STUNIR v0.9.0 - Additional Control Flow Features Design

Version: v0.9.0

Date: 2026-01-31

Status: In Progress

---

1. Overview

This document defines the design and implementation of additional control flow features for STUNIR v0.9.0:
- **break** statements (exit loop early)
- **continue** statements (skip to next iteration)
- **switch/case** statements (multi-way branching)

These features expand STUNIR's control flow capabilities beyond the existing if/else, while, and for loop support.

---

2. break Statement Design

2.1 Semantics

The `break` statement causes immediate exit from the innermost enclosing loop (while or for).

Behavior:
- Exits the current loop immediately
- Transfers control to the statement following the loop
- Only valid inside while/for loops
- Affects only the innermost loop in nested structures

2.2 Spec Format (JSON)

```
{
  "type": "break"
}
```

2.3 IR Representation

```
{
  "op": "break"
}
```

2.4 C Code Generation

```
break;
```

2.5 Example

Spec:

```
{
  "type": "while",
  "condition": "i < 10",
  "body": [
    {
      "type": "if",
      "condition": "i == 5",
      "then_block": [
        {"type": "break"}
      ]
    },
    {"type": "assign", "target": "i", "value": "i + 1"}
  ]
}
```

Generated C:

```
while (i < 10) {
  if (i == 5) {
    break;
  }
  i = i + 1;
}
```

---

3. continue Statement Design

3.1 Semantics

The `continue` statement skips the remaining statements in the current loop iteration and jumps to the loop condition check.

Behavior:
- Skips remaining statements in current iteration
- For `while`: jumps to condition check
- For `for`: executes increment, then checks condition
- Only valid inside while/for loops
- Affects only the innermost loop in nested structures

3.2 Spec Format (JSON)

```
{
  "type": "continue"
}
```

3.3 IR Representation

```
{
  "op": "continue"
}
```

3.4 C Code Generation

```
continue;
```

3.5 Example

Spec:

```
{
  "type": "for",
  "init": "i = 0",
  "condition": "i < 10",
  "increment": "i = i + 1",
  "body": [
    {
      "type": "if",
      "condition": "i % 2 == 0",
      "then_block": [
        {"type": "continue"}
      ]
    },
    {"type": "assign", "target": "sum", "value": "sum + i"}
  ]
}
```

Generated C:

```
for (i = 0; i < 10; i = i + 1) {
  if (i % 2 == 0) {
    continue;
  }
  sum = sum + i;
}
```

---

4. switch/case Statement Design

4.1 Semantics

The `switch` statement provides multi-way branching based on the value of an integer expression.

Behavior:
- Evaluates the switch expression once
- Compares against each case value in order
- Executes the first matching case block
- Falls through to next case unless `break` is used

- Executes `default` block if no cases match (optional)
- Only integer expressions are supported initially

4.2 Spec Format (JSON)

```
{
  "type": "switch",
  "expr": "x",
  "cases": [
    {
      "value": 1,
      "body": [
        {"type": "assign", "target": "result", "value": "100"},
        {"type": "break"}
      ]
    },
    {
      "value": 2,
      "body": [
        {"type": "assign", "target": "result", "value": "200"},
        {"type": "break"}
      ]
    }
  ],
  "default": [
    {"type": "assign", "target": "result", "value": "0"}
  ]
}
```

4.3 IR Representation

```
{
  "op": "switch",
  "expr": "x",
  "cases": [
    {
      "value": 1,
      "body": [
        {"op": "assign", "target": "result", "value": "100"},
        {"op": "break"}
      ]
    },
    {
      "value": 2,
      "body": [
        {"op": "assign", "target": "result", "value": "200"},
        {"op": "break"}
      ]
    }
  ],
  "default": [
    {"op": "assign", "target": "result", "value": "0"}
  ]
}
```

4.4 C Code Generation

```
switch (x) {
  case 1:
    result = 100;
    break;
  case 2:
    result = 200;
    break;
  default:
    result = 0;
}
```

4.5 Fall-through Behavior

With explicit fall-through:

```
{
  "type": "switch",
  "expr": "x",
  "cases": [
    {
      "value": 1,
      "body": [
        {"type": "assign", "target": "a", "value": "1"}
      ]
    },
    {
      "value": 2,
      "body": [
        {"type": "assign", "target": "b", "value": "2"},
        {"type": "break"}
      ]
    }
  ]
}
```

Generated C:

```
switch (x) {
  case 1:
    a = 1;
    /* fall through */
  case 2:
    b = 2;
    break;
}
```

---

5. Schema Updates

5.1 Spec Schema Updates

Add to statement type enum:
- `"break"` - break statement

- `"continue"` - continue statement
- `"switch"` - switch/case statement

5.2 IR Schema Updates

Add to op enum:
- `"break"` - break operation
- `"continue"` - continue operation
- `"switch"` - switch/case operation

---

6. Edge Cases and Validation

6.1 break/continue Outside Loops

Invalid:

```
{
  "type": "if",
  "condition": "x > 0",
  "then_block": [
    {"type": "break"}  // ERROR: break outside loop
  ]
}
```

Validation:
- Implementations SHOULD warn or error if break/continue appears outside a loop
- Generated C code will still be syntactically valid (C compiler will catch the error)

6.2 Nested Loops

Valid:

```
{
  "type": "while",
  "condition": "i < 10",
  "body": [
    {
      "type": "for",
      "init": "j = 0",
      "condition": "j < 5",
      "increment": "j = j + 1",
      "body": [
        {
          "type": "if",
          "condition": "j == 2",
          "then_block": [
            {"type": "break"}  // Breaks from inner for loop only
          ]
        }
      ]
    },
    {"type": "assign", "target": "i", "value": "i + 1"}
  ]
}
```

6.3 Switch with No Cases

Valid but unusual:

```
{
  "type": "switch",
  "expr": "x",
  "cases": [],
  "default": [
    {"type": "assign", "target": "result", "value": "0"}
  ]
}
```

Generated C:

```
switch (x) {
  default:
    result = 0;
}
```

6.4 Switch with Duplicate Case Values

Invalid (implementation should detect):

```
{
  "type": "switch",
  "expr": "x",
  "cases": [
    {"value": 1, "body": [...]},
    {"value": 1, "body": [...]}  // ERROR: duplicate case
  ]
}
```

---

7. Implementation Priorities

7.1 Phase 1: break/continue

1. Python reference implementation
2. Rust implementation
3. SPARK implementation
4. Testing across all pipelines

7.2 Phase 2: switch/case

1. Python reference implementation
2. Rust implementation
3. SPARK implementation
4. Testing across all pipelines

7.3 Phase 3: Comprehensive Testing

1. Single-level break/continue tests
2. Nested loop break/continue tests

3. Simple switch/case tests
4. Complex switch/case tests (multiple cases, fall-through, default)
5. Combined tests (break in switch, nested switch, etc.)

---

8. C Code Generation Guidelines

8.1 Indentation

- Maintain consistent 2-space indentation
- Proper indentation for nested structures

8.2 Comments

- Add `/* fall through */` comment for cases without break
- No special comments needed for break/continue

8.3 Formatting

```
// break
break;

// continue
continue;

// switch with proper formatting
switch (expr) {
  case value1:
    statement1;
    break;
  case value2:
    statement2;
    statement3;
    break;
  default:
    default_statement;
}
```

---

9. SPARK Considerations

9.1 Bounded Recursion

- switch/case adds one more level of nesting
- Must fit within Max_Recursion_Depth = 5
- Count switch as a control flow level

9.2 String Builder

- Use STUNIR_String_Builder for efficient code generation
- Pre-allocate buffer space for case labels

9.3 Array Bounds

- Max cases per switch: Limited by Step_Array bounds

- Implementation should handle up to 50 cases comfortably

---

10. Testing Strategy

10.1 Unit Tests

- break_while.json - break in while loop
- continue_while.json - continue in while loop
- break_for.json - break in for loop
- continue_for.json - continue in for loop
- break_nested.json - break in nested loops
- switch_simple.json - simple switch with 2-3 cases
- switch_multiple.json - switch with many cases
- switch_default.json - switch with default case
- switch_fallthrough.json - switch with fall-through
- switch_nested.json - nested switch statements

10.2 Integration Tests

- combined.json - mix of all features
- stress_test.json - deeply nested combinations

10.3 Cross-Pipeline Validation

- Python vs Rust output comparison
- Python vs SPARK output comparison
- Rust vs SPARK output comparison
- Verify functional equivalence

---

11. Documentation Requirements

11.1 User Documentation

- Update STUNIR spec documentation
- Add examples to user guide
- Update language reference

11.2 Developer Documentation

- Update IR schema documentation
- Add implementation notes
- Document SPARK-specific considerations

11.3 Release Notes

- Feature description
- Breaking changes (none expected)
- Migration guide (if needed)

---

12. Timeline

- Phase 1 (break/continue): 3-4 days
- Phase 2 (switch/case): 4-5 days
- Phase 3 (Testing): 2-3 days
- Phase 4 (Documentation): 2 days
- **Total: ~2 weeks**

---

13. Success Criteria

- ✅ All pipelines generate correct IR for break/continue
- ✅ All pipelines generate correct IR for switch/case
- ✅ Generated C code compiles without errors
- ✅ Generated C code produces correct runtime behavior
- ✅ SPARK maintains 100% status (no formal verification regressions)
- ✅ All test specs pass across all pipelines
- ✅ Cross-pipeline outputs are functionally equivalent
- ✅ Documentation is complete and accurate

End of Design Document