STUNIR Basic Code Generation

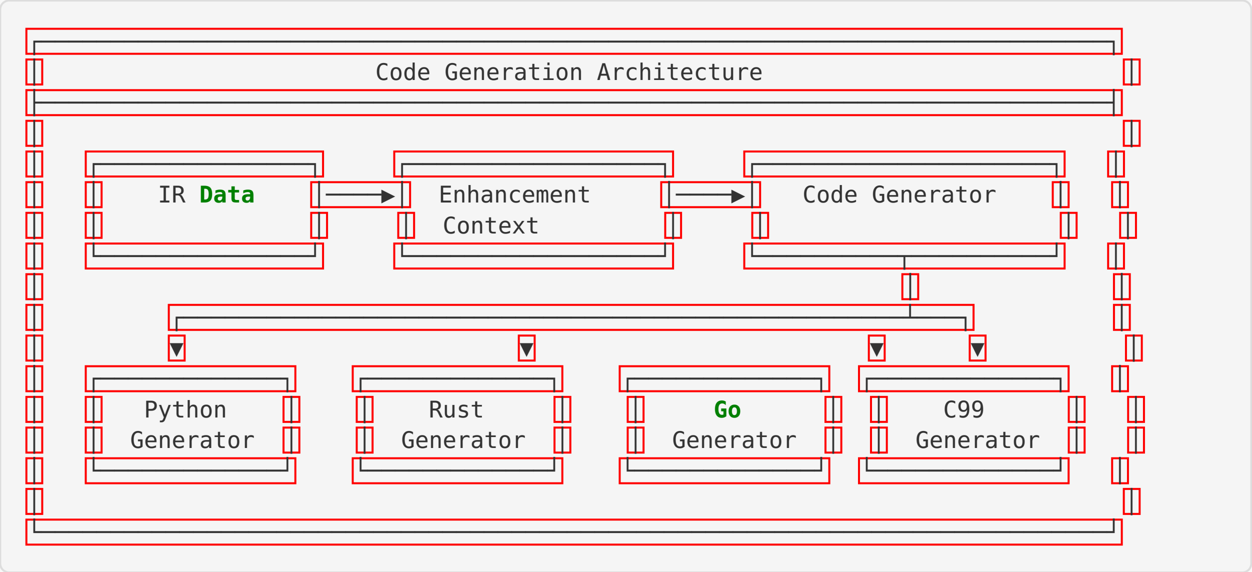
Phase 2 Implementation Documentation

Generated: 2026-01-28
Phase: 2 (Basic Code Generation)
Status: Complete

Overview

Phase 2 of STUNIR implements actual function body generation across 4 core languages: **Python**, **Rust**, **Go**, and **C99**. This replaces the previous stub generation with real executable code.

Architecture



Components

1. Statement Translator (`statement_translator.py`)

Base class for translating IR statements to target language code.

Supported Statement Types:
- `var_decl` - Variable declarations
- `const_decl` - Constant declarations
- `assign` - Simple assignments
- `return` - Return statements
- `expr_stmt` - Expression statements
- `call` - Function calls
- `break` / `continue` - Loop control

Abstract Methods:

```
translate_variable_declaration(var_name, var_type, init_value, mutable, indent)
translate_assignment(target, value, indent)
translate_compound_assignment(target, op, value, indent)
translate_return(value, indent)
translate_expression_statement(expr, indent)
```

2. Expression Translator (`expression_translator.py`)

Base class for translating IR expressions to target language code.

Supported Expression Types:
- `literal` - Numbers, strings, booleans
- `var` - Variable references
- `binary` - Arithmetic, comparison, logical operations
- `unary` - Negation, not
- `call` - Function/method calls
- `index` - Array indexing
- `member` - Member access
- `cast` - Type casts
- `ternary` - Conditional expressions

Abstract Methods:

```
translate_literal(value, lit_type)
translate_variable(name, var_type)
translate_binary_op(left, op, right)
translate_unary_op(op, operand)
translate_function_call(func_name, args, receiver)
```

3. Language-Specific Generators

Each language has a complete code generator that combines statement and expression translators.

Statement Translation

Variable Declarations

IR Type	Python	Rust	Go	C99
Mutable	`x: int = 0`	`let mut x: i32 = 0;`	`x := 0`	`int32_t x = 0;`
Immutable	`x: Final[int] = 0`	`let x: i32 = 0;`	`const x int32 = 0`	`const int32_t x = 0;`

IR Example:

```json
{
  "type": "var_decl",
  "var_name": "count",
  "var_type": "i32",
  "init": 0,
  "mutable": true
}
```

Assignments

Target	Python	Rust	Go	C99
Simple	`x = 42`	`x = 42;`	`x = 42`	`x = 42;`
Compound	`x += 1`	`x += 1;`	`x += 1`	`x += 1;`

Return Statements

Target	With Value	Void Return
Python	`return a + b`	`return`
Rust	`return a + b;`	`return;`
Go	`return a + b`	`return`
C99	`return a + b;`	`return;`

Expression Translation

Arithmetic Operations

All languages support the same arithmetic operators with minor syntax differences:

Operation	Python	Rust	Go	C99
Addition	`a + b`	`a + b`	`a + b`	`a + b`
Subtraction	`a - b`	`a - b`	`a - b`	`a - b`
Multiplication	`a * b`	`a * b`	`a * b`	`a * b`
Division	`a / b`	`a / b`	`a / b`	`a / b`
Modulo	`a % b`	`a % b`	`a % b`	`a % b`

Comparison Operations

Operation	All Languages
Equal	`a == b`
Not Equal	`a != b`
Less Than	`a < b`
Less Equal	`a <= b`
Greater Than	`a > b`
Greater Equal	`a >= b`

Logical Operations

Operation	Python	Rust/Go/C99
AND	`a and b`	`a && b`
OR	`a or b`	`a || b`
NOT	`not a`	`!a`

Type Mapping

Numeric Types

IR Type	Python	Rust	Go	C99
`i8`	`int`	`i8`	`int8`	`int8_t`
`i16`	`int`	`i16`	`int16`	`int16_t`
`i32`	`int`	`i32`	`int32`	`int32_t`
`i64`	`int`	`i64`	`int64`	`int64_t`
`u8`	`int`	`u8`	`uint8`	`uint8_t`
`u16`	`int`	`u16`	`uint16`	`uint16_t`
`u32`	`int`	`u32`	`uint32`	`uint32_t`
`u64`	`int`	`u64`	`uint64`	`uint64_t`
`f32`	`float`	`f32`	`float32`	`float`
`f64`	`float`	`f64`	`float64`	`double`

Other Types

IR Type	Python	Rust	Go	C99
`bool`	`bool`	`bool`	`bool`	`bool`
`string`	`str`	`String`	`string`	`char*`
`void`	`None`	`()`	``	`void`
`char`	`str`	`char`	`rune`	`char`

Usage

Basic Usage

```python
from tools.codegen import get_generator

# Create generator for target language
generator = get_generator('python')

# Define function IR
func_ir = {
    'name': 'add',
    'params': [
        {'name': 'a', 'type': 'i32'},
        {'name': 'b', 'type': 'i32'}
    ],
    'return_type': 'i32',
    'body': [
        {
            'type': 'return',
            'value': {
                'type': 'binary',
                'op': '+',
                'left': {'type': 'var', 'name': 'a'},
                'right': {'type': 'var', 'name': 'b'}
            }
        }
    ]
}

# Generate code
code = generator.generate_function(func_ir)
print(code)
```

Output (Python):

```python
def add(a: int, b: int) -> int:
    return (a + b)
```

With Enhancement Context

```python
from tools.codegen import get_generator
from tools.integration import EnhancementContext

# Create context with type information
context = EnhancementContext(original_ir=ir_data)

# Create generator with context
generator = get_generator('rust', enhancement_context=context)

# Generate code with enhanced type information
code = generator.generate_function(func_ir)
```

Module Generation

```python
module_ir = {
    'ir_module': 'math_utils',
    'ir_functions': [...],
    'ir_types': [...],
    'ir_exports': [...]
}

# Generate full module
code = generator.generate_module(module_ir)
```

Language-Specific Patterns

Python Generator

- Uses PEP 484 type hints
- Generates `__all__` export list
- Uses `Optional[]` for pointer types
- Generates `Final[]` for constants
- Idiomatic Python patterns (snake_case)

Rust Generator

- Handles `let mut` vs `let` for mutability
- Generates proper lifetime annotations (basic)
- Uses `.to_string()` for String literals
- Generates `#[derive(...)]` attributes
- Supports `pub` visibility

Go Generator

- Uses tabs for indentation (Go convention)
- Generates `package` declarations
- Uses short declaration (`:=`) when appropriate
- Capitalizes exported identifiers
- Generates `type Struct struct {}` patterns

C99 Generator

- Uses `<stdint.h>` fixed-width types
- Generates both `.c` and `.h` files
- Uses header guards (`#ifndef`)
- Proper `const` handling
- Designated initializers for structs

Extending to New Languages

To add a new language:

1. **Create Expression Translator:**

```python
class NewLangExpressionTranslator(ExpressionTranslator):
    TARGET = 'newlang'

    def translate_literal(self, value, lit_type):
        # Implement...

    def translate_binary_op(self, left, op, right):
        # Implement...
```

1. **Create Statement Translator:**

```python
class NewLangStatementTranslator(StatementTranslator):
    TARGET = 'newlang'
    STATEMENT_TERMINATOR = ';'

    def translate_variable_declaration(self, ...):
        # Implement...
```

1. **Create Code Generator:**

```python
class NewLangCodeGenerator:
    TARGET = 'newlang'
    FILE_EXTENSION = 'xyz'

    def __init__(self, enhancement_context=None):
        self.expr_translator = NewLangExpressionTranslator(enhancement_context)
        self.stmt_translator = NewLangStatementTranslator(enhancement_context)
        self.stmt_translator.set_expression_translator(self.expr_translator)
```

1. **Register in `__init__.py`:**

```python
from .newlang_generator import NewLangCodeGenerator

# Add to get_generator() dispatch
```

Testing

Running Tests

```
# Basic code generation tests
python -m pytest tests/codegen/test_basic_codegen.py -v

# Integration tests
python -m pytest tests/integration/test_phase2_integration.py -v

# All Phase 2 tests
python -m pytest tests/codegen tests/integration -v
```

Test Coverage

- 64 unit tests for basic code generation
- 19 integration tests for end-to-end validation
- Cross-language comparison tests
- Error handling tests
- EnhancementContext integration tests

Files Created

File	Description
`tools/codegen/__init__.py`	Package exports
`tools/codegen/statement_translator.py`	Statement translation base
`tools/codegen/expression_translator.py`	Expression translation base
`tools/codegen/python_generator.py`	Python code generator
`tools/codegen/rust_generator.py`	Rust code generator
`tools/codegen/go_generator.py`	Go code generator
`tools/codegen/c99_generator.py`	C99 code generator
`tests/codegen/test_basic_codegen.py`	Unit tests
`tests/integration/test_phase2_integration.py`	Integration tests

Next Steps (Phase 3)

Phase 3 will add:
- Control flow statements (if/else, while, for)
- Complex expressions (array access, struct fields)
- Advanced type handling
- Optimization hints integration
- More idiomatic patterns per language