STUNIR Semantic IR Parser User Guide

Version: 1.0.0
Last Updated: 2026-01-30

Overview

The STUNIR Semantic IR Parser transforms high-level specifications into semantically-rich Intermediate Reference (IR) suitable for code generation across 24 target categories.

Quick Start

Basic Usage

```
# Parse an embedded specification
python3 -m tools.semantic_ir.parse_spec \
  --input examples/specifications/embedded_example.json \
  --category embedded \
  --output embedded_ir.json

# Parse with validation
python3 -m tools.semantic_ir.parse_spec \
  --input spec.json \
  --category gpu \
  --validate \
  --verbose
```

Python API

```python
from tools.semantic_ir.parser import SpecParser
from tools.semantic_ir.types import ParserOptions

# Create parser
options = ParserOptions(category="embedded")
parser = SpecParser(options)

# Parse specification
ir = parser.parse_file("spec.json")

# Check for errors
if parser.get_errors():
    for error in parser.get_errors():
        print(error)

# Generate JSON
print(ir.to_json(pretty=True))
```

Supported Categories

The parser supports all 24 target categories:

Core Categories

1. **embedded** - Embedded systems (ARM, AVR, RISC-V, etc.)
2. **assembly** - Assembly languages (x86, ARM, MIPS, etc.)
3. **polyglot** - Polyglot languages (C89, C99, Rust, Zig)
4. **gpu** - GPU kernels (CUDA, ROCm, OpenCL, Metal, Vulkan)
5. **wasm** - WebAssembly

Language Families

1. **lisp** - Lisp dialects (Common Lisp, Scheme, Clojure, Racket, Emacs Lisp, Guile, Hy, Janet)
2. **prolog** - Prolog dialects (SWI-Prolog, GNU Prolog, SICStus, YAP, XSB, Ciao, B-Prolog, ECLiPSe)

Specialized Categories

1. **business** - Business languages (COBOL, BASIC)
2. **bytecode** - Bytecode targets (JVM, .NET)
3. **constraints** - Constraint programming
4. **expert_systems** - Expert systems
5. **fpga** - FPGA/HDL (VHDL, Verilog)
6. **functional** - Functional languages
7. **grammar** - Parser generators (ANTLR, PEG, BNF, EBNF, Yacc)
8. **lexer** - Lexer specifications
9. **parser** - Parser specifications
10. **mobile** - Mobile platforms
11. **oop** - Object-oriented languages
12. **planning** - Planning languages (PDDL)
13. **scientific** - Scientific computing (Fortran, Pascal)
14. **systems** - Systems languages (Ada, D)
15. **asm_ir** - Assembly IR
16. **beam** - BEAM VM (Erlang, Elixir)
17. **asp** - Answer Set Programming

Specification Format

Basic Structure

```json
{
  "schema": "https://stunir.dev/schemas/semantic_ir_v1_schema.json",
  "metadata": {
    "category": "embedded",
    "version": "1.0.0",
    "description": "My specification"
  },
  "functions": [
    {
      "name": "my_function",
      "parameters": [
        {"name": "param1", "type": "i32"}
      ],
      "return_type": "i32",
      "body": []
    }
  ],
  "types": [],
  "constants": [],
  "imports": []
}
```

Metadata Fields

- `category` (required): Target category
- `version` (optional): Specification version
- `description` (optional): Human-readable description
- Category-specific fields (see Category Guide)

Functions

```json
{
  "name": "add",
  "parameters": [
    {"name": "a", "type": "i32"},
    {"name": "b", "type": "i32"}
  ],
  "return_type": "i32",
  "body": [
    {
      "kind": "return",
      "expressions": [
        {
          "kind": "binary_op",
          "value": "+",
          "operands": [
            {"kind": "variable", "value": "a"},
            {"kind": "variable", "value": "b"}
          ]
        }
      ]
    }
  ]
}
```

Types

```json
{
  "name": "Point",
  "type": {
    "name": "Point",
    "is_primitive": false,
    "fields": {
      "x": "i32",
      "y": "i32"
    }
  }
}
```

CLI Reference

Commands

parse_spec

Parse specification file to Semantic IR.

Syntax:

```
python3 -m tools.semantic_ir.parse_spec [OPTIONS]
```

Options:
- `--input, -i FILE` - Input specification file (required)
- `--output, -o FILE` - Output IR file (default: stdout)
- `--category, -c CAT` - Target category (required)
- `--format, -f FMT` - Output format: json, pretty, compact (default: pretty)
- `--validate` - Validate generated IR
- `--verbose, -v` - Verbose output
- `--debug` - Debug mode
- `--no-type-inference` - Disable type inference
- `--max-errors N` - Maximum errors (default: 100)

Examples:

```
# Basic parsing
python3 -m tools.semantic_ir.parse_spec \
  --input spec.json \
  --category embedded \
  --output ir.json

# With validation and verbose output
python3 -m tools.semantic_ir.parse_spec \
  --input spec.json \
  --category gpu \
  --validate \
  --verbose

# Compact output to stdout
python3 -m tools.semantic_ir.parse_spec \
  --input spec.json \
  --category lisp \
  --format compact
```

Error Handling

Error Types

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

Error Format

```
ERROR: spec.json:10:5: Type mismatch: expected 'i32', got 'f64'
Suggestion: Cast the value to i32
```

Common Errors

Category Mismatch

```
ERROR: Category mismatch: expected 'embedded', got 'wasm'
```

Solution: Ensure `metadata.category` matches `--category` flag.

Missing Required Field

```
ERROR: Missing required field: 'name' in function definition
```

Solution: Add the required field to your specification.

Type Error

```
ERROR: Undefined type: 'MyType'
```

Solution: Define the type in the `types` section before using it.

Best Practices

1. Always Specify Category

Explicitly set the category in both metadata and CLI:

```json
{
  "metadata": {
    "category": "embedded"
  }
}
```

```
python3 -m tools.semantic_ir.parse_spec --category embedded ...
```

2. Use Type Inference

Let the parser infer types when possible:

```json
{
  "expressions": [
    {"kind": "literal", "value": 42}  // Type inferred as i32
  ]
}
```

3. Validate Output

Always validate generated IR:

```
python3 -m tools.semantic_ir.parse_spec --validate ...
```

4. Use Verbose Mode for Debugging

```
python3 -m tools.semantic_ir.parse_spec --verbose --debug ...
```

5. Organize Specifications

Keep category-specific fields separate:

```json
{
  "metadata": {
    "category": "embedded",
    "target_arch": "arm",  // Category-specific
    "memory": { ... }      // Category-specific
  },
  "functions": [ ... ]     // Generic
}
```

Advanced Usage

Incremental Parsing

```
options = ParserOptions(
    category="embedded",
    incremental=True  # Enable incremental parsing
)
parser = SpecParser(options)
```

Custom Error Handling

```
parser = SpecParser(options)
ir = parser.parse_file("spec.json")

for error in parser.get_errors():
    if error.severity == ErrorSeverity.ERROR:
        print(f"ERROR: {error}")
    elif error.severity == ErrorSeverity.WARNING:
        print(f"WARNING: {error}")
```

Performance Profiling

```
import time

options = ParserOptions(category="embedded", collect_metrics=True)
parser = SpecParser(options)

start = time.time()
ir = parser.parse_file("spec.json")
end = time.time()

print(f"Parsing time: {end - start:.2f}s")
print(f"Functions: {len(ir.functions)}")
print(f"Complexity: {ir.metadata.complexity_metrics['total']}")
```

Troubleshooting

Parser Fails to Load

Problem: `ModuleNotFoundError: No module named 'tools.semantic_ir'`

Solution: Ensure you’re running from the STUNIR repo root and have installed dependencies:

```
cd /path/to/stunir_repo
pip install -r requirements.txt
python3 -m tools.semantic_ir.parse_spec ...
```

Category Not Found

Problem: `KeyError: 'my_category'`

Solution: Check that the category is one of the 24 supported categories (see list above).

Slow Parsing

Problem: Parser takes too long for large specifications

Solution:
- Enable incremental parsing
- Disable type inference for first pass
- Split large specifications into modules

See Also

- Parser Architecture (SEMANTIC_IR_PARSER_ARCHITECTURE.md)
- API Documentation (SEMANTIC_IR_PARSER_API.md)
- Specification Format (SEMANTIC_IR_SPECIFICATION_FORMAT.md)
- Category Guide (SEMANTIC_IR_CATEGORY_GUIDE.md)
- Examples (../examples/specifications/)