let cmax = max(plasma)
let tmax_idx = index_of(plasma, cmax)
let tmax = solution.t[tmax_idx]

print("Tmax: " + str(tmax) + " hr")
print("Cmax: " + str(cmax) + " mg")

# Half-life
let t_half = log(2) / ke
print("Half-life: " + str(t_half) + " hr")

# AUC by trapezoidal rule
let auc = range(0, len(plasma) - 1)
  |> map(|i| (plasma[i] + plasma[i + 1]) / 2.0 * (solution.t[i + 1] -
solution.t[i]))
  |> sum()
print("AUC(0-24h): " + str(auc) + " mg*hr")
```

Example: Population Genetics (Hardy-Weinberg)

Simulate genotype frequencies and test for equilibrium across loci.

```
# Conceptual or diagnostic example; not directly executable.
# Observed genotype counts at 50 SNP loci
let loci = tsv("genotype_counts.tsv")
# columns: locus, AA, Aa, aa

let hwe_results = loci |> map(|loc| {
  let obs = [loc.AA, loc.Aa, loc.aa]
  let n = sum(obs)
  let p = (2 * loc.AA + loc.Aa) / (2 * n)
  let q = 1.0 - p

  let exp = [p * p * n, 2 * p * q * n, q * q * n]
  let test = chi_square(obs, exp)

  {
    locus: loc.locus,
    p_freq: p,
    q_freq: q,
    chi2: test.statistic,
    pvalue: test.pvalue,
  }
})

# Correct for multiple testing
let p_vals = hwe_results |> map(|r| r.pvalue)
let q_vals = p_adjust(p_vals, "BH")

let departures = range(0, len(hwe_results))
  |> map(|i| {...hwe_results[i], q_value: q_vals[i]})
  |> filter(|r| r.q_value < 0.05)

print(str(len(departures)) + " loci depart from HWE (FDR < 0.05):")
departures |> each(|r| print("  " + r.locus + " chi2=" + str(r.chi2)
                             + " q=" + str(r.q_value)))
```

Summary

BioLang's numerical toolbox covers the statistical tests and linear algebra operations that appear throughout computational biology. Descriptive stats, hypothesis tests with multiple-testing correction, matrix decompositions, and ODE integration are all available as built-in functions, keeping your analysis in a single language from raw data to publication-ready results.

Omics Packages

The BioLang source tree includes 37 importable packages spanning common omics and bioinformatics domains. Each package provides a curated, pipe-friendly API. Most operations use BioLang's native runtime; individual workflows may still call files, services, or external tools documented by that package.

Install a local package with `bl install`, or declare it in `biolang.toml`. Packages declaring `[lib].entry` can be imported by package name; other packages use their explicit `src/mod` entry:

```
bl install packages/variants
```

#223 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
import "singlecell"       as sc
import "variants/src/mod" as vcf
import "rnaseq/src/mod"   as rna
```

For a source checkout, set `BIOLANG_PATH` to the repository's `packages` directory before running examples. Installed packages are discovered under `~/.biolang/packages/`. Use `bl examples <package> --copy <directory>` to make an independent working copy of examples bundled with an installed package.

Current Package Catalog

Every package exposes code from `src/mod.bl`; packages with `[lib].entry` also support the short package-name import. The table is the complete catalog in the current source tree:

Package	Primary use	Package	Primary use
`annotation`	GTF/GFF parsing and feature annotation	`atac`	ATAC-seq QC, peaks, and accessibility

Package	Primary use	Package	Primary use
`cellchat`	Ligand-receptor communication scoring	`celltypes`	Marker-based cell-type annotation
`chipseq`	ChIP-seq peaks, FRiP, and consensus sets	`clustering`	General clustering workflows
`cnv`	Copy-number segments and gene-level calls	`crispr`	CRISPR screen QC and hit ranking
`deconvolution`	Bulk-mixture cell proportion estimates	`differential`	Differential expression testing
`drug`	Dose-response curves and synergy	`grn`	Gene-regulatory network inference
`gwas`	Summary statistics, loci, and inflation	`hic`	Hi-C matrices and contact analysis
`immune`	Immune signatures and repertoire summaries	`longread`	Long-read QC and length summaries
`metabolomics`	Feature matrices and metabolite analysis	`methylation`	CpG and region methylation analysis
`microbiome`	Taxonomic abundance and diversity	`motif`	Motif scans and enrichment
`multimodal`	Cross-modality integration	`network`	Biological graph analysis
`oric`	Bacterial origin-of-replication prediction	`pathway`	Pathway and gene-set enrichment
`phylo`	Newick trees and phylogenetic distances	`popgen`	Population-genetics statistics
`proteomics`	Protein abundance and differential analysis	`qpcr`	Ct, delta-Ct, and delta-delta-Ct

Package	Primary use	Package	Primary use
`rnaseq`	Bulk RNA-seq count processing	`scref`	Single-cell reference mapping
`singlecell`	Single-cell QC, normalization, and clustering	`spatial`	Spatial transcriptomics analysis
`statistics`	Reusable statistical helpers	`structure`	PDB and protein-structure analysis
`survival`	Kaplan-Meier and survival workflows	`variants`	VCF filtering and variant summaries
`velocity`	RNA-velocity workflows		

Use `bl metadata --format json` for core builtin signatures. Package APIs live in each package's `src/mod.bl`, which is the authoritative public surface.

---

1. Variant Analysis

The `variants` package wraps VCF parsing, filtering, and quality metrics. Use it for germline QC pipelines or to feed variant tables into downstream population genetics analysis.

#224 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "variants/src/mod" as vcf

let raw  = vcf.load("results/calls.vcf")
vcf.qc_report(raw)

let pass = vcf.load_filtered("results/calls.vcf", 30.0, true)
vcf.summary(pass)
vcf.titv(pass)

# Rare variant analysis (AF < 0.1%)
let rare = vcf.rare(pass, 0.001)
vcf.by_chrom(rare, "chr17")
```

Key functions: `load`, `load_filtered`, `summary`, `titv`, `rare`, `by_chrom`, `qc_report`

---

2. Bulk RNA-seq

The `rnaseq` package loads Salmon and featureCounts output, normalizes counts, and prepares matrices for differential expression analysis with the `differential` package.

#225 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
import "rnaseq/src/mod"       as rna
import "differential/src/mod" as de

let counts = rna.from_featurecounts("results/counts.txt")
rna.library_sizes(counts)
rna.correlation(counts)

let counts = rna.filter_genes(counts, 10, 3)
let norm   = rna.normalize_sf(counts)

let conditions = ["ctrl", "ctrl", "ctrl", "treated", "treated", "treated"]
let results = de.de_wald(norm, [0, 1, 2], [3, 4, 5])
results |> filter(|r| r.padj <= 0.05 && abs(r.log2fc) >= 1.0)
```

Key functions: `from_salmon`, `from_featurecounts`, `filter_genes`, `normalize_sf`, `library_sizes`, `correlation`

Single-cell RNA-seq

The `singlecell` package provides a sparse 10x workflow with synchronized cell and gene metadata. Filtering, normalization, variable-gene selection, PCA, neighbor-graph construction, and Leiden clustering can run without converting the expression matrix to a dense cells-by-genes array.

#226 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
import "singlecell" as sc

let cells = sc.load("filtered_feature_bc_matrix")
let result = cells
    |> sc.filter_genes(3)
    |> sc.filter_cells(200, 5000, 20.0)
    |> sc.normalize(10000.0)
    |> sc.variable_genes(2000)
    |> sc.run_pca(30)
    |> sc.neighbors(15)
    |> sc.cluster_leiden(15, 0.5)

println(sc.summary(result))
write_text("umap.svg", sc.plot_umap(result))
```

Use `sc.merge` before integration when samples share the same gene order. `sc.integrate` corrects the PCA embedding and records batch labels. Native AnnData Zarr exchange preserves CSR `X` matrices and observation/variable index names:

#227 ▶ Run ▶▶ Run All Above + This

```
write_anndata("analysis.zarr", result)
let restored = read_anndata("analysis.zarr")
```

Direct `.h5ad` I/O requires conversion with Python/anndata or a configured container. Arbitrary AnnData metadata columns and auxiliary layers are not yet copied by the native Zarr interchange. Validation scripts for Scanpy and Seurat are under `packages/singlecell/examples/validation`; compare cluster partitions by barcode using ARI rather than comparing arbitrary numeric cluster IDs.

Key functions: `load`, `filter_genes`, `filter_cells`, `normalize`, `variable_genes`, `run_pca`, `neighbors`, `cluster_leiden`, `merge`, `integrate`

---

3. Phylogenetics

The `phylo` package reads Newick trees, computes patristic distances, and reconstructs trees with UPGMA — useful for viral phylodynamics and species comparisons.

#228 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
import "phylo/src/mod" as ph

let tree = ph.load("data/sarscov2_sequences.nwk")
ph.n_leaves(tree)
ph.leaves(tree)

ph.distance(tree, "Alpha", "Delta")

let dmat    = ph.distance_matrix(tree)
let rebuilt = ph.build_upgma(ph.leaves(tree), dmat)
ph.faith_pd(tree)
```

Key functions: `load`, `n_leaves`, `leaves`, `distance`, `distance_matrix`, `build_upgma`, `faith_pd`

---

4. ChIP-seq & ATAC-seq

The `chipseq` package processes NarrowPeak files from MACS3, computes FRiP and TSS enrichment QC metrics, and builds consensus peak sets across replicates.

#229 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "chipseq/src/mod" as cs

let ctrl    = cs.load_narrowpeak("results/ctrl_peaks.narrowPeak")
let treated = cs.load_narrowpeak("results/treated_peaks.narrowPeak")

cs.qc_report(ctrl, 320000, 25000000)
cs.qc_report(treated, 280000, 22000000)

let consensus = cs.consensus([cs.merge(ctrl), cs.merge(treated)], 2)
cs.mean_peak_size(consensus)
cs.n_peaks(consensus)
```

Key functions: `load_narrowpeak`, `merge`, `consensus`, `qc_report`, `n_peaks`, `mean_peak_size`

---

5. Microbiome

The `microbiome` package computes alpha and beta diversity indices, performs rarefaction, and collapses OTU tables to arbitrary taxonomic ranks.

#230 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "microbiome/src/mod" as mb

let otus     = read_csv("data/feature_table.tsv")
let taxonomy = read_csv("data/taxonomy.tsv")

mb.alpha_table(otus, "shannon")
mb.alpha_table(otus, "chao1")

let rarefied = mb.rarefy_matrix(otus, 5000)
mb.beta(rarefied, "bray_curtis")

mb.top_taxa(taxonomy, 15, "genus")
mb.top_taxa(taxonomy, 10, "phylum")
```

Key functions: `alpha_table`, `beta`, `rarefy_matrix`, `top_taxa`, `relative_abundance`

---

6. Statistics

The `statistics` package extends the core statistical builtins with multiple-testing correction, permutation tests, bootstrap confidence intervals, and genomic inflation factors for GWAS.

#231 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "statistics/src/mod" as stat

let p_values = [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 0.9]

let adj_bh   = stat.bh(p_values)
let adj_bonf = stat.bonferroni(p_values)
stat.significant(adj_bh, 0.05)
stat.inflation(p_values)

let group_a = [2.1, 2.3, 1.9, 2.5, 2.2]
let group_b = [3.1, 2.8, 3.4, 2.9, 3.2]
stat.permtest(group_a, group_b, 10000)
stat.bootstrap(group_a, 5000, 0.95)
```

Key functions: `bh`, `bonferroni`, `significant`, `inflation`, `fisher`, `chi2`, `permtest`, `bootstrap`, `pearson`

---

7. RT-qPCR

The `qpcr` package implements the ΔCt and ΔΔCt method, standard-curve efficiency calculation, and geNorm stability scoring for reference gene selection.

#232 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "qpcr/src/mod" as qpcr

# Standard curve → amplification efficiency
let standard_cts  = [15.2, 18.5, 21.8, 25.1, 28.4]
let log_dilutions = [0.0, -1.0, -2.0, -3.0, -4.0]
qpcr.efficiency(standard_cts, log_dilutions)

# ΔCt and ΔΔCt fold change
let dct         = qpcr.dct(28.5, 22.0)
let fold_change = qpcr.ddct(dct, 6.2)

# Plate-level normalization and stability scoring
# let norm = qpcr.normalize(ct_table, ref_indices=[0, 1])
# qpcr.stability(ct_table, ref_indices=[0, 1])
```

Key functions: `dct`, `ddct`, `efficiency`, `normalize`, `stability`, `best_reference`

---

8. Proteomics

The `proteomics` package processes MaxQuant label-free quantification (LFQ) output through the full preprocessing pipeline and identifies differentially abundant proteins.

#233 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "proteomics/src/mod" as prot

# Load proteinGroups.txt → log2 → quantile normalize → impute
let mat = prot.full_pipeline("data/proteinGroups.txt")

# Differential abundance: samples 0-2 vs 3-5
let de   = prot.ttest(mat, [0, 1, 2], [3, 4, 5])
let hits = prot.significant(de, 1.0, 0.05)

# Prepare volcano plot data
prot.volcano(de, 1.0, 0.05)
```

Key functions: `load`, `log2_transform`, `quantile_normalize`, `impute`, `preprocess`, `full_pipeline`, `ttest`, `significant`, `volcano`

9. DNA Methylation

The `methylation` package handles RRBS and EPIC array data: beta/M-value conversion, differentially methylated region (DMR) calling, CpG density, and epigenetic clock prediction.

#234 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "methylation/src/mod" as meth

# beta_matrix: CpGs × samples
# let m_matrix = beta_matrix |> rows |> map(fn(r) -> meth.to_mvalue(r))

# Differential methylation (cases vs controls)
# let dmrs = meth.find_dmrs(beta_matrix, [0,1,2,3,4], [5,6,7,8,9])
# let diff = meth.diff_cpgs(beta_matrix, [0,1,2], [3,4,5])
# meth.significant_cpgs(diff, 0.15, 0.05)

# Epigenetic age (Horvath clock)
# meth.clock_age(beta_vec_353cpgs, horvath_coefs)

# CpG density in a window
let positions = [100, 120, 145, 300, 320, 340, 360]
meth.density(positions, 200)
```

Key functions: `to_mvalue`, `to_beta`, `find_dmrs`, `diff_cpgs`, `significant_cpgs`, `density`, `clock_age`

10. Protein Structure

The `structure` package parses PDB files, computes Kabsch-aligned RMSD, builds residue contact maps, assigns secondary structure (DSSP-lite), and extracts Ramachandran phi/psi angles.

#235 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "structure/src/mod" as st

let pred    = st.load("data/alphafold_prediction.pdb")
let exp     = st.load("data/experimental.pdb")

let pred_ca = st.ca_atoms(pred)
let exp_ca  = st.ca_atoms(exp)

st.rmsd(pred_ca, exp_ca)
st.ss_composition(pred_ca)
st.contacts(pred_ca, 8.0)
st.ramachandran(st.backbone(pred))
```

Key functions: `load`, `ca_atoms`, `backbone`, `chain`, `residues`, `rmsd`, `contacts`, `ss`, `ss_composition`, `ramachandran`

---

11. Biological Networks

The `network` package loads STRING or custom PPI edge lists and computes degree centrality, Brandes betweenness, BFS shortest paths, connected components, and hypergeometric disease module enrichment.

#236 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "network/src/mod" as net

let ppi = net.load_string("data/9606.protein.links.v12.0.txt")
  |> net.filter_score(0.7)

net.hub_genes(ppi, 20)
net.components(ppi)

let lcc        = net.largest_component(ppi)
let brca_genes = ["TP53", "BRCA1", "BRCA2", "ATM", "CHEK2", "PALB2"]
let disease_net = net.subnetwork(lcc, brca_genes)

net.degree(disease_net)
net.path(lcc, "TP53", "BRCA1")
net.enrichment(brca_genes, ppi, 20000)
```

Key functions: `load_string`, `load_csv`, `filter_score`, `subnetwork`, `degree`, `betweenness`, `path`, `components`, `hub_genes`, `largest_component`, `enrichment`

---

12. Population Genetics

The `popgen` package implements core population genetics statistics: HWE tests, Weir-Cockerham Fst, Tajima's D, LD r^2, allele frequency spectrum, and nucleotide diversity — all computed directly from genotype matrices.

#237 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "popgen/src/mod" as pg

# Hardy-Weinberg equilibrium test
pg.hwe(360, 480, 160)

# Fst between two populations
let ac_eur = [45, 12, 89, 3, 67]
let ac_afr = [78,  5, 92, 8, 34]
pg.fst(ac_eur, ac_afr, 200, 200)

# Tajima's D neutrality test
pg.tajima(15, 20, 1000)

# LD between two variants
let geno_a = [0, 1, 1, 2, 0, 1, 2, 0]
let geno_b = [0, 1, 1, 2, 0, 1, 1, 0]
pg.ld(geno_a, geno_b)
```

Key functions: `hwe`, `fst`, `tajima`, `ld`, `ld_matrix`, `afs`, `pi`, `watterson_theta`

---

cnv

The `cnv` package implements copy number variation analysis from whole-genome or whole-exome sequencing read-depth data. Use it to call amplifications and deletions in tumor samples, compute allele-specific copy numbers from B-allele frequencies, and summarise genome-wide CNV burden.

```
import "cnv/src/mod" as cnv
```

#238 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "cnv/src/mod" as cnv

# Per-bin log2 ratios from mosdepth output
# let ratios   = cnv.ratios(tumor_depths, normal_depths)

# Circular binary segmentation
# let segments = cnv.segment(ratios)

# Integer CN calls (diploid reference)
# cnv.call(segments, ploidy=2)

# Allele-specific CN from SNP heterozygotes
# let baf = read_csv("tumor_snps.tsv") |> col("baf")
# cnv.allelic(baf, ratios)

# Genome-wide summary
# cnv.summary(segments)
```

Key functions: `ratios`, `segment`, `call`, `allelic`, `summary`

---

hic

The `hic` package provides tools for Hi-C chromatin conformation capture data: ICE normalization of raw contact matrices, insulation score computation for TAD detection, and distance-decay analysis. Use it any time you need to identify topologically associating domains or inspect 3D genome organisation.

```
import "hic/src/mod" as hic
```

#239 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "hic/src/mod" as hic

# contact_matrix: N×N Table (genomic bins × bins, values = contact counts)

# ICE normalization
# let norm   = hic.normalize(contact_matrix)

# Insulation score (window = 10 bins)
# let scores = hic.insulation(norm, 10)

# TAD boundaries from insulation score minima
# let tads   = hic.boundaries(scores, delta=0.15)

# Distance decay: average contacts vs genomic separation
# hic.decay(norm)
```

Key functions: `normalize`, `insulation`, `boundaries`, `decay`, `expected`

atac

The `atac` package provides ATAC-seq quality control metrics derived from fragment size distributions. Use it to assess chromatin accessibility library quality — NFR enrichment, nucleosome phasing, and TSS enrichment — before proceeding to peak calling.

```
import "atac/src/mod" as atac
```

#240 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
import "atac/src/mod" as atac

# Fragment lengths from filtered BAM
# let frag_lengths = read_csv("fragments.tsv") |> col("tlen") |> map(fn(x) ->
abs(x))

# Full QC report
# atac.qc(frag_lengths)

# Fragment size histogram
# atac.sizes(frag_lengths)

# NFR enrichment score (> 1.5 = good quality)
# atac.nfr(frag_lengths)

# Nucleosome fraction breakdown (NFR / mono / di / tri)
# atac.nucleosomes(frag_lengths)

# TSS enrichment (requires per-fragment TSS distances)
# atac.tss(frag_lengths, tss_distances, flank=2000)
```

Key functions: `qc`, `nfr`, `sizes`, `nucleosomes`, `tss`

drug

The `drug` package fits dose-response curves and computes drug combination synergy scores. Use it for GDSC/PRISM-style pharmacogenomics analysis, IC50 estimation from viability data, and Bliss or Loewe synergy scoring for combination screens.

```
import "drug/src/mod" as drug
```

#241 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
import "drug/src/mod" as drug

let concentrations = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0, 10000.0]
let viabilities    = [98.0, 95.0, 88.0, 70.0, 45.0, 20.0, 8.0, 3.0]

# Fit 4-parameter logistic → {ic50, slope, top, bottom, r2}
let params = drug.fit(concentrations, viabilities)

# Area under dose-response curve (lower = more sensitive)
drug.auc(concentrations, viabilities)

# Combination synergy
drug.bliss(75.0, 60.0, 30.0)
drug.loewe(1.0, 2.0, 0.5, 1.0)
```

Key functions: `fit`, `curve`, `auc`, `bliss`, `loewe`, `rank`

---

gwas

The `gwas` package handles GWAS summary statistics: flexible column-name auto-detection, Manhattan and QQ plot data preparation, genomic inflation, and distance-based locus clumping. Use it to process output from PLINK, BOLT-LMM, SAIGE, or any standard GWAS tool.

```
import "gwas/src/mod" as gwas
```

#242 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
import "gwas/src/mod" as gwas

# Auto-detect column names (CHR/BP/SNP/P/BETA/SE/A1/A2/MAF)
let ss = gwas.load("data/my_gwas_sumstats.txt")

# Genomic inflation factor (should be ~1.0)
gwas.lambda(ss.pval)

# Genome-wide significant hits (p < 5×10⁻⁸)
let hits = gwas.hits(ss)

# Distance-based LD clumping → independent loci
let loci = gwas.clump(ss, 5e-8, 250)

# Data for Manhattan and QQ plots
gwas.manhattan(ss)
gwas.qq(ss.pval)
```

Key functions: `load`, `hits`, `clump`, `lambda`, `manhattan`, `qq`

---

annotation

The `annotation` package parses GTF/GFF3 gene annotation files and performs genomic interval operations. Use it to annotate ChIP-seq or ATAC-seq peaks with nearest genes, extract promoter regions, compute interval overlaps, and build Ensembl-to-gene-name lookup tables.

```
import "annotation/src/mod" as ann
```

#243 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
import "annotation/src/mod" as ann

# Load GENCODE or Ensembl GTF (auto-detects GTF vs GFF3)
let gtf   = ann.load("data/gencode.v45.annotation.gtf")

# Gene bodies and promoter windows
let genes = ann.genes(gtf)
let proms = ann.promoters(gtf, 2000, 200)

# Ensembl ID → gene name lookup
let idmap = ann.id_map(gtf)

# Annotate peaks with nearest gene + feature type
# let peaks = read_csv("data/macs3_peaks.narrowPeak")
# ann.annotate(peaks, gtf)

# Overlap ChIP-seq peaks with ATAC-seq peaks
# let atac_peaks = read_csv("data/atac_peaks.bed")
# ann.overlap(peaks, atac_peaks)
```

Key functions: `load`, `genes`, `promoters`, `id_map`, `annotate`, `overlap`

Chapter 13: Biological Database APIs

Bioinformatics analysis rarely lives in isolation. You query NCBI for gene annotations, check UniProt for protein function, pull variant consequences from Ensembl VEP, and look up pathways in KEGG. BioLang provides 16 built-in database client functions so you can fetch, cross-reference, and integrate biological data without leaving your script.

All bio API functions are builtins. No imports are needed.

NCBI

The National Center for Biotechnology Information hosts PubMed, Gene, Nucleotide, and dozens of other databases. BioLang provides three NCBI functions.

ncbi_search

Search any NCBI database (PubMed, Gene, Nucleotide, Protein, etc.).

#244 ▶ Run ▶▶ Run All Above + This

```
# requires: internet connection
# requires: NCBI_API_KEY (optional, increases rate limit)
# Search PubMed for recent CRISPR-Cas9 papers
# ncbi_search(db, query, max_results?)
let ids = ncbi_search("pubmed", "CRISPR-Cas9 delivery 2024", 20)
# ids => ["39012345", "39012346", ...]

# Get summaries for the IDs
let summaries = ncbi_summary(ids, "pubmed")
summaries |> each(|s| {
  print(s.uid)
})
```

ncbi_gene

Fetch detailed gene information by gene ID or symbol.

#245 ▶ Run ▶▶ Run All Above + This

```
# requires: internet connection
# requires: NCBI_API_KEY (optional, increases rate limit)
# ncbi_gene(symbol_or_query, max_results?)
let tp53 = ncbi_gene("TP53")
# If a single gene matches, returns a record:
# {id, symbol, name, description, organism, chromosome, location, summary}

print(tp53.name + " on chr" + tp53.chromosome + ": " + tp53.location)
```

ncbi_sequence

Retrieve nucleotide or protein sequences from NCBI.

#246 ▶ Run ▶▶ Run All Above + This

```
# requires: internet connection
# requires: NCBI_API_KEY (optional, increases rate limit)
# ncbi_sequence(accession) — returns FASTA text
let fasta = ncbi_sequence("NM_000546.6")
print("FASTA (first 100 chars): " + fasta[0..100])
```

Set the `NCBI_API_KEY` environment variable to get 10 requests/second instead of the default 3.

Ensembl

ensembl_gene

Look up a gene via the Ensembl REST API.

#247 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# requires: internet connection
# ensembl_symbol(species, symbol) — lookup by symbol
let brca1 = ensembl_symbol("human", "BRCA1")
# Returns: {id, symbol, description, species, biotype, start, end, strand,
chromosome}

print("Ensembl ID: " + brca1.id)
print("Location: " + brca1.chromosome + ":" + str(brca1.start) + "-" +
str(brca1.end))

# ensembl_gene(ensembl_id) — lookup by Ensembl ID
let same = ensembl_gene("ENSG00000012048")
print("Symbol: " + same.symbol)
```

ensembl_vep

Predict the functional consequences of variants using the Variant Effect Predictor.

#248 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# requires: internet connection
# ensembl_vep(hgvs) — predict variant consequences
let variants = [
  "17:g.43091434C>T",   # BRCA1 splice donor
  "7:g.140753336A>T",   # BRAF V600E
  "12:g.25245350C>A",   # KRAS G12V
]

let predictions = variants |> map(|v| ensembl_vep(v))

# Each result is a list of records with:
# {allele_string, most_severe_consequence, transcript_consequences: [...]}
predictions |> each(|pred| {
  if len(pred) > 0 then {
    let r = pred[0]
    print(r.allele_string + " => " + r.most_severe_consequence)
  }
})
```

UniProt

uniprot_search

Search the UniProt protein database.

#249 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# requires: internet connection
# uniprot_search(query, limit?)
let kinases = uniprot_search("kinase AND organism_id:9606", 50)
# Returns list of records: {accession, name, organism, sequence_length,
gene_names, function}

print(str(len(kinases)) + " human kinases found")
kinases |> take(5) |> each(|k| print(k.accession + " " + k.gene_names))
```

uniprot_entry

Get full details for a single UniProt accession.

#250 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# requires: internet connection
let entry = uniprot_entry("P04637")  # TP53
# Returns: {accession, name, organism, sequence_length, gene_names, function}

print(entry.name + ": " + str(entry.sequence_length) + " aa")
print("Genes: " + entry.gene_names)
print("Function: " + entry.function)

# Get protein features separately
let features = uniprot_features("P04637")
# Returns list of: {type, location, description}
let domains = features |> filter(|f| f.type == "Domain")
domains |> each(|d| print("  " + d.description + ": " + d.location))
```

UCSC Genome Browser

ucsc_sequence

Retrieve genomic sequences from the UCSC Genome Browser.

#251 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# requires: internet connection
# Get the sequence of the BRCA1 promoter region
# ucsc_sequence(genome, chrom, start, end)
let promoter = ucsc_sequence("hg38", "chr17", 43170245, 43172245)
# Returns DNA sequence as a string

print("BRCA1 promoter length: " + str(len(promoter)) + " bp")
let gc = gc_content(dna(promoter))
print("GC content: " + str(gc))
```

KEGG

kegg_find

Search KEGG databases (pathway, enzyme, compound, etc.).

#252 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# requires: internet connection
# kegg_find(db, query) — search KEGG databases
let pathways = kegg_find("pathway", "apoptosis human")
# Returns list of: {id, description}
```

kegg_get

Retrieve a specific KEGG entry.

#253 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# requires: internet connection
# kegg_get(entry_id) — returns raw KEGG text
let apoptosis = kegg_get("hsa04210")
# Returns the KEGG flat-file text for the entry
print("Entry text length: " + str(len(apoptosis)) + " chars")
```

STRING

string_network

Query protein-protein interaction networks from the STRING database.

#254 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# requires: internet connection
# string_network(identifiers, species)
# First argument must be a list of protein names
let network = string_network(["TP53"], 9606)
# Returns list of: {protein_a, protein_b, score}

network |> each(|i| print(i.protein_a + " <-> " + i.protein_b + " (" +
str(i.score) + ")"))
```

PDB

pdb_entry

Retrieve protein structure metadata from the Protein Data Bank.

#255 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# requires: internet connection
let structure = pdb_entry("1TUP")  # TP53 DNA-binding domain
# structure => {
#   id: "1TUP",
#   title: "Crystal structure of the p53 core domain...",
#   resolution: 2.2,
#   method: "X-RAY DIFFRACTION",
#   chains: [{id: "A", entity: "Tumor suppressor p53", length: 195}, ...],
#   ligands: [...],
# }

print(structure.title)
print("Resolution: " + str(structure.resolution) + " A")
```

Reactome

reactome_pathways

Find pathways associated with a gene or set of genes.

#256 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# requires: internet connection
# reactome_pathways(gene_or_genes)
let pathways = reactome_pathways("BRCA1")
# Returns pathway records

pathways |> each(|p| print(p))
```

Gene Ontology

go_term

Look up a GO term by its identifier.

#257 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# requires: internet connection
let term = go_term("GO:0006915")
# term => {id: "GO:0006915", name: "apoptotic process",
#          namespace: "biological_process",
#          definition: "A programmed cell death process..."}
```

go_annotations

Get GO annotations for a gene.

#258 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# requires: internet connection
# go_annotations(gene_or_accession)
let annotations = go_annotations("TP53")
# Returns list of: {go_id, term, aspect}

annotations |> take(10) |> each(|a| print("  " + a.go_id + " [" + a.aspect +
"]: " + a.term))
```

COSMIC

cosmic_gene

Query the Catalogue Of Somatic Mutations In Cancer. Requires `COSMIC_API_KEY`.

#259 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# requires: internet connection
# requires: COSMIC_API_KEY
# cosmic_gene(gene) — requires COSMIC_API_KEY env var
let cosmic = cosmic_gene("BRAF")
# Returns mutation data for the gene

print(cosmic)
```

NCBI Datasets

datasets_gene

Use the NCBI Datasets API for gene metadata.

#260 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# requires: internet connection
# datasets_gene(symbol_or_id)
let info = datasets_gene("EGFR")
print(info)
```

Environment Variables

Some APIs require or benefit from API keys:

Variable	Effect
`NCBI_API_KEY`	NCBI: 10 req/sec instead of 3
`COSMIC_API_KEY`	Required for COSMIC queries

Set these in your shell or in a `.env` file before running your script.

Example: Gene Annotation Pipeline

Fetch gene info from NCBI, cross-reference with UniProt, and pull pathways from KEGG.

#261 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# requires: internet connection
# requires: NCBI_API_KEY (optional, increases rate limit)
let gene_list = ["TP53", "BRCA1", "EGFR", "KRAS", "PIK3CA"]

let annotated = gene_list |> map(|symbol| {
  let ncbi = ncbi_gene(symbol)
  let up_hits = uniprot_search("gene:" + symbol + " AND organism_id:9606", 1)
  let prot_len = if len(up_hits) > 0 then up_hits[0].sequence_length else 0

  let pathways = kegg_find("pathway", symbol + " human")
    |> take(3)

  {
    symbol:         symbol,
    name:           ncbi.name,
    chromosome:     ncbi.chromosome,
    location:       ncbi.location,
    protein_length: prot_len,
    pathways:       pathways |> map(|p| p.description),
  }
})

annotated |> each(|g| {
  print(g.symbol + " (" + g.name + ") - chr" + g.chromosome)
  print("  Protein: " + str(g.protein_length) + " aa")
  print("  Pathways: " + str(g.pathways))
})

annotated |> write_json("gene_annotations.json")
```

Example: Variant Interpretation

Predict variant effects with Ensembl VEP and check for known cancer mutations in COSMIC.

#262 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# requires: internet connection
# requires: COSMIC_API_KEY (for cosmic_gene calls)
let variants = tsv("candidate_variants.tsv")
  |> map(|r| r.chrom + ":" + str(r.pos) + ":" + r.ref + ":" + r.alt)

let interpreted = variants |> map(|v| {
  let vep = ensembl_vep(v)

  let worst = vep.consequences
    |> sort_by(|c| c.impact_rank)
    |> first()

  let gene = worst.gene_symbol

  # Check COSMIC if it is a missense or nonsense variant
  let cosmic_info = if worst.consequence == "missense_variant" or
                       worst.consequence == "stop_gained" then
    cosmic_gene(gene)
      |> |cg| cg.mutations
      |> find(|m| m.aa_change == worst.amino_acid_change)
  else
    nil

  {
    variant: v,
    gene: gene,
    consequence: worst.consequence,
    impact: worst.impact,
    sift: worst.sift_prediction,
    polyphen: worst.polyphen_prediction,
    cosmic_count: if cosmic_info != nil then cosmic_info.count else 0,
    cosmic_known: cosmic_info != nil,
  }
})

# Flag high-impact or COSMIC-known variants
let flagged = interpreted
  |> filter(|v| v.impact == "HIGH" or v.cosmic_known)
  |> sort_by(|v| -v.cosmic_count)

print(str(len(flagged)) + " variants flagged for review")
flagged |> write_tsv("flagged_variants.tsv")
```

Example: Protein Interaction Network

Build a STRING interaction network for a set of differentially expressed genes and annotate with Reactome pathways.

#263 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# requires: internet connection
let de_genes = tsv("de_results.tsv")
  |> filter(|r| r.q_value < 0.01 and abs(r.log2fc) > 2.0)
  |> map(|r| r.gene)

# Get STRING interactions for the top 50 DE genes
let top_genes = de_genes |> take(50)
let network = string_network(top_genes, 9606)
# Returns list of: {protein_a, protein_b, score}

print(str(len(network)) + " interactions")

# Find hub genes (most connections)
let all_proteins = network |> map(|i| i.protein_a) + network |> map(|i|
i.protein_b)
let unique_proteins = all_proteins |> unique()
let degree = unique_proteins |> map(|p| {
  let edges = network |> filter(|i| i.protein_a == p || i.protein_b == p)
  {gene: p, degree: len(edges)}
})
  |> sort(|a, b| b.degree - a.degree)

print("Hub genes:")
degree |> take(10) |> each(|d| print("  " + d.gene + ": " + str(d.degree) + "
interactions"))

# Pathway enrichment for hub genes
let hub_genes = degree |> take(10) |> map(|d| d.gene)
let pathways = reactome_pathways(hub_genes)
  |> filter(|p| p.p_value < 0.05)
  |> sort_by(|p| p.p_value)

print("Enriched pathways:")
pathways |> take(10) |> each(|p| print("  " + p.name + " (p=" +
str(p.p_value) + ")"))
```

Summary

BioLang's built-in bio API functions let you query 12 major biological databases directly from your scripts. Combine them with pipes, maps, and filters to build annotation pipelines that cross-reference genes, variants, proteins, and pathways in a few lines of code. Set API keys via environment variables to increase rate limits where supported.

Chapter 14: Streams and Scale

Genomics data is large. A single whole-genome sequencing run produces hundreds of gigabytes of FASTQ. A population-scale VCF can hold billions of variant records. Loading everything into memory is not an option. BioLang addresses this with streams: lazy, single-pass iterators that process data record by record without materializing the entire dataset.

What Is a Stream?

A stream is a `StreamValue` – a lazy sequence of items produced on demand. File readers in BioLang return streams by default. Streams are consumed once: after you iterate through a stream, it is exhausted.

#264 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# read_fastq returns a stream, not a list
let reads = read_fastq("data/reads.fastq")

# This processes one record at a time -- constant memory
let high_quality = reads
  |> filter(|r| mean_phred(r.quality) >= 30)
  |> map(|r| {id: r.id, length: r.length, gc: gc_content(r.seq)})
  |> write_tsv("read_stats.tsv")
```

No matter how large the FASTQ file is, this script uses the same amount of memory. Each record flows through `filter`, then `map`, then is written out before the next record is read.

Stream Operations

Streams support the same functional operations as lists, but they execute lazily.

map

Transform each element.

#265 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
read_vcf("data/variants.vcf")
  |> map(|v| {chrom: v.chrom, pos: v.pos, qual: v.qual, af: v.info.AF})
  |> write_tsv("variant_summary.tsv")
```

filter

Keep only elements matching a predicate.

#266 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
read_bam("aligned.bam")
  |> filter(|r| r.mapping_quality >= 30 and not r.is_duplicate)
  |> count()
  |> |n| print(str(n) + " high-quality unique alignments")
```

take_while

Consume from a stream while a condition holds, then stop.

#267 ▶ Run ▶▶ Run All Above + This

```
# Read the first million reads from a FASTQ for quick QC
let sample = read_fastq("data/reads.fastq")
  |> take_while(|r, i| i < 1000000)

let quals = sample |> map(|r| mean_phred(r.quality))
print("Sampled stats: " + str(mean(quals)) + " mean Q")
```

each

Execute a side effect for every element (the stream analogue of a for loop).

#268 ▶ Run ▶▶ Run All Above + This

```
let bad_count = 0
read_fastq("data/reads.fastq")
  |> each(|r| {
       if mean_phred(r.quality) < 20 then
         bad_count = bad_count + 1
     })
print(str(bad_count) + " low-quality reads")
```

fold / reduce

Accumulate a value across the entire stream.

#269 ▶ Run ▶▶ Run All Above + This

```
let total_bases = read_fastq("data/reads.fastq")
  |> map(|r| r.length)
  |> sum()
print("Total bases: " + str(total_bases))
```

Chunked Processing

When you need batch operations on a stream, `chunk` groups elements into fixed-size lists. This is useful for writing output in blocks or batching API calls.

```
# Conceptual or diagnostic example; not directly executable.
# Process a FASTQ in chunks of 10,000 reads
read_fastq("data/reads.fastq")
  |> chunk(10000)
  |> each(|batch| {
       let mean_q = batch |> map(|r| mean_phred(r.quality)) |> mean()
       print("Batch: " + str(len(batch)) + " reads, "
             + "mean Q=" + str(mean_q))
     })
```

Chunking is also useful for writing to multiple output files.

#270 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# Split a FASTQ into files of 1 million reads each
let file_num = 0
read_fastq("data/reads.fastq")
  |> chunk(1000000)
  |> each(|batch| {
       let path = "split/chunk_" + str(file_num) + ".fastq.gz"
       batch |> write_fastq(path)
       file_num = file_num + 1
     })
print("Wrote " + str(file_num) + " chunk files")
```

Window Operations

`window` creates a sliding window over a list (or materialized portion of a stream). This is essential for computing positional statistics across a genome.

#271 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# Sliding window GC content
read_fasta("data/sequences.fasta") |> first() |> |r| r.seq |> into sequence
let win_size = 1000
let step = 500

let gc_track = window(sequence, win_size)
  |> enumerate()
  |> map(|pair| {
       pos: pair[0] * step,
       gc: gc_content(pair[1]),
     })

gc_track |> write_tsv("chr1_gc_content.tsv")
```

For windowed operations on streams (where you cannot look back), use `window` which maintains a buffer.

#272 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# Compute rolling average base quality across a BAM
read_bam("sample.bam")
  |> filter(|r| r.chrom == "chr17")
  |> sort_by(|r| r.pos)
  |> window(100)
  |> map(|win| {
       pos: win[50].pos,
       mean_mapq: mean(win |> map(|r| r.mapping_quality)),
     })
  |> write_tsv("chr17_mapq_rolling.tsv")
```

Parallel Operations

par_map

Apply a function to each element of a list in parallel, distributing work across available CPU cores.

#273 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
let chromosomes = ["chr1", "chr2", "chr3", "chr4", "chr5", "chr6",
                   "chr7", "chr8", "chr9", "chr10", "chr11", "chr12",
                   "chr13", "chr14", "chr15", "chr16", "chr17", "chr18",
                   "chr19", "chr20", "chr21", "chr22", "chrX", "chrY"]

let per_chrom_stats = par_map(chromosomes, |chrom| {
  let reads = read_bam("sample.bam") |> filter(|r| r.chrom == chrom)
  let count = reads |> count()
  let depth_table = tool("samtools", "depth -r " + chrom + " sample.bam")
  let depth = depth_table["depth"] |> mean()
  {chrom: chrom, reads: count, mean_depth: depth}
})

per_chrom_stats |> sort_by(|s| s.chrom) |> write_tsv("chrom_stats.tsv")
```

par_filter

Filter elements in parallel. Useful when the predicate itself is expensive.

```
# Conceptual or diagnostic example; not directly executable.
# requires: internet connection
let variants = tsv("all_variants.tsv")

let pathogenic = par_filter(variants, |v| {
  let vep = ensembl_vep(v.chrom + ":" + str(v.pos) + ":"
                        + v.ref + ":" + v.alt)
  let worst = vep.consequences |> first()
  worst.impact == "HIGH"
})

print(str(len(pathogenic)) + " high-impact variants")
```

Unit Helpers

BioLang provides genomic unit helpers that make size comparisons readable.

```
let region_size = mb(3)          # 3,000,000
let read_length = bp(150)        # 150
let window = kb(50)              # 50,000
let genome_size = gb(3)          # 3,000,000,000

# Use in calculations
let coverage = 1800000000 / genome_size  # ~0.6x
print("Coverage: " + str(coverage) + "x")

# Filter regions by size
let large_svs = read_vcf("data/variants.vcf")
  |> filter(|v| v.info.SVLEN != nil and abs(v.info.SVLEN) > mb(1))

print(str(large_svs |> count()) + " SVs larger than 1 Mb")
```

These helpers are simple multipliers but they prevent off-by-three-zeros bugs that plague genomics scripts.

Example: Process a 100GB FASTQ

Stream 4 billion reads, filter by quality, and compute statistics without loading the file into memory.

```
# Conceptual or diagnostic example; not directly executable.
let input = "novaseq_R1.fastq.gz"  # 100 GB compressed

let stats = {
  total: 0,
  passed: 0,
  total_bases: 0,
  gc_bases: 0,
  qual_sum: 0,
}

read_fastq(input)
  |> each(|r| {
       stats.total = stats.total + 1
       let q = mean_phred(r.quality)
       if q >= 30 then {
         stats.passed = stats.passed + 1
         stats.total_bases = stats.total_bases + r.length
         let gc = gc_content(dna(r.seq)) * r.length
         stats.gc_bases = stats.gc_bases + gc
         stats.qual_sum = stats.qual_sum + q
       }
     })

let pct_pass = stats.passed / stats.total * 100
let gc = stats.gc_bases / stats.total_bases * 100
let mean_q = stats.qual_sum / stats.passed

print("Total reads:  " + str(stats.total))
print("Passed Q>=30: " + str(stats.passed) + " (" + str(pct_pass) + "%)")
print("Mean quality: " + str(mean_q))
print("GC content:   " + str(gc) + "%")
```

This script processes the entire file in a single pass. Memory usage stays constant regardless of file size.

Example: Parallel Variant Annotation Across Chromosomes

Split variant annotation work by chromosome to use all cores.

```
# Conceptual or diagnostic example; not directly executable.
# requires: internet connection
let vcf_path = "cohort.vcf.gz"
let chromosomes = range(1, 23) |> map(|n| "chr" + str(n))
let chromosomes = concat(chromosomes, ["chrX", "chrY"])

let annotated = par_map(chromosomes, |chrom| {
  let variants = read_vcf(vcf_path)
    |> filter(|v| v.chrom == chrom)
    |> collect()

  let results = variants |> map(|v| {
    let vep = ensembl_vep(v.chrom + ":" + str(v.pos) + ":" + v.ref + ":" +
v.alt)
    let worst = vep.consequences
      |> sort_by(|c| c.impact_rank)
      |> first()
    {
      ...v,
      consequence: worst.consequence,
      impact: worst.impact,
      gene: worst.gene_symbol,
    }
  })

  print(chrom + ": " + str(len(results)) + " variants annotated")
  results
})
  |> flatten()

annotated |> write_tsv("annotated_variants.tsv")
print("Annotated " + str(len(annotated)) + " variants across "
      + str(len(chromosomes)) + " chromosomes")
```

Example: Sliding Window GC Content

Compute GC content in 1 kb windows across an entire chromosome.

#275 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
let chr_seq = read_fasta("data/sequences.fasta") |> first() |> |r| r.seq
let chr_len = len(chr_seq)
let win = kb(1)
let step = bp(500)

print("Chromosome 22: " + str(chr_len) + " bp")
print("Computing GC in " + str(win) + " bp windows, step " + str(step))

let gc_profile = window(chr_seq, win)
  |> enumerate()
  |> map(|pair| {
       start: pair[0] * step,
       end: pair[0] * step + win,
       gc: gc_content(pair[1]),
     })

# Identify GC-rich and GC-poor regions
let gc_rich = gc_profile |> filter(|w| w.gc > 0.60)
let gc_poor = gc_profile |> filter(|w| w.gc < 0.35)

print("GC-rich windows (>60%): " + str(len(gc_rich)))
print("GC-poor windows (<35%): " + str(len(gc_poor)))

gc_profile |> write_tsv("chr22_gc_profile.tsv")
```

Example: Batch Processing Thousands of Samples

Process a large sample cohort using chunked parallelism to avoid overwhelming system resources.

```
# Conceptual or diagnostic example; not directly executable.
let manifest = tsv("sample_manifest.tsv")  # 2,000 samples
print("Processing " + str(len(manifest)) + " samples")

let batch_size = 16  # process 16 samples at a time

let all_results = manifest
  |> chunk(batch_size)
  |> enumerate()
  |> map(|pair| {
       let batch_idx = pair[0]
       let batch = pair[1]
       print("Batch " + str(batch_idx + 1) + "/"
             + str(ceil(len(manifest) / batch_size)))

       par_map(batch, |sample| {
         # Align
         let sam = tool("bwa-mem2", "mem -t 2 GRCh38.fa " + sample.r1 + " " +
sample.r2)
         let bam = tool("samtools", "sort -@ 2 -o " + sample.sample_id +
".sorted.bam " + sam)
         tool("samtools", "index " + bam)

         # QC metrics
         let flagstat = tool("samtools", "flagstat " + bam)
         let depth_vals = tool("samtools", "depth " + bam)
         let mean_depth = depth_vals["depth"] |> mean()

         {
           id: sample.sample_id,
           bam: bam,
           mean_depth: mean_depth,
           status: if mean_depth > 20
                   then "PASS" else "FAIL",
         }
       })
     })
  |> flatten()

let passed = all_results |> filter(|r| r.status == "PASS")
let failed = all_results |> filter(|r| r.status == "FAIL")

print("Results: " + str(len(passed)) + " PASS, "
      + str(len(failed)) + " FAIL")

if len(failed) > 0 then {
  print("Failed samples:")
  failed |> each(|r| print("  " + r.id + " depth=" + str(r.mean_depth)
                           + " mapped=" + str(r.mapped_pct) + "%"))
}

all_results |> write_tsv("cohort_qc_results.tsv")
```

Memory Patterns to Know

Pattern	Memory	Use When
`read_fastq(f) |> filter(...) |> count()`	O(1)	Counting, simple stats
`read_vcf(f) |> collect()`	O(n)	Need random access
`read_bam(f) |> chunk(k) |> each(...)`	O(k)	Batch writing, API calls
`par_map(list, f)`	O(n)	List already in memory
`stream |> fold(init, f)`	O(1)	Aggregation

The rule of thumb: stay in stream mode as long as possible. Call `collect()` only when you genuinely need the full dataset in memory (for sorting, random access, or passing to a function that requires a list).

Summary

Streams let BioLang handle files of any size in constant memory. Combine them with `chunk` for batch processing, `window` for positional analysis, and `par_map` / `par_filter` for multi-core parallelism. The unit helpers `bp()`, `kb()`, `mb()`, and `gb()` keep genomic arithmetic readable. Together, these tools let you write scripts that scale from a test FASTQ to a population-scale dataset without changing the code.

SQLite & Notifications

BioLang includes built-in SQLite support and notification builtins so your pipelines can persist results and alert you when they finish — no external tools required.

SQLite

Bioinformatics workflows constantly produce tabular results: QC metrics, variant counts, sample manifests. SQLite gives you a zero-config embedded database to store, query, and compare results across runs.

Opening a Database

#276 ▶ Run ▶▶ Run All Above + This

```
# File-based (creates if missing)
let db = sqlite("project_results.db")

# In-memory (temporary, fast)
let scratch = sqlite()
```

Querying

`sql()` executes any SQL statement. SELECT queries return a Table; write statements return the number of affected rows. Use `?` placeholders for parameterized queries.

```
# Conceptual or diagnostic example; not directly executable.
# Create a table
sql(db, "CREATE TABLE IF NOT EXISTS qc (
  sample TEXT PRIMARY KEY,
  total_reads INTEGER,
  pass_reads INTEGER,
  pass_rate REAL
)")

# Insert data
sql(db, "INSERT INTO qc VALUES (?, ?, ?, ?)",
    ["tumor_01", 5000000, 4850000, 97.0])

# Query — returns a Table
let results = sql(db, "SELECT * FROM qc WHERE pass_rate > ?", [95.0])
print(results)
```

Since `sql()` returns a standard BioLang Table, you can pipe it directly:

#277 ▶ Run ▶▶ Run All Above + This

```
sql(db, "SELECT symbol, chrom, start, end FROM genes WHERE chrom = ?",
["chr17"])
  |> filter(|g| g.start > 40000000)
  |> sort_by(|g| g.start)
  |> print()
```

Bulk Insert

`sql_insert()` inserts an entire Table or list of records in a single transaction. This is much faster than individual INSERT statements.

#278 ▶ Run ▶▶ Run All Above + This

```
# From a Table (e.g., read from TSV)
let stats = tsv("variant_stats.tsv")
sql_insert(db, "variants", stats)

# From a list of records
let samples = [
  {id: "S1", depth: 42.3, mapped_pct: 98.1},
  {id: "S2", depth: 38.7, mapped_pct: 97.5},
]
let inserted = sql_insert(db, "qc_results", samples)
print(str(inserted) + " rows inserted")
```

Metadata

#279 ▶ Run ▶▶ Run All Above + This

```
# List all tables
sql_tables(db)
# ["qc", "variants", "qc_results"]

# Inspect a table's schema
sql_schema(db, "qc")
#  cid | name        | type    | notnull | pk
# -----+-------------+---------+---------+-----
#  0   | sample      | TEXT    | false   | true
#  1   | total_reads | INTEGER | false   | false
#  2   | pass_reads  | INTEGER | false   | false
#  3   | pass_rate   | REAL    | false   | false
```

Pipeline Example

Store QC results from every sample run, then query across all runs:

```
# Conceptual or diagnostic example; not directly executable.
let db = sqlite("lab_results.db")

sql(db, "CREATE TABLE IF NOT EXISTS qc (
  sample TEXT, run_date TEXT, total INTEGER, passing INTEGER, rate REAL
)")

pipeline sample_qc(sample_id, fastq_path) {
  stage stats {
    let reads = read_fastq(fastq_path) |> collect()
    let total = len(reads)
    let passing = reads |> filter(|r| mean_phred(r.quality) >= 25) |> len()
    let rate = passing / total * 100.0

    sql(db, "INSERT INTO qc VALUES (?, date('now'), ?, ?, ?)",
        [sample_id, total, passing, rate])

    {total: total, passing: passing, rate: rate}
  }
}

# After processing many samples, query trends
let low_quality = sql(db,
  "SELECT sample, rate FROM qc WHERE rate < ? ORDER BY rate",
  [95.0])
print("Samples below 95% pass rate:")
print(low_quality)
```

SQLite Builtins Summary

Builtin	Returns	Description
`sqlite(path?)`	DbHandle	Open/create database
`sql(db, query, params?)`	Table or Int	Execute SQL
`sql_insert(db, table, data)`	Int	Bulk insert (transactional)
`sql_tables(db)`	List	List table names
`sql_schema(db, table)`	Table	Column metadata
`sql_close(db)`	Nil	Close connection (optional)
`is_db(value)`	Bool	Type check

Notifications

Long-running pipelines (alignment, variant calling, cohort analysis) can take hours. BioLang's notification builtins send you a message when your analysis finishes or fails — Slack, Teams, Telegram, Discord, or email.

The `notify()` Builtin

`notify()` is the smart router. It reads `BIOLANG_NOTIFY` to determine which provider to use, then sends the message.

#280 ▶ Run ▶▶ Run All Above + This

```
# Simple string
notify("Alignment complete: 24 samples processed")

# Structured record — provider formats it natively
notify({
  title: "QC Pipeline Complete",
  status: "success",
  fields: {
    samples: 24,
    pass_rate: "96%",
    output: "/data/results/cohort.vcf.gz"
  }
})
```

If `BIOLANG_NOTIFY` is not set, `notify()` prints to stderr as a fallback.

Provider-Specific Builtins

Each provider has a dedicated builtin for direct use:

#281 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# Slack (env: SLACK_WEBHOOK)
slack("Variant calling finished: 1,234 SNPs, 456 indels")

# Microsoft Teams (env: TEAMS_WEBHOOK)
teams("RNA-seq pipeline complete: 12 samples normalized")

# Telegram (env: TELEGRAM_BOT_TOKEN, TELEGRAM_CHAT_ID)
telegram("Alignment done: tumor.sorted.bam")

# Discord (env: DISCORD_WEBHOOK)
discord("FASTQ QC complete: 98% pass rate")

# Email (env: SMTP_HOST, SMTP_USER, SMTP_PASS)
email("lab@example.com", "Pipeline Complete", "Analysis finished
successfully")
```

All webhook-based builtins accept an optional first argument for an explicit webhook URL, so you can skip env vars:

#282 ▶ Run ▶▶ Run All Above + This

```
slack("https://hooks.slack.com/services/xxx", "Pipeline done")
teams("https://outlook.office.com/webhook/xxx", "Pipeline done")
```

Structured Messages

Pass a record instead of a string for rich formatting. Each provider renders it natively (Slack Block Kit, Teams Adaptive Cards, Discord embeds):

#283 ▶ Run ▶▶ Run All Above + This

```
slack({
  title: "Cohort Analysis Complete",
  status: "success",
  fields: {
    samples: 48,
    variants: "2.3M",
    ti_tv: 2.1,
    output: "cohort.annotated.vcf.gz"
  }
})
```

Pipeline Integration

Use `notify()` at any point in a pipeline for progress updates:

#284 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# Variant calling pipeline with notifications
shell("bwa-mem2 mem -t 16 GRCh38.fa R1.fq.gz R2.fq.gz | samtools sort -o
aligned.bam")

shell("gatk HaplotypeCaller -R GRCh38.fa -I aligned.bam -O raw.vcf.gz")
let variants = read_vcf("data/variants.vcf") |> collect()
let snps = variants |> filter(|v| is_snp(v)) |> len()
let indels = variants |> filter(|v| is_indel(v)) |> len()

notify({
  title: "Variant Calling Complete",
  fields: {SNPs: snps, Indels: indels}
})
```

Setup

Set environment variables once in your shell profile:

```
# Pick your provider
export BIOLANG_NOTIFY=slack

# Slack
export SLACK_WEBHOOK=https://hooks.slack.com/services/T.../B.../xxx

# Telegram
export TELEGRAM_BOT_TOKEN=123456:ABC-DEF
export TELEGRAM_CHAT_ID=-1001234567890

# Email (for notify() with BIOLANG_NOTIFY=email)
export SMTP_HOST=smtp.gmail.com
export SMTP_USER=you@gmail.com
export SMTP_PASS=app-password
export NOTIFY_EMAIL_TO=lab@example.com
```

Notification Builtins Summary

Builtin	Provider	Env Vars
`notify(msg)`	Auto	`BIOLANG_NOTIFY` + provider vars
`slack(msg)`	Slack	`SLACK_WEBHOOK`
`teams(msg)`	Teams	`TEAMS_WEBHOOK`

Builtin	Provider	Env Vars
`telegram(msg)`	Telegram	`TELEGRAM_BOT_TOKEN`, `TELEGRAM_CHAT_ID`
`discord(msg)`	Discord	`DISCORD_WEBHOOK`
`email(to, subj, body)`	SMTP	`SMTP_HOST`, `SMTP_USER`, `SMTP_PASS`

Knowledge Graphs

BioLang includes a built-in graph data structure for modeling biological networks — protein interactions, gene regulatory networks, metabolic pathways, and more.

Creating Graphs

#285 ▶ Run ▶▶ Run All Above + This

```
# Empty undirected graph
let g = graph()

# Directed graph
let g = graph(true)
```

Adding Nodes and Edges

Nodes are identified by string IDs and can carry arbitrary attributes:

#286 ▶ Run ▶▶ Run All Above + This

```
let g = graph()
let g = add_node(g, "BRCA1", {biotype: "protein_coding", chrom: "chr17"})
let g = add_node(g, "TP53", {biotype: "protein_coding", chrom: "chr17"})
let g = add_edge(g, "BRCA1", "TP53", {score: 0.99, source: "STRING"})
```

Adding an edge auto-creates nodes that don't exist yet:

#287 ▶ Run ▶▶ Run All Above + This

```
let g = graph()
let g = add_edge(g, "A", "B")  # both A and B are created
has_node(g, "A")                # true
```

Querying the Graph

#288 ▶ Run ▶▶ Run All Above + This

```
# Build a small PPI network
let g = graph()
let g = add_edge(g, "BRCA1", "TP53", {score: 0.99})
let g = add_edge(g, "TP53", "MDM2", {score: 0.97})
let g = add_edge(g, "BRCA1", "BARD1", {score: 0.95})
let g = add_edge(g, "MDM2", "CDKN2A", {score: 0.85})

# Direct neighbors
neighbors(g, "TP53")           # ["BRCA1", "MDM2"]

# Node degree
degree(g, "BRCA1")             # 2

# Shortest path (BFS)
shortest_path(g, "BARD1", "CDKN2A")  # ["BARD1", "BRCA1", "TP53", "MDM2",
"CDKN2A"]

# All nodes and edges
nodes(g)                       # ["BARD1", "BRCA1", "CDKN2A", "MDM2", "TP53"]
edges(g)                       # Table with from, to, weight columns
```

Graph Analysis

Connected Components

Find disconnected subnetworks:

#289 ▶ Run ▶▶ Run All Above + This

```
let g = graph()
let g = add_edge(g, "BRCA1", "TP53")
let g = add_edge(g, "BRCA1", "BARD1")
let g = add_node(g, "ISOLATED_GENE")
let g = add_edge(g, "CDK2", "CCND1")

let components = connected_components(g)
print("Number of components: " + str(len(components)))
# 3: [BRCA1, TP53, BARD1], [ISOLATED_GENE], [CDK2, CCND1]
```

Induced Subgraph

Extract a subgraph containing only specified nodes and their connecting edges:

#290 ▶ Run ▶▶ Run All Above + This

```
let g = graph()
let g = add_edge(g, "A", "B")
let g = add_edge(g, "B", "C")
let g = add_edge(g, "C", "D")

let sub = subgraph(g, ["A", "B", "C"])
has_edge(sub, "A", "B")    # true
has_edge(sub, "C", "D")    # false (D not in subgraph)
```

Node Attributes

#291 ▶ Run ▶▶ Run All Above + This

```
let g = graph()
let g = add_node(g, "EGFR", {
    biotype: "protein_coding",
    chrom: "chr7",
    pathway: "EGFR signaling"
})

let attrs = node_attr(g, "EGFR")
print(attrs.pathway)    # "EGFR signaling"
```

Removing Nodes and Edges

#292 ▶ Run ▶▶ Run All Above + This

```
let g = graph()
let g = add_edge(g, "A", "B")
let g = add_edge(g, "B", "C")

# Remove a single edge
let g = remove_edge(g, "A", "B")
has_edge(g, "A", "B")    # false

# Remove a node (also removes all connected edges)
let g = remove_node(g, "B")
has_node(g, "B")          # false
```

Directed vs Undirected

By default, graphs are undirected — edges go both ways:

#293 ▶ Run ▶▶ Run All Above + This

```
let g = graph()
let g = add_edge(g, "A", "B")
neighbors(g, "B")    # ["A"] — B sees A as neighbor
```

For directed graphs (e.g., regulatory networks):

#294 ▶ Run ▶▶ Run All Above + This

```
let g = graph(true)
let g = add_edge(g, "TF", "TARGET")
neighbors(g, "TF")      # ["TARGET"]
neighbors(g, "TARGET")  # [] — directed, no reverse edge
```

Real-World Example: STRING Network Analysis

#295 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# requires: internet connection
# Fetch protein interactions from STRING
let network = string_network(["BRCA1"], 9606)

# Build graph from API results — network is list of {protein_a, protein_b,
score}
let g = graph()
let g = network |> reduce(g, |g, edge|
    add_edge(g, edge.protein_a, edge.protein_b, {score: edge.score})
)

# Find hub genes (highest degree)
let gene_degrees = nodes(g) |> map(|n| {gene: n, deg: degree(g, n)})
gene_degrees
  |> sort_by(|r| -r.deg)
  |> take(10)
  |> each(|r| print(r.gene + ": " + str(r.deg) + " interactions"))

# Check connectivity
let components = connected_components(g)
print("Connected components: " + str(len(components)))
```

Builtin Reference

Function	Args	Description
`graph()`	`[directed]`	Create empty graph (default: undirected)
`add_node(g, id, [attrs])`	graph, Str, [Record]	Add node with optional attributes
`add_edge(g, from, to, [attrs])`	graph, Str, Str, [Record]	Add edge (auto-creates nodes)
`remove_node(g, id)`	graph, Str	Remove node and its edges
`remove_edge(g, from, to)`	graph, Str, Str	Remove first matching edge
`neighbors(g, id)`	graph, Str	List of neighbor node IDs
`degree(g, id)`	graph, Str	Number of edges for a node
`shortest_path(g, from, to)`	graph, Str, Str	BFS shortest path (nil if none)
`connected_components(g)`	graph	List of node-ID lists per component
`nodes(g)`	graph	Sorted list of all node IDs
`edges(g)`	graph	Table with from, to, weight columns
`has_node(g, id)`	graph, Str	Bool
`has_edge(g, from, to)`	graph, Str, Str	Bool
`subgraph(g, ids)`	graph, List[Str]	Induced subgraph
`node_attr(g, id)`	graph, Str	Attributes record for a node

PDB Structures & Enrichment Analysis

BioLang provides direct access to the RCSB Protein Data Bank and built-in gene set enrichment analysis.

PDB Structures

Fetching Entries

#296 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# requires: internet connection
let entry = pdb_entry("4HHB")
print(entry.title)          # "THE CRYSTAL STRUCTURE OF HUMAN
DEOXYHAEMOGLOBIN"
print(entry.method)         # "X-RAY DIFFRACTION"
print(entry.resolution)     # 1.74
print(entry.organism)       # "Homo sapiens"
print(entry.release_date)   # "1984-07-17"
```

Searching

#297 ▶ Run ▶▶ Run All Above + This

```
# requires: internet connection
let ids = pdb_search("insulin receptor")
print("Found " + str(len(ids)) + " structures")
ids |> take(5) |> each(|id| print(id))
```

Chains and Entities

#298 ▶ Run ▶▶ Run All Above + This

```
# requires: internet connection
# Get all polymer entities (chains) in a structure
let chains = pdb_chains("4HHB")
chains |> each(|c|
    print(c.description + " (" + c.entity_type + "): " + str(len(c.sequence))
+ " residues")
)

# Get a specific entity
let entity = pdb_entity("4HHB", 1)
print(entity.description)
print(entity.sequence)

# Get sequence as Protein value
let seq = pdb_sequence("4HHB", 1)
print(len(seq))             # sequence length
```

Real-World Example: Compare Hemoglobin Chains

#299 ▶ Run ▶▶ Run All Above + This

```
# requires: internet connection
let alpha = pdb_entity("4HHB", 1)
let beta = pdb_entity("4HHB", 2)
print("Alpha chain: " + str(len(alpha.sequence)) + " residues")
print("Beta chain: " + str(len(beta.sequence)) + " residues")
print("Alpha type: " + alpha.entity_type)
```

PubMed Integration

Search PubMed and retrieve abstracts:

#300 ▶ Run ▶▶ Run All Above + This

```
# requires: internet connection
# requires: NCBI_API_KEY (optional, increases rate limit)
# Search for recent CRISPR papers
let results = pubmed_search("CRISPR Cas9 therapy", 5)
print("Total results: " + str(results.count))

# Fetch abstract for a specific paper
let abstract = pubmed_abstract(results[0])
print(abstract)
```

Literature Review Pipeline

#301 ▶ Run ▶▶ Run All Above + This

```
# requires: internet connection
# requires: NCBI_API_KEY (optional, increases rate limit)
# Search for papers about a gene of interest
let gene = "BRCA1"
let results = pubmed_search(gene + " cancer therapy 2024", 20)

print("Found " + str(results.count) + " papers for " + gene)
results.ids
  |> take(5)
  |> each(|pmid|
    let text = pubmed_abstract(pmid)
    print("PMID " + pmid + ":")
    print(text)
    print("---")
  )
```

Enrichment Analysis

Over-Representation Analysis (ORA)

Test whether your gene list is enriched for specific biological pathways:

#302 ▶ Run ▶▶ Run All Above + This

```
# Load gene sets (GMT format from MSigDB, GO, KEGG, etc.)
let gene_sets = read_gmt("h.all.v2024.1.Hs.symbols.gmt")

# Your differentially expressed genes
let de_genes = ["BRCA1", "TP53", "CDK2", "CCND1", "RB1", "E2F1", "PCNA",
"MCM2"]

# Run ORA with background size (total genes measured)
let results = enrich(de_genes, gene_sets, 20000)

# Filter significant results
results
  |> filter(|r| r.fdr < 0.05)
  |> each(|r| print(r.term + ": p=" + str(r.p_value) + " FDR=" + str(r.fdr)))
```

The `enrich()` function uses the hypergeometric test with Benjamini-Hochberg FDR correction.

Output columns: `term`, `overlap`, `p_value`, `fdr`, `genes`

Gene Set Enrichment Analysis (GSEA)

For ranked gene lists (e.g., by fold change or t-statistic):

#303 ▶ Run ▶▶ Run All Above + This

```
# Prepare ranked table with gene and score columns
let ranked = tsv("de_results.tsv")
  |> select("gene", "log2fc")
  |> rename("log2fc", "score")
  |> sort_by(|r| -r.score)

# Load gene sets
let gene_sets = read_gmt("c2.cp.kegg.v2024.1.Hs.symbols.gmt")

# Run GSEA (1000 permutations)
let results = gsea(ranked, gene_sets)

# Top enriched pathways
results
  |> filter(|r| r.fdr < 0.25)
  |> each(|r| print(r.term + ": NES=" + str(r.nes) + " FDR=" + str(r.fdr)))
```

Output columns: `term`, `es` (enrichment score), `nes` (normalized ES), `p_value`, `fdr`, `leading_edge`

GMT File Format

The GMT (Gene Matrix Transposed) format is tab-delimited:

```
PATHWAY_NAME\tDescription\tGENE1\tGENE2\tGENE3\t...
```

Standard sources:

- **MSigDB** (Broad Institute) — Hallmark, KEGG, Reactome, GO sets
- **Enrichr** — curated gene set libraries
- **WikiPathways** — community-curated pathways

Real-World Example: RNA-seq Enrichment Pipeline

#304 ▶ Run ▶▶ Run All Above + This

```
# 1. Read differential expression results
let de = tsv("deseq2_results.tsv")

# 2. Get significant genes
let sig_genes = de
  |> filter(|r| r.padj < 0.05 and abs(r.log2FoldChange) > 1)
  |> map(|r| r.gene)
  |> collect()

print("Significant DE genes: " + str(len(sig_genes)))

# 3. Load pathway gene sets
let hallmark = read_gmt("h.all.v2024.1.Hs.symbols.gmt")
let kegg = read_gmt("c2.cp.kegg.v2024.1.Hs.symbols.gmt")

# 4. Run ORA on both
let h_results = enrich(sig_genes, hallmark, 20000)
let k_results = enrich(sig_genes, kegg, 20000)

# 5. Report significant pathways
print("\n=== Hallmark Pathways ===")
h_results |> filter(|r| r.fdr < 0.05) |> each(|r|
    print(r.term + " (overlap=" + str(r.overlap) + ", FDR=" + str(r.fdr) +
")")
)

print("\n=== KEGG Pathways ===")
k_results |> filter(|r| r.fdr < 0.05) |> each(|r|
    print(r.term + " (overlap=" + str(r.overlap) + ", FDR=" + str(r.fdr) +
")")
)
```

Builtin Reference

PDB

Function	Args	Returns	Description
`pdb_entry(id)`	Str	Record	Fetch PDB entry metadata
`pdb_search(query)`	Str	List[Str]	Full-text search, returns PDB IDs
`pdb_entity(id, entity_id)`	Str, Int	Record	Get specific polymer entity
`pdb_sequence(id, entity_id)`	Str, Int	Protein	Get entity sequence as Protein value
`pdb_chains(id)`	Str	List[Record]	Get all polymer entities for a structure

PubMed

Function	Args	Returns	Description
`pubmed_search(query, [max])`	Str, [Int]	Record{count, ids}	Search PubMed articles
`pubmed_abstract(pmid)`	Str	Str	Fetch abstract text

Enrichment

Function	Args	Returns	Description
`read_gmt(path)`	Str	Map{name -> List[Str]}	Parse GMT gene set file
`enrich(genes, sets, bg)`	List, Map, Int	Table	ORA with hypergeometric test + BH FDR
`ora(genes, sets, bg)`	List, Map, Int	Table	Alias for `enrich()`
`gsea(ranked, sets)`	Table, Map	Table	GSEA with permutation test + BH FDR

Chapter 17: Literate Notebooks

BioLang notebooks (`.bln` files) combine Markdown prose with executable code blocks. They're ideal for documenting analyses, creating reproducible reports, and sharing methods with collaborators.

Running a Notebook

```
bl notebook analysis.bln
```

Prose sections are rendered with ANSI formatting in the terminal. Code blocks are executed in order, with output interleaved between prose.

The .bln Format

A `.bln` file is plain text. Everything outside a code block is Markdown prose. Code blocks can use either fenced syntax or dash delimiters.

Fenced Code Blocks

Use standard Markdown fences with `biolang`, `bl`, or no language tag:

````
## Load Data

Read the FASTQ file and compute basic stats.

```biolang
let reads = read_fastq("data/reads.fastq")
print(f"Loaded {len(reads)} reads")
```

## Filter

Keep high-quality reads.

```bl
let filtered = reads |> filter(|r| mean_phred(r.quality) > 25)
print(f"Kept {len(filtered)} reads")
```
````

Dash Delimiters

The original `.bln` syntax uses `---` on its own line:

```
## Load Data
---
let reads = read_fastq("data/reads.fastq")
print(f"Loaded {len(reads)} reads")
---
## Results
The output above shows the read count.
```

Both styles can be mixed in the same notebook. Fenced blocks with other language tags (e.g. ```` ```python ````) are treated as prose and not executed.

State Carries Over

Variables defined in one code block are available in all later blocks. This lets you build up an analysis step by step, explaining each part in prose.

Cell Directives

Add special comment annotations at the top of a code block to control behavior:

Directive	Effect
`# @hide`	Execute but don't display the code
`# @skip`	Don't execute this block
`# @echo`	Print the code before executing
`# @hide-output`	Execute (and show code) but suppress printed output

Directives are stripped from the code before execution. Multiple directives can be combined:

````
```biolang
# @hide
let config = {min_quality: 30, reference: "GRCh38"}
```

```biolang
# @echo
let reads = read_fastq("data/reads.fastq")
  |> filter(|r| mean_phred(r.quality) >= config.min_quality)
print(f"Filtered: {len(reads)} reads")
```
````

With `# @hide`, the config block runs silently. With `# @echo`, readers see both the code and its output.

HTML Export

Generate a self-contained HTML report:

```
bl notebook analysis.bln --export html > report.html
```

The HTML output includes:

- Inline CSS with a dark theme
- BioLang syntax highlighting (keywords, strings, comments, pipes)
- Rendered Markdown prose
- Code output blocks
- No external dependencies – a single standalone file

This is useful for sharing results via email or publishing to a website.

Jupyter Interop

Import from Jupyter

Convert an existing `.ipynb` notebook to `.bln`:

```
bl notebook experiment.ipynb --from-ipynb > experiment.bln
```

- Markdown cells become prose sections
- Code cells become fenced `biolang` blocks
- Outputs are discarded (they'll be regenerated)

Export to Jupyter

Convert a `.bln` to `.ipynb`:

```
bl notebook analysis.bln --to-ipynb > analysis.ipynb
```

- Prose becomes Markdown cells
- Code blocks become code cells with `"language": "biolang"` metadata
- Uses nbformat v4, compatible with JupyterLab and VS Code

Example: GC Content Report

Here's a complete notebook that analyzes GC content:

````
# GC Content Analysis

This notebook computes per-contig GC content and flags outliers.

## Setup

```biolang
# @hide
let threshold = 2.0
```

## Load Data

```biolang
let seqs = read_fasta("data/sequences.fasta")
print(f"Loaded {len(seqs)} sequences")
```

## Statistics

```biolang
# @echo
let gc_values = seqs |> map(|s| gc_content(s.seq))
let mu = mean(gc_values)
let sigma = stdev(gc_values)
print(f"Mean GC: {mu:.3f} +/- {sigma:.4f}")
```

## Outliers

Contigs more than 2 std devs from the mean may indicate
contamination or horizontal gene transfer.

```biolang
let outliers = seqs
  |> filter(|s| abs(gc_content(s.seq) - mu) > threshold * sigma)
print(f"Found {len(outliers)} outlier contigs")
```
````

Run it:

```
bl notebook gc_analysis.bln
```

Export it:

```
bl notebook gc_analysis.bln --export html > gc_report.html
```

Tips

- **Use `# @hide` for setup** – configuration, imports, and helper functions that readers don't need to see
- **Use `# @echo` for key steps** – shows both code and output for the important analysis logic
- **Use `# @skip` during development** – temporarily disable expensive cells
- **The HTML export is self-contained** – share the `.html` file directly
- **Convert existing Jupyter notebooks** with `--from-ipynb` to migrate analyses

Chapter 15: Plugins

BioLang ships with a rich set of built-in functions, but bioinformatics is vast. You may need to wrap a niche aligner, call an R statistical package, or connect to a lab-specific LIMS API. The plugin system lets you extend BioLang with external tools written in Python, R, TypeScript/Deno, or native binaries, all communicating over a simple JSON protocol on stdin/stdout.

Plugin Architecture

A BioLang plugin is a standalone program that:

1. Reads JSON requests from stdin
2. Processes the request
3. Writes JSON responses to stdout

BioLang manages the plugin lifecycle: it starts the subprocess when the plugin is first called, keeps it alive for subsequent calls, and shuts it down when the script ends. This avoids process startup overhead on repeated invocations.

The Plugin Manifest

Every plugin has a `plugin.json` file that describes its interface.

```
{
  "name": "blast-wrapper",
  "version": "1.0.0",
  "description": "BLAST+ sequence search wrapper",
  "kind": "python",
  "entry": "blast_plugin.py",
  "functions": [
    {
      "name": "blastn",
      "description": "Nucleotide BLAST search",
      "params": {
        "query": {"type": "string", "description": "Path to query FASTA"},
        "db": {"type": "string", "description": "BLAST database name or
path"},
        "evalue": {"type": "number", "default": 1e-5, "description": "E-value
threshold"},
        "max_hits": {"type": "integer", "default": 10, "description":
"Maximum hits to return"},
        "threads": {"type": "integer", "default": 4, "description": "CPU
threads"}
      },
      "returns": {
        "type": "array",
        "items": {
          "type": "object",
          "properties": {
            "subject": "string",
            "identity": "number",
            "evalue": "number",
            "bitscore": "number",
            "alignment_length": "integer"
          }
        }
      }
    },
    {
      "name": "blastp",
      "description": "Protein BLAST search",
      "params": {
        "query": {"type": "string", "description": "Path to query FASTA"},
        "db": {"type": "string", "description": "BLAST database name or
path"},
        "evalue": {"type": "number", "default": 1e-5},
        "max_hits": {"type": "integer", "default": 10}
      },
      "returns": {
        "type": "array"
      }
    },
    {
      "name": "makeblastdb",
      "description": "Create a BLAST database from a FASTA file",
      "params": {
        "input": {"type": "string", "description": "Input FASTA path"},
        "dbtype": {"type": "string", "enum": ["nucl", "prot"], "description":
"Sequence type"},
        "out": {"type": "string", "description": "Output database prefix"}
      },
      "returns": {
        "type": "object"
      }
    }
  ]
}
```

The `kind` field tells BioLang how to launch the plugin:

Kind	Interpreter
`python`	`python3` (or `python` on Windows)
`r`	`Rscript`
`deno`	`deno run`
`typescript`	`deno run` (same as deno)
`native`	Direct executable

Communication Protocol

BioLang sends JSON messages to the plugin's stdin and reads JSON responses from stdout. Each message is a single line of JSON terminated by a newline.

Request Format

```
{"id": "req_001", "function": "blastn", "params": {"query": "input.fa", "db":
"nt", "evalue": 1e-10}}
```

Response Format

Success:

```
{"id": "req_001", "result": [{"subject": "NM_000546.6", "identity": 99.5,
"evalue": 0.0, "bitscore": 1842}]}
```

Error:

```
{"id": "req_001", "error": {"code": "BLAST_FAILED", "message": "Database nt
not found"}}
```

Lifecycle Messages

BioLang sends a shutdown message when the script ends:

```
{"id": "shutdown", "function": "__shutdown__", "params": {}}
```

The plugin should clean up resources and exit.

Installing Plugins

From a Directory

```
bl add blast-wrapper --path ./my-plugins/blast-wrapper
```

This copies the local plugin into `~/.biolang/plugins/blast-wrapper/`. The current `bl add` command accepts a local path; clone a Git repository first, then pass its directory with `--path`.

Removing Plugins

```
bl remove blast-wrapper
```

Listing Installed Plugins

From the command line:

```
bl plugins
```

From the REPL:

```
:plugins
```

Both show the plugin name, version, kind, and available functions.

Using Plugins in Scripts

Once installed, import a plugin by name and call its functions.

```
# Conceptual or diagnostic example; not directly executable.
import "blast-wrapper"

# Build a custom database
makeblastdb(input: "my_sequences.fa", dbtype: "nucl", out: "custom_db")

# Search against it
let hits = blastn(query: "query.fa", db: "custom_db",
                  evalue: 1e-20, max_hits: 5)

hits |> each(|h| print(h.subject + " identity=" + str(h.identity)
                       + "% evalue=" + str(h.evalue)))
```

Plugin functions integrate seamlessly with pipes and other BioLang features.

#305 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "blast-wrapper"

read_fasta("data/sequences.fasta")
  |> filter(|r| len(r.seq) > kb(1))
  |> write_fasta("long_contigs.fa")

blastn(query: "long_contigs.fa", db: "nt", evalue: 1e-30, threads: 8)
  |> filter(|h| h.identity > 95.0)
  |> sort_by(|h| -h.bitscore)
  |> write_tsv("blast_hits.tsv")
```

Example: Python BLAST Wrapper Plugin

Here is the complete Python implementation for the `blast-wrapper` plugin described above.

Directory Structure

```
blast-wrapper/
  plugin.json        # manifest (shown earlier)
  blast_plugin.py    # implementation
```

blast_plugin.py

```
import json
import subprocess
import sys
import csv
from io import StringIO


def blastn(params):
    cmd = [
        "blastn",
        "-query", params["query"],
        "-db", params["db"],
        "-evalue", str(params.get("evalue", 1e-5)),
        "-max_target_seqs", str(params.get("max_hits", 10)),
        "-num_threads", str(params.get("threads", 4)),
        "-outfmt", "6 sseqid pident evalue bitscore length",
    ]
    result = subprocess.run(cmd, capture_output=True, text=True)
    if result.returncode != 0:
        return {"error": {"code": "BLAST_FAILED", "message":
result.stderr.strip()}}

    hits = []
    reader = csv.reader(StringIO(result.stdout), delimiter="\t")
    for row in reader:
        hits.append({
            "subject": row[0],
            "identity": float(row[1]),
            "evalue": float(row[2]),
            "bitscore": float(row[3]),
            "alignment_length": int(row[4]),
        })
    return {"result": hits}


def blastp(params):
    cmd = [
        "blastp",
        "-query", params["query"],
        "-db", params["db"],
        "-evalue", str(params.get("evalue", 1e-5)),
        "-max_target_seqs", str(params.get("max_hits", 10)),
        "-outfmt", "6 sseqid pident evalue bitscore length",
    ]
    result = subprocess.run(cmd, capture_output=True, text=True)
    if result.returncode != 0:
        return {"error": {"code": "BLAST_FAILED", "message":
result.stderr.strip()}}

    hits = []
    reader = csv.reader(StringIO(result.stdout), delimiter="\t")
    for row in reader:
        hits.append({
            "subject": row[0],
            "identity": float(row[1]),
            "evalue": float(row[2]),
            "bitscore": float(row[3]),
            "alignment_length": int(row[4]),
        })
    return {"result": hits}
```

```
def makeblastdb(params):
    cmd = [
        "makeblastdb",
        "-in", params["input"],
        "-dbtype", params["dbtype"],
        "-out", params["out"],
    ]
    result = subprocess.run(cmd, capture_output=True, text=True)
    if result.returncode != 0:
        return {"error": {"code": "MAKEBLASTDB_FAILED", "message":
result.stderr.strip()}}
    return {"result": {"database": params["out"], "status": "created"}}


DISPATCH = {
    "blastn": blastn,
    "blastp": blastp,
    "makeblastdb": makeblastdb,
}


def main():
    for line in sys.stdin:
        line = line.strip()
        if not line:
            continue
        request = json.loads(line)

        func_name = request["function"]
        if func_name == "__shutdown__":
            break

        handler = DISPATCH.get(func_name)
        if handler is None:
            response = {"id": request["id"],
                        "error": {"code": "UNKNOWN_FUNCTION",
                                  "message": f"No function: {func_name}"}}
        else:
            resp = handler(request["params"])
            response = {"id": request["id"], **resp}

        sys.stdout.write(json.dumps(response) + "\n")
        sys.stdout.flush()


if __name__ == "__main__":
    main()
```

Install and use:

```
# Conceptual or diagnostic example; not directly executable.
bl add blast-wrapper --path ./blast-wrapper

# In a BioLang script:
import "blast-wrapper"

let hits = blastn(query: "primers.fa", db: "refseq_genomic",
                  evalue: 1e-5, max_hits: 20)
hits |> filter(|h| h.identity == 100.0)
    |> each(|h| print("Perfect match: " + h.subject))
```

Example: R DESeq2 Plugin for Differential Expression

Directory Structure

```
deseq2-plugin/
  plugin.json
  deseq2_plugin.R
```

plugin.json

```
{
  "name": "deseq2",
  "version": "1.0.0",
  "description": "DESeq2 differential expression analysis",
  "kind": "r",
  "entry": "deseq2_plugin.R",
  "functions": [
    {
      "name": "deseq2_de",
      "description": "Run DESeq2 differential expression on a count matrix",
      "params": {
        "counts_file": {"type": "string", "description": "Path to raw counts
TSV (genes x samples)"},
        "metadata_file": {"type": "string", "description": "Path to sample
metadata TSV"},
        "design": {"type": "string", "default": "~ condition", "description":
"DESeq2 design formula"},
        "contrast": {"type": "array", "description": "Contrast vector, e.g.
['condition', 'treated', 'control']"},
        "alpha": {"type": "number", "default": 0.05, "description": "Adjusted
p-value threshold"},
        "lfc_threshold": {"type": "number", "default": 0.0, "description":
"Log2 fold change threshold for lfcShrink"}
      },
      "returns": {
        "type": "object",
        "properties": {
          "results_file": "string",
          "n_significant": "integer",
          "n_up": "integer",
          "n_down": "integer"
        }
      }
    },
    {
      "name": "deseq2_normalize",
      "description": "Return DESeq2-normalized counts",
      "params": {
        "counts_file": {"type": "string"},
        "metadata_file": {"type": "string"},
        "design": {"type": "string", "default": "~ condition"}
      },
      "returns": {
        "type": "object",
        "properties": {
          "normalized_file": "string"
        }
      }
    }
  ]
}
```

deseq2_plugin.R

```
library(jsonlite)
library(DESeq2)

deseq2_de <- function(params) {
  counts <- read.delim(params$counts_file, row.names = 1)
  metadata <- read.delim(params$metadata_file, row.names = 1)

  # Ensure sample order matches
  metadata <- metadata[columns(counts), , drop = FALSE]

  dds <- DESeqDataSetFromMatrix(
    countData = counts,
    colData = metadata,
    design = as.formula(params$design)
  )

  dds <- DESeq(dds)

  contrast <- params$contrast
  res <- results(dds, contrast = contrast, alpha = params$alpha)

  if (params$lfc_threshold > 0) {
    res <- lfcShrink(dds, contrast = contrast, res = res, type = "ashr")
  }

  res_df <- as.data.frame(res)
  res_df$gene <- rownames(res_df)
  out_file <- sub("\\.tsv$", "_deseq2_results.tsv", params$counts_file)
  write.table(res_df, out_file, sep = "\t", quote = FALSE, row.names = FALSE)

  sig <- res_df[!is.na(res_df$padj) & res_df$padj < params$alpha, ]

  list(
    result = list(
      results_file = out_file,
      n_significant = len(sig),
      n_up = sum(sig$log2FoldChange > 0),
      n_down = sum(sig$log2FoldChange < 0)
    )
  )
}

deseq2_normalize <- function(params) {
  counts <- read.delim(params$counts_file, row.names = 1)
  metadata <- read.delim(params$metadata_file, row.names = 1)
  metadata <- metadata[columns(counts), , drop = FALSE]

  dds <- DESeqDataSetFromMatrix(
    countData = counts,
    colData = metadata,
    design = as.formula(params$design)
  )
  dds <- estimateSizeFactors(dds)
  norm_counts <- counts(dds, normalized = TRUE)

  out_file <- sub("\\.tsv$", "_normalized.tsv", params$counts_file)
  write.table(as.data.frame(norm_counts), out_file, sep = "\t", quote =
FALSE)

  list(result = list(normalized_file = out_file))
}
```

```
dispatch <- list(
  deseq2_de = deseq2_de,
  deseq2_normalize = deseq2_normalize
)

# Main loop: read JSON from stdin, dispatch, write JSON to stdout
con <- file("stdin", "r")
while (TRUE) {
  line <- readLines(con, n = 1)
  if (length(line) == 0) break

  request <- fromJSON(line)
  if (request$`function` == "__shutdown__") break

  handler <- dispatch[[request$`function`]]
  if (is.null(handler)) {
    response <- list(
      id = request$id,
      error = list(code = "UNKNOWN_FUNCTION",
                   message = paste("No function:", request$`function`))
    )
  } else {
    resp <- tryCatch(
      handler(request$params),
      error = function(e) list(error = list(code = "R_ERROR", message =
e$message))
    )
    response <- c(list(id = request$id), resp)
  }

  cat(toJSON(response, auto_unbox = TRUE), "\n")
  flush(stdout())
}
close(con)
```

Using the DESeq2 Plugin

```
# Conceptual or diagnostic example; not directly executable.
# requires: internet connection (for ncbi_gene calls below)
# requires: NCBI_API_KEY (optional, increases rate limit)
bl add deseq2 --path ./deseq2-plugin

# In a BioLang script:
import "deseq2"

# Run differential expression
let de = deseq2_de(
  counts_file: "raw_counts.tsv",
  metadata_file: "sample_metadata.tsv",
  design: "~ condition",
  contrast: ["condition", "treated", "control"],
  alpha: 0.05,
  lfc_threshold: 1.0
)

print(str(de.n_significant) + " DE genes ("
      + str(de.n_up) + " up, " + str(de.n_down) + " down)")

# Read the results back into BioLang for further analysis
let results = tsv(de.results_file)
  |> filter(|r| r.padj != nil and r.padj < 0.01)
  |> sort_by(|r| r.padj)

# Cross-reference top hits with bio APIs
results |> take(10) |> each(|r| {
  let gene_info = ncbi_gene(r.gene)
  print(r.gene + " log2FC=" + str(r.log2FoldChange)
        + " padj=" + str(r.padj)
        + " — " + gene_info.name)
})

# Get normalized counts for downstream analysis
let norm = deseq2_normalize(
  counts_file: "raw_counts.tsv",
  metadata_file: "sample_metadata.tsv"
)
print("Normalized counts: " + norm.normalized_file)
```

Writing a TypeScript/Deno Plugin

For plugins that connect to REST APIs or need async I/O, Deno is a good choice.

lims-connector/plugin.json

```
{
  "name": "lims-connector",
  "version": "1.0.0",
  "description": "Connect to lab LIMS for sample metadata",
  "kind": "deno",
  "entry": "lims_plugin.ts",
  "functions": [
    {
      "name": "lims_samples",
      "description": "Fetch sample metadata from LIMS",
      "params": {
        "project": {"type": "string", "description": "LIMS project ID"},
        "status": {"type": "string", "default": "all", "description": "Filter
by status"}
      },
      "returns": {"type": "array"}
    },
    {
      "name": "lims_submit_results",
      "description": "Push analysis results back to LIMS",
      "params": {
        "sample_id": {"type": "string"},
        "results_file": {"type": "string"},
        "qc_status": {"type": "string", "enum": ["pass", "fail", "warning"]}
      },
      "returns": {"type": "object"}
    }
  ]
}
```

lims_plugin.ts

```
import { readLines } from "https://deno.land/std/io/mod.ts";

const LIMS_URL = Deno.env.get("LIMS_API_URL") ||
"https://lims.example.com/api";
const LIMS_TOKEN = Deno.env.get("LIMS_API_TOKEN") || "";

interface Request {
  id: string;
  function: string;
  params: Record<string, unknown>;
}

async function limsSamples(params: Record<string, unknown>) {
  const project = params.project as string;
  const status = (params.status as string) || "all";

  const url = `${LIMS_URL}/projects/${project}/samples?status=${status}`;
  const resp = await fetch(url, {
    headers: { "Authorization": `Bearer ${LIMS_TOKEN}` },
  });

  if (!resp.ok) {
    return { error: { code: "LIMS_ERROR", message: await resp.text() } };
  }

  const data = await resp.json();
  return { result: data.samples };
}

async function limsSubmitResults(params: Record<string, unknown>) {
  const sampleId = params.sample_id as string;
  const resultsFile = params.results_file as string;
  const qcStatus = params.qc_status as string;

  const results = await Deno.readTextFile(resultsFile);

  const resp = await fetch(`${LIMS_URL}/samples/${sampleId}/results`, {
    method: "POST",
    headers: {
      "Authorization": `Bearer ${LIMS_TOKEN}`,
      "Content-Type": "application/json",
    },
    body: JSON.stringify({ results: JSON.parse(results), qc_status: qcStatus
}),
  });

  if (!resp.ok) {
    return { error: { code: "LIMS_ERROR", message: await resp.text() } };
  }

  return { result: { sample_id: sampleId, status: "submitted" } };
}

const dispatch: Record<string, (p: Record<string, unknown>) =>
Promise<unknown>> = {
  lims_samples: limsSamples,
  lims_submit_results: limsSubmitResults,
};

for await (const line of readLines(Deno.stdin)) {
  if (!line.trim()) continue;
```

```
  const request: Request = JSON.parse(line);
  if (request.function === "__shutdown__") break;

  const handler = dispatch[request.function];
  let response;
  if (!handler) {
    response = {
      id: request.id,
      error: { code: "UNKNOWN_FUNCTION", message: `No function:
${request.function}` },
    };
  } else {
    const resp = await handler(request.params);
    response = { id: request.id, ...resp };
  }

  console.log(JSON.stringify(response));
}
```

Using the LIMS Plugin

#306 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "lims-connector"

# Fetch samples from the LIMS
let samples = lims_samples(project: "WGS-2024-042", status: "sequenced")

# Process each sample
let results = samples |> par_map(|sample| {
  let bam = tool("bwa-mem2", "-t 8 -x sr " + "GRCh38.fa " + sample.fastq_r1 +
" " + sample.fastq_r2)
    |> tool("samtools", "sort -@ 4")
  let stats = tool("samtools", "flagstat " + bam)
  let depth = tool("samtools", "depth -a " + bam) |> mean()

  let qc = if stats.mapped_pct > 90 and depth > 30 then "pass"
           else if stats.mapped_pct > 80 then "warning"
           else "fail"

  let result = {sample_id: sample.id, mapped_pct: stats.mapped_pct,
                mean_depth: depth, bam: bam}
  result |> write_json(sample.id + "_results.json")

  # Push results back to LIMS
  lims_submit_results(sample_id: sample.id,
                      results_file: sample.id + "_results.json",
                      qc_status: qc)
  result
})
```

Plugin Discovery

BioLang searches for plugins in this order:

1. `./plugins/` in the current working directory
2. `~/.biolang/plugins/` in the user home directory
3. Paths listed in the `BIOLANG_PLUGIN_PATH` environment variable

Each directory is scanned for subdirectories containing a `plugin.json` file.

Plugin Best Practices

Keep plugins focused. A plugin should wrap one tool or one API. If you have BLAST and HMMER, make them separate plugins.

Use stderr for logging. BioLang captures stdout for the JSON protocol. Diagnostic messages, progress indicators, and debug output should go to stderr.

Handle errors gracefully. Return error responses instead of crashing. The `error` response format lets BioLang surface meaningful messages to the user.

Document parameters. The `description` fields in `plugin.json` are shown by `:plugins` in the REPL and `bl plugins` on the command line.

Pin versions. If your plugin depends on an external tool, document the required version in the manifest description so users know what to install.

Summary

The plugin system extends BioLang with any language that can read and write JSON on stdio. Write a `plugin.json` manifest, implement the stdin/stdout dispatch loop, and install with `bl add`. Plugins integrate fully with pipes, parallel operations, and the rest of BioLang. Python plugins wrap command-line bioinformatics tools, R plugins bring statistical packages like DESeq2, and Deno/TypeScript plugins connect to REST APIs and external services.

Chapter 18B: Packages, Migration, and CLI Tooling

BioLang's command-line interface covers source validation, package setup, cross-language migration, notebook conversion, editor integration, environment diagnostics, and machine-readable API discovery.

Check Before You Run

`bl check` parses one or more scripts without executing them:

```
bl check main.bl src/qc.bl src/report.bl
```

Use it in continuous integration and before running a workflow against large inputs. `bl run --verbose` shows execution steps, while `bl run --events` emits versioned JSON Lines events for desktop clients and automation:

```
bl run --verbose main.bl
bl run --events main.bl
```

Packages

A package is BioLang source code described by `biolang.toml`. Initialize the current directory:

```
mkdir sequence-qc
cd sequence-qc
bl init --name sequence-qc
```

The command creates:

```
sequence-qc/
  biolang.toml
  main.bl
```

Dependencies can be local paths or Git repositories:

```
[package]
name = "sequence-qc"
version = "0.1.0"

[dependencies]
shared = { path = "../shared" }
variants = { git = "https://github.com/example/variants.git", branch = "main"
}
```

Run `bl install` to install manifest dependencies. A specific local package or Git package can also be installed directly:

```
bl install ../shared
bl install variants --git https://github.com/example/variants.git --branch
main
```

Packages may bundle runnable examples. They remain available after installation, without retaining the package source repository:

```
bl examples variants
bl examples variants --copy variants-examples
```

The first command lists the installed package's example files. The second copies the complete tree, including nested data or validation files, into a new or empty working directory. During package development, the command also accepts a local package directory in place of the installed name.

Version-only registry dependencies are recognized in the manifest but the current CLI does not fetch them from a registry.

Packages Versus Plugins

Packages contain BioLang modules and are installed under `~/.biolang/packages/`. Plugins are separate processes that communicate with

BioLang through the plugin JSON protocol and are installed under `~/.biolang/plugins/`.

```
bl add aligner --path ./plugins/aligner
bl plugins
bl remove aligner
```

Use a package for reusable BioLang logic. Use a plugin when an external program, another language runtime, or process isolation is required.

Import Python, R, and Notebooks

`bl import` converts Python, R, Jupyter, and R Markdown sources to BioLang. Always inspect the generated code: conversion preserves intent where possible, but library-specific calls can require manual replacement.

```
bl import analysis.py --validate -o analysis.bl
bl import analysis.R --validate -o analysis.bl
bl import report.ipynb --validate -o report.bl
bl import report.Rmd --validate -o report.bl
```

The source format is inferred from the extension. Use `--from` to override it, or provide a name when reading standard input:

```
bl import legacy.txt --from python --validate -o legacy.bl
python generate.py | bl import - --from python --name generated.py --validate
```

`--validate` reports remaining BioLang diagnostics and exits nonzero when the conversion still needs attention. `--json` returns the converted content and validation result as structured JSON for editor integrations.

Notebook Interchange

BioLang runs `.bln`, `.bl.md`, and `.ipynb` notebooks:

```
bl notebook analysis.bln
bl notebook analysis.bl.md
```

Convert between BioLang and Jupyter notebooks or export a self-contained HTML report:

```
bl notebook study.ipynb --from-ipynb > study.bln
bl notebook study.bln --to-ipynb > study.ipynb
bl notebook study.bln --export html > study.html
```

Notebook cells share one session and execute in document order.

Environment and Editor Integration

Run the environment doctor before relying on native tools, containers, or network capabilities:

```
bl doctor
```

Editors start the language server with `bl lsp`. Tooling that needs completion, signature, and builtin documentation can consume the same structured metadata:

```
bl metadata --format json > biolang-metadata.json
```

Use `bl version` to show the installed release and check for updates, then `bl upgrade` to install the latest available release.

Reproducible Migration Checklist

1. Convert with `bl import --validate`.
2. Replace Python or R library calls that have no direct BioLang equivalent.
3. Confirm file paths and generate or document every expected input.
4. Run `bl check` over all generated `.bl` files.
5. Compare key statistics against the original Python or R workflow.
6. Record BioLang and package versions with the analysis outputs.

Benchmarks & Correctness

BioLang is benchmarked against Python (BioPython) and R (Bioconductor) on 30 bioinformatics tasks spanning sequence I/O, k-mer analysis, interval overlaps, variant processing, and multi-step pipelines. All results are reproducible from the `benchmarks/` directory.

Test Environment

Linux (WSL2)

- Intel Core i9-12900K, 16 GB RAM
- BioLang 0.3.0, Python 3.12.3, R 4.3.3

Windows 11

- Intel Core i9-12900K, 32 GB RAM
- BioLang 0.3.0, Python 3.12.4

Results Summary

Where BioLang Wins

BioLang's Rust I/O engine (noodles) and native 2-bit DNA encoding deliver the biggest gains on:

Task	BioLang	Python	Speedup
ENCODE Peak Overlap	0.363s	2.574s	**7.1x**
Protein K-mers	0.191s	1.331s	**7.0x**
FASTA Parse (30 KB)	0.138s	0.926s	**6.7x**
FASTA gzipped	0.141s	0.930s	**6.6x**
E. coli Genome	0.176s	1.081s	**6.1x**

Task	BioLang	Python	Speedup
GC Content (51 MB)	0.830s	2.771s	**3.3x**
K-mer Counting (21-mers)	6.551s	21.01s	**3.2x**
FASTQ QC Pipeline	2.349s	5.059s	**2.2x**

Where Python Wins

Python's `csv`, `re`, and `dict` modules are highly optimized C extensions. On text-heavy parsing tasks:

Task	BioLang	Python	Result
VCF Filtering	0.349s	0.166s	Py 2.1x faster
ClinVar Variants	0.661s	0.265s	Py 2.5x faster
CSV Join + Group-by	0.281s	0.156s	Py 1.8x faster
GFF3 Ensembl chr22	0.453s	0.171s	Py 2.6x faster

Windows Notes

Windows process creation adds ~1s overhead per invocation, compressing speedup ratios for sub-second tasks. For accurate algorithmic comparison, refer to Linux results. CPU-bound tasks that exceed this floor (k-mer counting 3.0x, ENCODE overlap 2.9x, QC pipeline 2.5x) still show clear wins.

Code Conciseness

BioLang scripts average 50-70% fewer lines of code than equivalent Python for the same analysis task. This comes from pipe-first syntax, built-in bio types, and higher-order functions on streams.

Correctness Validation

Performance without correctness is meaningless. BioLang includes two correctness validation suites — synthetic and real-world — that compare outputs

against Python (BioPython) and R (Bioconductor) as independent gold standards.

Synthetic Data Validation

Uses generated test data with controlled inputs for deterministic comparison:

Task	What it checks	Tolerance	R
`gc_content`	GC% per sequence from FASTA	float ±1e-6	yes
`kmer_count`	Canonical 5-mer counts from DNA	exact integer	–
`vcf_filter`	Filter VCF by QUAL>=30, count per chrom	exact integer	yes
`reverse_complement`	Reverse complement of DNA sequences	exact string	yes
`translate`	DNA→protein translation	exact string	yes
`csv_groupby`	Group-by aggregation (count, mean)	float ±1e-6	yes
`gff_features`	Count features by type from GFF	exact integer	yes
`sequence_stats`	N50, total length, GC from FASTA	float ±1e-6	yes
`bed_intervals`	BED parse, span, merge overlapping	exact integer	yes

Real-World Data Validation

Uses actual biological data from NCBI and ClinVar to test edge cases that synthetic data misses — non-standard bases, multi-allelic variants, overlapping bacterial genes, and variable naming conventions:

Task	Real Data Source	Tolerance	R
`gc_content`	S. cerevisiae genome (16 chromosomes)	float ±1e-6	yes
`kmer_count`	E. coli K-12 genome (50 KB)	exact integer	–

Task	Real Data Source	Tolerance	R
`vcf_filter`	ClinVar VCF (5,000 variants, pathogenic filter)	exact integer	yes
`reverse_complement`	S. cerevisiae (5 chroms, 200bp each)	exact string	yes
`translate`	S. cerevisiae (3 chroms, 99bp each)	exact string	yes
`csv_groupby`	ClinVar variants CSV (group by significance)	float ±1e-6	yes
`gff_features`	E. coli K-12 GFF3 annotation	exact integer	yes
`sequence_stats`	S. cerevisiae genome	float ±1e-6	yes
`bed_intervals`	E. coli gene BED (derived from GFF)	exact integer	yes

Real-world data is downloaded automatically via `python download_real_data.py` (~25 MB total from NCBI FTP).

How It Works

Each task has three implementations — BioLang, Python, and R — that compute the same result and output JSON to stdout. A recursive comparator checks:

- **Floats**: ±1e-6 tolerance
- **Integers**: exact match
- **Strings**: exact match
- **Dicts/lists**: recursive key-by-key comparison

Running Validation

```
# Synthetic data validation
cd benchmarks/correctness
./validate.sh [bl_binary] [python_binary] [rscript_binary]

# Real-world data validation
python download_real_data.py
./validate_real.sh [bl_binary] [python_binary] [rscript_binary]

# Windows
.\validate.ps1 [-BL bl] [-PY python] [-RS Rscript]
.\validate_real.ps1 [-BL bl] [-PY python] [-RS Rscript]
```

R tests are skipped automatically if R/Bioconductor is not installed.

Reproducing Benchmarks

```
# Generate synthetic test data
python benchmarks/generate_data.py

# Run all benchmarks (Linux)
cd benchmarks && ./run_all.sh

# Run correctness validation (synthetic)
cd benchmarks/correctness && ./validate.sh

# Run correctness validation (real-world)
python download_real_data.py && ./validate_real.sh
```

Results are saved to `benchmarks/results/latest/{linux,windows}/` with per-category breakdown:

- `language/` — sequence I/O, k-mers, protein, intervals, variants, file I/O, data wrangling
- `pipelines/` — QC pipeline, variant pipeline, annotation, multi-sample, RNA-seq

Methodology

- **Timing**: Best of 3 wall-clock runs
- **Data**: Mix of synthetic (generated) and real-world (NCBI, ClinVar, ENCODE, Ensembl)

- **K-mers**: BioLang uses canonical (strand-agnostic) 21-mers; Python uses forward-only — BioLang does strictly more work
- **Fair comparison**: Same input files, same output format, same machine, cold cache between runs
- **Correctness**: Two independent validation suites (synthetic + real-world) ensure identical biological answers

Appendix A: Builtin Reference

BioLang ships with a comprehensive standard library of builtins designed for bioinformatics workflows. Every function listed here is available without imports – they are part of the language runtime.

Sequence Operations

Builtins that operate on bio-typed sequences (`dna`, `rna`, `protein`).

Builtin	Description
`complement(seq) -> Dna | Rna`	Watson-Crick complement of a nucleotide sequence
`reverse_complement(seq) -> Dna | Rna`	Reverse complement – the opposing strand
`transcribe(seq) -> Rna`	Transcribe DNA to RNA (T to U)
`translate(seq) -> Protein`	Translate an RNA or DNA coding sequence to amino acids
`gc_content(seq) -> Float`	GC fraction of a nucleotide sequence (0.0 – 1.0)
`find_motif(seq, pattern) -> List[Int]`	All start positions where `pattern` occurs in `seq`
`hamming_distance(a, b) -> Int`	Number of mismatched positions between equal-length sequences
`edit_distance(a, b) -> Int`	Edit distance between two sequences
`find_orfs(seq, min_len?) -> List[Record]`	Open reading frames with start, stop, and frame fields
`restriction_sites(seq, enzyme?) -> List[Record]`	Recognition sites for restriction enzymes
`tm(seq) -> Float`	Melting temperature estimate for a short oligonucleotide

Builtin	Description
`slice(seq, start, end) -> Dna | Rna | Protein`	Extract a subsequence by 0-based coordinates

#307 ▶ Run ▶▶ Run All Above + This

```
# Example: quick primer analysis
let primer = dna"ATCGATCGATCG"
let rc     = reverse_complement(primer)
let temp   = tm(primer)
print("Primer Tm = " + str(temp) + "C, reverse complement = " + str(rc))
```

Collection Operations

General-purpose operations on lists, records, and sets.

Builtin	Description
`len(coll) -> Int`	Number of elements in a list, string, or sequence
`push(list, item) -> List`	Append an element, returning a new list
`pop(list) -> List`	Remove the last element, returning a new list
`concat(a, b) -> List`	Concatenate two lists
`flatten(nested) -> List`	Flatten one level of nesting
`reverse(list) -> List`	Reverse element order
`contains(coll, item) -> Bool`	True if `item` is present
`index_of(list, item) -> Int | Nil`	First index of `item`, or nil
`last(list) -> Any`	Last element
`first(list) -> Any`	First element
`head(list, n) -> List`	First `n` elements
`tail(list, n) -> List`	Last `n` elements

Builtin	Description
`unique(list) -> List`	Remove duplicates, preserving order
`zip(a, b) -> List`	Pair elements from two lists into a list of tuples
`enumerate(list) -> List`	Pair each element with its 0-based index
`chunk(list, size) -> List[List]`	Split into fixed-size sublists
`window(list, size) -> List[List]`	Sliding window of the given size
`scan(list, init, fn) -> List`	Running accumulation (like reduce but keeps intermediates)
`range(start, end, step?) -> List`	Integer range
`set(list) -> Set`	Convert a list to a deduplicated set
`keys(record) -> List[Str]`	Field names of a record
`values(record) -> List`	Field values of a record
`has_key(record, key) -> Bool`	True if the record contains the named field
`sort_by(list, fn) -> List`	Sort by a key function
`group_by(list, fn) -> Record`	Group elements by a key function into a record of lists
`partition(list, fn) -> [List, List]`	Split into elements that pass and fail a predicate

#308 ▶ Run ▶▶ Run All Above + This

```
# Example: enumerate quality-filtered reads
let good_reads = read_fastq("data/reads.fastq")
  |> filter(|r| mean_phred(r.quality) > 30)
  |> enumerate()
  |> head(5)
```

Higher-Order Functions

Functions that accept other functions as arguments – the backbone of BioLang's pipeline style.

Builtin	Description
`map(coll, fn) -> List`	Apply `fn` to every element
`filter(coll, fn) -> List`	Keep elements where `fn` returns true
`reduce(coll, init, fn) -> Any`	Fold elements into a single value
`sort(coll, fn?) -> List`	Sort, optionally by comparator
`each(coll, fn) -> Nil`	Execute `fn` for side effects on every element
`flat_map(coll, fn) -> List`	Map then flatten one level
`take_while(coll, fn) -> List`	Take leading elements while predicate holds
`any(coll, fn) -> Bool`	True if `fn` returns true for at least one element
`all(coll, fn) -> Bool`	True if `fn` returns true for every element
`none(coll, fn) -> Bool`	True if `fn` returns true for no elements
`find(coll, fn) -> Any | Nil`	First element satisfying `fn`
`find_index(coll, fn) -> Int | Nil`	Index of first element satisfying `fn`
`par_map(coll, fn) -> List`	Parallel map across available cores
`par_filter(coll, fn) -> List`	Parallel filter across available cores
`mat_map(matrix, fn) -> Matrix`	Apply `fn` element-wise to a matrix
`try_call(fn, args) -> Result`	Call `fn` with `args`, capturing errors instead of panicking

#309 ▶ Run ▶▶ Run All Above + This

```
# Example: parallel GC content across a genome's chromosomes
let chromosomes = [
  {name: "chrA", seq: dna"ATGCGCGTAA"},
  {name: "chrB", seq: dna"ATATATGCAT"}
]
let gc_values = chromosomes
  |> par_map(|chr| {name: chr.name, gc: gc_content(chr.seq)})
  |> sort_by(|r| r.gc)
```

Table Operations

Tabular data manipulation inspired by dataframe semantics – designed for sample sheets, variant tables, and expression matrices.

Builtin	Description
`table(data) -> Table`	Create a table from a list of records
`select(tbl, ...cols) -> Table`	Pick columns by name
`mutate(tbl, name, fn) -> Table`	Add or transform a column
`summarize(grouped, |key, rows| {...}) -> Table`	Aggregate grouped data via closure
`group_by(tbl, col) -> GroupedTable`	Group rows by a column value (table variant)
`inner_join(a, b, on) -> Table`	Keep rows whose key occurs in both tables
`left_join(a, b, on) -> Table`	Keep every left row and attach matching right columns
`csv(path) -> Table`	Read a CSV file into a table
`tsv(path) -> Table`	Read a TSV file into a table
`write_tsv(tbl, path) -> Nil`	Write a table to CSV
`write_tsv(tbl, path) -> Nil`	Write a table to TSV
`len(tbl) -> Int`	Number of rows
`ncols(tbl) -> Int`	Number of columns
`columns(tbl) -> List[Str]`	Column name list

Builtin	Description
`row_names(tbl) -> List[Str]`	Row name list (if set)

#310 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# Example: summarize variant counts per chromosome
tsv("variants.tsv")
  |> group_by("chrom")
  |> summarize(|chrom, rows| {count: len(rows), mean_qual: mean(col(rows,
"quality"))})
  |> write_tsv("chrom_summary.csv")
```

Bio File I/O

Read and write standard bioinformatics file formats. Readers return lazy streams that integrate with pipes; writers flush on completion.

Builtin	Description
`read_fasta(path) -> Table`	Parse FASTA; columns are `id`, `description`, `seq`, and `length`
`read_fastq(path) -> Table`	Parse FASTQ; columns are `id`, `description`, `seq`, `length`, and `quality`
`read_vcf(path) -> Table`	Parse VCF; columns include `chrom`, `pos`, `ref`, `alt`, `qual`, and `info`
`read_bed(path) -> Table`	Parse BED; columns include `chrom`, `start`, and `end`
`read_gff(path) -> Table`	Parse GFF/GTF; columns include `seqid`, `type`, `start`, `end`, and raw `attributes` text
`write_fasta(records, path) -> Nil`	Write records to FASTA format
`write_fastq(records, path) -> Nil`	Write records to FASTQ format
`write_bed(records, path) -> Nil`	Write records to BED format

#311 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
# Example: filter FASTQ reads by quality and write survivors
read_fastq("data/reads.fastq")
  |> filter(|r| mean_phred(r.quality) > 30)
  |> write_fastq("sample_R1.filtered.fq")
```

Genomic Intervals

Interval arithmetic for coordinate-based genomic analysis. Intervals carry `chrom`, `start`, `end`, and optional `strand` and `data` fields.

Builtin	Description
`interval(chrom, start, end, strand?) -> Interval`	Create a genomic interval
`interval_tree(intervals) -> IntervalTree`	Build an index for fast overlap queries
`query_overlaps(tree, chrom, start, end) -> Table`	Rows overlapping a half-open query range
`count_overlaps(tree, chrom, start, end) -> Int`	Number of rows overlapping a query range
`coverage(tree) -> Table`	Coverage segments with `chrom`, `start`, `end`, and `depth`
`merge_intervals(intervals, dist?) -> List[Interval]`	Merge overlapping or nearby intervals
`intersect(a, b) -> List[Interval]`	Pairwise intersection of two interval sets
`subtract(a, b) -> List[Interval]`	Regions in `a` not covered by `b`

#312 ▶ Run ▶▶ Run All Above + This

```
# Example: find promoter-peak overlaps
let promoters = read_bed("data/regions.bed") |> map(|r| interval(r.chrom,
r.start, r.end))
let peaks     = read_bed("data/exons.bed") |> map(|r| interval(r.chrom,
r.start, r.end))
let tree      = interval_tree(peaks)
let hits      = promoters |> flat_map(|p| query_overlaps(tree, p.chrom,
p.start, p.end))
print("Found " + str(len(hits)) + " promoter-peak overlaps")
```

Variants

Builtins for working with genetic variant records. Variant objects carry `chrom`, `pos`, `ref`, `alt`, `qual`, and `info` fields.

Builtin	Description
`variant(chrom, pos, ref, alt) -> Variant`	Construct a variant record
`is_snp(v) -> Bool`	True if single-nucleotide polymorphism
`is_indel(v) -> Bool`	True if insertion or deletion
`is_transition(v) -> Bool`	True if purine-purine or pyrimidine-pyrimidine substitution
`is_transversion(v) -> Bool`	True if purine-pyrimidine substitution
`variant_type(v) -> Str`	Classification string: “snp”, “ins”, “del”, “mnv”, “complex”
`is_het(v) -> Bool`	True if heterozygous genotype
`is_hom_ref(v) -> Bool`	True if homozygous reference
`is_hom_alt(v) -> Bool`	True if homozygous alternate
`is_multiallelic(v) -> Bool`	True if more than one alt allele
`parse_vcf_info(info_str) -> Record`	Parse a VCF INFO field string into a record
`variant_summary(variants) -> Record`	Aggregate counts of SNPs, indels, Ti/Tv ratio, het/hom ratio

#313 ▶ Run ▶▶ Run All Above + This

```
# Example: compute Ti/Tv ratio for a VCF
let vars = vcf_filter(vcf_parse(read_text("data/variants.vcf")), 30)
let summary = variant_summary(vars)
let snp_rows = summary |> filter(|row| row.type_ == "SNP")
let snp_count = if len(snp_rows) > 0 { snp_rows[0].count } else { 0 }
print(summary)
print("Ti/Tv = " + str(titv_ratio(vars)) + ", SNPs = " + str(snp_count))
```

Statistics

Statistical functions for quality control, expression analysis, and hypothesis testing.

Builtin	Description
`mean(xs) -> Float`	Arithmetic mean
`median(xs) -> Float`	Median value
`stdev(xs) -> Float`	Sample standard deviation
`variance(xs) -> Float`	Sample variance
`quantile(xs, q) -> Float`	Quantile at fraction `q` (0.0 – 1.0)
`min(xs) -> Num`	Minimum value
`max(xs) -> Num`	Maximum value
`sum(xs) -> Num`	Sum of all elements
`cor(xs, ys) -> Float`	Pearson correlation coefficient
`ttest(xs, ys) -> Record`	Two-sample t-test; returns `{statistic, pvalue}`
`chi_square(observed, expected) -> Record`	Chi-squared test; returns `{statistic, pvalue, df}`
`wilcoxon(xs, ys) -> Record`	Wilcoxon rank-sum test
`anova(groups) -> Record`	One-way ANOVA across groups
`fisher_exact(a, b, c, d) -> Record`	Fisher's exact test on a 2x2 contingency table
`p_adjust(pvals, method?) -> List[Float]`	Multiple testing correction (default: Benjamini-Hochberg)
`lm(xs, ys) -> Record`	Simple linear regression; returns `{slope, intercept, r_squared}`
`ks_test(xs, ys) -> Record`	Kolmogorov-Smirnov test
`mean_phred(quals) -> Float`	Mean Phred quality score from a quality string

#314 ▶ Run ▶▶ Run All Above + This

```
# Example: differential expression significance
let control   = [5.2, 4.8, 5.1, 4.9]
let treatment = [8.1, 7.5, 8.3, 7.9]
let result    = ttest(control, treatment)
print("p-value = " + str(result.pvalue))
```

Linear Algebra

Matrix operations for expression matrices, PCA, distance calculations, and numerical biology.

Builtin	Description
`matrix(data) -> Matrix`	Create a matrix from a list of lists (row-major)
`transpose(m) -> Matrix`	Transpose rows and columns
`mat_mul(a, b) -> Matrix`	Matrix multiplication
`determinant(m) -> Float`	Determinant of a square matrix
`inverse(m) -> Matrix`	Matrix inverse
`eigenvalues(m) -> List[Float]`	Eigenvalues of a square matrix
`svd(m) -> Record`	Singular value decomposition; returns `{u, s, vt}`
`solve(a, b) -> Matrix`	Solve the linear system Ax = b
`trace(m) -> Float`	Sum of diagonal elements
`norm(m, p?) -> Float`	Matrix or vector norm (default: Frobenius / L2)
`rank(m) -> Int`	Numerical rank
`eye(n) -> Matrix`	n x n identity matrix
`zeros(rows, cols) -> Matrix`	Matrix of zeros
`ones(rows, cols) -> Matrix`	Matrix of ones
`diag(values) -> Matrix`	Diagonal matrix from a list of values

Builtin	Description
`mat_map(m, fn) -> Matrix`	Apply `fn` element-wise

#315 ▶ Run ▶▶ Run All Above + This

```
# Example: PCA on a gene expression matrix
let expr = tsv("examples/sample-data/counts.tsv") |> table()
let m    = matrix(expr |> select("gene_a", "gene_b", "gene_c"))
let decomp = svd(m)
print("Top 3 singular values: " + str(head(decomp.s, 3)))
```

Math

Standard mathematical functions available for scoring, normalization, and modeling.

Builtin	Description
`abs(x) -> Num`	Absolute value
`ceil(x) -> Int`	Round up to nearest integer
`floor(x) -> Int`	Round down to nearest integer
`round(x, digits?) -> Float`	Round to `digits` decimal places (default: 0)
`sqrt(x) -> Float`	Square root
`log(x) -> Float`	Natural logarithm
`log2(x) -> Float`	Base-2 logarithm (common in fold-change analysis)
`log10(x) -> Float`	Base-10 logarithm
`exp(x) -> Float`	Euler's number raised to `x`
`pow(base, exp) -> Float`	Exponentiation
`sin(x) -> Float`	Sine
`cos(x) -> Float`	Cosine
`tan(x) -> Float`	Tangent

Builtin	Description
`ode_solve(fn, y0, t_span, dt?) -> List[Record]`	Numerical ODE integration (Runge-Kutta)

#316 ▶ Run ▶▶ Run All Above + This

```
# Example: log2 fold-change between conditions
let control   = 12.5
let treatment = 50.0
let lfc = log2(treatment / control)
print("Log2 fold-change = " + str(lfc))
```

String Operations

Text manipulation for parsing identifiers, annotations, and formatted output.

Builtin	Description
`split(s, delim) -> List[Str]`	Split string on delimiter
`join(list, delim) -> Str`	Join list elements into a string
`trim(s) -> Str`	Strip leading and trailing whitespace
`upper(s) -> Str`	Convert to uppercase
`lower(s) -> Str`	Convert to lowercase
`starts_with(s, prefix) -> Bool`	True if `s` begins with `prefix`
`ends_with(s, suffix) -> Bool`	True if `s` ends with `suffix`
`replace(s, from, to) -> Str`	Replace all occurrences
`regex_match(s, pattern) -> Bool`	True if regex `pattern` matches
`format(template, ...args) -> Str`	Printf-style formatting

BioLang also supports **f-strings** for inline interpolation:

#317 ▶ Run ▶▶ Run All Above + This

```
# Example: parse a FASTA header
let header = ">sp|P12345|MYG_HUMAN Myoglobin OS=Homo sapiens"
let parts  = split(header, "|")
let acc    = parts[1]
print("Accession: " + acc)
```

Type Operations

Runtime type inspection and conversion – useful for dynamic dispatch in pipelines that handle mixed bio types.

Builtin	Description
`type(val) -> Str`	Runtime type name as a string
`is_dna(val) -> Bool`	True if val is a DNA sequence
`is_rna(val) -> Bool`	True if val is an RNA sequence
`is_protein(val) -> Bool`	True if val is a protein sequence
`is_interval(val) -> Bool`	True if val is a genomic interval
`is_variant(val) -> Bool`	True if val is a variant record
`is_record(val) -> Bool`	True if val is a record
`is_list(val) -> Bool`	True if val is a list
`is_table(val) -> Bool`	True if val is a table
`is_nil(val) -> Bool`	True if val is nil
`is_int(val) -> Bool`	True if val is an integer
`is_float(val) -> Bool`	True if val is a float
`is_str(val) -> Bool`	True if val is a string
`is_bool(val) -> Bool`	True if val is a boolean
`into(val, target_type) -> Any`	Convert between compatible types

#318 ▶ Run ▶▶ Run All Above + This

```
# Example: route processing based on sequence type
let seq = read_fasta("data/sequences.fasta") |> first() |> |r| r.seq
if is_dna(seq) then
  print("DNA, GC = " + str(gc_content(seq)))
else if is_protein(seq) then
  print("Protein, length = " + str(len(seq)))
```

Bio APIs

Remote database queries for annotation enrichment. All API builtins are async-aware and return structured records.

Builtin	Description
`ncbi_search(db, query, max?) -> List[Str]`	Search NCBI Entrez databases (returns ID list)
`ncbi_gene(symbol, max?) -> Record or List[Str]`	Gene lookup: Record if single match, else ID list
`ncbi_sequence(acc) -> Str`	Fetch sequence by accession as FASTA text
`ensembl_gene(ensembl_id) -> Record`	Ensembl gene lookup by Ensembl ID
`ensembl_symbol(species, symbol) -> Record`	Ensembl gene lookup by species and symbol
`ensembl_vep(variants) -> List[Record]`	Variant Effect Predictor annotation
`uniprot_search(query, max?) -> List[Record]`	Search UniProt by keyword or accession
`uniprot_entry(acc) -> Record`	Full UniProt entry
`ucsc_sequence(genome, chrom, start, end) -> Dna`	Fetch genomic sequence from UCSC DAS
`kegg_get(entry) -> Record`	Retrieve a KEGG database entry
`kegg_find(db, query) -> List[Record]`	Search KEGG databases

Builtin	Description
`string_network(proteins, species) -> List[Record]`	STRING interactions: {protein_a, protein_b, score}
`pdb_entry(pdb_id) -> Record`	Fetch PDB structure metadata
`reactome_pathways(gene) -> List[Record]`	Reactome pathway memberships for a gene
`go_term(go_id) -> Record`	Gene Ontology term details
`go_annotations(gene, species?) -> List[Record]`	GO annotations for a gene
`cosmic_gene(symbol) -> Record`	COSMIC cancer gene census entry
`datasets_gene(symbol, taxon?) -> Record`	NCBI Datasets gene data
`biomart_query(dataset, attributes, filters?) -> Table`	BioMart query returning a table
`nfcore_list(sort_by?, limit?) -> List[Record]`	List nf-core pipelines
`nfcore_search(query, limit?) -> List[Record]`	Search nf-core pipelines by name/topic
`nfcore_info(name) -> Record`	Detailed nf-core pipeline metadata
`nfcore_releases(name) -> List[Record]`	Release history for an nf-core pipeline
`nfcore_params(name) -> Record`	Parameter schema for an nf-core pipeline
`biocontainers_search(query, limit?) -> List[Record]`	Search BioContainers registry
`biocontainers_popular(limit?) -> List[Record]`	Popular BioContainers tools
`biocontainers_info(tool) -> Record`	Detailed tool info with versions
`biocontainers_versions(tool) -> List[Record]`	All versions with container image URIs

Builtin	Description
`galaxy_search(query, limit?) -> List[Record]`	Search Galaxy ToolShed repositories
`galaxy_popular(limit?) -> List[Record]`	Popular Galaxy ToolShed tools
`galaxy_categories() -> List[Record]`	Galaxy ToolShed tool categories
`galaxy_tool(owner, name) -> Record`	Galaxy ToolShed repository details
`nf_parse(path) -> Record`	Parse a Nextflow .nf file into a structured Record
`nf_to_bl(record) -> Str`	Generate BioLang pipeline code from parsed Nextflow
`galaxy_to_bl(record) -> Str`	Generate BioLang pipeline code from Galaxy workflow
`api_endpoints() -> Record`	Show current API endpoint URLs

#319 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
# requires: internet connection
# Example: annotate a gene list with pathway data
let genes = ["BRCA1", "TP53", "EGFR"]
genes |> each(|g| {
  let pathways = reactome_pathways(g)
  print(g + ": " + str(len(pathways)) + " pathways")
})
```

Utility

General-purpose helpers for debugging, timing, unit conversion, and serialization.

Builtin	Description
`print(val) -> Nil`	Print a value followed by a newline
`assert cond, msg?`	Abort with `msg` if `cond` is false
`sleep(ms) -> Nil`	Pause execution for `ms` milliseconds

Builtin	Description
`now() -> Str`	Current UTC time in ISO 8601 format
`timestamp() -> Int`	Current Unix timestamp in seconds; subtract two values to measure elapsed time
`bp(n) -> Int`	Identity; documents that `n` is in base pairs
`kb(n) -> Int`	Convert kilobases to base pairs (`n * 1000`)
`mb(n) -> Int`	Convert megabases to base pairs (`n * 1_000_000`)
`gb(n) -> Int`	Convert gigabases to base pairs (`n * 1_000_000_000`)
`json_stringify(val) -> Str`	Serialize any value to a JSON string
`json_parse(s) -> Any`	Parse a JSON string into a BioLang value
`env(name) -> Str | Nil`	Read an environment variable
`exit(code?) -> Never`	Terminate the process with an exit code (default: 0)

#320 ▶ Run ▶▶ Run All Above + This

```
# Example: time a heavy operation
let t0 = timestamp()
let result = read_fasta("data/sequences.fasta")
  |> flat_map(|r| find_orfs(r.seq, 300))
print("Found " + str(len(result)) + " ORFs in " + str(timestamp() - t0) +
"s")
```

Single-Cell Analysis

Builtins for single-cell RNA-seq workflows. Higher-level operations are available via the `singlecell`, `cellchat`, `spatial`, `velocity`, `celltypes`, `multimodal`, and `grn` packages.

Builtin	Arity	Description
`lr_score(matrix, cell_labels, lr_pairs)`	3	Pairwise ligand-receptor scoring between clusters

Builtin	Arity	Description
`lr_aggregate(lr_scores, pathway_map)`	2	Pathway-level aggregation of LR scores
`spatial_neighbors(coords, k)`	2	2-D k-NN adjacency from spatial coordinates
`spatial_moransi(expr_vec, spatial_adj)`	2	Moran's I spatial autocorrelation statistic
`reference_classify(query, ref_matrix, ref_labels)`	3	Cosine k-NN label transfer from a reference dataset
`pseudobulk_aggregate(matrix, cell_labels, sample_labels)`	3	Sum cells×genes counts per (cluster, sample); rows are genes
`wnn_graph(matrix_a, matrix_b, k)`	3	Weighted nearest-neighbor graph for multimodal integration
`velocity_estimate(spliced, unspliced)`	2	RNA velocity (β·u − s) per gene per cell

#321 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "singlecell" as sc
import "cellchat"   as cc

let obj = sc.load("data/pbmc3k/filtered_gene_bc_matrices/hg19/")
  |> sc.filter_cells(200, 5000, 20.0)
  |> sc.normalize
  |> sc.variable_genes(2000)
  |> sc.neighbors(15)
  |> sc.cluster

let scores = cc.score(obj)
cc.top(scores, 20)
cc.senders(scores)
cc.pathways(scores) |> take(10)
```

Variants & Population Genetics

Builtins for VCF-level variant analysis and population genetics statistics. The `variants` and `popgen` packages provide higher-level workflows.

Variant Parsing

Builtin	Arity	Description
`vcf_parse(text)`	1	Parse VCF text into a table of variant records
`vcf_filter(variants, min_qual, pass_only?)`	1-3	Filter a parsed VCF table by minimum QUAL and optional `PASS` status
`titv_ratio(variants)`	1	Transition / transversion ratio
`variant_summary(variants)`	1	Aggregate SNP, insertion, deletion, and MNV counts
`allele_freq(variants, field?)`	1-2	Extract an INFO allele-frequency field (default `AF`)

Population Genetics

Builtin	Arity	Description
`hwe_test(n_aa, n_ab, n_bb)`	3	Hardy-Weinberg chi-square test
`fst_weir_cockerham(pop1, pop2)`	2	Weir-Cockerham Fst from matching `[n_ref, n_total]` rows
`tajima_d(site_differences, n_sequences)`	2	Tajima's D neutrality test statistic
`ld_r2(geno_a, geno_b)`	2	Linkage disequilibrium r^2 between two variants
`allele_freq_spectrum(counts, n_sequences?)`	1-2	Folded site frequency spectrum
`nucleotide_diversity(geno_matrix)`	1	Nucleotide diversity (π) across sites

#322 ▶ CLI Only

Requires CLI — run with: bl run script.bl

```
import "variants" as vcf
import "popgen"   as pg

let pass = vcf.load_filtered("results/calls.vcf", 30.0, true)
vcf.summary(pass)
vcf.titv(pass)

pg.hwe(360, 480, 160)
pg.fst([45, 12, 89], [78, 5, 92], 200, 200)
pg.tajima(15, 20, 1000)
```

Bulk RNA-seq

Builtins for loading and normalising bulk RNA-seq quantification output.

Builtin	Arity	Description
`parse_salmon(quant_sf)`	1	Parse Salmon `quant.sf` text into a gene expression table
`parse_featurecounts(counts_txt)`	1	Parse featureCounts output into a counts table
`size_factors(count_matrix)`	1	DESeq2-style median-ratio size factors
`filter_low_counts(count_matrix, min_count, min_samples)`	3	Remove genes with fewer than `min_count` in `min_samples` samples
`tpm_matrix(count_matrix, gene_lengths)`	2	Convert raw counts to TPM
`sample_correlation(count_matrix)`	1	Pairwise Pearson correlation between samples

#323 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
import "rnaseq"       as rna
import "differential" as de

let counts = rna.from_featurecounts("results/counts.txt")
let counts = rna.filter_genes(counts, 10, 3)
let norm   = rna.normalize_sf(counts)

let conditions = ["ctrl", "ctrl", "ctrl", "treated", "treated", "treated"]
let results = de.de_wald(norm, [0, 1, 2], [3, 4, 5])
results |> filter(|r| r.padj <= 0.05 && abs(r.log2fc) >= 1.0)
```

Phylogenetics

Builtins for Newick tree parsing and distance-based phylogenetics.

Builtin	Arity	Description
`nw_parse(newick_str)`	1	Parse a Newick string into a tree record
`tree_leaves(tree)`	1	List all leaf names in the tree
`patristic_distance(tree, leaf_a, leaf_b)`	3	Sum of branch lengths between two leaves
`nw_to_distance_matrix(tree)`	1	All-pairs patristic distance matrix
`upgma(leaves, dist_matrix)`	2	UPGMA tree reconstruction from a distance matrix

#324 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "phylo" as ph

let tree = ph.load("data/sarscov2_sequences.nwk")
ph.n_leaves(tree)
ph.distance(tree, "Alpha", "Delta")

let dmat    = ph.distance_matrix(tree)
let rebuilt = ph.build_upgma(ph.leaves(tree), dmat)
```

ChIP-seq / ATAC-seq

Builtins for peak-level quality control and consensus peak generation.

Builtin	Arity	Description
`merge_peaks(peaks)`	1	Merge overlapping NarrowPeak records
`consensus_peaks(peak_lists, min_samples)`	2	Peaks present in at least `min_samples` replicates
`frip_score(reads_in_peaks, total_reads)`	2	Fraction of Reads In Peaks quality metric
`tss_enrichment(peaks, tss_sites)`	2	TSS enrichment score for ATAC QC
`peak_annotation(peaks, annotations)`	2	Annotate peaks with nearest genomic features

#325 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "chipseq" as cs

let ctrl    = cs.load_narrowpeak("results/ctrl_peaks.narrowPeak")
let treated = cs.load_narrowpeak("results/treated_peaks.narrowPeak")

cs.qc_report(ctrl, 320000, 25000000)

let consensus = cs.consensus([cs.merge(ctrl), cs.merge(treated)], 2)
cs.n_peaks(consensus)
```

Microbiome

Builtins for alpha/beta diversity, rarefaction, and taxonomic composition.

Builtin	Arity	Description
`alpha_diversity(count_vec, method)`	2	Shannon, Simpson, or Chao1 alpha diversity
`beta_diversity(matrix_a, matrix_b, method)`	3	Bray-Curtis or Jaccard beta diversity
`rarefaction(count_vec, depth)`	2	Subsample counts to `depth` without

Builtin	Arity	Description
		replacement
`relative_abundance(count_vec)`	1	Fractional abundances (sums to 1.0)
`taxonomic_collapse(otu_table, taxonomy, level)`	3	Aggregate OTU counts at a given taxonomic rank

#326 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
import "microbiome" as mb

let otus = read_csv("data/feature_table.tsv")
mb.alpha_table(otus, "shannon")

let rarefied = mb.rarefy_matrix(otus, 5000)
mb.beta(rarefied, "bray_curtis")

let taxonomy = read_csv("data/taxonomy.tsv")
mb.top_taxa(taxonomy, 10, "genus")
```

Statistics (Extended)

Higher-level statistical testing beyond the core `ttest`, `chi_square`, and `p_adjust` builtins. The `statistics` package wraps these with convenient defaults.

Builtin	Arity	Description
`bh_adjust(p_values)`	1	Benjamini-Hochberg FDR correction, order preserved
`bonferroni_adjust(p_values)`	1	Bonferroni correction, clamped to 1.0
`fisher_exact(a, b, c, d)`	4	Two-sided Fisher's exact test; returns `{p_value, odds_ratio}`
`chi_square(observed, expected)`	2	Chi-squared goodness-of-fit; returns `{statistic, df, p_value}`
`permutation_test(a, b, n)`	3	Permutation test with `n` shuffles; returns p-value

Builtin	Arity	Description
`bootstrap_ci(values, n, conf)`	3	Percentile bootstrap CI; returns `{mean, lower, upper, std_err}`
`genomic_inflation(p_values)`	1	Genomic inflation factor λ from GWAS p-values
`pearson_correlation(x, y)`	2	Pearson r, zero-variance safe

#327 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "statistics" as stat

let p_values = [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 0.9]
let adj_bh   = stat.bh(p_values)
stat.significant(adj_bh, 0.05)
stat.inflation(p_values)

let group_a = [2.1, 2.3, 1.9, 2.5, 2.2]
let group_b = [3.1, 2.8, 3.4, 2.9, 3.2]
stat.permtest(group_a, group_b, 10000)
```

RT-qPCR

Builtins for ΔCt/ΔΔCt analysis, standard-curve efficiency, and geNorm reference gene selection.

Builtin	Arity	Description
`delta_ct(sample_ct, ref_ct)`	2	ΔCt = sample_ct − reference_ct
`delta_delta_ct(sample_dct, control_dct)`	2	Fold change = 2^(−ΔΔCt)
`pcr_efficiency(cts, log_dilutions)`	2	Efficiency from standard curve; returns `{efficiency, slope}`
`reference_normalize(ct_table, ref_indices)`	2	Per-sample normalization using reference gene mean
`genorm_stability(ct_table, ref_indices)`	2	geNorm M-score stability ranking

#328 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "qpcr" as qpcr

let standard_cts  = [15.2, 18.5, 21.8, 25.1, 28.4]
let log_dilutions = [0.0, -1.0, -2.0, -3.0, -4.0]
qpcr.efficiency(standard_cts, log_dilutions)

let dct         = qpcr.dct(28.5, 22.0)
let fold_change = qpcr.ddct(dct, 6.2)
```

Proteomics

Builtins for MaxQuant LFQ data: loading, normalization, imputation, and differential abundance.

Builtin	Arity	Description
`load_maxquant(text)`	1	Parse MaxQuant `proteinGroups.txt` into a matrix
`log2_transform(matrix)`	1	Log2-transform all intensity values
`quantile_normalize(matrix)`	1	Quantile normalization across samples
`impute_minvalue(matrix, percentile)`	2	Impute missing values with a low-percentile constant
`protein_ttest(matrix, group_a, group_b)`	3	Per-protein two-sample t-test; returns `{log2fc, p_value}`
`volcano_data(test_result, fc, p)`	3	Classify each protein as up/down/unchanged

#329 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "proteomics" as prot

let mat  = prot.full_pipeline("data/proteinGroups.txt")
let de   = prot.ttest(mat, [0, 1, 2], [3, 4, 5])
let hits = prot.significant(de, 1.0, 0.05)
prot.volcano(de, 1.0, 0.05)
```

DNA Methylation

Builtins for RRBS/EPIC array data: beta/M-value conversion, DMR calling, CpG density, and epigenetic clocks.

Builtin	Arity	Description
`beta_to_mvalue(beta_vec)`	1	Convert β values to M-values (logit)
`mvalue_to_beta(m_vec)`	1	Convert M-values back to β (inverse logit)
`dmr_find(beta_matrix, group_a, group_b, min_delta, min_cpgs)`	5	Find differentially methylated regions
`cpg_density(positions, window)`	2	CpG density per window of base pairs
`epigenetic_age(beta_vec, coefs)`	2	Predicted age from a linear clock model (e.g. Horvath)
`differential_methylation(beta_matrix, group_a, group_b)`	3	Site-level ΔBeta and p-value for each CpG

#330 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "methylation" as meth

# let m_matrix = beta_matrix |> rows |> map(fn(r) -> meth.to_mvalue(r))
# let dmrs = meth.find_dmrs(beta_matrix, [0,1,2,3,4], [5,6,7,8,9])
# let diff = meth.diff_cpgs(beta_matrix, [0,1,2], [3,4,5])
# meth.significant_cpgs(diff, 0.15, 0.05)

let positions = [100, 120, 145, 300, 320, 340, 360]
meth.density(positions, 200)
```

Protein Structure

Builtins for PDB parsing, structural alignment (Kabsch RMSD), contact maps, and secondary structure assignment.

Builtin	Arity	Description
`pdb_parse(text)`	1	Parse PDB ATOM records into a table with chain/residue/coordinate fields
`rmsd(coords_a, coords_b)`	2	Optimal Kabsch RMSD between two Cα coordinate sets
`contact_map(coords, dist)`	2	Boolean contact matrix at `dist` Å cutoff
`secondary_structure(coords)`	1	DSSP-lite assignment: H (helix), E (sheet), C (coil) per residue
`backbone_angles(coords)`	1	Phi/psi dihedral angles for Ramachandran analysis

#331 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
import "structure" as st

let pred    = st.load("data/alphafold_prediction.pdb")
let exp     = st.load("data/experimental.pdb")
let pred_ca = st.ca_atoms(pred)
let exp_ca  = st.ca_atoms(exp)

st.rmsd(pred_ca, exp_ca)
st.ss_composition(pred_ca)
st.contacts(pred_ca, 8.0)
```

Biological Networks

Builtins for protein-protein interaction network analysis: centrality, shortest paths, connected components, and disease module enrichment.

Builtin	Arity	Description
`load_ppi(text)`	1	Parse STRING TSV edge list into a table of `{source, target, score}`
`degree_centrality(edges)`	1	Degree (number of neighbours) for every node
`betweenness_centrality(edges)`	1	Brandes BFS betweenness for every node

Builtin	Arity	Description
`shortest_path(edges, src, tgt)`	3	BFS shortest path between two nodes
`connected_components(edges)`	1	Union-Find component labels for every node
`network_enrichment(gene_list, edges, bg_size)`	3	Hypergeometric enrichment of a gene set in the network

#332 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
import "network" as net

let ppi = net.load_string("data/9606.protein.links.v12.0.txt")
  |> net.filter_score(0.7)

net.hub_genes(ppi, 20)
let lcc = net.largest_component(ppi)

let brca = ["TP53", "BRCA1", "BRCA2", "ATM", "CHEK2"]
net.path(lcc, "TP53", "BRCA1")
net.enrichment(brca, ppi, 20000)
```

Copy Number Variation (cnv)

Builtins for CNV analysis from WGS/WES read-depth data.

Builtin	Arity	Description
`log2_ratios(tumor, normal)`	2	log2((tumor+1)/(normal+1)) per bin
`cbs_segment(ratios)`	1	Circular binary segmentation → Table of segments
`cn_call(segments, ploidy)`	2	Integer CN from log2-ratio segments
`allele_specific_cn(baf, ratio)`	2	Major/minor allele CN from B-allele freq + ratio
`cnv_summary(segments)`	1	Fraction altered, n_segments, mean ratio

#333 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
import "cnv" as cnv

# let ratios   = cnv.ratios(tumor_depths, normal_depths)
# let segments = cnv.segment(ratios)
# cnv.call(segments, ploidy=2)
# cnv.summary(segments)

# Allele-specific CN from SNP heterozygotes
# let baf = read_csv("tumor_snps.tsv") |> col("baf")
# cnv.allelic(baf, ratios)
```

Hi-C & 3D Genomics (hic)

Builtins for chromatin conformation capture data.

Builtin	Arity	Description
`ice_normalize(matrix)`	1	Iterative correction (ICE) of contact matrix
`insulation_score(matrix, window)`	2	Diamond insulation score per bin
`tad_boundaries(scores, min_delta)`	2	Local minima in insulation score → TAD boundaries
`distance_decay(matrix)`	1	Mean contact frequency vs genomic distance
`expected_contacts(matrix)`	1	Distance-decay expected contact matrix

#334 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "hic" as hic

# let norm   = hic.normalize(contact_matrix)
# let scores = hic.insulation(norm, 10)
# let tads   = hic.boundaries(scores, delta=0.15)
# hic.decay(norm)
```

ATAC-seq (atac)

Builtins for ATAC-seq quality control and fragment analysis.

Builtin	Arity	Description
`fragment_size_dist(lengths)`	1	Histogram of fragment lengths in 10-bp bins
`nfr_enrichment(lengths)`	1	NFR/mono-nucleosome ratio (>1.5 = good quality)
`nucleosome_fractions(lengths)`	1	Sub-NFR / NFR / mono / di / tri fractions
`tss_enrichment_score(lengths, dists, flank)`	3	TSS signal / background ratio
`atac_qc(lengths)`	1	Combined QC record: NFR fraction, median size, etc.

#335 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
import "atac" as atac

# let frag_lengths = read_csv("fragments.tsv") |> col("tlen") |> map(fn(x) ->
abs(x))
# atac.qc(frag_lengths)
# atac.nfr(frag_lengths)
# atac.sizes(frag_lengths)
# atac.nucleosomes(frag_lengths)
```

Drug Response (drug)

Builtins for dose-response modelling and combination synergy.

Builtin	Arity	Description
`fit_ic50(concentrations, viabilities)`	2	4-parameter logistic fit → `{ic50, slope, top, bottom, r2}`
`dose_response_curve(concs, ic50, slope, top, bottom)`	5	Evaluate 4PL model at given concentrations

Builtin	Arity	Description
`auc_response(concentrations, viabilities)`	2	Area under dose-response curve (trapezoidal, log10-normalized)
`bliss_synergy(viab_a, viab_b, viab_combo)`	3	Bliss independence synergy score
`loewe_synergy(ic50_a, ic50_b, conc_a, conc_b, obs)`	5	Loewe additivity synergy (1 - CI)
`drug_rank(ic50_table, ascending)`	2	Rank drugs by IC50

#336 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
import "drug" as drug

let concs  = [0.001, 0.01, 0.1, 1.0, 10.0, 100.0, 1000.0, 10000.0]
let viabs  = [98.0, 95.0, 88.0, 70.0, 45.0, 20.0, 8.0, 3.0]

let params = drug.fit(concs, viabs)
drug.auc(concs, viabs)
drug.bliss(75.0, 60.0, 30.0)
drug.loewe(1.0, 2.0, 0.5, 1.0)
```

GWAS (gwas)

Builtins for genome-wide association study summary statistics.

Builtin	Arity	Description
`parse_sumstats(text)`	1	Auto-detect column names (CHR/BP/SNP/P/BETA/SE/A1/A2/MAF)
`manhattan_data(sumstats)`	1	Cumulative positions + -log10(p) for Manhattan plot
`qq_data(pvals)`	1	Expected vs observed -log10(p) for QQ plot
`clump(sumstats, p_threshold, window_kb)`	3	Greedy distance-based LD clumping
`top_loci(sumstats, p_threshold)`	2	Genome-wide significant hits

Builtin	Arity	Description
`lambda_gc(pvals)`	1	Genomic inflation factor λ

#337 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
import "gwas" as gwas

let ss   = gwas.load("data/my_gwas_sumstats.txt")
gwas.lambda(ss.pval)
let hits = gwas.hits(ss)
let loci = gwas.clump(ss, 5e-8, 250)
gwas.manhattan(ss)
gwas.qq(ss.pval)
```

Genomic Annotation (annotation)

Builtins for GTF/GFF3 parsing and genomic interval annotation.

Builtin	Arity	Description
`parse_gtf(text)`	1	Parse GTF or GFF3 → Table with gene_id, gene_name, chrom, start, end, strand
`gene_bodies(gtf)`	1	Collapse transcripts to gene-level min/max intervals
`promoters(gtf, upstream, downstream)`	3	Strand-aware TSS ± window intervals
`interval_overlap(query, subject)`	2	Chrom-aware interval overlap between two Tables
`annotate_peaks(peaks, gtf)`	2	Nearest gene + Promoter/Intragenic/Distal classification
`gene_id_map(gtf)`	1	Ensembl ID → gene name mapping Table

#338 ▶ CLI Only Requires CLI — run with: bl run script.bl

```
import "annotation" as ann

let gtf   = ann.load("data/gencode.v45.annotation.gtf")
let genes = ann.genes(gtf)
let proms = ann.promoters(gtf, 2000, 200)
let idmap = ann.id_map(gtf)

# let peaks = read_csv("peaks.narrowPeak")
# ann.annotate(peaks, gtf)
# ann.overlap(peaks, atac_peaks)
```

Appendix B: Operator Reference

BioLang operators are listed below from **highest precedence** (binds tightest) to **lowest**. Within the same precedence level, operators are left-associative unless noted otherwise.

Precedence Table

1. Access and Call (highest)

Symbol	Name	Description	Example
`.`	Field access	Access a record field or method	`variant.chrom`
`()`	Call	Invoke a function or builtin	`gc_content(seq)`
`[]`	Index	Index into a list, string, or sequence	`reads[0]`

2. Exponentiation

Symbol	Name	Description	Example
`**`	Power	Raise to a power (right-associative)	`2 ** 10` gives 1024

3. Unary

Symbol	Name	Description	Example
`-`	Negate	Arithmetic negation	`-log2(fc)`
`!`	Logical NOT	Boolean negation	`!is_snp(v)`
`not`	Logical NOT (word)	Same as `!`, reads more naturally in predicates	`not is_indel(v)`
`~`	Bitwise NOT	Bitwise complement	`~flags`

4. Multiplicative

Symbol	Name	Description	Example
`*`	Multiply	Arithmetic multiplication	`coverage * depth`
`/`	Divide	Arithmetic division	`gc_count / len(seq)`
`%`	Modulo	Remainder after division; useful for reading frame math	`pos % 3`

5. Additive

Symbol	Name	Description	Example
`+`	Add	Arithmetic addition; also concatenates strings	`start + kb(1)`
`-`	Subtract	Arithmetic subtraction	`end - start`

6. Type Cast

Symbol	Name	Description	Example
`as`	Cast	Convert between compatible types	`qual as Int`

7. Ranges

Symbol	Name	Description	Example
`..`	Range (exclusive)	Half-open range; useful for genomic coordinates	`0..len(seq)`
`..=`	Range (inclusive)	Closed range	`1..=22` for autosomal chromosomes

8. Bit Shifts

Symbol	Name	Description	Example
`<<`	Left shift	Shift bits left	`1 << 4`
`>>`	Right shift	Shift bits right; useful for SAM flag decoding	`flags >> 8`

9. Bitwise AND

Symbol	Name	Description	Example
`&`	Bitwise AND	Test individual flag bits in BAM flags	`flags & 0x4` to check unmapped

10. Bitwise XOR

Symbol	Name	Description	Example
`^`	Bitwise XOR	Exclusive or	`a ^ b`

11. Comparison

Symbol	Name	Description	Example
`==`	Equal	Structural equality	`variant_type(v) == "snp"`
`!=`	Not equal	Structural inequality	`v.chrom != "chrM"`
`<`	Less than	Numeric or lexicographic comparison	`v.qual < 30`
`>`	Greater than		`gc_content(seq) > 0.6`
`<=`	Less or equal		`len(seq) <= kb(1)`
`>=`	Greater or equal		`depth >= 10`

Symbol	Name	Description	Example
`~`	Regex match	True if the left string matches the right pattern	`header ~ "^>chr[0-9]+"`

12. Logical AND

Symbol	Name	Description	Example
`&&`	AND	Short-circuiting logical and	`is_snp(v) && v.qual > 30`
`and`	AND (word)	Same as `&&`, reads naturally in filters	`is_snp(v) and v.qual > 30`

13. Logical OR

Symbol	Name	Description	Example
`||`	OR	Short-circuiting logical or	`is_het(v) || is_hom_alt(v)`
`or`	OR (word)	Same as `||`	`is_het(v) or is_hom_alt(v)`

14. Null Coalesce

Symbol	Name	Description	Example
`??`	Null coalesce	Use the right-hand value when the left is nil	`v.info["AF"] ?? 0.0`

15. Pipe

Symbol	Name	Description	Example
`|>`	Pipe	Pass the left-hand value as the first argument to the right-hand function	`reads |> filter(|r| mean_phred(r.quality) > 30)`

Symbol	Name	Description	Example
`|>>`	Tap pipe	Like pipe, but passes the value through unchanged; used for side effects	`reads |>> print |> len()`
`|> into`	Pipe bind	Evaluate the left side, bind the result to a name, and return it	`data |> filter(|x| x > 0) |> into clean`

The pipe operators are the idiomatic way to build multi-step analysis pipelines in BioLang. The `|> into` variant lets you capture an intermediate result without breaking left-to-right reading order — `expr |> into name` is equivalent to `let name = expr`:

#339 ▶ CLI Only — Requires CLI — run with: bl run script.bl

```
read_fastq("data/reads.fastq")
  |> filter(|r| mean_phred(r.quality) > 25)
  |>> |reads| print("After QC: " + str(len(reads)) + " reads")
  |> map(|r| {id: r.id, gc: gc_content(r.seq)})
  |> write_tsv("gc_report.csv")
```

16. Assignment (lowest)

Symbol	Name	Description	Example
`=`	Assign	Bind a value to a name	`let gc = gc_content(seq)`
`+=`	Add-assign	Increment in place	`count += 1`
`-=`	Subtract-assign	Decrement in place	`remaining -= 1`
`*=`	Multiply-assign	Multiply in place	`score *= weight`
`/=`	Divide-assign	Divide in place	`total /= n`
`?=`	Nil-assign	Assign only if the target is currently nil	`cache ?= compute()`

17. Structural (not expression operators)

These symbols appear in specific syntactic positions and do not participate in general expression precedence.

Symbol	Name	Description	Example
`=>`	Fat arrow	Separates patterns from bodies in `match` and `given` arms	`match v { "snp" => handle_snp() }`
`->`	Arrow	Return type annotation in function signatures	`fn gc(seq: Dna) -> Float`

Newlines and Statement Termination

BioLang uses **newlines as statement terminators** – there are no semicolons. A newline ends the current expression unless it is suppressed.

Newline Suppression Rules

A newline does **not** terminate a statement when it appears:

1. **After a binary operator.** The expression continues on the next line.

```
let total = a +
    b +
    c
```

2. **After an opening delimiter** (`(`, `[`, `{`). The expression continues until the matching closing delimiter.

```
let record = {
    chrom: "chr1",
    start: 1000,
    end: 2000
}
```

3. **After a pipe operator** (`|>` or `|>>`). The pipeline continues on the next line.

```
reads
  |> filter(|r| mean_phred(r.quality) > 30)
  |> map(|r| r.seq)
```

4. **After a comma** inside argument lists, list literals, and record literals.

```
let xs = [
    1,
    2,
    3
]
```

5. **After keywords that expect a continuation**: `then`, `else`, `do`, `and`, `or`.

These rules mean that multi-line pipelines and data structures work naturally without any explicit continuation characters.

Blank Lines

Blank lines (lines containing only whitespace) are ignored between statements. Use them freely to organize code into logical sections.

Comments

Syntax	Name	Description
`#`	Line comment	Everything from `#` to end of line is ignored
`##`	Doc comment	Attached to the following declaration; extractable by documentation tools

```
# Conceptual or diagnostic example; not directly executable.
# This is a regular comment

## Compute GC content for a DNA sequence.
## Returns a float between 0.0 and 1.0.
fn gc(seq: Dna) -> Float
  gc_content(seq)
end
```

Doc comments (`##`) attach to the immediately following `fn`, `let`, or `type` declaration and are preserved in the AST for tooling and auto-generated documentation.