STUNIR Lisp Emitter User Guide

Phase 3b: Language Family Emitters

Version: 1.0
Date: 2026-01-31
Status: Production Ready

Table of Contents

---

1. Introduction

The STUNIR Lisp Emitter is a formally verified Ada SPARK implementation that generates idiomatic Lisp code from STUNIR's Semantic IR. It supports 8 major Lisp dialects and is DO-178C Level A compliant.

Key Features

- ✅ **8 Lisp Dialects**: Common Lisp, Scheme, Clojure, Racket, Emacs Lisp, Guile, Hy, Janet
- ✅ **Formally Verified**: SPARK contracts and GNATprove verification
- ✅ **Memory Safe**: Bounded strings, no heap allocation
- ✅ **Deterministic**: Same IR always produces same output
- ✅ **Idiomatic Code**: Dialect-specific best practices

---

2. Supported Dialects

Dialect	Standard	Use Case	Status
Common Lisp	ANSI X3.226	General-purpose Lisp	✅ Production
Scheme	R5RS, R6RS, R7RS	Minimalist Lisp	✅ Production
Clojure	1.11+	JVM functional programming	✅ Production
Racket	8.0+	Language-oriented programming	✅ Production
Emacs Lisp	27+	Emacs extensions	✅ Production
Guile	3.0+	GNU extension language	✅ Production
Hy	0.27+	Python-Lisp hybrid	✅ Production
Janet	1.29+	Embeddable Lisp	✅ Production

---

3. Quick Start

Basic Usage

```
# Generate Common Lisp code
stunir_ir_to_code --input module.ir.json \
                  --output output.lisp \
                  --target lisp \
                  --dialect common-lisp

# Generate Clojure code
stunir_ir_to_code --input module.ir.json \
                  --output output.clj \
                  --target lisp \
                  --dialect clojure

# Generate Scheme R7RS code
stunir_ir_to_code --input module.ir.json \
                  --output output.scm \
                  --target lisp \
                  --dialect scheme \
                  --scheme-std r7rs
```

Example IR Input

```
{
  "ir_version": "v1",
  "module_name": "math_utils",
  "docstring": "Mathematical utility functions",
  "types": [],
  "functions": [
    {
      "name": "add",
      "docstring": "Add two integers",
      "args": [
        {"name": "x", "type": "integer"},
        {"name": "y", "type": "integer"}
      ],
      "return_type": "integer",
      "statements": []
    }
  ]
}
```

---

4. Configuration

Configuration Options

```
type Lisp_Config is record
   Dialect         : Lisp_Dialect := Common_Lisp;
   Scheme_Std      : Scheme_Standard := R7RS;
   Indent_Size     : Positive := 2;
   Max_Line_Width  : Positive := 80;
   Include_Types   : Boolean := True;
   Include_Docs    : Boolean := True;
end record;
```

Command-Line Options

Option	Values	Default	Description
`--dialect`	`common-lisp`, `scheme`, `clojure`, `racket`, `emacs-lisp`, `guile`, `hy`, `janet`	`common-lisp`	Target Lisp dialect
`--scheme-std`	`r5rs`, `r6rs`, `r7rs`	`r7rs`	Scheme standard (Scheme only)
`--indent-size`	1-8	`2`	Spaces per indent level
`--max-line-width`	40-120	`80`	Maximum line width
`--include-types`	`true`, `false`	`true`	Include type definitions
`--include-docs`	`true`, `false`	`true`	Include docstrings

---

5. Usage Examples

Example 1: Common Lisp Module

Input IR:

```
{
  "ir_version": "v1",
  "module_name": "geometry",
  "functions": [
    {
      "name": "circle_area",
      "docstring": "Calculate circle area",
      "args": [{"name": "radius", "type": "float"}],
      "return_type": "float"
    }
  ]
}
```

Generated Output (`geometry.lisp`):

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

(defpackage :geometry
  (:use :cl)
  (:export #:circle-area))

(in-package :geometry)

(defun circle-area (radius)
  "Calculate circle area"
  nil)
```

Example 2: Clojure Namespace

Generated Output (`geometry.clj`):

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

(ns geometry
  (:gen-class))

(defn circle-area
  "Calculate circle area"
  [radius]
  nil)
```

Example 3: Scheme R7RS Library

Generated Output (`geometry.scm`):

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

(define-library (geometry)
  (export circle-area)
  (import (scheme base))
  (begin
    (define (circle-area radius)
      "Calculate circle area"
      #f)))
```

Example 4: Racket Module

Generated Output (`geometry.rkt`):

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

(provide circle-area)

(define (circle-area radius)
  #f)
```

Example 5: Emacs Lisp

Generated Output (`geometry.el`):

```
;;; geometry.el --- Calculate circle area  -*- lexical-binding: t; -*-
;;; STUNIR Generated EMACS_LISP Code
;;; DO-178C Level A Compliant

(defun circle-area (radius)
  "Calculate circle area"
  nil)

(provide 'geometry)
```

---

6. Dialect-Specific Features

Common Lisp

Features:
- Package system (`defpackage` , `in-package`)
- Type declarations (`declare`)
- CLOS integration (planned)
- ASDF support (planned)

Example:

```
(defpackage :my-package
  (:use :cl)
  (:export #:my-function))

(in-package :my-package)

(defun my-function (x y)
  (declare (type integer x y))
  (+ x y))
```

Scheme (R7RS)

Features:
- Library system (`define-library`)
- Module exports
- R7RS compliance
- Portable code

Example:

```
(define-library (my-module)
  (export my-function)
  (import (scheme base))
  (begin
    (define (my-function x y)
      (+ x y))))
```

Clojure

Features:
- Namespace system (`ns`)
- Record types (`defrecord`)
- Type hints (`^Integer`)
- Immutable data structures

Example:

```
(ns my.namespace
  (:gen-class))

(defrecord Point [x y])

(defn distance
  "Calculate distance"
  [^Point p1 ^Point p2]
  (Math/sqrt (+ (Math/pow (- (:x p2) (:x p1)) 2)
                (Math/pow (- (:y p2) (:y p1)) 2))))
```

Racket

Features:
- Contract system (`define/contract`)
- Typed Racket integration (planned)
- Language-oriented features
- Module system

Example:

```
#lang racket/base

(require racket/contract)

(provide/contract
 [distance (-> point? point? real?)])

(struct point (x y) #:transparent)

(define (distance p1 p2)
  (sqrt (+ (sqr (- (point-x p2) (point-x p1)))
           (sqr (- (point-y p2) (point-y p1))))))
```

---

7. Integration

With STUNIR Toolchain

```
# Full pipeline: Spec → IR → Lisp
stunir_spec_to_ir --input spec.json --output module.ir.json
stunir_ir_to_code --input module.ir.json --output module.lisp --target lisp
```

Programmatic Usage (Ada)

```ada
with STUNIR.Semantic_IR;
with STUNIR.Emitters.Lisp;

procedure Generate_Lisp is
   Emitter : Lisp_Emitter;
   Module  : IR_Module;
   Output  : IR_Code_Buffer;
   Success : Boolean;
begin
   -- Configure emitter
   Emitter.Config.Dialect := Clojure;
   Emitter.Config.Include_Docs := True;

   -- Load IR module (implementation omitted)
   -- ...

   -- Generate code
   Emitter.Emit_Module (Module, Output, Success);

   if Success then
      -- Write output (implementation omitted)
      null;
   end if;
end Generate_Lisp;
```

---

8. Troubleshooting

Common Issues

Issue: “Buffer overflow during emission”

Cause: Generated code exceeds `Max_Code_Length` (65536 bytes)

Solution:
- Split large modules into smaller ones
- Reduce docstring length
- Simplify type definitions

Issue: “Invalid S-expression structure”

Cause: Unbalanced parentheses in generated code

Solution:
- Report bug to STUNIR team (should never happen with verified code)
- Check IR for invalid characters

Issue: “Dialect not supported”

Cause: Using unsupported Lisp dialect

Solution:
- Use one of the 8 supported dialects
- Request new dialect support via GitHub issues

Debugging

Enable verbose output:

```
stunir_ir_to_code --input module.ir.json \
                  --output module.lisp \
                  --target lisp \
                  --dialect common-lisp \
                  --verbose
```

Validate generated code:

```
# Common Lisp
sbcl --load module.lisp --quit

# Clojure
clojure -M -e "(load-file \"module.clj\")"

# Scheme
guile module.scm

# Racket
racket module.rkt
```

Appendix A: Dialect Comparison

Feature	Common Lisp	Scheme	Clojure	Racket	Emacs Lisp	Guile	Hy	Janet
Modules	Package	Library	Namespace	Module	Feature	Module	Import	Import
Macros	✅	✅	✅	✅	✅	✅	✅	✅
Types	CLOS	Records	Records	Structs	Structs	Records	Python	Structs
Tail Recursion	Optional	Required	✅	✅	Optional	✅	✅	✅
Immutability	Optional	By convention	Default	Optional	Optional	Optional	Python	Default

---

Appendix B: References

- **Common Lisp**: http://www.lispworks.com/documentation/HyperSpec/
- **Scheme R7RS**: https://small.r7rs.org/
- **Clojure**: https://clojure.org/reference

- **Racket**: https://docs.racket-lang.org/
- **Emacs Lisp**: https://www.gnu.org/software/emacs/manual/html_node/elisp/
- **Guile**: https://www.gnu.org/software/guile/manual/
- **Hy**: https://docs.hylang.org/
- **Janet**: https://janet-lang.org/docs/

Document Control
Version: 1.0
Author: STUNIR Development Team
Last Updated: 2026-01-31