STUNIR Semantic IR Examples Guide

Version: 1.0.0
Status: Implementation Complete
Last Updated: 2026-01-30

Overview

This guide provides detailed walkthroughs of Semantic IR examples, demonstrating key concepts and patterns for different target categories.

Example Structure

All examples follow this structure:

```
{
  "schema_version": "1.0.0",
  "ir_version": "1.0.0",
  "root": {
    // Module IR
  },
  "metadata": {
    // Generation metadata
  }
}
```

Example 1: Simple Function (Native Target)

File: `examples/semantic_ir/simple_function.json`

Purpose

Demonstrates basic function with integer addition—the “Hello World” of Semantic IR.

Key Concepts

- Function declarations
- Binary expressions
- Return statements
- Variable references

Structure Breakdown


```
{
  "declarations": [
    {
      "kind": "function_decl",
      "name": "add",
      "function_type": {
        "parameters": [
          {"name": "a", "type": {"kind": "primitive_type", "primitive": "i32"}},
          {"name": "b", "type": {"kind": "primitive_type", "primitive": "i32"}}
        ],
        "return_type": {"kind": "primitive_type", "primitive": "i32"}
      },
      "body": {
        "kind": "block_stmt",
        "statements": [
          {
            "kind": "return_stmt",
            "value": {
              "kind": "binary_expr",
              "op": "+",
              "left": {"kind": "var_ref", "name": "a"},
              "right": {"kind": "var_ref", "name": "b"}
            }
          }
        ]
      }
    }
  ]
}
```


Generated Code (C)

```c
int32_t add(int32_t a, int32_t b) {
    return a + b;
}
```

Generated Code (Rust)

```rust
pub fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

Example 2: Embedded Startup (Safety-Critical)

File: `examples/semantic_ir/embedded_startup.json`

Purpose

Demonstrates embedded system startup code with interrupt handling for DO-178C Level A systems.

Key Concepts

- Entry point functions
- Interrupt handlers

- Safety-critical attributes
- Stack usage annotations
- Memory section placement

Interrupt Handler

```
{
  "kind": "function_decl",
  "name": "isr_handler",
  "attributes": {
    "interrupt_vector": 5,
    "stack_usage": 128,
    "priority": 10,
    "embedded": {
      "save_context": true,
      "naked_function": false
    }
  }
}
```

Generated Code (C for ARM Cortex-M)

```
/* STUNIR Generated Code - DO-178C Level A */
__attribute__((interrupt("IRQ")))
__attribute__((section(".text")))
void isr_handler(void) {
    /* Stack usage: 128 bytes */
    /* Priority: 10 */
    /* Context automatically saved */
}
```

Safety Annotations

- **DAL A**: Design Assurance Level A (highest)
- **Stack bound**: 128 bytes guaranteed
- **Interrupt vector**: 5 (hardware-specific)
- **Context save**: Automatic register preservation

Example 3: GPU Vector Addition

File: `examples/semantic_ir/gpu_kernel.json`

Purpose

Demonstrates GPU kernel for parallel vector addition.

Key Concepts

- Kernel execution model
- Workgroup configuration
- Global memory access
- Pointer types with address spaces

Kernel Declaration


```
{
  "kind": "function_decl",
  "name": "vector_add_kernel",
  "parameters": [
    {
      "name": "a",
      "type": {
        "kind": "pointer_type",
        "pointee": {"kind": "primitive_type", "primitive": "f32"},
        "address_space": "global",
        "mutability": "immutable"
      }
    }
  ],
  "attributes": {
    "gpu": {
      "execution_model": "kernel",
      "workgroup_size": [256, 1, 1],
      "shared_memory": 0,
      "max_registers": 32
    }
  }
}
```


Generated Code (OpenCL)

```
__kernel void vector_add_kernel(
    __global const float* a,
    __global const float* b,
    __global float* result,
    int n
) {
    int i = get_global_id(0);
    if (i < n) {
        result[i] = a[i] + b[i];
    }
}
```

Generated Code (CUDA)

```
__global__ void vector_add_kernel(
    const float* __restrict__ a,
    const float* __restrict__ b,
    float* __restrict__ result,
    int n
) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) {
        result[i] = a[i] + b[i];
    }
}
```

Example 4: WebAssembly Module

File: `examples/semantic_ir/wasm_module.json`

Purpose

Demonstrates WebAssembly module with imports and exports.

Key Concepts

- WASM imports/exports
- Module boundaries
- Type mappings (i32, i64, f32, f64)

WASM Function

```
{
  "kind": "function_decl",
  "name": "multiply",
  "attributes": {
    "wasm": {
      "wasm_export": "multiply",
      "wasm_type": "i32"
    }
  }
}
```

Generated Code (WAT)

```
(module
  (import "env" "log" (func $log (param i32)))
  (func $multiply (export "multiply") (param $x i32) (param $y i32) (result i32)
    local.get $x
    local.get $y
    i32.mul
  )
)
```

JavaScript Interface

```
const wasmInstance = await WebAssembly.instantiate(wasmModule, {
    env: {
        log: (x) => console.log(x)
    }
});

const result = wasmInstance.exports.multiply(6, 7); // 42
```

Example 5: Functional Factorial (Lisp)

File: `examples/semantic_ir/lisp_expression.json`

Purpose

Demonstrates recursive functional programming with tail recursion.

Key Concepts

- Recursive function calls
- Ternary expressions (if-then-else)

- Pure functions
- Tail recursion optimization

Factorial Function

```
{
  "kind": "function_decl",
  "name": "factorial",
  "body": {
    "kind": "ternary_expr",
    "condition": {
      "kind": "binary_expr",
      "op": "<=",
      "left": {"kind": "var_ref", "name": "n"},
      "right": {"kind": "integer_literal", "value": 1}
    },
    "then_expr": {"kind": "integer_literal", "value": 1},
    "else_expr": {
      "kind": "binary_expr",
      "op": "*",
      "left": {"kind": "var_ref", "name": "n"},
      "right": {
        "kind": "function_call",
        "function": "factorial",
        "arguments": [
          {
            "kind": "binary_expr",
            "op": "-",
            "left": {"kind": "var_ref", "name": "n"},
            "right": {"kind": "integer_literal", "value": 1}
          }
        ]
      }
    }
  },
  "attributes": {
    "functional": {
      "pure": true,
      "tail_recursive": true
    }
  }
}
```

Generated Code (Common Lisp)

```
(defun factorial (n)
  "STUNIR Generated - Pure, Tail Recursive"
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
```

Generated Code (Scheme)

```
(define (factorial n)
  ;; STUNIR Generated - Pure, Tail Recursive
  (if (<= n 1)
      1
      (* n (factorial (- n 1)))))
```

Advanced Patterns

Pattern 1: Struct Member Access

```
{
  "kind": "member_expr",
  "object": {
    "kind": "var_ref",
    "name": "point",
    "type": {"kind": "type_ref", "name": "Point3D"}
  },
  "member": "x",
  "type": {"kind": "primitive_type", "primitive": "f32"}
}
```

Pattern 2: Array Indexing

```
{
  "kind": "array_access",
  "array": {
    "kind": "var_ref",
    "name": "buffer",
    "type": {
      "kind": "array_type",
      "element_type": {"kind": "primitive_type", "primitive": "u8"},
      "size": 256
    }
  },
  "index": {
    "kind": "var_ref",
    "name": "i",
    "type": {"kind": "primitive_type", "primitive": "i32"}
  }
}
```

Pattern 3: Type Casting

```
{
  "kind": "cast_expr",
  "operand": {
    "kind": "var_ref",
    "name": "x",
    "type": {"kind": "primitive_type", "primitive": "i32"}
  },
  "target_type": {"kind": "primitive_type", "primitive": "f32"}
}
```

Creating Your Own Examples

Step 1: Define Module Structure

```
from semantic_ir.modules import IRModule, ModuleMetadata
from semantic_ir.ir_types import IRNodeKind, TargetCategory

module = IRModule(
    node_id="n_mod_my_example",
    kind=IRNodeKind.MODULE,
    name="my_example",
    metadata=ModuleMetadata(
        target_categories=[TargetCategory.NATIVE]
    )
)
```

Step 2: Add Declarations

```
from semantic_ir.declarations import FunctionDecl, Parameter
from semantic_ir.nodes import PrimitiveTypeRef, TypeKind
from semantic_ir.ir_types import IRPrimitiveType

func = FunctionDecl(
    node_id="n_func_my_func",
    kind=IRNodeKind.FUNCTION_DECL,
    name="my_function",
    return_type=PrimitiveTypeRef(
        kind=TypeKind.PRIMITIVE,
        primitive=IRPrimitiveType.I32
    ),
    parameters=[]
)

module.add_declaration(func.node_id)
```

Step 3: Serialize to JSON

```
import json

# Serialize
json_str = module.model_dump_json(exclude_none=True, indent=2)

# Save to file
with open("my_example.json", 'w') as f:
    f.write(json_str)
```

Step 4: Validate

```
python tools/semantic_ir/validator.py my_example.json
```

Testing Examples

Validation Test

```
def test_example_validation():
    validator = SemanticIRValidator()

    for example_file in Path("examples/semantic_ir").glob("*.json"):
        result = validator.validate_file(example_file)
        assert result.status == ValidationStatus.VALID, \
            f"{example_file.name} validation failed: {result.message}"
```

Round-Trip Test

```
def test_example_round_trip():
    # Load example
    with open("examples/semantic_ir/simple_function.json", 'r') as f:
        original_json = json.load(f)

    # Parse to Python objects
    module = IRModule(**original_json["root"])

    # Serialize back
    reconstructed_json = json.loads(module.model_dump_json(exclude_none=True))

    # Compare
    assert reconstructed_json["node_id"] == original_json["root"]["node_id"]
    assert reconstructed_json["name"] == original_json["root"]["name"]
```

See Also

- Schema Guide (SEMANTIC_IR_SCHEMA_GUIDE.md)
- Validation Guide (SEMANTIC_IR_VALIDATION_GUIDE.md)
- Specification (SEMANTIC_IR_SPECIFICATION.md)
- Example Files (../examples/semantic_ir/)