We are designing a **translational language layer** that starts as a **CLIPS-compatible DSL** and evolves into a **modern, agentic rule-based system** with distributed semantics and CSP-style messaging. Here's a breakdown of the architectural and language design implications, plus some terminology that might help frame your approach:

🧠 Phase 1: CLIPS Subset Mapping

Goal

Create a language (let's call it *CAW*) that:

- Maps 1:1 to a subset of CLIPS
- Supports rule definition, fact assertion, and inference
- Can be transpiled into CLIPS syntax or executed via a CLIPS engine

Techniques

- **Transpilation**: Source-to-source compilation from CAW → CLIPS
- **Semantic preservation**: Ensure rule behavior and fact propagation match CLIPS expectations
- **Grammar design**: Likely a PEG or LALR parser that supports modular rule blocks, scoped facts, and type-safe templates

🚀 Phase 2: Language Expansion for AI + Distribution

New Language Features

- **Agent abstraction**: `Expert(...)` becomes a first-class construct
- **Namespaces**: Hierarchical domains like `Physics.Nuclear.Particle._` for scoped rule contexts
- **Distributed facts**: Backed by a vector DB (e.g., Qdrant, Weaviate, Milvus) for similarity-based retrieval
- **Message passing**: Inspired by CSP (Communicating Sequential Processes), using `!` for asynchronous sends

Pseudocode Semantics

let albert = Expert(Physics.Nuclear.Particle._)
let marie = Expert(Chemistry.Nuclear._)

marie.research(radium) ! albert.research _

This implies:

- Expert(domain) creates a rule engine scoped to a domain
- marie.research(radium) triggers a rule evaluation or fact assertion
- ! albert.research _ sends a message to albert to evaluate research with the result or context from marie

Underlying Concepts

- **Actor model**: Each Expert is an actor with its own rule base and message queue
- **CSP channels**: Model ! and ? as channel tell or ask, with optional reply semantics
- **Fact routing**: Facts can be tagged with origin, confidence, and vector embeddings for similarity-based dispatch

🧱 Suggested Language Design Terms

Concept	Term
Mapping from CAW to CLIPS	**Transpilation**
Expanded language layer	**Metalinguistic abstraction**
Domain-scoped rule engines	**Agentic modules** or **Expert agents**
Message passing between engines	**CSP-style dispatch** or **asynchronous rule invocation** (support both synchronous and asynchronous)
Distributed fact store	**Vectorized knowledge base (kvCache?)**
Fact + rule encapsulation	**Knowledge capsule** or **semantic unit (Roost)**

* Need to scaffold a grammar for CAW and

* sketch out the runtime architecture for these agents.

* Prototype the message dispatch layer

* design integratation of vector DBs for act similarity.

🪶 Language Identity: Caw

Core Themes

- **Ravens**: Messengers, memory-keepers, and watchers—perfect metaphors for distributed expert agents.
- **Irish Mythology**: Draw from the Tuatha Dé Danann, the Morrígan (goddess of fate and war, often appearing as a raven), and the concept of **geasa** (binding magical obligations).
- **Knowledge as flight**: Rules and facts are feathers; agents are wings; inference is the wind.

🧙 Syntax Inspiration

Agent Declaration

```
let albert = Expert(Physics.Nuclear.Particle._)
let marie = Expert(Chemistry.Nuclear._)
```

Message Passing

```
marie.research(radium) ! albert.research _
```

Rule Definition (Mythic Style)

```
rune "DecayLaw" when
  fact Particle(type: "radium", state: "unstable")
then
  assert Particle(state: "decaying")
```

Fact Assertion

```
feather Particle(type: "radium", state: "unstable")
```

🧩 Language Layers

Layer	Purpose	Mythic Metaphor
Caw-Core	CLIPS-compatible subset of language features	Raven's memory
Caw-Flock	Distributed agents, message passing, workflow definition	Flock of ravens
Caw-Vault	Vectorized fact store, Kvcache of knowledge base + assertions/conclusions	Bru na knowledge
Caw-Geas	Binding rules, obligations, contracts between agents	Magical contracts
Caw-Sky	CSP-style channels, tell and ask with synchronous and asynchronous invocation	Currents of wind

📜 Naming Conventions

- **feather**: Fact
- **rune**: Rule
- **flock**: Group of agents
- **cairn**: Knowledge capsule or module
- **skyline**: Communication layer
- **echo**: Message receipt or reply
- **veil**: Namespace or domain boundary

🪶 Caw Language Grammar (Draft v0.1)

`ebnf
Identifier ::= [a-zA-Z_][a-zA-Z0-9_.]* // Namespaced identifiers
TypeName ::= [A-Z][a-zA-Z0-9]* // PascalCase types
Literal ::= StringLiteral | NumberLiteral | BooleanLiteral
StringLiteral ::= '"' .*? '"'
NumberLiteral ::= [0-9]+ ('.' [0-9]+)?
BooleanLiteral ::= 'true' | 'false'

🎭 Lexical Elements

`ebnf
Identifier ::= [a-zA-Z_][a-zA-Z0-9_.]* // Namespaced identifiers
TypeName ::= [A-Z][a-zA-Z0-9]* // PascalCase types
Literal ::= StringLiteral | NumberLiteral | BooleanLiteral
StringLiteral ::= '"' .*? '"'
NumberLiteral ::= [0-9]+ ('.' [0-9]+)?
BooleanLiteral ::= 'true' | 'false'

🧠 Core Constructs

`ebnf
Program ::= { Declaration | Statement }

Declaration ::= 'feather' Identifier ':' TypeName '=' Expression
| 'rune' Identifier RuleBlock
| 'let' Identifier '=' AgentInit

RuleBlock ::= 'when' ConditionBlock 'then' ActionBlock

ConditionBlock ::= { Expression }
ActionBlock ::= { Statement }

Statement ::= Expression
| Assertion
| MessageSend
| AgentCall

Assertion ::= 'assert' Identifier '(' FieldList ')'

FieldList ::= { Identifier ':' Expression }

Expression ::= Literal
| Identifier
| FunctionCall
| AgentCall

FunctionCall ::= Identifier '(' [Expression { ',' Expression }] ')'
AgentCall ::= Identifier '.' Identifier '(' [Expression] ')'
MessageSend ::= Expression '!' Expression

🧙 Agent System (Scala-style Actor Model)

`ebnf
AgentInit   ::= 'Expert' '(' DomainPath ')'
DomainPath   ::= Identifier { '.' Identifier } ['._']

- Expert(...) creates a scoped rule engine.
- agent.method(args) invokes a rule or function.
- expr ! agent.method(_) sends a message asynchronously.

🧬 Type System

`ebnf
TypeDecl   ::= 'type' TypeName '=' TypeExpr
TypeExpr   ::= PrimitiveType
              | RecordType
              | UnionType
              | VectorType
              | FunctionType

PrimitiveType ::= 'String' | 'Number' | 'Boolean'
RecordType   ::= '{' Identifier ':' TypeExpr { ',' Identifier ':' TypeExpr } '}'
UnionType   ::= TypeExpr '|' TypeExpr
VectorType   ::= '[' TypeExpr ']'
FunctionType ::= '(' TypeExpr { ',' TypeExpr } ')' '=>' TypeExpr

- Supports **structural typing**, **union types**, and **function types**.
- Can be inferred or explicitly declared.
- Future extension: typeclasses or traits for polymorphic dispatch.

🧪 Example Snippet

```
type Particle = {
  type: String,
  state: String
}

feather radium: Particle = {
  type: "radium",
  state: "unstable"
}

rune "DecayLaw" when
  radium.state == "unstable"
then
  assert Particle(state: "decaying")

let albert = Expert(Physics.Nuclear.Particle._)
let marie = Expert(Chemistry.Nuclear._)

marie.research(radium) ! albert.research _
```

🪶 Caw PEG Parser (v0.1)

```
Program       <- Statement*

# === Statements ===
Statement      <- Declaration / Rule / ExpressionStmt

Declaration    <- 'let' Identifier '=' AgentInit
               / 'feather' Identifier ':' Type '=' Expression
               / TypeDecl

Rule           <- 'rune' StringLiteral 'when' ConditionBlock 'then' ActionBlock

ExpressionStmt <- Expression

# === Rule Blocks ===
ConditionBlock <- Expression+
ActionBlock    <- Statement+
```

```
# === Expressions ===
Expression     <- MessageSend / AgentCall / FunctionCall / Literal / Identifier

FunctionCall   <- Identifier '(' ArgList? ')'
AgentCall      <- Identifier '.' Identifier '(' ArgList? ')'
MessageSend    <- Expression '!' Expression

ArgList        <- Expression (',' Expression)*

# === Agent System ===
AgentInit      <- 'Expert' '(' DomainPath ')'
DomainPath     <- Identifier ('.' Identifier)* ('._')?

# === Type System ===
TypeDecl       <- 'type' Type '=' TypeExpr
TypeExpr       <- PrimitiveType / RecordType / UnionType / VectorType / FunctionType

PrimitiveType  <- 'String' / 'Number' / 'Boolean'
RecordType     <- '{' FieldTypeList '}'
FieldTypeList  <- Identifier ':' TypeExpr (',' Identifier ':' TypeExpr)*
UnionType      <- TypeExpr '|' TypeExpr
VectorType     <- '[' TypeExpr ']'
FunctionType   <- '(' TypeExprList ')' '=>' TypeExpr
TypeExprList   <- TypeExpr (',' TypeExpr)*

# === Literals ===
Literal        <- StringLiteral / NumberLiteral / BooleanLiteral
StringLiteral  <- '"' (!'"' .)* '"'
NumberLiteral  <- [0-9]+ ('.' [0-9]+)?
BooleanLiteral <- 'true' / 'false'

# === Identifiers ===
Identifier     <- [a-zA-Z_][a-zA-Z0-9_.]*
Type           <- [A-Z][a-zA-Z0-9]*

# === Whitespace and Comments ===
_              <- [ \t\r\n]*
Comment        <- '#' (![\r\n] .)*
```

🧪 Example Parse Tree (for reference)

```
let albert = Expert(Physics.Nuclear.Particle._)

type Particle = {
  type: String,
  state: String
}

feather radium: Particle = {
  type: "radium",
  state: "unstable"
}

rune "DecayLaw" when
  radium.state == "unstable"
then
  assert Particle(state: "decaying")

marie.research(radium) ! albert.research _
```

🧰 Next Steps

- **Parser implementation**: Tree-sitter for fast incremental parsing, or Rust pest for expressive grammar.
- **AST builder**: Map parse tree to typed AST nodes (Rule, Agent, Fact, Message).
- **Type checker**: Validate feather declarations and rule inputs/outputs.
- **Runtime**: Agent registry, message queue, rule engine, vector DB hooks.