# KERN Language Development Plan

# Project Overview

KERN (Knowledge Execution & Reasoning Notation) is a deterministic intelligence execution language designed to encode business logic, rules, and workflows compactly. It's a logic-centric, graph-native execution language optimized for machine reasoning rather than human comfort.

# Core Objectives

- Create a deterministic, rule-based execution language
- Build a graph-based execution model
- Ensure PSI.brain can easily parse, analyze, refactor, and generate KERN
- Achieve extreme performance with minimal storage requirements
- Maintain auditability and security

# Development Phases

## Phase 1: Foundation (Grammar, AST, Validation)

**Duration:** 4-6 weeks

#### 1.1 Lexer and Parser Implementation

- ☑ Implement lexer based on EBNF grammar specification
- ☑ Create token definitions for all lexical elements
- ☑ Build recursive descent parser
- ☑ Handle all grammar productions (entities, rules, flows, constraints)
- ☑ Add comprehensive error reporting for syntax errors

#### 1.2 Abstract Syntax Tree (AST) Definition

- ☑ Define AST node structures for all language constructs
- ☑ Create node types for entities, rules, flows, constraints
- ☑ Implement AST visitor pattern for traversal

- ☑ Add position tracking for debugging
- ☑ Design serialization/deserialization for PSI integration

#### 1.3 Static Validation

- ☑ Implement type checking algorithms
- ☑ Create symbol table management
- ☑ Add scope resolution validation
- ☑ Implement dependency analysis
- ☑ Design rule conflict detection
- ☑ Create bytecode compatibility validation

#### 1.4 Testing Framework

- ☑ Set up unit testing infrastructure
- ☑ Create test cases for all grammar productions
- ☑ Implement parser error recovery tests
- ☑ Add validation edge case tests

## Phase 2: Rule Engine and Graph Builder (4-6 weeks)

#### 2.1 Execution Graph Data Model

- ☑ Define graph node structures for operations, rules, conditions
- ☑ Implement edge definitions for data and control flow
- ☑ Create graph builder from AST
- ☑ Design graph optimization algorithms
- ☑ Implement cycle detection algorithms

#### 2.2 Rule Engine Implementation

- ☑ Create rule matching algorithms
- ☑ Implement pattern matching engine
- ☑ Design rule priority system
- ☑ Build conflict resolution mechanisms
- ☑ Implement rule execution scheduling
- ☑ Add recursion prevention with explicit limits

### 2.3 Flow Execution Engine

- [x] Implement flow pipeline execution
- [x] Create demand-driven evaluation
- [x] Build lazy evaluation strategies
- [x] Design context passing mechanisms
- [x] Implement control flow operations (if/then/else, loop, break, halt)

### 2.4 Testing

- [ ] Create rule evaluation test cases
- [ ] Test graph building algorithms
- [ ] Validate execution order
- [ ] Test rule conflict scenarios

## Phase 3: Virtual Machine and Bytecode (6-8 weeks)

### 3.1 Bytecode Instruction Set Implementation

- [x] Implement fixed-width instruction encoding (8 bytes per instruction)
- [x] Create instruction decoder
- [x] Build all control flow instructions (NOP, JMP, JMP_IF, HALT)
- [x] Implement data and symbol instructions (LOAD_SYM, LOAD_NUM, MOVE, COMPARE)
- [x] Add graph operations instructions
- [x] Create rule execution instructions
- [x] Implement context and state instructions
- [x] Build error handling instructions
- [x] Add external interface instructions

### 3.2 Virtual Machine Core

- [x] Design register-based execution model (R0-R15, CTX, ERR, PC, FLAG)
- [ ] Implement instruction fetch-execute cycle
- [ ] Create memory management system
- [ ] Build context management
- [ ] Implement error handling as data

- [ ] Add introspection hooks for PSI
- [ ] Create step-by-step execution capability

### 3.3 Bytecode Compiler

- [ ] Build AST to bytecode translator
- [ ] Implement register allocation
- [ ] Create bytecode optimization passes
- [ ] Add bytecode serialization
- [ ] Implement bytecode verification

### 3.4 Performance and Security

- [ ] Implement memory limits
- [ ] Add execution step limits
- [ ] Create sandboxed execution environment
- [ ] Implement security validation
- [ ] Add performance monitoring

### 3.5 Testing

- [ ] Create bytecode execution tests
- [ ] Test VM instruction execution
- [ ] Validate performance targets (<10ms startup)
- [ ] Test security sandboxing

# Phase 4: Tooling and Ecosystem (4-6 weeks)

### 4.1 Development Tools

- [ ] Create command-line compiler
- [ ] Build debugger with step execution
- [ ] Implement profiler
- [ ] Create bytecode inspector
- [ ] Build graph visualizer
- [ ] Add syntax highlighting support

### 4.2 Integration Tools

- [ ] Create external function adapters
- [ ] Implement serialization/deserialization tools
- [ ] Build PSI integration APIs
- [ ] Create import/export utilities

#### 4.3 Documentation and Examples

- [ ] Write comprehensive language reference
- [ ] Create tutorial examples
- [ ] Build API documentation
- [ ] Create best practices guide

#### 4.4 Testing

- [ ] End-to-end integration tests
- [ ] Performance benchmarking
- [ ] Tooling functionality tests
- [ ] User acceptance testing

# Technical Architecture

---

# Core Components

1. **Lexer/Parser**: Converts source code to AST
2. **Validator**: Ensures semantic correctness
3. **Graph Builder**: Creates execution graphs from AST
4. **Rule Engine**: Executes rule-based logic
5. **Bytecode Compiler**: Translates graphs to bytecode
6. **Virtual Machine**: Executes bytecode
7. **Tooling**: Development and debugging utilities

# Data Flow

Source Code → Lexer → Parser → AST → Validator → Graph Builder → Bytecode Compiler → VM → Execution

## Performance Targets

- Startup time: < 10ms
- Execution: Near-native performance
- Memory: Bounded and predictable
- Compilation: Fast and deterministic

# Implementation Technologies

- **Language**: Rust/C++ for performance-critical components
- **Build System**: Cargo/CMake
- **Testing**: Built-in testing framework
- **Serialization**: Custom binary format for bytecode
- **Memory Management**: Manual or smart pointers

# Risk Mitigation

- Follow KERN design principles strictly to avoid feature creep
- Implement comprehensive testing at each phase
- Maintain backward compatibility from early versions
- Regular PSI integration validation
- Performance monitoring throughout development

# Success Metrics

- Successful compilation of all KERN grammar constructs
- Performance targets met
- PSI integration working seamlessly
- Security sandboxing effective
- Tooling providing good developer experience
- Deterministic execution guaranteed