STUNIR Semantic IR Parser Architecture

Status: Phase 2 Implementation (Parser Implementation)
Version: 1.0.0
Last Updated: 2026-01-30

Overview

This document defines the architecture for STUNIR's Semantic IR parsers, which transform high-level specifications into semantically-rich Intermediate Reference (IR) suitable for code generation across 24 target categories.

Design Philosophy

1. **Multi-Stage Pipeline**: Clear separation between lexical, syntactic, and semantic analysis
2. **Language Polyglotism**: Reference implementation in Python, production implementations in Ada SPARK (safety-critical), Rust (performance), and Haskell (correctness)
3. **Category Extensibility**: Pluggable category parsers for 24 target domains
4. **Error Recovery**: Graceful error handling with actionable diagnostics
5. **Incremental Parsing**: Support for partial re-parsing of modified specifications

Parsing Stages

Stage 1: Specification File Loading and Validation

Purpose: Load specification files and validate structural integrity.

Responsibilities:
- Read JSON/YAML specification files
- Validate against base schema (`schemas/semantic_ir_v1_schema.json`)
- Extract metadata (category, version, dependencies)
- Validate file structure and required fields

Outputs:
- Raw specification AST (Abstract Syntax Tree)
- Validation errors (if any)
- Source location map for error reporting

Error Handling:
- File not found
- Invalid JSON/YAML syntax
- Schema validation failures
- Missing required fields

Stage 2: AST Construction from Specification

Purpose: Build an Abstract Syntax Tree from the validated specification.

Responsibilities:
- Parse specification content into typed AST nodes

- Attach source locations to all nodes
- Resolve references within the specification
- Build symbol tables for identifiers

Outputs:
- Typed AST with nodes for:
- Functions (name, parameters, return type, body)
- Types (primitives, structs, enums, unions)
- Constants and global variables
- Imports/dependencies
- Category-specific constructs

Error Handling:
- Duplicate symbol definitions
- Invalid syntax in category-specific sections
- Unresolved references

Stage 3: Semantic Analysis and Type Inference

Purpose: Perform semantic validation and infer missing type information.

Responsibilities:
- Type checking for expressions and statements
- Type inference for untyped variables
- Validate control flow (unreachable code, missing returns)
- Check category-specific semantic rules
- Resolve polymorphic types
- Validate memory safety constraints (for embedded targets)

Outputs:
- Annotated AST with:
- Inferred types for all expressions
- Control flow graph
- Data flow analysis results
- Semantic error list

Error Handling:
- Type mismatches
- Undefined variables
- Invalid control flow
- Category-specific semantic violations

Stage 4: Semantic IR Generation

Purpose: Transform the annotated AST into Semantic IR format.

Responsibilities:
- Generate Semantic IR nodes from AST
- Apply category-specific transformations
- Compute IR metadata (complexity metrics, optimization hints)
- Generate IR validation report
- Serialize to JSON format

Outputs:
- Semantic IR JSON conforming to `schemas/semantic_ir_v1_schema.json`
- IR validation report
- Transformation statistics

Error Handling:
- Unsupported language features
- Category incompatibilities
- IR generation failures

Parser Architecture

Core Components

```
┌─────────────────────────────────────────────────────────┐
│                        SpecParser                       │
│  (Main orchestrator for all parsing stages)             │
└─────────────────────────────────────────────────────────┘
                          │
         ┌────────────────┼────────────────┐
         │                │                │
         ▼                ▼                ▼
┌─────────────┐   ┌──────────────┐   ┌──────────────┐
│ ASTBuilder  │   │ Semantic     │   │ IRGenerator  │
│             │   │ Analyzer     │   │              │
└─────────────┘   └──────────────┘   └──────────────┘
         │                │                │
         │                │                │
         ▼                ▼                ▼
┌─────────────────────────────────────────────────────────┐
│          Category-Specific Parsers (24)                 │
│  Embedded, Assembly, Polyglot, GPU, WASM, Lisp,         │
│  Prolog, Business, Bytecode, Constraints, Expert        │
│  Systems, FPGA, Functional, Grammar, Lexer, Parser,     │
│  Mobile, OOP, Planning, Scientific, Systems, ASM IR,    │
│  BEAM, ASP                                              │
└─────────────────────────────────────────────────────────┘
```

SpecParser (Main Parser Class)

Responsibilities:
- Coordinate all parsing stages
- Manage error accumulation and reporting
- Provide high-level parsing API
- Handle incremental parsing

API:

```python
class SpecParser:
    def __init__(self, category: str, options: ParserOptions)
    def parse_file(self, path: str) -> SemanticIR
    def parse_string(self, content: str) -> SemanticIR
    def validate_ir(self, ir: SemanticIR) -> ValidationResult
    def get_errors(self) -> List[ParseError]
```

ASTBuilder

Responsibilities:
- Construct typed AST from raw specification
- Maintain symbol tables
- Track source locations

API:

```
class ASTBuilder:
    def build_ast(self, spec: Dict) -> AST
    def build_function(self, func_spec: Dict) -> FunctionNode
    def build_type(self, type_spec: Dict) -> TypeNode
    def resolve_reference(self, ref: str) -> Optional[ASTNode]
```

SemanticAnalyzer

Responsibilities:
- Type checking and inference
- Control flow analysis
- Data flow analysis
- Category-specific validation

API:

```
class SemanticAnalyzer:
    def analyze(self, ast: AST) -> AnnotatedAST
    def check_types(self, node: ASTNode) -> TypeCheckResult
    def infer_type(self, expr: Expression) -> Type
    def validate_control_flow(self, func: FunctionNode) -> CFGResult
```

IRGenerator

Responsibilities:
- Transform annotated AST to Semantic IR
- Apply optimization hints
- Generate IR metadata

API:

```
class IRGenerator:
    def generate_ir(self, ast: AnnotatedAST) -> SemanticIR
    def generate_function_ir(self, func: FunctionNode) -> IRFunction
    def generate_type_ir(self, type_node: TypeNode) -> IRType
    def compute_metadata(self, ir: SemanticIR) -> IRMetadata
```

Category-Specific Parsers

Each of the 24 target categories has a specialized parser that extends the base parser:

Category Parser Interface

```python
class CategoryParser(ABC):
    @abstractmethod
    def validate_spec(self, spec: Dict) -> ValidationResult

    @abstractmethod
    def build_category_ast(self, spec: Dict) -> CategoryAST

    @abstractmethod
    def analyze_category_semantics(self, ast: CategoryAST) -> AnalysisResult

    @abstractmethod
    def generate_category_ir(self, ast: CategoryAST) -> CategoryIR
```

Category Parser Registry

```python
CATEGORY_PARSERS = {
    "embedded": EmbeddedParser,
    "assembly": AssemblyParser,
    "polyglot": PolyglotParser,
    "gpu": GPUParser,
    "wasm": WASMParser,
    "lisp": LispParser,
    "prolog": PrologParser,
    # ... 17 more categories
}
```

Error Handling and Recovery

Error Types

1. **Lexical Errors**: Invalid characters, malformed tokens
2. **Syntax Errors**: Invalid structure, missing required fields
3. **Semantic Errors**: Type mismatches, undefined symbols
4. **Category Errors**: Category-specific validation failures

Error Recovery Strategies

1. **Panic Mode**: Skip to next stable point (e.g., next function definition)
2. **Error Productions**: Recognize common errors and suggest fixes
3. **Best-Effort Parsing**: Continue parsing even with errors to report multiple issues

Error Reporting

```python
class ParseError:
    location: SourceLocation  # file, line, column
    error_type: ErrorType     # lexical, syntax, semantic, category
    message: str              # human-readable error message
    suggestion: Optional[str] # suggested fix
    severity: ErrorSeverity   # error, warning, info
```

Source Location Tracking

All AST nodes and IR nodes include source location information:

```python
class SourceLocation:
    file: str
    line: int
    column: int
    length: int
```

This enables:
- Precise error reporting
- IDE integration (go-to-definition, find references)
- Incremental parsing (re-parse only changed sections)

Incremental Parsing

For large specifications or IDE integration:

1. **Checkpointing**: Save parser state at stable points (function boundaries)
2. **Partial Re-parsing**: Re-parse only modified sections
3. **Dependency Tracking**: Track which definitions depend on others
4. **Cache Management**: Cache parsed results with invalidation

Implementation Languages

Python (Reference Implementation)

- **Location**: `tools/semantic_ir/parser.py`
- **Purpose**: Reference implementation, rapid prototyping
- **Strengths**: Clear code, fast iteration, extensive libraries
- **Testing**: Unit tests with pytest

Ada SPARK (Safety-Critical)

- **Location**: `tools/spark/src/semantic_ir/semantic_ir-parser.ads/adb`
- **Purpose**: DO-178C Level A compliance
- **Strengths**: Formal verification, memory safety, concurrency safety
- **Contracts**: Preconditions, postconditions, invariants
- **Testing**: SPARK proofs + unit tests

Rust (Performance)

- **Location**: `tools/rust/semantic_ir/parser.rs`
- **Purpose**: High-performance parsing
- **Strengths**: Zero-cost abstractions, memory safety, concurrency
- **Error Handling**: Result types, comprehensive error types
- **Testing**: Unit tests + property-based tests (proptest)

Haskell (Correctness)

- **Location**: `tools/haskell/src/STUNIR/SemanticIR/Parser.hs`
- **Purpose**: Formal correctness, type-level guarantees

- **Strengths**: Parser combinators, algebraic data types, type safety
- **Libraries**: Megaparsec for parsing, Aeson for JSON
- **Testing**: QuickCheck property tests + HUnit

Performance Characteristics

Target Performance

- **Small specs** (<1KB): <10ms parsing time
- **Medium specs** (1-10KB): <100ms parsing time
- **Large specs** (>10KB): <1s parsing time

Optimization Strategies

1. **Lazy Parsing**: Parse only what's needed for current operation
2. **Memoization**: Cache expensive computations
3. **Parallel Parsing**: Parse independent sections concurrently
4. **Zero-Copy Parsing** (Rust): Avoid string allocations where possible

Testing Strategy

Unit Tests

- Test each parsing stage independently
- Test each category parser
- Test error handling and recovery

Integration Tests

- End-to-end parsing of complete specifications
- Test all 24 categories
- Test cross-category references

Property-Based Tests

- Round-trip: Spec → IR → Spec
- Invariant preservation: IR always valid
- Error completeness: All errors reported

Fuzzing

- Generate random specifications
- Test parser robustness
- Discover edge cases

Documentation Strategy

1. **API Documentation**: Docstrings/comments for all public APIs
2. **Architecture Guide** (this document)
3. **User Guide**: How to write specifications
4. **Category Guides**: Specification format for each category
5. **Examples**: Complete examples for all categories
6. **Troubleshooting Guide**: Common errors and solutions

Future Enhancements

1. **Streaming Parser**: Parse large specifications without loading into memory
2. **Language Server Protocol (LSP)**: IDE integration
3. **Tree-sitter Integration**: Syntax highlighting and structural editing
4. **Graphical Specification Editor**: Visual specification creation
5. **Specification Refactoring**: Automated specification transformations

Conclusion

This parser architecture provides a solid foundation for transforming high-level specifications into Semantic IR. The multi-stage pipeline, category extensibility, and multi-language implementation ensure robustness, safety, and performance across all 24 target categories.