Tutorial 3: Advanced STUNIR Features

Duration: 20 minutes
Level: Intermediate
Prerequisites: Completed Tutorials 1 & 2

Video Script

Introduction (0:00 - 1:00)

> "Welcome to the advanced tutorial! Today we'll explore STUNIR's powerful features: multi-target generation, custom emitters, batch processing, and the manifest system."

Multi-Target Generation (1:00 - 5:00)

Concept:

> "STUNIR can generate code for multiple languages from a single IR. This ensures consistency across implementations."

Supported targets:
- Python (3.8+)
- Rust
- C89/C99
- x86/ARM Assembly
- Go, TypeScript (via custom emitters)

Demo - Generate all targets:

```
# Generate for all built-in targets
for target in python rust c89 c99 x86 arm; do
  python3 tools/emitters/emit_code.py \
    --target $target \
    --ir module.ir.json \
    --output output/$target/
  echo "Generated: $target"
done
```

Type mapping explanation:

IR Type	Python	Rust	C99	Go
i32	int	i32	int32_t	int32
f64	float	f64	double	float64
bool	bool	bool	bool	bool
str	str	String	char*	string

Creating Custom Emitters (5:00 - 10:00)

Explain the emitter framework:

> "STUNIR's emitter framework lets you create generators for any language. Let's create a TypeScript emitter."

Create `typescript_emitter.py`:

```python
from typing import Dict, Any

class TypeScriptEmitter:
    """Custom emitter for TypeScript."""

    TYPE_MAP = {
        'i32': 'number',
        'f64': 'number',
        'bool': 'boolean',
        'str': 'string',
        'void': 'void'
    }

    def emit(self, ir: Dict[str, Any]) -> str:
        lines = [
            f"// Generated TypeScript module: {ir['module']['name']}",
            f"// Version: {ir['module']['version']}",
            ""
        ]

        for func in ir.get('functions', []):
            sig = func.get('signature', {})
            params = ', '.join(
                f"{p['name']}: {self.TYPE_MAP.get(p['type'], 'any')}"
                for p in sig.get('params', [])
            )
            ret = self.TYPE_MAP.get(sig.get('returns', 'void'), 'any')

            lines.append(
                f"export function {func['name']}({params}): {ret} {{"
            )
            lines.append("    throw new Error('Not implemented');")
            lines.append("}")
            lines.append("")

        return '\n'.join(lines)
```

Use the custom emitter:

```python
import json
from typescript_emitter import TypeScriptEmitter

# Load IR
with open('module.ir.json') as f:
    ir = json.load(f)

# Emit
emitter = TypeScriptEmitter()
code = emitter.emit(ir)
print(code)
```

Batch Processing (10:00 - 14:00)

When to use batch processing:

> "When you have multiple modules to process, batch mode is more efficient than processing them one by one."

Demo - Batch script:

```
#!/bin/bash
# batch_build.sh - Process multiple specs

SPECS_DIR="specs"
OUTPUT_DIR="output"

# Find all spec files
find "$SPECS_DIR" -name "*.spec.json" | while read spec; do
  name=$(basename "$spec" .spec.json)
  echo "Processing: $name"

  # Generate IR
  python3 tools/ir_emitter/emit_ir.py \
    "$spec" "$OUTPUT_DIR/ir/${name}.ir.json"

  # Generate targets
  python3 tools/emitters/emit_code.py \
    --target python \
    --ir "$OUTPUT_DIR/ir/${name}.ir.json" \
    --output "$OUTPUT_DIR/python/"
done

# Generate aggregate manifest
python3 manifests/ir/gen_ir_manifest.py --ir-dir "$OUTPUT_DIR/ir/"
```

Python batch processing:

```
from concurrent.futures import ThreadPoolExecutor
import os

def process_spec(spec_path):
    """Process a single spec."""
    # ... processing logic ...
    return {'spec': spec_path, 'success': True}

def batch_process(spec_dir, max_workers=4):
    """Process all specs in parallel."""
    specs = [f for f in os.listdir(spec_dir) if f.endswith('.spec.json')]

    with ThreadPoolExecutor(max_workers=max_workers) as executor:
        results = list(executor.map(
            process_spec,
            [os.path.join(spec_dir, s) for s in specs]
        ))

    return results
```

The Manifest System (14:00 - 18:00)

Manifest types:

Type	Purpose	File
IR	Track IR artifacts	`ir_manifest.json`
Targets	Track generated code	`targets_manifest.json`
Receipts	Track all receipts	`receipts_manifest.json`
Pipeline	Track build stages	`pipeline_manifest.json`

Manifest structure:

```
{
  "schema": "stunir.manifest.ir.v1",
  "manifest_epoch": 1738000000,
  "entries": [
    {
      "name": "module.ir.json",
      "path": "asm/ir/module.ir.json",
      "hash": "abc123...",
      "size": 1234
    }
  ],
  "manifest_hash": "def456..."
}
```

Verification workflow:

```
# Verify individual manifests
python3 manifests/ir/verify_ir_manifest.py receipts/ir_manifest.json
python3 manifests/targets/verify_targets_manifest.py receipts/targets_manifest.json

# Strict verification (all manifests)
./scripts/verify_strict.sh --strict

# Check specific artifact
python3 -c "
import json
import hashlib

with open('asm/ir/module.ir.json', 'rb') as f:
    actual = hashlib.sha256(f.read()).hexdigest()
print(f'Computed: {actual}')
"
```

Provenance Tracking (18:00 - 19:30)

What is provenance?

> "Provenance tracks the complete history of how artifacts were created - including tool versions, timestamps, and input hashes."

Generate provenance header:

```
# Generate C provenance header
./tools/native/haskell/stunir-native/stunir-native \
  gen-provenance \
  --spec spec.json \
  --out provenance.h \
  --extended
```

Output:

```
#ifndef STUNIR_PROVENANCE_H
#define STUNIR_PROVENANCE_H

#define STUNIR_PROV_BUILD_EPOCH 1738000000
#define STUNIR_PROV_SPEC_DIGEST "abc123..."
#define STUNIR_PROV_TOOL_VERSION "2.0.0"

#endif
```

Wrap Up (19:30 - 20:00)

Summary:
- ✅ Multi-target generation
- ✅ Custom emitter creation
- ✅ Batch processing
- ✅ Manifest system
- ✅ Provenance tracking

Next steps:

> "The next tutorial covers troubleshooting common issues. Happy building!"

Code Templates

Custom Emitter Template

```
class MyCustomEmitter:
    TYPE_MAP = {...}

    def emit(self, ir): ...
    def emit_function(self, func): ...
```

Batch Script Template

```
find specs/ -name "*.spec.json" | parallel python3 process.py {}
```