

STUNIR Language Specification

Version: 1.0

Status: Phase 6D Implementation

STUNIR (Deterministic Multi-Language Code Generation) is a domain-specific language for specifying compilers, code generators, and IR transformations.

Table of Contents

1. [Overview](#)
 2. [Lexical Structure](#)
 3. [Syntax](#)
 4. [Types](#)
 5. [Declarations](#)
 6. [Statements](#)
 7. [Expressions](#)
 8. [IR Definitions](#)
 9. [Target Definitions](#)
 10. [Pattern Matching](#)
-

Overview

STUNIR enables declarative specification of:

- **Intermediate Representations (IR):** Define IR node structures
- **Type Systems:** Define types and type mappings
- **Code Generators:** Define target-specific emission rules
- **Transformations:** Define IR-to-IR transformations

Design Goals

1. **Determinism:** Same input always produces same output
 2. **Self-hosting:** STUNIR can describe itself
 3. **Multi-target:** Generate code for multiple languages
 4. **Type-safe:** Strong static typing
-

Lexical Structure

Character Set

STUNIR source files are UTF-8 encoded.

Whitespace

Whitespace (spaces, tabs, newlines) is ignored except as token separators.

```
WHITESPACE = [ \t\r\n]+
```

Comments

Line comments start with `//` and extend to end of line.

Block comments start with `/*` and end with `*/`.

```
COMMENT_LINE = //[^\n]*
COMMENT_BLOCK = /\*([^\*]|\\*[/])*\*/
```

Keywords

module	import	from	export	as	
type	function	ir	target	const	
let	var	if	else	while	
for	in	match	return	emit	
child	op	true	false	null	
i8	i16	i32	i64		
u8	u16	u32	u64		
f32	f64	bool	string	void	any

Operators

+	-	*	/	%		// Arithmetic
==	!=	<	>	<=	>=	// Comparison
&&		!				// Logical
&		^	~	<<	>>	// Bitwise
=	+=	-=	*=	/=	%=	// Assignment
->	=>	?	:			// Special

Punctuation

```
( ) { } [ ]
, ; .
```

Identifiers

```
IDENTIFIER = [a-zA-Z_][a-zA-Z0-9_]*
```

Literals

```
INTEGER = [0-9]+
FLOAT = [0-9]+[.][0-9]+([eE][+-]?[0-9]+)?
STRING = "([^\n\r]|\\n|\\r)*"
```

Syntax

Program Structure

```
program ::= module_decl { declaration }

module_decl ::= 'module' IDENTIFIER ';'
              | 'module' IDENTIFIER '{' module_body '}'

module_body ::= { import_decl | export_decl | declaration }

import_decl ::= 'import' dotted_name [ 'as' IDENTIFIER ] ';'
              | 'from' dotted_name 'import' identifier_list ';'

export_decl ::= 'export' identifier_list ';'
              | 'export' '*' ';'

dotted_name ::= IDENTIFIER { '.' IDENTIFIER }
```

Declarations

```
declaration ::= type_def
              | function_def
              | ir_def
              | target_def
              | const_def
```

Types

Primitive Types

Type	Description	Size
i8	Signed 8-bit integer	1 byte
i16	Signed 16-bit integer	2 bytes
i32	Signed 32-bit integer	4 bytes
i64	Signed 64-bit integer	8 bytes
u8	Unsigned 8-bit integer	1 byte
u16	Unsigned 16-bit integer	2 bytes
u32	Unsigned 32-bit integer	4 bytes
u64	Unsigned 64-bit integer	8 bytes
f32	32-bit floating point	4 bytes
f64	64-bit floating point	8 bytes
bool	Boolean	1 byte
string	UTF-8 string	variable
void	No value	0 bytes
any	Dynamic type	variable

Type Expressions

```

type_expr ::= basic_type
          | IDENTIFIER [ '<' type_list '>' ]
          | '[' type_expr ']'
          | '(' type_list ')' '->' type_expr
          | type_expr '?'
          | type_expr '|' type_expr

```

Type Definitions

Type Alias

```

type MyInt = i32;
type StringList = [string];
type Predicate = (i32) -> bool;

```

Struct Type

```

type Point {
  x: i32;
  y: i32;
}

type Person {
  name: string;
  age: i32;
  email: string?; // Optional field
}

```

Variant Type (Sum Type)

```

type Option<T> {
  | Some(T)
  | None
}

type Result<T, E> {
  | Ok(T)
  | Err(E)
}

type Expr {
  | Literal(i32)
  | Binary(string, Expr, Expr)
  | Unary(string, Expr)
}

```

Declarations

Function Definition

```

function_def ::= 'function' IDENTIFIER '(' param_list ')' [ ':' type_expr ] block

param_list ::= [ param { ',' param } ]

param ::= IDENTIFIER ':' type_expr [ '=' expression ]

```

Examples:

```

function add(a: i32, b: i32): i32 {
  return a + b;
}

function greet(name: string, greeting: string = "Hello"): string {
  return greeting + ", " + name + "!";
}

function log(message: string): void {
  // ...
}

```

Constant Definition

```
const PI: f64 = 3.14159;
const MAX_SIZE: i32 = 1024;
```

Statements

Variable Declaration

```
var_decl ::= 'let' IDENTIFIER [ ':' type_expr ] '=' expression ';'
          | 'var' IDENTIFIER [ ':' type_expr ] [ '=' expression ] ';'

```

```
let x = 42;           // Immutable, type inferred
let y: i32 = 42;      // Immutable, explicit type
var z: i32;           // Mutable, uninitialized
var w: i32 = 0;       // Mutable, initialized
```

Control Flow

If Statement

```
if condition {
    // then branch
} else if other_condition {
    // else if branch
} else {
    // else branch
}
```

While Loop

```
while condition {
    // loop body
}
```

For Loop

```
for item in collection {
    // loop body
}
```

Match Statement

```
match value {
  0 => "zero",
  1 => "one",
  n => "other: " + n,
}

match expr {
  Literal(n) => n,
  Binary(op, left, right) => {
    // ...
  },
  _ => 0, // Default case
}
```

Return Statement

```
return value;
return; // For void functions
```

Emit Statement

Used in target definitions to emit code:

```
emit expression;
emit "literal code";
```

Expressions

Operator Precedence

Precedence	Operators	Associativity
1 (lowest)	<code>?:</code>	Right
2	<code>\ \ </code>	Left
3	<code>&&</code>	Left
4	<code>==</code> <code>!=</code>	Left
5	<code><</code> <code>></code> <code><=</code> <code>>=</code>	Left
6	<code>+</code> <code>-</code>	Left
7	<code>*</code> <code>/</code> <code>%</code>	Left
8	<code>!</code> <code>-</code> <code>~</code> (unary)	Right
9 (highest)	<code>.</code> <code>[]</code> <code>()</code>	Left

Literals

```
42           // Integer
3.14         // Float
"hello"      // String
true, false  // Boolean
null        // Null
[1, 2, 3]    // Array
{x: 1, y: 2} // Object
```

Arithmetic

```
a + b // Addition
a - b // Subtraction
a * b // Multiplication
a / b // Division
a % b // Modulo
-a    // Negation
```

Comparison

```
a == b // Equal
a != b // Not equal
a < b  // Less than
a > b  // Greater than
a <= b // Less than or equal
a >= b // Greater than or equal
```


Logical

```
a && b // And
a || b // Or
!a    // Not
```

Ternary

```
condition ? then_value : else_value
```

Member Access

```
object.field
array[index]
function(args)
```

IR Definitions

IR definitions declare intermediate representation nodes:

```
ir_def ::= 'ir' IDENTIFIER [ '<' identifier_list '>' ] '{' ir_body '}'

ir_body ::= { ir_field | ir_child | ir_op }

ir_field ::= IDENTIFIER ':' type_expr ';'
ir_child ::= 'child' IDENTIFIER ':' type_expr ';'
ir_op ::= 'op' IDENTIFIER '(' param_list ')' [ ':' type_expr ] ';'

```

Example:

```
ir BinaryOp {
    op: string;           // Field
    child left: Expr;     // Child node
    child right: Expr;    // Child node
    op evaluate(): i32;    // Operation
}

ir Function {
    name: string;
    params: [string];
    return_type: Type;
    child body: Block;
}
```

Target Definitions

Target definitions specify code generation for a target language:

```

target_def ::= 'target' IDENTIFIER '{' target_body '}'

target_body ::= { target_option | emit_rule }

target_option ::= IDENTIFIER ':' expression ';'
emit_rule ::= 'emit' IDENTIFIER '(' param_list ')' block

```

Example:

```

target Python {
    extension: ".py";
    indent: "    ";

    emit BinaryOp(node: BinaryOp) {
        emit node.left;
        emit " " + node.op + " ";
        emit node.right;
    }

    emit Function(func: Function) {
        emit "def " + func.name + "(";
        for i in 0..func.params.length {
            if i > 0 {
                emit ", ";
            }
            emit func.params[i];
        }
        emit "):\n";
        emit func.body;
    }
}

```

Pattern Matching

Patterns are used in match statements:

```

pattern ::= IDENTIFIER
         | literal
         | IDENTIFIER '(' pattern_list ')'
         | '[' pattern_list ']'
         | '-'

pattern_list ::= [ pattern { ',' pattern } ]

```

Examples:

```
match value {
  0 => "zero",           // Literal pattern
  x => "value: " + x,    // Variable pattern
  _ => "default",        // Wildcard pattern
}

match expr {
  Literal(n) => n,           // Constructor pattern
  Binary("+", a, b) => a + b, // Nested pattern
  [first, second] => first + second, // Array pattern
}
```

Complete Example

```

module calculator;

// Type definitions
type Operator {
  | Add
  | Sub
  | Mul
  | Div
}

// IR node definitions
ir NumberExpr {
  value: i32;
}

ir BinaryExpr {
  op: Operator;
  child left: Expr;
  child right: Expr;
  op evaluate(): i32;
}

type Expr {
  | Num(NumberExpr)
  | Bin(BinaryExpr)
}

// Evaluator
function evaluate(expr: Expr): i32 {
  match expr {
    Num(n) => n.value,
    Bin(b) => {
      let l = evaluate(b.left);
      let r = evaluate(b.right);
      match b.op {
        Add => l + r,
        Sub => l - r,
        Mul => l * r,
        Div => l / r,
      }
    }
  }
}

// Code generation target
target C {
  extension: ".c";

  emit NumberExpr(n: NumberExpr) {
    emit n.value;
  }

  emit BinaryExpr(b: BinaryExpr) {
    emit "(";
    emit b.left;
    match b.op {
      Add => emit " + ",
      Sub => emit " - ",
      Mul => emit " * ",
      Div => emit " / ",
    }
    emit b.right;
  }
}

```

```
        emit " ";
    }
}

// Main function
function main(): i32 {
    let expr = Bin({
        op: Add,
        left: Num({value: 2}),
        right: Bin({
            op: Mul,
            left: Num({value: 3}),
            right: Num({value: 4})
        })
    });

    return evaluate(expr); // Returns 14
}
```

References

- [Bootstrap Package](#) (../bootstrap/README.md)
- [Example Programs](#) (../examples/stunir/)
- [Grammar IR](#) (../ir/grammar/README.md)
- [Parser Generation](#) (../ir/parser/README.md)
- [Lexer Generation](#) (../ir/lexer/README.md)