ALCHEM-J

Formal Language Specification (Draft 0xA1)

January 2026 – J. Roberto Jiménez & Kimi-K2

1. Overview

ALCHEM-J is a domain-specific language embedded in JAX for composing **algorithmic primitives** used in machine learning, optimization, and scientific computing. It provides a **formal grammar** and **denotational semantics** to describe complex computational workflows as "spells"—structured compositions of pure, PyTree-preserving transformations.

The language enables **type-safe symbolic expression** of algorithms, automatic compilation to XLA, and systematic extension via a global registry.

---

2. Alphabet (Tokens)

The token set `T` defines the primitive symbols:

text

T ≜ {
𝕀, ℝ, 𝕃, 𝕊, ℤ, ℂ, φ, # Operator symbols
⊕, ⊗, ∘, # Combinators
(,), λ, ·, →, =, # Syntax & lambda calculus
0–9, . # Numeric literals
}

Each symbol is a **token** in the grammar; `𝕀, ℝ, …` are **operator identifiers**, while `⊕, ⊗, ∘` are **composition combinators**.

---

3. Primitive Operators (Registry)

Each symbol maps to a pure function of signature:

text

Φ : Key × PyTree × PyTree → PyTree

Symbol	Call Pattern	Description
$\mathbb{I}$	`(key, x, p)`	**ISO/RIME update** – invariant state-update rule from optimization theory.
$\mathbb{R}$	`(key, x, p)`	**RL-policy update** – implements PPO, A2C, or other reinforcement learning steps.
$\mathbb{L}$	`(key, _, p)`	**Lévy random vector** – generates stable-distribution noise (heavy-tailed).
$\mathbb{S}$	`(_, x, p)`	**SVD low-rank projection** – factorizes `x` via Singular Value Decomposition.
$\mathbb{Z}$	`(_, x, p)`	**Zeta-transform of x** – applies analytic number-theoretic transformation.
$\mathbb{C}$	`(_, x, p)`	**Chaotic map iteration** – evolves `x` by a chaotic dynamical system.
φ	`(key, g, θ)`	**Meta-gradient correction** – adjusts gradients `g` given meta-parameters `θ`.

Here:

- `Key` is a JAX PRNG key.
- `x` is the primary input PyTree (e.g., model parameters, state).
- `p` is a static parameter PyTree (hyperparameters, constants).
- `_` indicates the argument is ignored or not required.

---

4. Composition Combinators

Combinators combine operators into new operators while preserving the **PyTree → PyTree** property.

Combinator	Type Signature	Semantics
$\oplus$	$(\Phi_1, \Phi_2) \to \Phi_{\|}$	**Parallel fusion** – both operators applied independently, results merged additively.
$\otimes$	$(\Phi_1, \Phi_2) \to \Phi_{seq}$	**Sequential fusion** – Φ_1 then Φ_2 applied to the result.
$\circ$	$(\phi, \Phi) \to \Phi_{cond}$	**Conditional fusion** – Φ only if predicate ϕ holds on input.

Typing rules ensure PyTree-preservation:

- `⊕` requires `Φ₁` and `Φ₂` produce compatible PyTrees for tree-addition.
- `⊗` requires output type of `Φ₁` matches input type of `Φ₂`.
- `∘` requires `ϕ : PyTree → bool`.

---

5. Syntax Grammar

The grammar defines well-formed "spells" (expressions):

text

Spell ::= Sym Params | Spell Comb Spell | '(' Spell ')'
Sym ::= 𝕀 | ℝ | 𝕃 | 𝕊 | ℤ | ℂ | φ
Comb ::= ⊕ | ⊗ | ∘
Params ::= '(' ArgList ')' | ε
ArgList ::= Key '=' Val (',' ArgList)*
Val ::= ℝ | ℕ | String | PyTree literal

Examples:

- `𝕀(rate=0.01)` – ISO update with learning rate 0.01.
- `(𝕀 ⊗ 𝕃(α=1.5))` – ISO update followed by Lévy noise.
- `φ ∘ ℝ` – RL update only if meta-gradient condition `φ` holds.

---

6. Denotational Semantics

Let `ℛ` be the **registry** mapping symbols to functions.
The meaning function `⟦·⟧` maps spells to `(Key × PyTree × PyTree → PyTree)`.

text

⟦𝕊(rank=k)⟧ ≜ λ(k,x,p). ℛ𝕊 # direct lookup
⟦S₁ ⊕ S₂⟧ ≜ λ(k,x,p). ℛ[⊕](⟦S₁⟧, ⟦S₂⟧)(k,x,p)
⟦S₁ ⊗ S₂⟧ ≜ λ(k,x,p). ℛ[⊗](⟦S₁⟧, ⟦S₂⟧)(k,x,p)
⟦φ ∘ S⟧ ≜ λ(k,x,p). ℛ[∘](φ, ⟦S⟧)(k,x,p)

Where:

- `ℛ[⊕](f,g)(k,x,p) = f(k,x,p) + g(k,x,p)` (tree-additive merge)
- `ℛ[⊗](f,g)(k,x,p) = g(k, f(k,x,p), p)` (sequential chaining)
- `ℛ[∘](ϕ,f)(k,x,p) = f(k,x,p) if ϕ(x) else x`

---

7. Type Soundness

Theorem (PyTree Preservation):
For any well-formed spell `S`, `⟦S⟧` is total and `⟦S⟧(k,x,p) ∈ PyTree`.

Proof sketch:

- Base case: Each registered `Φ` is PyTree-preserving by definition.
- Inductive step: Each combinator (`⊕`, `⊗`, `∘`) preserves the PyTree property:
  - `⊕` uses tree-addition (PyTree + PyTree → PyTree).
  - `⊗` composes PyTree → PyTree functions.
  - `∘` returns either original PyTree or transformed PyTree.
- Induction on grammar completes the proof.

---

8. Execution Model

The runtime follows a **JAX-native compilation pipeline**:

1. **Parse spell** → Abstract Syntax Tree (AST).
2. **Lookup registry** → Map symbols to JAX expressions.
3. **Trace** using `jax.make_jaxpr` → Generate XLA HLO.
4. **Compile** with `jit` / `pmap` / `shard_map` → Device kernel.
5. **Cache** kernels keyed by `(AST, static_params)` for reuse.

This ensures **high-performance execution** on CPU/GPU/TPU with JAX's optimizations.

---

9. Example Derivations

(1) FQL-RIME-Lévy

text

Spell: (𝕀 ⊗ 𝕃(α=1.5) ⊗ ℝ)(flow=4)
Denotation: λ(k,x,p). ℝ(k, 𝕃(k, 𝕀(k,x,p), p), p)

Interpretation:
Apply ISO update, add Lévy noise (stability α=1.5), then apply RL-policy update. The `flow=4` parameter is passed as static config.

(2) MRBMO-PPO-Siege

text

Spell: $(\mathbb{I} \circ \varphi \otimes \mathbb{R})(\text{siege}\geq 0.85)$
Denotation: $\lambda(k,x,p)$. if $\text{siege}(x) \geq 0.85$ then $\mathbb{R}(k, \mathbb{I}(k,x,p), p)$ else x

Interpretation:
If the `siege` condition (computed from `x`) exceeds 0.85, apply ISO update followed by RL update; otherwise, return `x` unchanged.

---

10. Registry Extension

Users may **extend ALCHEM-J** with custom operators:

python

@alchemj.register("𝔻")
def my_step(key: Key, x: PyTree, p: PyTree) -> PyTree:
# Custom transformation
return transformed_x

Requirement:
The registered function must satisfy `Φ : Key × PyTree × PyTree → PyTree`.
Extensions are **backward-compatible** and do not break existing spells.

---

11. Design Philosophy

- **Formal yet practical** – Rigorous semantics enable proof of properties while running on accelerators.
- **Compositional** – Complex algorithms built from simple, pure primitives.
- **JAX-native** – Leverages JAX transformations (`jit` , `grad` , `vmap` , `pmap`).
- **Extensible** – Users add operators without modifying core language.

---

12. Applications

- **Meta-learning** – Composing gradient updates, noise injection, and projection.
- **Reinforcement learning** – Flexible policy optimization pipelines.
- **Scientific computing** – Chaotic systems, randomized linear algebra, and stochastic optimization.
- **Automated algorithm design** – Search over spell space for novel compositions.
