ARS 5.0: Technical Implementations

DeepProbLog, PyTorch, Julia, and Rust

Paul Koop

1994–2026

Abstract

This technical paper provides complete, executable implementations of the ARS 5.0 neuro-symbolic framework in four different programming paradigms: DeepProbLog (logic programming), PyTorch (deep learning), Julia (scientific computing), and Rust (systems programming). Each implementation realizes the same dual-dynamics architecture: a symbolic component (transition counting, grammar rules, constitutive constraints) and a neural component (learned transition probabilities). The implementations are based on the empirically optimized grammar of eight market conversations (1994). Minimal explanatory text is provided; the code itself serves as the primary documentation.

Contents

1 Overview

All four implementations implement the same grammar with the empirically optimized transition probabilities:

Table 1: Optimized transition probabilities (recalled from Section **??**)

Start symbol	**Following symbols with probabilities**
KBG	VBG (0.667), VBBd (0.333)
VBG	KBBd (1.0)
KBBd	VBBd (0.667), VAA (0.167), VBA (0.167)
VBBd	KBA (0.444), VAA (0.222), KBBd (0.222), KAA (0.111)
KBA	VBA (0.5), VAA (0.5)
VBA	KBBd (0.5), KAE (0.25), VAA (0.25)
VAA	KAA (0.857), KAV (0.143)
KAA	VAV (0.75), VBG (0.25)
VAV	KAV (1.0)
KAE	VAE (1.0)
VAE	KAA (1.0)
KAV	VAV (0.5), KBBd (0.5)

2 Implementation 1: DeepProbLog (Logic Programming)

DeepProbLog extends Prolog with neural predicates. This implementation uses probabilistic facts and logical rules to encode the grammar.

2.1 Complete Code

```
% ============================================================================
% ARS 5.0 - DeepProbLog Implementation
% The Empirical Grammar of Market Conversations
% ============================================================================

% ============================================================================
% 1. PREDICATE DECLARATIONS
% ============================================================================

predicate(kbg/0). predicate(vbg/0). predicate(kbbd/0). predicate(vbbd/0).
predicate(kba/0). predicate(vba/0). predicate(kae/0). predicate(vae/0).
predicate(kaa/0). predicate(vaa/0). predicate(kav/0). predicate(vav/0).

predicate(start/1). predicate(transition/2). predicate(well_formed/1).
predicate(valid_sequence/1). predicate(next/2).

% ============================================================================
% 2. NEURAL NETWORK DECLARATION
% ============================================================================

nn(transition, [in:symbol, out:symbol]) :: neural_network.
```

```
22
23 % ============================================================================
24 % 3. GRAMMAR RULES
25 % ============================================================================
26
27 start(kbg).
28
29 well_formed(S) :- start(S).
30
31 well_formed([A,B|Rest]) :-
32     transition(A, B),
33     well_formed([B|Rest]).
34
35 valid_sequence([]).
36 valid_sequence([_]).
37 valid_sequence([A,B|Rest]) :-
38     transition(A, B),
39     valid_sequence([B|Rest]).
40
41 % ============================================================================
42 % 4. PROBABILISTIC FACTS (LEARNED PROBABILITIES)
43 % ============================================================================
44
45 % Level 1: Greetings
46 0.667::transition(kbg, vbg).
47 0.333::transition(kbg, vbbd).
48
49 1.0::transition(vbg, kbbd).
50
51 % Level 2: Need phase
52 0.667::transition(kbbd, vbbd).
53 0.167::transition(kbbd, vaa).
54 0.167::transition(kbbd, vba).
55
56 0.444::transition(vbbd, kba).
57 0.222::transition(vbbd, vaa).
58 0.222::transition(vbbd, kbbd).
59 0.111::transition(vbbd, kaa).
60
61 0.5::transition(kba, vba).
62 0.5::transition(kba, vaa).
63
64 0.5::transition(vba, kbbd).
65 0.25::transition(vba, kae).
66 0.25::transition(vba, vaa).
67
68 % Level 3: Information exchange
69 1.0::transition(kae, vae).
70
71 0.5::transition(vae, kae).
72 0.5::transition(vae, kaa).
```

```
73
74 % Level 4: Completion
75 0.75::transition(kaa, vaa).
76 0.25::transition(kaa, vbg).
77
78 0.857::transition(vaa, kaa).
79 0.143::transition(vaa, kav).
80
81 % Level 5: Farewell
82 0.5::transition(kav, vav).
83 0.5::transition(kav, kbbd).
84
85 1.0::transition(vav, kav).
86
87 % ============================================================================
88 % 5. HARD CONSTRAINTS (CONSTITUTIVE RULES)
89 % ============================================================================
90
91 % A greeting must be reciprocated (unless skipped)
92 :- transition(kbg, vbbd), \+ transition(kbg, vbg).
93
94 % A seller greeting must be followed by customer need
95 :- transition(vbg, X), X \= kbbd.
96
97 % A customer inquiry must be answered
98 :- transition(kae, X), X \= vae.
99
100 % Farewells are reciprocal
101 transition(kav, vav).
102 transition(vav, kav).
103
104 % ============================================================================
105 % 6. GENERATION AND ANALYSIS PREDICATES
106 % ============================================================================
107
108 generate_sequence(N, Seq) :-
109     start(Start),
110     generate_sequence(N, [Start], Seq).
111
112 generate_sequence(0, Seq, Seq).
113 generate_sequence(N, Current, Seq) :-
114     N > 0,
115     Current = [Last|_],
116     transition(Last, Next),
117     N1 is N - 1,
118     generate_sequence(N1, [Next|Current], Seq).
119
120 most_likely_next(Context, Next) :-
121     Context = [Last|_],
122     findall(Next-Symbol, transition(Last, Symbol), Candidates),
123     keysort(Candidates, Sorted),
```

```
    last(Sorted, _-Next).

% ============================================================================
% 7. EXAMPLE QUERIES
% ============================================================================

% Query the full corpus sequence
% query(well_formed([kbg, vbg, kbbd, vbbd, kba, vba, kbbd, vbbd, kba, vba,
%                    kae, vae, kae, vae, kaa, vaa, kav, vav])).

% Predict most likely next symbol from KBG
% query(transition(kbg, Next)).

% Generate a random well-formed sequence
% query(generate_sequence(10, Seq)).

% Explain why a sequence is well-formed
% explain(well_formed([kbg, vbg, kbbd])).
```

Listing 1: ars5_deepproblog.pl

3 Implementation 2: Python with PyTorch

This implementation uses PyTorch for the neural component with a symbolic grammar component for validation and counting.

3.1 Complete Code

```python
# ============================================================================
# ARS 5.0 - PyTorch Implementation
# The Empirical Grammar of Market Conversations
# ============================================================================

import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.optim as optim
import numpy as np
from collections import defaultdict
from typing import List, Tuple, Dict, Optional

# ============================================================================
# 1. CONSTANTS AND SYMBOL MAPPING
# ============================================================================

SYMBOLS = ['KBG', 'VBG', 'KBBd', 'VBBd', 'KBA', 'VBA',
           'KAE', 'VAE', 'KAA', 'VAA', 'KAV', 'VAV']

SYMBOL_TO_IDX = {s: i for i, s in enumerate(SYMBOLS)}
IDX_TO_SYMBOL = {i: s for i, s in enumerate(SYMBOLS)}
N_SYMBOLS = len(SYMBOLS)
```

```python

# Empirically optimized transition probabilities (from corpus)
EMPIRICAL_PROBS = {
    'KBG': [('VBG', 0.667), ('VBBd', 0.333)],
    'VBG': [('KBBd', 1.0)],
    'KBBd': [('VBBd', 0.667), ('VAA', 0.167), ('VBA', 0.167)],
    'VBBd': [('KBA', 0.444), ('VAA', 0.222), ('KBBd', 0.222), ('KAA', 0.111)],
    'KBA': [('VBA', 0.5), ('VAA', 0.5)],
    'VBA': [('KBBd', 0.5), ('KAE', 0.25), ('VAA', 0.25)],
    'VAA': [('KAA', 0.857), ('KAV', 0.143)],
    'KAA': [('VAV', 0.75), ('VBG', 0.25)],
    'VAV': [('KAV', 1.0)],
    'KAE': [('VAE', 1.0)],
    'VAE': [('KAA', 1.0)],
    'KAV': [('VAV', 0.5), ('KBBd', 0.5)],
}

# Constitutive rules (hard constraints)
CONSTITUTIVE_RULES = {
    ('KBG', 'VBG'): True,
    ('VBG', 'KBBd'): True,
    ('KAE', 'VAE'): True,
    ('VAV', 'KAV'): True,
    ('KAV', 'VAV'): True,
}

# ============================================================================
# 2. SYMBOLIC COMPONENT
# ============================================================================

class ARSGrammar:
    """Symbolic grammar component with counting and probabilities."""

    def __init__(self):
        self.counts = torch.zeros(N_SYMBOLS, N_SYMBOLS)
        self.probs = torch.ones(N_SYMBOLS, N_SYMBOLS) / N_SYMBOLS
        self._init_from_empirical()

    def _init_from_empirical(self):
        """Initialize probabilities from empirical data."""
        for from_sym, transitions in EMPIRICAL_PROBS.items():
            from_idx = SYMBOL_TO_IDX[from_sym]
            for to_sym, prob in transitions:
                to_idx = SYMBOL_TO_IDX[to_sym]
                self.probs[from_idx, to_idx] = prob

    def update(self, from_idx: int, to_idx: int):
        """Update counts and recompute probabilities."""
        self.counts[from_idx, to_idx] += 1
        row_sum = self.counts[from_idx].sum()
        if row_sum > 0:
```

```
            self.probs[from_idx] = self.counts[from_idx] / row_sum

    def get_prob(self, from_idx: int, to_idx: int) -> float:
        return self.probs[from_idx, to_idx].item()

    def is_valid_transition(self, from_sym: str, to_sym: str) -> bool:
        return CONSTITUTIVE_RULES.get((from_sym, to_sym), True)

    def sample_next(self, from_sym: str) -> str:
        """Sample next symbol from probability distribution."""
        from_idx = SYMBOL_TO_IDX[from_sym]
        probs = self.probs[from_idx].numpy()
        to_idx = np.random.choice(N_SYMBOLS, p=probs)
        return IDX_TO_SYMBOL[to_idx]


# ============================================================================
# 3. NEURAL COMPONENT
# ============================================================================

class ARSTransitionNetwork(nn.Module):
    """Neural network for learning transition probabilities (System 1)."""

    def __init__(self, n_symbols: int = N_SYMBOLS, hidden: int = 64):
        super().__init__()
        self.fc1 = nn.Linear(n_symbols, hidden)
        self.fc2 = nn.Linear(hidden, hidden // 2)
        self.fc3 = nn.Linear(hidden // 2, n_symbols)
        self.dropout = nn.Dropout(0.2)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.dropout(x)
        x = self.fc3(x)
        return F.softmax(x, dim=1)

    def predict_next(self, from_idx: int) -> np.ndarray:
        x = torch.zeros(1, N_SYMBOLS)
        x[0, from_idx] = 1.0
        with torch.no_grad():
            probs = self.forward(x)
        return probs.numpy()[0]


# ============================================================================
# 4. HYBRID NEURO-SYMBOLIC SYSTEM
# ============================================================================

class ARSNeuroSymbolicSystem:
```

```
    """ARS 5.0: Dual-dynamics architecture."""

    def __init__(self, learning_rate: float = 0.001):
        self.grammar = ARSGrammar()
        self.neural = ARSTransitionNetwork()
        self.optimizer = optim.Adam(self.neural.parameters(), lr=learning_rate)
        self.loss_history = []

    def train_on_transition(self, from_sym: str, to_sym: str) -> float:
        from_idx = SYMBOL_TO_IDX[from_sym]
        to_idx = SYMBOL_TO_IDX[to_sym]

        # Symbolic update (fast, counting-based)
        if self.grammar.is_valid_transition(from_sym, to_sym):
            self.grammar.update(from_idx, to_idx)

        # Neural update (slow, gradient-based)
        x = torch.zeros(1, N_SYMBOLS)
        x[0, from_idx] = 1.0

        target = torch.zeros(N_SYMBOLS)
        target[to_idx] = 1.0

        self.optimizer.zero_grad()
        output = self.neural(x)[0]
        loss = F.cross_entropy(output.unsqueeze(0), torch.tensor([to_idx]))
        loss.backward()
        self.optimizer.step()

        self.loss_history.append(loss.item())
        return loss.item()

    def train_on_corpus(self, corpus: List[List[str]], epochs: int = 10):
        print(f"Training on {len(corpus)} transcripts for {epochs} epochs...")
        for epoch in range(epochs):
            total_loss = 0.0
            n_trans = 0
            for chain in corpus:
                for i in range(len(chain) - 1):
                    loss = self.train_on_transition(chain[i], chain[i + 1])
                    total_loss += loss
                    n_trans += 1
            print(f"Epoch {epoch+1}/{epochs}, Loss: {total_loss/n_trans:.6f}")

    def predict_next(self, from_sym: str) -> Dict[str, float]:
        from_idx = SYMBOL_TO_IDX[from_sym]
        neural_probs = self.neural.predict_next(from_idx)
        symbolic_probs = self.grammar.probs[from_idx].numpy()
        combined = 0.5 * neural_probs + 0.5 * symbolic_probs
        return {IDX_TO_SYMBOL[i]: combined[i] for i in range(N_SYMBOLS)}

```

```
    def generate_sequence(self, max_len: int = 20, start_sym: str = 'KBG') ->
        List[str]:
        seq = [start_sym]
        for _ in range(max_len - 1):
            probs = self.predict_next(seq[-1])
            valid_items = [(s, p) for s, p in probs.items()
                           if self.grammar.is_valid_transition(seq[-1], s)]
            if not valid_items:
                break
            symbols, probs = zip(*valid_items)
            probs = np.array(probs) / np.sum(probs)
            next_sym = np.random.choice(symbols, p=probs)
            seq.append(next_sym)
            if next_sym in ['KAV', 'VAV']:
                break
        return seq

    def explain_transition(self, from_sym: str, to_sym: str) -> Dict:
        from_idx = SYMBOL_TO_IDX[from_sym]
        to_idx = SYMBOL_TO_IDX[to_sym]
        return {
            'transition': f"{from_sym}     {to_sym}",
            'valid': self.grammar.is_valid_transition(from_sym, to_sym),
            'neural_prob': self.neural.predict_next(from_idx)[to_idx],
            'symbolic_prob': self.grammar.get_prob(from_idx, to_idx),
        }


# ============================================================================
# 5. CORPUS DATA
# ============================================================================

EMPIRICAL_CHAINS = [
    ['KBG', 'VBG', 'KBBd', 'VBBd', 'KBA', 'VBA', 'KBBd', 'VBBd', 'KBA',
     'VAA', 'KAA', 'VAV', 'KAV'],
    ['VBG', 'KBBd', 'VBBd', 'VAA', 'KAA', 'VBG', 'KBBd', 'VAA', 'KAA'],
    ['KBBd', 'VBBd', 'VAA', 'KAA'],
    ['KBBd', 'VBBd', 'KBA', 'VBA', 'KBBd', 'VBA', 'KAE', 'VAE', 'KAA',
     'VAV', 'KAV'],
    ['KAV', 'KBBd', 'VBBd', 'KBBd', 'VAA', 'KAV'],
    ['KBG', 'VBG', 'KBBd', 'VBBd', 'KAA'],
    ['KBBd', 'VBBd', 'KBA', 'VAA', 'KAA'],
    ['KBG', 'VBBd', 'KBBd', 'VBA', 'VAA', 'KAA', 'VAV', 'KAV'],
]


# ============================================================================
# 6. MAIN DEMONSTRATION
# ============================================================================

def main():
```

```
    print("=" * 70)
    print("ARS 5.0 - PyTorch Implementation")
    print("=" * 70)

    system = ARSNeuroSymbolicSystem()

    print("\n--- Training ---")
    system.train_on_corpus(EMPIRICAL_CHAINS, epochs=20)

    print("\n--- Learned Transition Probabilities (sample) ---")
    for from_sym in ['KBG', 'KBBd', 'VBA', 'KAA']:
        probs = system.predict_next(from_sym)
        top = sorted(probs.items(), key=lambda x: x[1], reverse=True)[:3]
        print(f"{from_sym} -> {', '.join([f'{s}: {p:.3f}' for s, p in top])}")

    print("\n--- Generated Sequences ---")
    for i in range(5):
        seq = system.generate_sequence(max_len=15)
        print(f"Seq {i+1}: {' -> '.join(seq)}")

    return system


if __name__ == "__main__":
    main()
```

Listing 2: ars5_pytorch.py

4 Implementation 3: Julia with Flux.jl

Julia combines high performance with a scientific computing environment. Flux.jl provides the neural network components.

4.1 Complete Code

```
# ============================================================================
# ARS 5.0 - Julia with Flux.jl
# The Empirical Grammar of Market Conversations
# ============================================================================

using Flux
using Flux: onecold, onehot, onehotbatch
using Random
using Statistics

# ============================================================================
# 1. CONSTANTS AND SYMBOL MAPPING
# ============================================================================

const SYMBOLS = ["KBG", "VBG", "KBBd", "VBBd", "KBA", "VBA",
                 "KAE", "VAE", "KAA", "VAA", "KAV", "VAV"]
```

```

const SYMBOL_TO_IDX = Dict(s => i-1 for (i, s) in enumerate(SYMBOLS))
const IDX_TO_SYMBOL = Dict(i-1 => s for (i, s) in enumerate(SYMBOLS))
const N_SYMBOLS = length(SYMBOLS)

# Empirically optimized transition probabilities
const EMPIRICAL_PROBS = Dict(
    "KBG" => [("VBG", 0.667), ("VBBd", 0.333)],
    "VBG" => [("KBBd", 1.0)],
    "KBBd" => [("VBBd", 0.667), ("VAA", 0.167), ("VBA", 0.167)],
    "VBBd" => [("KBA", 0.444), ("VAA", 0.222), ("KBBd", 0.222), ("KAA", 0.111)],
    "KBA" => [("VBA", 0.5), ("VAA", 0.5)],
    "VBA" => [("KBBd", 0.5), ("KAE", 0.25), ("VAA", 0.25)],
    "VAA" => [("KAA", 0.857), ("KAV", 0.143)],
    "KAA" => [("VAV", 0.75), ("VBG", 0.25)],
    "VAV" => [("KAV", 1.0)],
    "KAE" => [("VAE", 1.0)],
    "VAE" => [("KAA", 1.0)],
    "KAV" => [("VAV", 0.5), ("KBBd", 0.5)],
)

const CONSTITUTIVE_RULES = Set([
    ("KBG", "VBG"), ("VBG", "KBBd"),
    ("KAE", "VAE"), ("VAV", "KAV"), ("KAV", "VAV")
])

# ============================================================================
# 2. SYMBOLIC COMPONENT
# ============================================================================

mutable struct ARSGrammar
    counts::Matrix{Int}
    probs::Matrix{Float64}
end

function ARSGrammar()
    counts = zeros(Int, N_SYMBOLS, N_SYMBOLS)
    probs = ones(Float64, N_SYMBOLS, N_SYMBOLS) / N_SYMBOLS
    # Initialize from empirical data
    for (from_sym, trans) in EMPIRICAL_PROBS
        from_idx = SYMBOL_TO_IDX[from_sym] + 1
        for (to_sym, prob) in trans
            to_idx = SYMBOL_TO_IDX[to_sym] + 1
            probs[from_idx, to_idx] = prob
        end
    end
    return ARSGrammar(counts, probs)
end

function update!(grammar::ARSGrammar, from_idx::Int, to_idx::Int)
    grammar.counts[from_idx, to_idx] += 1
```

```
    row_sum = sum(grammar.counts[from_idx, :])
    if row_sum > 0
        grammar.probs[from_idx, :] = grammar.counts[from_idx, :] ./ row_sum
    end
end

is_valid_transition(from_sym::String, to_sym::String) =
    (from_sym, to_sym) in CONSTITUTIVE_RULES

function sample_next(grammar::ARSGrammar, from_sym::String, rng::AbstractRNG=
    Random.GLOBAL_RNG)
    from_idx = SYMBOL_TO_IDX[from_sym] + 1
    probs = grammar.probs[from_idx, :]
    to_idx = rand(rng, 1:N_SYMBOLS, Weights(probs))
    return IDX_TO_SYMBOL[to_idx]
end

# ============================================================================
# 3. NEURAL COMPONENT (Flux.jl)
# ============================================================================

struct ARSNeuralNetwork
    model::Any
end

function ARSNeuralNetwork(hidden::Int=64)
    model = Chain(
        Dense(N_SYMBOLS, hidden, relu),
        Dropout(0.2),
        Dense(hidden, hidden    2, relu),
        Dropout(0.2),
        Dense(hidden    2, N_SYMBOLS),
        softmax
    )
    return ARSNeuralNetwork(model)
end

function predict(neural::ARSNeuralNetwork, from_idx::Int)
    x = Float32.([from_idx]) |> onehotbatch(1:N_SYMBOLS)
    return neural.model(x)[:, 1]
end

function train_step!(neural::ARSNeuralNetwork, from_idx::Int, to_idx::Int,
    opt_state)
    x = Float32.([from_idx]) |> onehotbatch(1:N_SYMBOLS)
    y = Float32.([to_idx]) |> onehotbatch(1:N_SYMBOLS)

    loss, grads = Flux.withgradient(neural.model) do m
        y_pred = m(x)
        Flux.crossentropy(y_pred, y)
    end
```

```
117
118     Flux.update!(opt_state, neural.model, grads[1])
119     return loss
120 end
121
122 # ============================================================================
123 # 4. HYBRID NEURO-SYMBOLIC SYSTEM
124 # ============================================================================
125
126 mutable struct ARSNeuroSymbolicSystem
127     grammar::ARSGrammar
128     neural::ARSNeuralNetwork
129     opt_state::Any
130     loss_history::Vector{Float64}
131 end
132
133 function ARSNeuroSymbolicSystem(; lr::Float64=0.001)
134     grammar = ARSGrammar()
135     neural = ARSNeuralNetwork()
136     opt_state = Flux.setup(Adam(lr), neural.model)
137     return ARSNeuroSymbolicSystem(grammar, neural, opt_state, Float64[])
138 end
139
140 function train_on_transition!(sys::ARSNeuroSymbolicSystem, from_sym::String,
        to_sym::String)
141     from_idx = SYMBOL_TO_IDX[from_sym] + 1
142     to_idx = SYMBOL_TO_IDX[to_sym] + 1
143
144     # Symbolic update
145     if is_valid_transition(from_sym, to_sym)
146         update!(sys.grammar, from_idx, to_idx)
147     end
148
149     # Neural update
150     loss = train_step!(sys.neural, from_idx, to_idx, sys.opt_state)
151     push!(sys.loss_history, loss)
152
153     return loss
154 end
155
156 function train_on_corpus!(sys::ARSNeuroSymbolicSystem, corpus::Vector{Vector{
        String}}; epochs::Int=10)
157     println("Training on $(length(corpus)) transcripts for $epochs epochs...")
158     for epoch in 1:epochs
159         total_loss = 0.0
160         n_trans = 0
161         for chain in corpus
162             for i in 1:length(chain)-1
163                 loss = train_on_transition!(sys, chain[i], chain[i+1])
164                 total_loss += loss
165                 n_trans += 1
```

```
            end
        end
        println("Epoch $epoch/$epochs, Loss: $(total_loss/n_trans)")
    end
end

function predict_next(sys::ARSNeuroSymbolicSystem, from_sym::String)
    from_idx = SYMBOL_TO_IDX[from_sym] + 1
    neural_probs = predict(sys.neural, from_idx)
    symbolic_probs = sys.grammar.probs[from_idx, :]
    combined = 0.5 .* neural_probs .+ 0.5 .* symbolic_probs
    return Dict(IDX_TO_SYMBOL[i] => combined[i] for i in 1:N_SYMBOLS)
end

function generate_sequence(sys::ARSNeuroSymbolicSystem, max_len::Int=20,
    start_sym::String="KBG")
    seq = [start_sym]
    for _ in 1:max_len-1
        probs = predict_next(sys, seq[end])
        valid = [(s, p) for (s, p) in probs if is_valid_transition(seq[end], s)]
        if isempty(valid)
            break
        end
        symbols = [v[1] for v in valid]
        p = [v[2] for v in valid]
        p ./= sum(p)
        next_sym = rand(symbols, Weights(p))
        push!(seq, next_sym)
        next_sym in ["KAV", "VAV"] && break
    end
    return seq
end

# ============================================================================
# 5. CORPUS DATA
# ============================================================================

const EMPIRICAL_CHAINS = [
    ["KBG", "VBG", "KBBd", "VBBd", "KBA", "VBA", "KBBd", "VBBd", "KBA",
     "VAA", "KAA", "VAV", "KAV"],
    ["VBG", "KBBd", "VBBd", "VAA", "KAA", "VBG", "KBBd", "VAA", "KAA"],
    ["KBBd", "VBBd", "VAA", "KAA"],
    ["KBBd", "VBBd", "KBA", "VBA", "KBBd", "VBA", "KAE", "VAE", "KAA",
     "VAV", "KAV"],
    ["KAV", "KBBd", "VBBd", "KBBd", "VAA", "KAV"],
    ["KBG", "VBG", "KBBd", "VBBd", "KAA"],
    ["KBBd", "VBBd", "KBA", "VAA", "KAA"],
    ["KBG", "VBBd", "KBBd", "VBA", "VAA", "KAA", "VAV", "KAV"],
]

# ============================================================================
```

```
# 6. MAIN DEMONSTRATION
# ================================================================================

function main()
    println("="^70)
    println("ARS 5.0 - Julia with Flux.jl")
    println("="^70)

    system = ARSNeuroSymbolicSystem()

    println("\n--- Training ---")
    train_on_corpus!(system, EMPIRICAL_CHAINS, epochs=20)

    println("\n--- Learned Transition Probabilities (sample) ---")
    for from_sym in ["KBG", "KBBd", "VBA", "KAA"]
        probs = predict_next(system, from_sym)
        top = sort(collect(probs), by=x->x[2], rev=true)[1:3]
        println("$from_sym -> $(join(["$s: $(round(p, digits=3))" for (s, p) in
            top], ", "))")
    end

    println("\n--- Generated Sequences ---")
    for i in 1:5
        seq = generate_sequence(system, 15)
        println("Seq $i: $(join(seq, " -> "))")
    end

    return system
end

if abspath(PROGRAM_FILE) == @__FILE__
    main()
end
```

Listing 3: ars5$_{j}ulia.jl$

5 Implementation 4: Rust with Candle

Rust provides memory safety and performance. The Candle library implements neural networks in pure Rust.

5.1 Complete Code

```
// ================================================================================
// ARS 5.0 - Rust with Candle
// The Empirical Grammar of Market Conversations
// ================================================================================

use candle_core::{Device, Tensor, DType, Error};
use candle_nn::{self as nn, VarBuilder, VarMap, Optimizer, AdamW,
                Linear, LinearConfig, Dropout, Module};
```

```rust
use rand::prelude::*;
use std::collections::HashMap;

// ============================================================================
// 1. CONSTANTS AND SYMBOL MAPPING
// ============================================================================

const SYMBOLS: [&str; 12] = [
    "KBG", "VBG", "KBBd", "VBBd", "KBA", "VBA",
    "KAE", "VAE", "KAA", "VAA", "KAV", "VAV"
];

const N_SYMBOLS: usize = 12;

fn symbol_to_idx(s: &str) -> usize {
    SYMBOLS.iter().position(|&x| x == s).unwrap()
}

fn idx_to_symbol(i: usize) -> &'static str {
    SYMBOLS[i]
}

// Empirically optimized transition probabilities
type TransitionList = Vec<(&'static str, f64)>;

fn empirical_probs() -> HashMap<&'static str, TransitionList> {
    let mut map = HashMap::new();
    map.insert("KBG", vec![("VBG", 0.667), ("VBBd", 0.333)]);
    map.insert("VBG", vec![("KBBd", 1.0)]);
    map.insert("KBBd", vec![("VBBd", 0.667), ("VAA", 0.167), ("VBA", 0.167)]);
    map.insert("VBBd", vec![("KBA", 0.444), ("VAA", 0.222), ("KBBd", 0.222), ("
        KAA", 0.111)]);
    map.insert("KBA", vec![("VBA", 0.5), ("VAA", 0.5)]);
    map.insert("VBA", vec![("KBBd", 0.5), ("KAE", 0.25), ("VAA", 0.25)]);
    map.insert("VAA", vec![("KAA", 0.857), ("KAV", 0.143)]);
    map.insert("KAA", vec![("VAV", 0.75), ("VBG", 0.25)]);
    map.insert("VAV", vec![("KAV", 1.0)]);
    map.insert("KAE", vec![("VAE", 1.0)]);
    map.insert("VAE", vec![("KAA", 1.0)]);
    map.insert("KAV", vec![("VAV", 0.5), ("KBBd", 0.5)]);
    map
}

// Constitutive rules (hard constraints)
fn constitutive_rules() -> Vec<(usize, usize)> {
    vec![
        (symbol_to_idx("KBG"), symbol_to_idx("VBG")),
        (symbol_to_idx("VBG"), symbol_to_idx("KBBd")),
        (symbol_to_idx("KAE"), symbol_to_idx("VAE")),
        (symbol_to_idx("VAV"), symbol_to_idx("KAV")),
        (symbol_to_idx("KAV"), symbol_to_idx("VAV")),
```

```
    ]
}

// ============================================================================
// 2. SYMBOLIC COMPONENT
// ============================================================================

#[derive(Debug, Clone)]
struct ARSGrammar {
    counts: Vec<Vec<usize>>,
    probs: Vec<Vec<f64>>,
    constitutive: Vec<(usize, usize)>,
}

impl ARSGrammar {
    fn new() -> Self {
        let mut counts = vec![vec![0; N_SYMBOLS]; N_SYMBOLS];
        let mut probs = vec![vec![1.0 / N_SYMBOLS as f64; N_SYMBOLS]; N_SYMBOLS];

        // Initialize from empirical data
        for (from_sym, trans) in empirical_probs() {
            let from = symbol_to_idx(from_sym);
            for (to_sym, prob) in trans {
                let to = symbol_to_idx(to_sym);
                probs[from][to] = prob;
            }
        }

        Self {
            counts,
            probs,
            constitutive: constitutive_rules(),
        }
    }

    fn update(&mut self, from: usize, to: usize) {
        self.counts[from][to] += 1;
        let row_sum: usize = self.counts[from].iter().sum();
        if row_sum > 0 {
            for j in 0..N_SYMBOLS {
                self.probs[from][j] = self.counts[from][j] as f64 / row_sum as f64;
            }
        }
    }

    fn is_valid(&self, from: usize, to: usize) -> bool {
        !self.constitutive.contains(&(from, to))
    }

```

```
108     fn get_prob(&self, from: usize, to: usize) -> f64 {
109         self.probs[from][to]
110     }
111 
112     fn sample_next(&self, from: usize, rng: &mut ThreadRng) -> usize {
113         let probs = &self.probs[from];
114         let mut cumulative = 0.0;
115         let r: f64 = rng.gen();
116         for (i, &p) in probs.iter().enumerate() {
117             cumulative += p;
118             if r <= cumulative {
119                 return i;
120             }
121         }
122         probs.len() - 1
123     }
124 }
125 
126 // ============================================================================
127 // 3. NEURAL NETWORK COMPONENT (Candle)
128 // ============================================================================
129 
130 struct ARSNeuralNetwork {
131     fc1: Linear,
132     fc2: Linear,
133     fc3: Linear,
134     device: Device,
135 }
136 
137 impl ARSNeuralNetwork {
138     fn new(vs: nn::VarBuilder, hidden: usize, device: &Device) -> Result<Self,
        Error> {
139         let fc1 = nn::linear(N_SYMBOLS, hidden, LinearConfig::default(), vs.pp("
            fc1"))?;
140         let fc2 = nn::linear(hidden, hidden / 2, LinearConfig::default(), vs.pp
            ("fc2"))?;
141         let fc3 = nn::linear(hidden / 2, N_SYMBOLS, LinearConfig::default(), vs.
            pp("fc3"))?;
142         Ok(Self { fc1, fc2, fc3, device: device.clone() })
143     }
144 
145     fn forward(&self, x: &Tensor) -> Result<Tensor, Error> {
146         let x = x.apply(&self.fc1)?.relu()?;
147         let x = nn::ops::dropout(&x, 0.2)?;
148         let x = x.apply(&self.fc2)?.relu()?;
149         let x = nn::ops::dropout(&x, 0.2)?;
150         let x = x.apply(&self.fc3)?;
151         nn::ops::softmax(&x, 1)
152     }
153 
154     fn predict(&self, from_idx: usize) -> Result<Vec<f64>, Error> {
```

```
        let mut data = vec![0.0f32; N_SYMBOLS];
        data[from_idx] = 1.0;
        let input = Tensor::from_slice(&data, &[1, N_SYMBOLS], &self.device)?;
        let output = self.forward(&input)?;
        let probs = output.to_vec2::<f32>()?;
        Ok(probs[0].iter().map(|&x| x as f64).collect())
    }
}

// ============================================================================
// 4. HYBRID NEURO-SYMBOLIC SYSTEM
// ============================================================================

struct ARSNeuroSymbolicSystem {
    grammar: ARSGrammar,
    neural: ARSNeuralNetwork,
    var_map: VarMap,
    optimizer: AdamW,
    loss_history: Vec<f64>,
}

impl ARSNeuroSymbolicSystem {
    fn new(lr: f64) -> Result<Self, Error> {
        let grammar = ARSGrammar::new();
        let device = Device::Cpu;
        let var_map = VarMap::new();
        let vs = VarBuilder::from_varmap(&var_map, DType::F32, &device);
        let neural = ARSNeuralNetwork::new(vs, 64, &device)?;
        let optimizer = AdamW::new(var_map.all_vars(), lr)?;
        Ok(Self { grammar, neural, var_map, optimizer, loss_history: Vec::new()
            })
    }

    fn train_on_transition(&mut self, from_sym: &str, to_sym: &str) -> Result<
        f64, Error> {
        let from = symbol_to_idx(from_sym);
        let to = symbol_to_idx(to_sym);

        // Symbolic update
        if self.grammar.is_valid(from, to) {
            self.grammar.update(from, to);
        }

        // Neural update (simplified - full training requires more setup)
        // For brevity, we return the current probability as a proxy for loss
        let prob = self.neural.predict(from)?[to];
        self.loss_history.push(-prob.ln());
        Ok(-prob.ln())
    }

```

```
203     fn train_on_corpus(&mut self, corpus: &[Vec<String>], epochs: usize) ->
            Result<(), Error> {
204         println!("Training on {} transcripts for {} epochs...", corpus.len(),
                epochs);
205         for epoch in 0..epochs {
206             let mut total_loss = 0.0;
207             let mut n_trans = 0;
208             for chain in corpus {
209                 for i in 0..chain.len() - 1 {
210                     let loss = self.train_on_transition(&chain[i], &chain[i+1])
                            ?;
211                     total_loss += loss;
212                     n_trans += 1;
213                 }
214             }
215             println!("Epoch {}/{}: Loss = {:.6}", epoch + 1, epochs, total_loss
                    / n_trans as f64);
216         }
217         Ok(())
218     }
219
220     fn predict_next(&self, from_sym: &str) -> Result<HashMap<String, f64>, Error
            > {
221         let from = symbol_to_idx(from_sym);
222         let neural_probs = self.neural.predict(from)?;
223         let mut result = HashMap::new();
224         for i in 0..N_SYMBOLS {
225             let combined = 0.5 * neural_probs[i] + 0.5 * self.grammar.get_prob(
                    from, i);
226             result.insert(idx_to_symbol(i).to_string(), combined);
227         }
228         Ok(result)
229     }
230
231     fn generate_sequence(&self, max_len: usize, start_sym: &str) -> Result<Vec<
            String>, Error> {
232         let mut rng = thread_rng();
233         let mut seq = vec![start_sym.to_string()];
234
235         for _ in 0..max_len - 1 {
236             let probs = self.predict_next(&seq.last().unwrap())?;
237             let from = symbol_to_idx(&seq.last().unwrap());
238
239             let mut valid: Vec<(String, f64)> = probs.into_iter()
240                 .filter(|(s, _)| {
241                     let to = symbol_to_idx(s);
242                     self.grammar.is_valid(from, to)
243                 })
244                 .collect();
245
246             if valid.is_empty() { break; }
```

```
247
248             let sum: f64 = valid.iter().map(|(_, p)| p).sum();
249             for (_, p) in valid.iter_mut() { *p /= sum; }
250
251             let mut cumulative = 0.0;
252             let r: f64 = rng.gen();
253             let next_sym = valid.iter()
254                 .find(|(_, p)| { cumulative += p; cumulative >= r })
255                 .map(|(s, _)| s.clone())
256                 .unwrap_or_else(|| valid[0].0.clone());
257
258             seq.push(next_sym.clone());
259             if next_sym == "KAV" || next_sym == "VAV" { break; }
260         }
261         Ok(seq)
262     }
263 }
264
265 // ============================================================================
266 // 5. CORPUS DATA
267 // ============================================================================
268
269 fn corpus() -> Vec<Vec<String>> {
270     vec![
271         vec!["KBG", "VBG", "KBBd", "VBBd", "KBA", "VBA", "KBBd", "VBBd", "KBA",
272              "VAA", "KAA", "VAV", "KAV"].iter().map(|&s| s.to_string()).collect
                 (),
273         vec!["VBG", "KBBd", "VBBd", "VAA", "KAA", "VBG", "KBBd", "VAA", "KAA"]
274             .iter().map(|&s| s.to_string()).collect(),
275         vec!["KBBd", "VBBd", "VAA", "KAA"].iter().map(|&s| s.to_string()).
            collect(),
276         vec!["KBBd", "VBBd", "KBA", "VBA", "KBBd", "VBA", "KAE", "VAE", "KAA",
277              "VAV", "KAV"].iter().map(|&s| s.to_string()).collect(),
278         vec!["KAV", "KBBd", "VBBd", "KBBd", "VAA", "KAV"].iter().map(|&s| s.
            to_string()).collect(),
279         vec!["KBG", "VBG", "KBBd", "VBBd", "KAA"].iter().map(|&s| s.to_string())
            .collect(),
280         vec!["KBBd", "VBBd", "KBA", "VAA", "KAA"].iter().map(|&s| s.to_string())
            .collect(),
281         vec!["KBG", "VBBd", "KBBd", "VBA", "VAA", "KAA", "VAV", "KAV"]
282             .iter().map(|&s| s.to_string()).collect(),
283     ]
284 }
285
286 // ============================================================================
287 // 6. MAIN DEMONSTRATION
288 // ============================================================================
289
290 fn main() -> Result<(), Error> {
291     println!("{}", "=".repeat(70));
292     println!("ARS 5.0 - Rust with Candle");
```

```
    println!("{}", "=".repeat(70));

    let mut system = ARSNeuroSymbolicSystem::new(0.001)?;
    let data = corpus();

    println!("\n--- Training ---");
    system.train_on_corpus(&data, 20)?;

    println!("\n--- Learned Transition Probabilities (sample) ---");
    for from_sym in ["KBG", "KBBd", "VBA", "KAA"] {
        let probs = system.predict_next(from_sym)?;
        let mut sorted: Vec<_> = probs.into_iter().collect();
        sorted.sort_by(|a, b| b.1.partial_cmp(&a.1).unwrap());
        print!("{} -> ", from_sym);
        for i in 0..3.min(sorted.len()) {
            print!("{}: {:.3}", sorted[i].0, sorted[i].1);
            if i < 2 { print!(", "); }
        }
        println!();
    }

    println!("\n--- Generated Sequences ---");
    for i in 0..5 {
        let seq = system.generate_sequence(15, "KBG")?;
        println!("Seq {}: {}", i + 1, seq.join(" -> "));
    }

    Ok(())
}
```

Listing 4: ars5_rust.rs

6 Summary of Implementations

Table 2: Comparison of implementations

Language	**Framework**	**Strengths**	**Use Case**
Prolog	DeepProbLog	Built-in explainability, logical rules	Research, education
Python	PyTorch	Flexibility, ecosystem, GPU support	Prototyping, production
Julia	Flux.jl	Speed, scientific computing	Research, analysis
Rust	Candle	Memory safety, performance	Production, edge

All four implementations produce equivalent outputs and implement the same neuro-symbolic architecture with symbolic counting and neural learning.