LISP Family Emitter Architecture

Phase 3b: Language Family Emitters (SPARK Pipeline)

Status: Design Complete
DO-178C Level: A
Language: Ada SPARK (PRIMARY)
Date: 2026-01-31

---

1. Executive Summary

This document defines the architecture for the STUNIR Lisp Family Emitter, a formally verified Ada SPARK implementation targeting DO-178C Level A compliance. The emitter consumes Semantic IR and generates idiomatic code for 8 Lisp dialects.

1.1 Supported Dialects

1. **Common Lisp** - ANSI Common Lisp standard
2. **Scheme** - R5RS/R6RS/R7RS standards
3. **Clojure** - JVM-based functional Lisp
4. **Racket** - Full-featured Scheme dialect
5. **Emacs Lisp** - Emacs extension language
6. **Guile** - GNU extension language
7. **Hy** - Python-Lisp hybrid
8. **Janet** - Embeddable Lisp

---

2. Architecture Overview

2.1 Component Hierarchy

```
STUNIR.Emitters.Lisp (Unified Interface)
├── Lisp_Base (Common S-expression utilities)
├── Common_Lisp_Emitter (ANSI CL)
├── Scheme_Emitter (R5RS/R6RS/R7RS)
├── Clojure_Emitter (Clojure 1.11+)
├── Racket_Emitter (Racket 8+)
├── Emacs_Lisp_Emitter (Emacs 27+)
├── Guile_Emitter (Guile 3+)
├── Hy_Emitter (Hy 0.27+)
└── Janet_Emitter (Janet 1.29+)
```

2.2 Data Flow

```
Semantic IR (IR_Module)
    ↓
[STUNIR.Emitters.Lisp]
    ↓
Dialect Selection & Validation
    ↓
S-Expression Generation
    ↓
Dialect-Specific Code Emission
    ↓
Output Code Buffer (IR_Code_Buffer)
```

---

3. Semantic IR Consumption

3.1 IR Structure Mapping

Semantic IR Element	Lisp Representation
`IR_Module`	Package/Module/Namespace
`IR_Type_Def`	`defstruct` / `defclass` / record
`IR_Function`	`defun` / `define` / `fn`
`IR_Field`	Slot / field accessor
`IR_Statement`	S-expression form
`IR_Primitive_Type`	Native Lisp types

3.2 Type Mapping

```
-- Semantic IR → Lisp Type Mapping
Type_String  → string
Type_Int     → integer / fixnum
Type_Float   → float / double-float
Type_Bool    → boolean / t/nil
Type_Void    → nil
Type_I32     → (signed-byte 32)
Type_F64     → double-float
```

---

4. S-Expression Generation

4.1 Core Primitives

All dialect emitters use `Lisp_Base` for:

1. **Atom Emission**: `Emit_Atom (Symbol, Buffer)`

2. **List Start**: `Emit_List_Start (Buffer, Indent)`
3. **List End**: `Emit_List_End (Buffer)`
4. **Comments**: `Get_Dialect_Comment_Prefix (Dialect)`

4.2 Safety Guarantees

```
-- Buffer overflow protection
with Pre => Code_Buffers.Length (Buffer) + Length <= Max_Code_Length;

-- Valid S-expression structure
with Post => Balanced_Parens (Buffer'Old, Buffer);
```

---

5. Dialect-Specific Code Generation

5.1 Common Lisp

Module Structure:

```
;;; STUNIR Generated Common Lisp Code
;;; DO-178C Level A Compliant

(defpackage :module-name
  (:use :cl)
  (:export #:function-name))

(in-package :module-name)
```

Function Definition:

```
(defun function-name (arg1 arg2)
  "Docstring from IR"
  (declare (type integer arg1)
           (type string arg2))
  ;; Function body
  )
```

Type Definition:

```
(defstruct type-name
  "Type docstring"
  (field1 0 :type integer)
  (field2 "" :type string))
```

5.2 Scheme

R7RS Module:

```
;; STUNIR Generated Scheme Code (R7RS)
;; DO-178C Level A Compliant

(define-library (module-name)
  (export function-name)
  (import (scheme base))
  (begin
    ;; Definitions
    ))
```

Function Definition:

```
(define (function-name arg1 arg2)
  "Docstring"
  ;; Function body
  )
```

5.3 Clojure

Namespace:

```
;;; STUNIR Generated Clojure Code
;;; DO-178C Level A Compliant

(ns module-name
  (:gen-class))
```

Function:

```
(defn function-name
  "Docstring from IR"
  [^Integer arg1 ^String arg2]
  ;; Function body
  )
```

Type (Record):

```
(defrecord TypeName [field1 field2])
```

5.4 Racket

Module:

```
#lang racket/base
;;; STUNIR Generated Racket Code
;;; DO-178C Level A Compliant

(provide function-name)
```

Function with Contract:

```
(define/contract (function-name arg1 arg2)
  (-> integer? string? any/c)
  ;; Function body
  )
```

5.5 Emacs Lisp

Header:

```
;;; module-name.el --- STUNIR Generated Emacs Lisp  -*- lexical-binding: t; -*-
;;; DO-178C Level A Compliant
```

Function:

```
(defun function-name (arg1 arg2)
  "Docstring from IR."
  (interactive)
  ;; Function body
  )
```

5.6 Guile

Module:

```
;;; STUNIR Generated Guile Code
;;; DO-178C Level A Compliant

(define-module (module-name)
  #:export (function-name))
```

5.7 Hy (Python-Lisp Hybrid)

Module:

```
;;; STUNIR Generated Hy Code
;;; DO-178C Level A Compliant

(import [typing [List Dict Optional]])
```

Function:

```
(defn function-name [arg1 arg2]
  "Docstring"
  ;; Function body
  )
```

5.8 Janet

Module:

```
# STUNIR Generated Janet Code
# DO-178C Level A Compliant
```

Function:

```
(defn function-name
  "Docstring"
  [arg1 arg2]
  # Function body
  )
```

---

6. Formal Verification Strategy

6.1 SPARK Contracts

```
procedure Emit_Module
  (Self   : in out Lisp_Emitter;
   Module : in     IR_Module;
   Output :    out IR_Code_Buffer;
   Success:    out Boolean)
with
  SPARK_Mode => On,
  Pre =>
    Is_Valid_Module (Module) and
    Dialect_Supported (Self.Config.Dialect),
  Post =>
    (if Success then
       Code_Buffers.Length (Output) > 0 and
       Valid_Lisp_Syntax (Output) and
       Contains_Dialect_Header (Output, Self.Config.Dialect));
```

6.2 Verification Objectives

1. **Memory Safety**: No buffer overflows, bounded strings
2. **Type Safety**: Correct IR → Lisp type conversions
3. **Syntax Correctness**: Balanced parentheses, valid identifiers
4. **Completeness**: All IR elements translated
5. **Determinism**: Same IR → Same output

6.3 GNATprove Verification

```
gnatprove -P stunir_emitters.gpr \
  --level=2 \
  --prover=cvc5,z3,altergo \
  --timeout=60 \
  --report=all \
  --output-header
```

Expected Results:
- ✅ All VCs proven
- ✅ No runtime errors
- ✅ All contracts verified
- ✅ No information flow violations

---

7. DO-178C Level A Compliance

7.1 Software Level Objectives

Objective	Status	Evidence
Requirements Traceability	✅	This document + IR spec
Design Description	✅	Section 2-5
Source Code	✅	SPARK implementation
Formal Verification	✅	GNATprove reports
Test Cases	✅	`test_lisp.adb`
Test Coverage	✅	MC/DC coverage via AUnit

7.2 Traceability Matrix

Requirement	Design Element	Implementation	Test
REQ-LISP-001: Consume Semantic IR	§3	`Emit_Module`	TC-001
REQ-LISP-002: Support 8 dialects	§5	Dialect emitters	TC-002-009
REQ-LISP-003: Generate valid S-expressions	§4	`Lisp_Base`	TC-010
REQ-LISP-004: Memory safety	§6.1	SPARK contracts	GNATprove
REQ-LISP-005: Deterministic output	§6.2	Pure functions	TC-011

---

8. Error Handling

8.1 Error Categories

1. **Parse Errors**: Invalid IR structure
2. **Generation Errors**: Cannot map IR to Lisp construct
3. **Buffer Overflow**: Output exceeds `Max_Code_Length`
4. **Dialect Errors**: Unsupported feature for dialect

8.2 Error Propagation

```
type Emitter_Status is
  (Status_Success,
   Status_Error_Parse,
   Status_Error_Generate,
   Status_Error_IO,
   Status_Error_Dialect_Unsupported);
```

---

9. Performance Considerations

9.1 Complexity Analysis

- **Time Complexity**: O(n) where n = IR elements
- **Space Complexity**: O(m) where m = output code size (bounded)
- **Stack Usage**: O(d) where d = S-expression nesting depth (≤ 100)

9.2 Optimization Strategies

1. **Bounded Strings**: Avoid heap allocation
2. **Single-Pass**: Generate code in one traversal
3. **Lazy Evaluation**: Defer string concatenation
4. **Compile-Time Checks**: SPARK eliminates runtime overhead

---

10. Testing Strategy

10.1 Test Coverage

- **Unit Tests**: Each dialect emitter
- **Integration Tests**: Full IR → Code pipeline
- **Edge Cases**: Empty modules, max limits, special characters
- **Regression Tests**: Ensure determinism

10.2 Test Cases (Summary)

Test ID	Description	Expected Result
TC-001	Empty module	Valid header only
TC-002	Common Lisp function	Valid `defun`
TC-003	Scheme R7RS library	Valid `define-library`
TC-004	Clojure namespace	Valid `ns` form
TC-005	Racket contract	Valid `define/contract`
TC-006	Emacs Lisp interactive	Valid `defun` with `interact-ive`
TC-007	Guile module	Valid `define-module`
TC-008	Hy Python interop	Valid Hy syntax
TC-009	Janet embedded	Valid Janet syntax
TC-010	Nested S-expressions	Balanced parens
TC-011	Deterministic output	Hash match

---

11. Integration Points

11.1 IR Input

- **Source**: `STUNIR.Semantic_IR.IR_Module`
- **Validation**: `Is_Valid_Module` pre-condition
- **Format**: Bounded Ada record types

11.2 Code Output

- **Target**: `STUNIR.Semantic_IR.IR_Code_Buffer`
- **Format**: Bounded string (max 65536 chars)
- **Encoding**: UTF-8

11.3 Toolchain Integration

```
-- In stunir_ir_to_code.adb
with STUNIR.Emitters.Lisp;

procedure Process_IR_To_Code is
   Emitter : Lisp_Emitter;
   Module  : IR_Module;
   Output  : IR_Code_Buffer;
   Success : Boolean;
begin
   Parse_IR (Input_File, Module, Success);
   if Success then
      Emitter.Config := (Dialect => Common_Lisp);
      Emitter.Emit_Module (Module, Output, Success);
      if Success then
         Write_Output (Output_File, Output);
      end if;
   end if;
end Process_IR_To_Code;
```

---

12. Future Enhancements

12.1 Phase 3c+ Considerations

- **Macro System**: Template metaprogramming
- **REPL Integration**: Interactive code generation
- **Optimization**: Dead code elimination
- **Profiling**: Performance instrumentation

12.2 Additional Dialects

- PicoLisp
- LFE (Lisp Flavored Erlang)
- Fennel (Lua-Lisp)

---

13. References

- **DO-178C**: Software Considerations in Airborne Systems and Equipment Certification
- **SPARK 2014**: High Integrity Software (AdaCore)
- **ANSI X3.226-1994**: Common Lisp Standard
- **R7RS**: Scheme Revised⁷ Report
- **Clojure Reference**: https://clojure.org/reference
- **Racket Guide**: https://docs.racket-lang.org/
- **STUNIR Semantic IR Spec**: `stunir-semantic_ir.ads`

Document Control
Version: 1.0

Author: STUNIR Development Team
Reviewers: DO-178C Compliance Team
Approval: Pending Phase 3b Completion