SLIECE User Guide

Summary

Introduction

Overview of SLIECE

SLIECE is a minimalist, expression-oriented programming language designed for clarity, purity, and just a little bit of mischief.

It takes its name from the fusion of slice (as in data slicing, precision, and computation) and lie (a nod to elegance, abstraction, and perhaps the small illusions of programming itself).

SLIECE encourages developers to think functionally — every operation returns a value, and functions are the foundation of all computation.

Its syntax is compact and readable, making it an ideal environment for both experimentation and structured problem-solving.

SLIECE programs are stored in files with the .lie extension, and each program must define a main function — the single entry point where execution begins.

Whether you're performing arithmetic, manipulating lists, or exploring abstract computation, SLIECE invites you to think clearly — even if the truth is sometimes a lie.

Philosophy and Design Goals

SLIECE is built on three principles:

Simplicity — Code should be easy to read, write, and reason about. Minimal syntax, predictable behavior, and no unnecessary keywords.

Purity — Functions are self-contained and deterministic. No side effects, no surprises — the same input always produces the same output.

Transparency through abstraction — While SLIECE hides certain implementation details for clarity, it never sacrifices understanding.

Every abstraction can be broken down into smaller, comprehensible pieces (or slices).

Together, these ideas make SLIECE both a teaching language and a serious platform for logical and mathematical exploration.

Inspirations and Context

SLIECE is inspired by modern minimalist languages like Zig, Rust, and Python, but its philosophy aligns more closely with Lisp and ML — emphasizing structure through simplicity.

The .lie file extension is a playful homage to the phrase “The cake is a lie”, a reminder that programming languages often hide complexity behind elegant syntax.

But make no mistake — beneath its humor, SLIECE is a language for those who seek clarity through constraint.

By forcing developers to define a main function and use pure expressions, SLIECE rewards discipline with precision.

Getting Started

Installation

To install SLICE, you have to build the language yourself.

Here the steps to do that :

- Download the source code from GitHub :

https://github.com/MYAMBO/GLaDOS.git

- Make sure you have Haskell and stack installed on your machine
- Run the following command :

Make

After that, you're supposed to have two binaries: glados-vm and glados-compiler

File Extension (.lie)

All the files for slices language must have the .lie extension to be compiled. Each .lie file can contain one or more declarations (functions, variables, or both). Moreover, the file extension have a Visual Studio Code associated to help you with the grammar adding colors.

Program Structure

A SLIECE program is built from **declarations** and **expressions**, according to the language grammar.

A minimal program structure looks like this:

```
$ main
-> "Hello, SLIECE!"
```

Here:

- $ main defines the **main function**, which is required in every program.
- -> "Hello, SLIECE!" is the **expression** returned by the function.

Example with variables and a custom function:

```
define Int32 number = 10

func add<Int32 a, Int32 b> => Int32

$ add
-> a + b

$ main
-> add(number, 5)
```

Explanation:

- define declares a variable.
- func declares a function signature.
- $ add defines the actual function logic.
- $ main runs automatically and produces the final program output.

The main Function Requirement

Every SLIECE program must contain a **main function**.

It acts as the **entry point** of execution — the first function that runs when you execute the program.

Example:

```
$                                                    main
->        "This        is        the        entry        point!"
```

If your program does not contain a main function, the compiler will throw an error:

```
[!] Function "main " is not defined.
```

The main function can call other functions, perform computations, and return any valid SLIECE expression.

Running a Program

Once your .lie file is ready, you can run it using the SLIECE interpreter and compiler.

Example:

File: main.lie

```
$ main
-> "Hello, SLIECE!"
```

Compile:

```
./glados-compiler main.lie
```

You will can run the compiled file with :

```
./glados-vm output.bin
```

Output:

```
Hello, SLIECE!
```

Language Basics

Variables and Declarations

Variables in SLIECE are declared using the define keyword, followed by:

- a **type specifier**
- a **variable name**
- and an **initial value**

Syntax

```
define       <type>       <identifier>       =       <expression>
```

Example

```
define            Int32            age            =            21
define          String          name          =          "Alice"
define Bool isStudent = True
```

Variables must always have a type. For example, the following will produce an error:

```
define                  number                  =                  10
```

Error: Missing type specifier in variable declaration.

Variables are immutable by default. Once defined, their value cannot be changed — a design choice made to improve program predictability and security.

Data Types

SLIECE provides a strong and explicit type system. Each variable, parameter, and return value must have a declared type.

Integer Types

Type	Size	Signed
Int8	8 bits	Yes
Int16	16 bits	Yes
Int32	32 bits	Yes
Int64	64 bits	Yes
UInt8	8 bits	No
UInt16	16 bits	No
UInt32	32 bits	No
UInt64	64 bits	No

Floating-Point Types

Type	Precision
Float	32-bit precision
Double	64-bit precision

Other Built-in Types

Type	Description
Bool	Boolean value (True or False)
String	Sequence of characters
List	Collection of elements of the same type

Example

```
define Float pi = 3.1415
define Bool active = True
define String greeting = "Hello!"
define List numbers = [1, 2, 3, 4, 5]
```

Expressions and Operators

Expressions are the heart of computation in SLIECE. They combine variables, literals, and operators to produce new values.

Arithmetic Operators

Operator	Meaning	Example	Result
+	Addition	2 + 3	5
-	Subtraction	5 - 2	3
*	Multiplication	4 * 2	8
/	Division	8 / 4	2

Comparison Operators

Operator	Meaning	Example	Result
==	Equal	5 == 5	True
!=	Not equal	3 != 2	True
<	Less than	1 < 2	True
>	Greater than	4 > 7	False
<=	Less or equal	2 <= 2	True
>=	Greater or equal	3 >= 5	False

Logical Operators

Operator	Meaning	Example	Result
and	Logical AND	True && False	False
not	Logical NOT	!True	False

Example

```
define          Int32          a          =          10
define          Int32           b          =           5
define Bool result = a > b && b != 0
```

Strings

Strings are fundamental container types in SLIECE. Strings are enclosed in double quotes ".

Example:

```
define String msg = "SLIECE says hello!"
$ main
-> msg
```

Strings can be concatenated using the + operator:

```
$ main
-> "The cake " + "is a lie."
```

Output:

```
The cake is a lie.
```

Comments and Formatting

SLIECE supports comments to improve code readability. Comments are ignored by the compiler and can appear anywhere in your code.

Single-line Comments

Start with # and continue until the end of the line:

```
#            This            is            a            comment
define    Int32    x    =    10        #    Inline    comment
```

Formatting Rules

Indentation is recommended for readability but not required for parsing.

Statements should be separated by newlines.

Each function definition must begin with $ on its own line, followed by a newline.

Example:

```
#       Function       that       doubles       a       number
func          double<Int32          x>          =>          Int32

$                                                          double
->                  x                  *                   2

$                                                            main
->                                                      double(21)
```

Output:

```
42
```

Functions

Function Declarations

Function declarations describe the **signature** of a function: its **name**, **parameters**, and **return type**.
They are defined using the func keyword.

Syntax

```
func <identifier> < <params> > => <type>
```

Example

```
func add<Int32 a, Int32 b> => Int32
```

Explanation:

- add is the function's name.
- It accepts two parameters: a and b, both of type Int32.
- It returns an Int32 value.

A declaration tells the compiler what a function looks like, even before it's defined.

Function Definitions

The definition provides the actual body (logic) of a declared function. In SLIECE, function definitions always start with a $ symbol, followed by the function name on a new line.

Syntax

```
$                                                    <identifier>
<NEWLINE>
<statement-list>
<NEWLINE>
->                                                   <expression>
```

Example

```
func      add<Int32      a,      Int32      b>      =>      Int32

$                                                          add
->                    a                    +                    b
```

Explanation:

`func add<...> => Int32` declares the function.

`$ add` begins its definition.

`-> a + b` returns the result of the addition.

The body can also include intermediate statements if necessary (following <statement-list>), though SLIECE currently favors concise, single-expression functions.

The -> Return Expression

The -> operator in SLIECE defines the return value of a function. It always appears at the end of a function body or statement chain.

Example

```
func greet<String name> => String

$ greet
-> "Hello " + name
```

When the function greet("Lexa") is called, it returns:

```
Hello Lexa
```

You can think of -> as a direct expression arrow — it evaluates the right-hand side and produces the final result.

Nested Example

x

Here, square(4) returns 16, demonstrating composition of pure functions.

Function Calls

To call a function, use its identifier followed by parentheses () enclosing arguments, separated by commas.

Syntax

```
<identifier>(<expression>      {      ,      <expression>      })
```

Example

```
func      add<Int32      a,      Int32      b>      =>      Int32

$                                                          add
->                    a                    +                    b

$                                                         main
->                           add(2,                         3)
```

Output:

```
5
```

Function calls can be nested inside expressions:

```
$                                                         main
->                  add(add(1,                 2),                 3)
```

Output:

```
6
```

You can also pass literal values, variables, or results of other functions as arguments.

Example: Pure Function Behavior

SLIECE functions are pure — they do not modify external state or rely on mutable data. This makes programs easier to reason about, test, and secure.

Example

```
func      sum<Int32      a,      Int32      b>      =>      Int32

$                                                          sum
->                    a                    +                 b

func          double<Int32          x>          =>          Int32

$                                                        double
->                          sum(x,                          x)

$                                                         main
->                                                  double(21)
```

Output:

42

Explanation:

Both sum and double are pure functions.

The same inputs always produce the same outputs (double(21) always returns 42).

There are no side effects — no variable changes, no global state modification.

Purity in SLIECE helps ensure that:

Program behavior is deterministic.

Compilation and optimization are safer and faster.

Security analysis becomes simpler and more reliable.

The main function

Purpose and Role

In SLIECE, the main function is the entry point of every program. When the compiler executes a .lie file, it automatically looks for the main function and runs it first.

Without a main function, the compiler cannot determine where to start, and will raise an error:

```
[!] Function "main " is not defined.
```

This design enforces a clear, predictable program structure and prevents ambiguity during execution.

Syntax and Structure

The main function follows the same syntax as any other SLIECE function, but with no parameters and a single return expression.

Syntax

```
$ main
-> <expression>
```

Optionally, main may include internal statements or calls to other functions:

```
$ main
x == 10 -> x * 2
-> "Done"
```

However, only the final -> expression defines the program's **return output**.

Example Programs Using main

Example 1 — Hello World

```
$                                                    main
->                        "Hello,                        SLIECE!"
```

Output:

```
Hello,                                                SLIECE!
```

Example 2 — Function Call

```
func      add<Int32      a,      Int32      b>      =>      Int32

$                                                      add
->                    a                    +                    b

$main
->                           add(5,                           7)
```

Output:

```
12
```

Example 3 — Conditional Execution

```
define             Int32             x             =             3

$                                                      main
x             >             0             ->             "Positive"
->                                                "Non-positive"
```

Output:

```
Positive
```

Control and Flow

Conditional Statements

SLIECE does not use traditional if keywords. Instead, conditions are expressed directly using comparison operators followed by an arrow expression.

Syntax

```
<condition>                    ->                    <expression>
```

If the condition evaluates to True, the expression is executed.

Example

```
define              Int32              x              =              10

$                                                                    main
x          >          5          ->          "Large          number"
->                          "Small                          number"
```

Output:

```
Large number
```

Expression-Based Logic

Each conditional statement is expression-based, meaning the result of a logical evaluation directly determines the return value.

Example

```
define Bool isHungry = True

$ main
isHungry == True -> "Time to eat!"
-> "No cake today."
```

Output:

```
Time to eat!
```

You can also use logical operators to build more complex expressions:

```
x > 0 and x < 10 -> "Within range"
-> "Out of range"
```

Combining Conditions

Multiple conditional statements can be chained to simulate “if / else if / else” flows.

Example

```
define Int32 score = 85

$ main
score >= 90 -> "Excellent"
score >= 70 -> "Good"
-> "Needs improvement"
```

Output:

```
Good
```

This structure ensures clarity and simplicity — every branch remains an explicit expression, preserving SLIECE’s functional purity.

Complete Examples

Hello World

```
$ main

-> "Hello, SLIECE!"
```

Output :

```
Hello, SLIECE!
```

Mathematical Operations

```
define Int32 a = 6

define Int32 b = 3


$ main

-> a * b + (a / b)
```

Output:

```
21
```

Functional Composition

```
func add<Int32 a, Int32 b> => Int32

$ add

-> a + b


func square<Int32 x> => Int32

$ square

-> x * x


$ main

-> square(add(2, 3))
```

Output:

```
25
```

Multi-File Project Example

math.lie

```
func      add<Int32      a,      Int32      b>      =>      Int32
$                                                          add
->                    a                    +                 b
```

main.lie

```
define             Int32             x             =             5
define             Int32             y             =             10

$                                                              main
->                           add(x,                              y)
```

Command:

```
sliece                        run                        main.lie
```

Output:

```
15
```

Compilation and Execution Process

Parsing

The SLIECE compiler first **parses** the source code using its BNF grammar. It converts the code into an **Abstract Syntax Tree (AST)**, validating tokens and structure.

Steps:

1. Tokenize (split text into keywords, identifiers, operators)
2. Parse (build syntax tree from grammar)
3. Check for syntax errors

Execution Flow

Execution starts at $ main.
The interpreter recursively evaluates all function calls and expressions in **a pure, deterministic order** — without side effects.

Simplified steps:

1. Locate $ main
2. Evaluate each statement in sequence
3. Resolve all -> expressions
4. Produce a final value as output

Output Rules

The value of the last evaluated expression in $ main becomes the **program output**.

Example:

```
$                                                                  main
->                      2                      +                     2
```

Output:

```
4
```

Security and Design

Inspirations and Safety Review

SLIECE draws inspiration from:

- **Haskell** (purity and immutability)
- **Python** (readability and indentation)
- **C** (type safety and memory security)

Security issues found in other languages (like null dereferencing or type coercion) are **prevented** by SLIECE's strict syntax and explicit types.

Security Mechanisms Implemented

- **Immutable Variables** — no reassignment once defined.
- **No Side Effects** — functions cannot modify external state.
- **Strong Typing** — every value must have a declared type.
- **Deterministic Execution** — same inputs → same outputs.
- **No Implicit Casting** — all type conversions must be explicit.

Potential Risks and Limitations

- Limited error handling (no runtime exception model yet).
- No sandboxing — file or network operations must be external.
- Recursive calls could cause stack overflows for large inputs.
- Performance optimization is minimal in early builds.

Despite these, SLIECE's design ensures **predictable and secure execution** by default.

Accessibility

Readability Principles

SLIECE's syntax is designed to be:

- Minimal and consistent
- Easy to read aloud for screen readers
- Free of ambiguous punctuation (e.g., no braces {})

Font and Editor Recommendations

To improve readability:

- Use a **monospaced font** (e.g., JetBrains Mono, Fira Code, Consolas)
- Prefer high-contrast themes for visibility
- Enable syntax highlighting for .lie files in your editor

Screen Reader Compatibility

- Keywords like define, func, and main are easily pronounced by text-to-speech software.
- Avoid nonstandard Unicode characters.
- Line breaks and indentation should be preserved for logical structure.

Accessibility Considerations for Developers

- Avoid cryptic variable names; use descriptive identifiers.
- Keep functions small and purpose-driven.
- Structure code with clear line spacing and minimal nesting.

Example of accessible style:

```
#          Accessible          and          readable          code
func          greet<String          name>          =>          String

$                                                               greet
->             "Hello,              "             +             name

$                                                               main
-> greet("World")
```