LéPiX Language Specification

Ceci n'est pas un Photoshop

Fatima Koly (fak2116)
Manager
fak2116@barnard.edu

Gabrielle Taylor (gat2118)
Language Guru
gat2118@columbia.edu

Jackie Lin (jl4162)
Tester
jl4162@columbia.edu

Akshaan Kakar (ak3808)
Codegen
ak3808@columbia.edu

ThePhD (jm3689)
System Architect
jm3689@columbia.edu

https://github.com/ThePhD/lepix

October 26, 2016

Contents

Part I

Introduction

Chapter 1

Tutorial

1.1 Hello, World!

This is an example of a Hello World program in LéPiX. It creates an array from an initializer, and then proceeds to save it to the directory of the running program under the name "hello.bmp":

```
fun main () : int {
    // 2 dimensional array
    // of integers, initialized as a string
    // from a "bitmap"
    var arr : int[[]] = "\
|   | ||| |   |   |||   |   | ||| ||| |   ||
|   | |   |   |   | |   |   | | | | | |   | |
||||| ||  |   |   | |   | | | | | ||  |   | |
|   | |   |   |   | |   ||  || | | | | |   | |
|   | ||| ||| ||| |||   |   | ||| | | ||| || ";
    lib.save("hello.bmp", arr);
}
```

Listing 1.1: hello world

1.2 Variables and Declarations

1.2.1 Variables

Variables are made with the and var declaration. You can declare and assign variables by giving them a name and then referencing that name in other places.

```
fun main () : int {
      var a : int = 24 * 2 + 1;
      // a == 49
      var b : int = a % 8;
      // b == 1
      var c : int[[5, 2]] = [
      0, 2, 4, 6, 8, 10;
          1, 3, 5, 7, 9, 11;
     ];
      var value : int = a + b + c[0, 4];
      // value == 58
      return value;
}
```

Listing 1.2: variable declaration and manipulation

1.2.2 Mutability

Variables can also be declared immutable or unchanging by declaring them with let. That is, let is the same as a var const, and let mutable.

```
fun main () : int {
      let a : float = 31.5;
      var const b : float = 0.5;
      var c : int = 0;
      c = lib.trunc(a + b);
      // compiler error: 'var const' variable is immutable
      b = 2.5;
      // compiler error: 'let' variable is immutable
      a = 1.1;
      return c;
}
```

Listing 1.3: mutability

1.3 Control Flow

Control flow is important for programs to exhibit more complex behaviors. LéPiX has for and while constructs for looping, as well as if, else if, else statements. They can be used as in the following sample:

```
fun main () : int {
    for (var x : int = 0 to 10) {
        var x : int = lib.random_int(0, 40);
        if (x < 20) {
            lib.print("It's less than 20!");
        }
        else {
            lib.print("It's equal to or greater than 20");
        }
    }
}
```

"intro/tutorial/code/flow.hak"

1.4 Functions

1.4.1 Defining and Declaring Functions

Functions can be called with a simple syntax. The goal is to make it easy to pass arguments and specify types on those arguments, as well as the return type. All functions are defined by starting with the fun keyword, followed by an identifier including the name, before an optional list of parameters.

```
fun sum (arr : int[]) : int {
    int a = 2;
    int b = 3;
    return a + b;
}

fun numbers () : int[] {
    return [ 1, 2, 3 ];
}

fun main () : int {
    return sum(numbers());
}
```

Listing 1.4: functions

1.4.2 Parameters and Arguments

All arguments given to a function for a function call are passed by value, unless the reference symbol & is written just before the argument, as shown in the below example. This allows a person to manipulate a value that was passed in directly, rather than receiving a copy of it the argument.

```
fun fibonacci_to (n : int, &storage : int[]) : int {
    int index = 0;
    var result : int = 0;
    var n_2 : int = 0;
    var n_1 : int = 1;
    while (n > 0) {
        result = n_1 + n_2;
        storage[index] = result;
        n_2 = n_1;
        n_1 = result;
        --n;
        ++index;
    }
    return result;
}

fun main () : int {
    var storage : int [3] = [];
    var x : int = fibonacci_to(3, storage);
    return x;
}
```

Listing 1.5: arguments

Part II

Reference Manual

Chapter 2

Expressions, Operations and Types

2.1 Variable Names and Identifiers

2.1.1 Identifiers

1. All names for all identifiers in a LéPiX program must be composed of a single start alpha codepoint followed by either zero or more of a digit or an alpha codepoint. Any identifier that does not follow this scheme and does not form a valid keyword, literal or definition is considered ill-formed.

2. All identifiers that containing two underscores `__` in any part of the name are reserved for usage by the compiler implementation details and may not be used by programs. If an identifier has two underscores the program is considered ill-formed.

3. All identifiers prefixed by '`lib.`' (i.e., belong in the `lib` namespace) are reserved by the standard to the standard library and nothing may be defined in that namespace by the program, aside from implementations of the standard library.

2.2 Literals

2.2.1 Kinds of Literals

There are many kinds of literals. They are:

literal:

boolean-literal
integer-literal
floating-literal
string-literal

2.2.2 Boolean Literals

1. A boolean literal are the keywords `true` or `false`.

2.2.3 Integer Literals

1. An integer literal is a valid sequence of digits with some optional alpha characters that change the interpretation of the supplied literal.
2. A decimal integer literal uses digits '0' through '9' to define a base-10 number.
3. A hexidecimal integer literal uses digits '0' through '9', 'A' through 'F' (case insensitive) to define a base-16 number. It must be prefixed by `0x` or `0x`.
4. An octal integer literal uses digits '0' and '7' to define a base-8 number. It must be prefixed by `0c` or `0C`.
5. A binary integer literal uses digits 0 and 1 to define a base-2 number. It must be prefixed by `0b` (case sensitive).
6. An n-digit integer literal uses the characters below to define a base-n number. It must be prefixed by `0n` or `0N`. It must be suffixed by `#n`, where n is the desired base. The character set defined for these bases

goes up to 63 characters, giving a maximum arbitrary base of 63. The characters which are:

$0 - 9$, A $-$ Z, a $-$ z, _

7. Arbitrary bases for n-digit must be base-10 numbers.

8. Groups of digits may be separated by a `'` and do not change the integer literal at all.

2.2.4 Floating Literals

1. A floating literal has two primary forms, utilizing digits as defined in 2.2.3.

2. The first form must have a dot '.' preceded by an integer literal and/or suffixed by an integer literal. It must have one or the other, and may not omit both the prefixing or suffixing integer literal.

3. The second form follows 2, but includes the exponent symbol `e` and another integer literal describing that exponent. Both the exponent and integer literal must be present in this form, but if the exponent is included then the dot is not necessary and may be prefixed with only an integer literal or just an integer literal and a dot.

2.2.5 String Literals

1. A string literal is started with a single '`'`' or double '`"`' quotation mark and does not end until the next matching single '`'`' or double '`"`' quotation mark character, with respect to what the string was started with. This includes any and all spacing characters, including newline characters.

2. Newline characters in a multi-line string will be included in the string as an ASCII Line Feed `\n` character.

3. A string literal must remove the leading space on each line that are equivalent to all other lines in the text, and any empty leading space at the start of the string.

4. A string literal may retain the any leading space and common indentation by prefixing the opening single or double quotation mark with an ‘R’.

2.3 Variable Declarations

2.3.1 let and var declarations

variable-initialization:

> *let | var (mutable | const)$_{optional}$ < identifier > : <type>;*

1. A variable can be declared using the let and var keywords, an identifier as defined in 2.1.1 and optionally followed by a colon ‘:’ and type name. This is called a variable declaration.
2. A variable declared with let is determined to be immutable. Immutable variables cannot have their values re-assigned after declaration and initialization.
3. A variable declared with var is immutable. Mutable variables can have their values re-assigned after declaration and initialization.
4. let mutable is equivalent to var const.
5. It is valid to initialize or assign to a mutable variable from an immutable variable.
6. A declaration can appear at any scope in the program.

2.4 Initialization

2.4.1 Variable Initialization

variable-declaration:

> *let | var (mutable | const)$_{optional}$ < identifier > : <type> = (expression);*

1. Initialization is the assignment of an expression on the right side to a variable declaration.

2. If the expression cannot directly initialize or be coerced to initialize the type on the left, then the program is ill-formed.

2.4.2 Assignment

assignment-expression:

expression = *expression*

2.5 Access

2.5.1 Member Access

member-access-expression:

(*expression*) . <*identifier*>

1. Member access is performed with the dot '.' operator.

2. If the expression does not evaluate to a type that can be accessed with the dot operator, the program is ill-formed.

3. If the identifier is not available per lookup rules in 2.5.2 on the evaluated type, the program is ill-formed.

2.5.2 Member Lookup

1. When a member is accessed through the dot operator as in 2.5.1, a name must be found that matches the supplied identifier . If there is none,

2.6 Parenthesis

parenthesis-expression:

> *'(' expression ')'*

1. Parentheses define expression groupings and supersede precedence rules in 2.1.

2.7 Arithmetic Expressions

2.7.1 Binary Arithmetic Operations

addition-expression:

> *expression* + *expression*

subtraction-expression:

> *expression* − *expression*

division-expression:

> *expression* / *expression*

multiplication-expression:

> *expression* * *expression*

modulus-expression:

> *expression* % *expression*

1. Symbolic expression to perform the commonly understood mathematical operations on two operands.

2. All operations are left-associative.

2.7.2 Unary Arithmetic Operations

unary-minus-expression:

$-expression$

1. Unary minus is typically interpreted as negation of the single operand.
2. All operations are left-associative.

2.8 Incremental Expressions

2.8.1 Incremental operations

post-increment-expression:

$(\ expression\ )++$

pre-increment-expression:

$++(\ expression\ )$

post-decrement-expression:

$(\ expression\ )--$

pre-decrement-expression:

$--(\ expression\ )$

1. Symbolic expression that should semantically evaluate to
   (expression) = (expression) + 1.
2. (expression) is only evaluated once.

2.9 Logical Expressions

2.9.1 Binary Compound Boolean Operators

and-expression:

expression *and* *expression*

expression *&&* *expression*

or-expression:

expression *or* *expression*

expression || *expression*

1. Symbolic expressions to check for logical conjunction and disjunction.
2. For the and−expression, short-circuiting logic is applied if the expression on the left evaluates to false. The right hand expression will not be evaluated.
3. For the or−expression, short-circuiting logic is applied if the expression on the left evaluates to true. The right hand expression will not be evaluated.
4. All operations are left associative.

2.9.2 Binary Relational Operators

equal-to-expression:

expression == *expression*

not-equal-to-expression:

expression != *expression*

less-than-expression:

expression < *expression*

greater-than-expression:

expression > *expression*

less-than-equal-to-expression:

expression <= *expression*

greater-than-equal-to-expression:

expression >= expression

1. Symbolic expression to perform relational operations meant to do comparisons.
2. All operations are left-associative.

2.9.3 Unary Logical Operators

inversion-expression:

! expression

complement-expression:

˜expression

1. Symbolic expression to perform unary logic operations, such as logical complement and logical inversion.

2.10 Bitwise Operations

2.10.1 Binary Boolean Operators

bitwise-and-expression:

expression & expression

bitwise-or-expression:

expression | expression

bitwise-xor-expression:

expression ˆ expression

1. Symbolic expressions to perform logical / bitwise and, or, and exclusive-or operations.
2. All operations are left associative.

2.11 Operator and Expression Precedence

Precedence is defined as follows:

2.12 Expression and Operand Conversions

2.12.1 Boolean Conversions

1. Expressions that are expected to evaluate to booleans for the purposes of Flow Control as defined in 5.3 and for common relational and logical operations as in 2.9 will have their rules checked against the following:
   (a) If the evaluated value is already a boolean, use the value directly.
   (b) If the evaluated value is of an integral type, then any such type which compares equivalent to the integral literal `0` will be `false`: otherwise, it is `true`.
   (c) If the evaluated value is of a floating point type, then any such type which compares equivalent to the floating point literal `0.0` will be `false`: otherwise, it is `true`.
   (d) Otherwise, if there is no defined conversion, then the program is ill-formed.

2.12.2 Mathematical Conversions

1. Conversion will not be implicitly done on any operands.
2. If there is no operator defined for those two types exactly, then the program is ill-formed.

Table 2.1: Precedence Table

Precedence	Operator	Variants	Associativity
1	++ −− ( ) [ ] .	Postfix	Left to Right
2	++ −− + - ! ∼	Prefix, Unary Operations	Right to Left
3	* / %		
4	+ -		
5	<< >>	Binary Operations	Left to Right
6	< <= > >=		
7	== !=		
8	&		
9	ˆ		
10	|		
11	||		
12	= += -= *= /= %= <<= >>= &= ˆ= |=	Assignments	Right-To-Left

Chapter 3

Functions

3.1 Functions and Function Declarations

Functions are independent code that perform a particular task and can be reused across programs. They can appear in any order and in one or many source files, but cannot be split among source files.

Function declarations tell the compiler how a function should be called, while function definitions define what the function does.

3.1.1 Function Definitions

```
fun <identifier> ([<parameter_declarations>]) : <return_type>
{
<function_body>
[return <expression>;]
}
```

1. All function definitions in LéPiX are of the above form where they begin with the keyword fun, followed by the identifier, a list of optional parameter declarations enclosed in parentheses, optionally the return type, and the function body with an optional return statement.

2. return types can be variable types or void.

3. Functions that return void can either omit the return statement or leave it in or return the value unit, as shown:

```
        fun zero ( &arr:int[] ) : void {
            for (var i : int = 0 to arr.length)
            {
             arr[i] = 0;
            }
        }

```

```
    fun zero ( &arr:int[] ) : void {
        for (var i : int = 0 to arr.length)
        {
         arr[i] = 0;
        }
        return;
    }

```

4. Functions that return any other variable type must include a return statement and the expression in the return statement must evaluate to the same type as the return type or be convertible to the return type:

```
        fun add ( arg1:float , arg2:float ) : float
        {
            return arg1 + arg2;
        }

```

5. In the function add, arg1 and arg2 are passed by value. In the function zero, arr is passed by reference.

6. Function input parameters can be passed by value, for all variable types, or by reference, only for arrays and array derived variable types. See 3.1.3 for more about passing by value and reference.

3.1.2 Function Declarations

1. All function declarations in LéPiX are of the form

```
        fun <identifier> ([<parameter_declarations>]) : <
    return_type>;

```

2. The function declaration for the add function from 3.1.1 would be

```
        fun add ( arg1:float , arg2:float ) : float;

```

3. Function declarations are identical to function definitions except for the absence or presence of the code body. Function declarations are optional, but useful to include when functions are used across multiple source files to ensure that functions are called appropriately.

3.1.3 Function Scope and Parameters

1. Variables are declared as usual within the body of a function. The variables declared within the body of a function exist only in the scope of the function and are discarded when they go out of scope.

2. External variables are passed into functions as parameters. All variable types except arrays and array derived variable types are passed by value. Arrays and array derived variable types can be passed by both value and reference.

3. Passing value copies the object, meaning changes are made to the copy within the function and not the original. Passing by reference gives a pointer to the original object to the function, meaning changes are to the original within the function.

4. To pass by value to a function, use the variable name: `add ( x, y );`

5. To pass by reference to a function, use the symbol & and the variable name, as in `zero ( &arr );`.

Chapter 4

Data Types

4.1 Data Types

The types of the language are divided into two categories: primitive types and data types derived from those primitive types. The primitive types are the boolean type, the integral type int, and the floating-point type float. The derived types are struct, Array, and image and pixel, which are both special instances of arrays.

4.1.1 Primitive Data Types

1. `int`

   By default, the int data type is a 32-bit signed two's complement integer, which has a minimum value of -2^{31} and a maximum value of 2^{32}.

2. `float`

   The float data type is a single precision 32-bit IEEE 754 floating point.

3. `boolean`

   The boolean data type has possible values true and false.

4.1.2 Derived Data Types

Besides the primitive data types, the derived types include arrays, structs, images, and pixels.

1. `array`

   An array is a container object that holds a fixed number of values of a single type. Multi-dimensional arrays are also supported. They need to have arrays of the same length at each level.

2. `pixel`

   A pixel data type is a wrapper for an array that will contain the representation for each pixel of an image. It will contain the rgb values, each as a separate int, and the gray value of a pixel.

3. `image`

   The image data type is just an alias for a 2-dimensional array. The 2-d array will define the size of an image and contains a pixel as each of its data elements.

4. `struct`

   A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling. Structures help to organize data because they permit a group of related variables to be treated as a unit instead of as separate entities.

Chapter 5

Program Structure and Control Flow

5.1 Statements

Any expression followed by a semicolon becomes a statement. For example, the expressions `x = 2`, `lib.save(...)`, `return x` become statements

```
x = 2;
lib.save(...);
return  x;
```

The semicolon is used in this way as a statement terminator.

5.2 Blocks and Scope

Braces { and } are used to group statements in to blocks. Braces that surround the contents of a function are an example of grouping statements like this. Statements in the body of a `for`, `while`, `if` or `switch` statement are also surrounded in braces, and therefore also contained in a block. Variables declared within a block exist only in that block. A semicolon is not required after the right brace.

5.2.1 Blocks

At any point in a program, braces can be used to create a block. For example,

```
1 var x: int = 2;
2 var y: int = 4;
3 var result: int;
4 {
5     var z: int = 6
6     result = x + y + z;
7 }
```

In this trivial example, the statements on lines 5 and 6 live within their own block. Blocks do have access to named definitions of their surrounding scope, however variables define within a block exist only within that block.

5.2.2 Scope

Scopes are defined as the collection of identifiers and available within the current lexicographic block[1]. Every program is implicitly surrounded by braces, which define the 'global block', and creating a 'global scope'.

5.2.3 Variable Scope

Variables are in scope only within their own block. In the example in Section 5.2.1, `z` is declared within the braces. Variables declared within blocks last only within lifetime of that block. If we attempted to access `z` outside of this block, this would cause an error to occur.

Variables are constructed, that is, stored in memory when they are first encountered in their scope, and destructed at the end of the scope in the reverse order they were encountered in.

[1]This is usually between two curly braces{}

5.2.4 Function Scope

Function definitions define a new block, which each have their own scope. Function definitions have access to any variables within their surrounding scope, however anything defined in the function definition's block is not accessible in the surrounding blocks. Variables defined in a parameter list belong to the definition-scope of the function.

5.2.5 Control Flow Scope

Control flow also introduces a new block with its own scope. Variables initialized in any control flow statement, that is within the parenthesis before the block, belong to the control flow block and are not accessible in the surrounding block. In the statement `for (var x = 0 to 5) { ... }`, `x` only exists within that for loop and destructed after the loop ends.

5.3 Namespaces

1. Namespaces are essentially blocks that allow identifiers to be prefix with an arbitrary nesting of names. They are declared with the `namespace` keyword, followed by several identifiers delimited by a dot '.' symbol.
2. Accessing variables and functions inside of a namespace must have the name of the namespace prefixed before the name of the desired identifier.
3. The namespaces `lib` and `compiler` is reserved for use by standard library implementations and the compiler.

5.4 if

If statements are used to make decisions in control flow. The syntax for an if statement is as follows:

```
if (expression)
    statement
else
    alternative statements
```

Variations on this syntax are permitted. For example, the `else` block of the if statement is optional. If the expression is evaluated and returns `true`, then the first portion of the if statement is executed. Otherwise, if there is an else the portion after it is executed, and if there is none then the function continues at the next statement. Parenthesis are optional after the if block if there is a single statement. If there are multiple statements, parenthesis are needed.

Variables can be initialized inside the expression portion of the if statement as long as the final expression in a semi-colon delimited list evaluates to a Boolean value.

```
if (var x = 20; var y = 50; x < y) {
    statements
}
```

The scope for variables `x` and `y` is within that particular if statement.

If statements can also be nested so that multiple conditions can be tested. For example,

```
if (x < 0)
    y = -1
else if (x > 0)
    y = 1
else
    y = 0
```

5.5 switch

Switch statements can be used as an alternative to a nested if statement. The general structure of a switch statement is as follows:

```
switch (variable) {
    case (constant expression):
        statements
        end;
    case (constant expression):
        statements
        end;
    case (constant expression):
        statements
        end;
    default: statements
}
```

The variable is compared against the constant expression for each case, and if it is equal to this expression then the statements in that case are executed. If the variable does not match any of the cases then the default case is executed. The statements in each case must be followed by an `end` statement.

As with if statements, if a variable is declared within the switch like `switch (var x = other_variable; x) { ... }`, the scope for variable `x` is within that particular switch block.

5.6 while

While loops are used to repeat a block of code until some condition is met. The general structure of a while loop is as follows:

```
while (condition)  {
    statements
}
```

The condition is an expression that is evaluated on each iteration of the loop. If condition evaluates to a non-zero value then the statements in the loop block are executed, and then condition re-evaluated to determine if execution should continue in the while loop or not. As soon as condition evaluates to zero then execution continues outside the while loop.

A while loop can support expressions that are more complex that a single expression. A while loop that looks like `while (var x = 20; x <30) {...}` is a valid while loop. The scope for variable `x` is within that particular while block.

An important point when using a while loop is that the statements should affect the value of the variable being evaluated in the condition in order to leave the loop, as in the example below. Other than that, a `break` or `continue` statement must be used. This will be discussed in Section 5.8.

```
while (var x = 1; x <= 10) {
    arr[x] = 1;
    x = x + 1;
}
```

5.7 for

For loops are another way to repeat a group of statements multiple times. The general structure resembles the following:

```
for (variable = lower_bound to upper_bound by size) {
    statements
}
```

The `by` keyword and argument `size` are optional and used to specify how much the variable should change by each iteration of the loop. For example, `for (x = 1 to 10 by 2) {...}` will increment x by two each iteration rather than the default value of 1. Variables can be declared in the loop declaration, as in `for (var x = 1 to 10) {...}`. For loops can also be used to decrement by swapping the positions of the lower_bound and upper_bound arguments, and using a negative value for the size (if using the `by` keyword).

The while loop in Section 5.6 could be expressed as a for loop as follows.

```
for (var x = 1 to 10) {
    arr[x] = 1;
}
```

C-style for loops are also supported, as in

```
for (var x = 1; x <= 10; x++) {
    arr[x] = 1;
}
```

5.8 break and continue

Break and continue statements are used to exit a loop immediately, before the specified condition has been reached. Break statements are used as follows:

```
while (...) {
     statements_above
     break;
     statements_below
}
```

In the example above, statements_above would be executed only once. The statements_below would never be executed. Break statements are usually included inside of an if statement within the loop to immediately exit on a particular condition.

Continue statements jump to the end of the loop body and begin the next iteration. The statements below the `continue` are not executed, and the loop condition is immediately reevaluated.

```
for (...) {
     statements_above
     if (condition) {
          continue;
     }
     statements_below
}
```

In the example above, statements_above would always be executed. The statements_below would be executed on iterations where the if condition was false, since if the if condition were true execution would jump back to the top of the for loop.

Break statements can be used to exit nested loops by jumping out of multiple scopes by adding an integer value after the `break` keyword.

```
for (...) {
    for (...) {
        statements_above
        if (condition)
            break 2;
        statements_below
    }
}
```

The example above will allow the user to break out of both for loops with only one break statement.[2]

[2]This could be considered a "poor man's goto" and should be used with the programmer's utmost discretion.

Chapter 6

Parallel Execution

Since a large number of elementary operations in the realm of image processing are embarrassingly parallel matrix operations, the LéPiX language supports a simple parallelization scheme.

6.1 Parallel Execution Model

1. Parallel Execution is when two computations defined by the language are run at the exact same time by the abstract virtual machine, capable of accessing the same memory space.
2. The primary parallel primitive is parallel-marked loops. Most parallel work is fundamentally loopable work.
3. Use of parallel primitives does not guarantee parallel execution: computation specified to run in parallel may run sequentially.[1]

6.2 Syntax

1. The syntax for a parallelized loop is a simple extension to all loop syntax. In order to parallelize a loop, the keyword `parallel` can be

[1]This could be due to hardware limitations, operating system limitations, and other factors.

added before the keywords `for` or `while` in the loop statement.

2. In the situation where there are variables that must be shared by all the threads, a comma separated list of variable identifiers can be specified in parentheses using the keyword shared as in
`parallel ( thread_count = n ) shared( ... shared list ... ) for | while.`

3. In the case of nested for loops, only the outermost loop carrying the parallel keyword is parallel.

6.3 Threads

1. When a for-loop or while-loop is specified as ”parallel”, the compiler uses the range of indices in the for loop and divides this range into equal shares based on how many cores are available on the machine. Each share is assigned to a thread, which executed the statements within the loop for its assigned range of indices.

2. Each thread that is spawned in this way will have its own scope, which is created when the thread is spawned and destroyed when the thread is killed.

3. Each thread has its own copy of each variable that is declares within the scope of the loop statements.

4. In the case when variables are marked with the shared or atomic variables, the load splitting occurs similarly among threads, but the variables marked shared are locked using a binary semaphore. In this way, the threads will acquire locks on a shared variable, perform read/write operations and then release the lock so that other threads may acquire the locks.[2]

[2]A simple example use case for shared variables is the parallel summing of an array of values. The accumulator variable which will hold the final sum must be shared among threads and will therefore have to be read from and written to in an atomic manner.

Part III

Grammar Specification

Chapter 7

Grammar

7.1 Lexical Definitions and Conventions

A program consists of one or more translation units, which are translated in two phases, namely the preprocessing step and the lexing step. The preprocessing step entails carrying out directives which begin with # in a C-like style. The lexing step reduces the program to a sequence of tokens.

7.1.1 Tokens

1. Tokens belong to _categories. These are whitespace, keywords, operators, integer literals, floating point literals, string literals, identifiers, and brackets.
2. Whitespace tokens are used to separate other tokens and are ignored in any case where they do not occur between other non-whitespace tokens.

7.1.2 Comments

1. Comments come in two flavors: single-line and multi-line.
2. Single line comment begin with // and continue until the next newline character is found. Multi-line comments begin with /* and end with

*/. They are nested.

3. Comments are treated as whitespace tokens, but for various purposes may still appear between other whitespace tokens in a program's token stream.

7.1.3 Identifiers

1. Identifiers are composed of letters, numbers and the underscore character (_) but must begin with a letter. Identifiers beginning with underscores and numbers will be reserved for use within the implementation of the language.

7.1.4 Keywords

1. A set of identifiers has been reserved for use as keywords and cannot be used in other cases. The list of keywords is in the table below.

int	float	void	bool
unit	char	codepoint	string
vector	matrix	vector	vec
var	let	if	else
for	while	by	to
return	true	false	mutable
const	fun	struct	maybe
protected	public	private	shared
as	of	parallel	atomic

7.1.5 Literals

1. Literals are of three types: integer literals, floating literals, and string literals, as detailed in 2.2.1. All of them use the following definitions for their digits:

   ⟨*decimal-digit*⟩ ::= one of
   0 1 2 3 4 5 6 7 8 9

$\langle$*hexidecimal-digit*$\rangle$ ::= one of
    0 1 2 3 4 5 6 7 8 9
    A B C D E F
    a b c d e f

$\langle$*binary-digit*$\rangle$ ::= one of
    0 1

$\langle$*octal-digit*$\rangle$ ::= one of
    0 1 2 3 4 5 6 7

$\langle$*n-digit*$\rangle$ ::= one of
    0 1 2 3 4 5 6 7 8 9
    A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
    a b c d e f g h i j k l m n o p q r s t u v w x y z
    '_'

$\langle$*decimal-digit-sequence*$\rangle$ ::= $\langle\rangle$
 | $\langle$*decimal-digit*$\rangle$ $\langle$*decimal-digit-sequence*$\rangle$

$\langle$*binary-digit-sequence*$\rangle$ ::= $\langle\rangle$
 | $\langle$*binary-digit*$\rangle$ $\langle$*binary-digit-sequence*$\rangle$

$\langle$*octal-digit-sequence*$\rangle$ ::= $\langle\rangle$
 | $\langle$*octal-digit*$\rangle$ $\langle$*octal-digit-sequence*$\rangle$

$\langle$*hexidecimal-digit-sequence*$\rangle$ ::= $\langle\rangle$
 | $\langle$*hexidecimal-digit*$\rangle$ $\langle$*hexidecimal-digit-sequence*$\rangle$

$\langle$*n-digit-sequence*$\rangle$ ::= $\langle\rangle$
 | $\langle$*n-digit*$\rangle$ $\langle$*n-digit-sequence*$\rangle$

2. Integer literals consist of sequences of digits are always interpreted as decimal numbers. They can be represented by the following lexical compositions:

   $\langle$*integer-literal*$\rangle$ ::= $\langle$*decimal-literal*$\rangle$
    | $\langle$*binary-literal*$\rangle$
    | $\langle$*octal-literal*$\rangle$
    | $\langle$*hexidecimal-literal*$\rangle$
    | $\langle$*n-digit-literal*$\rangle$

   $\langle$*decimal-literal*$\rangle$ ::= $\langle$*decimal-digit-sequence*$\rangle$

⟨*binary-literal*⟩ ::= 0b ⟨*binary-digit-sequence*⟩
  | 0B ⟨*binary-digit-sequence*⟩

⟨*octal-literal*⟩ ::= 0c ⟨*octal-digit-sequence*⟩
  | 0C ⟨*octal-digit-sequence*⟩

⟨*hexidecimal-literal*⟩ ::= 0x ⟨*hexidecimal-digit-sequence*⟩
  | 0X ⟨*hexidecimal-digit-sequence*⟩

⟨*n-digit-literal*⟩ ::= 0n ⟨*n-digit-sequence*⟩ ‘#’
  | 0N ⟨*n-digit-sequence*⟩

3. Floating point literals can be specified using digits and a decimal points or in scientific notation. The following regular expression represents the set of acceptable floating-point constants.

   ⟨*e-part*⟩ ::= e ⟨*+ |−″ -*⟩ ⟨*integral-literal*⟩

   ⟨*floating-literal*⟩ ::= ⟨*integral-literal*⟩$_{opt}$. ⟨*integral-literal*⟩$_{opt}$
      ⟨*e-part*⟩$_{opt}$
     | ⟨*integral-literal*⟩ .$_{opt}$ ⟨*integral-literal*⟩$_{opt}$ ⟨*e-part*⟩

4. String literals are sections of quote-delimited items. They are defined as follows:

   ⟨*single-quote*⟩ ::= ’

   ⟨*double-quote*⟩ ::= ”

   ⟨*raw-specifier*⟩ ::= R$_{opt}$

   ⟨*character*⟩ ::= ⟨*escape-character*⟩ ⟨*source-character*⟩

   ⟨*character-sequence*⟩ ::= ⟨⟩
     | ⟨*character*⟩ ⟨*character-sequence*⟩

   ⟨*string-literal*⟩ ::= ⟨*raw-specifier*⟩ ⟨*double-quote*⟩
      ⟨*character-sequence*⟩ ⟨*double-quote*⟩
     | ⟨*raw-specifier*⟩ ⟨*single-quote*⟩ ⟨*character-sequence*⟩ ⟨*single-quote*⟩

7.2 Expressions

The following sections formalize the types of expressions that can be used in a LéPiX program and also specify completely, the precedence of operators and left or right associativity.

7.2.1 Primary Expression

$\langle$*primary_expression*$\rangle$ ::= $\langle$*identifier*$\rangle$
  | $\langle$*integer-constant*$\rangle$
  | $\langle$*float-constant*$\rangle$
  | (expression)

1. A primary expression are composed of either a constant, an identifier, or an expression in enclosing parentheses.

7.2.2 Postfix Expressions

$\langle$*postfix_expression*$\rangle$ ::= $\langle$*primary_expression*$\rangle$
  | $\langle$*postfix_expresion*$\rangle$ (argument_list)
  | $\langle$*postfix_expression*$\rangle$ [expression]
  | $\langle$*postfix_expression*$\rangle$. identifier $\langle$*argument_list*$\rangle$ ::= $\langle\rangle$
  | $\langle$*argument_list*$\rangle$, $\langle$*postfix_expression*$\rangle$

1. A postfix expression consist of primary expression followed by postfix operators. The operators in postfix expressions are left-associative.

Array Indexing

Array indexing consists of a postfix expression, followed by an expression enclosed in square brackets. The expression in the brackets must evaluate to an integer which will represent the index to be accessed. The value returned by indexing is the value in the array at the specified index.

Function Calls

A function call is a postfix expression (representing the name of a defined function) followed by a (possibly empty) list of arguments enclosed in parentheses. The list of arguments is represented as a comma separated list of postfix expressions.

Structure access

The name of a structure followed by a dot and an identifier name is a postfix expression. The value of the expression is the value of the named member of the structure that is being accessed.

7.2.3 Unary Expression

A unary expression consists of postfix_expression preceded by a unary operator (- ˜, !, * ,&). Expressions with unary operations are left-associative. The unary operation is carried out after the postfix expression has been evaluated.

$\langle$*unary_operator*$\rangle$ ::= ˜
  | !
  | -
  | *

$\langle$*unary_expression*$\rangle$ ::= $\langle$*unary_operator*$\rangle$ $\langle$*postfix_expression*$\rangle$

The function of each unary operator has been summarized in the table below:

-	Unary minus
˜	Bitwise negation operator
ˆ	Logical negation operator
*	Indirection operator

7.2.4 Casting

The LéPiX language supports the casting of an integer to a floating point value and vice versa. It also supports casting of an integer value to a boolean value and vice versa. Integer to float casting creates a floating point constant with the same value as the integer. Casting a floating point value to an integer rounds down to the nearest integral value. Casting a boolean value to an integer gives 1 if the value is true and 0 if it is false. Casting an integer to a boolean value yields false if the value is 0 and true otherwise.

⟨*cast_expression*⟩ ::= ⟨*unary_expression*⟩
  | ⟨*unary_expression*⟩ as ⟨*type_name*⟩

7.2.5 Multiplicative Expressions

The multiplication (*), division (/) and modulo (%) operators are left associative.

⟨*multiplicative_expression*⟩ ::= ⟨*cast_expression*⟩
  | ⟨*multiplicative_expression*⟩ * ⟨*cast_expression*⟩
  | ⟨*multiplicative_expression*⟩ / ⟨*cast_expression*⟩
  | ⟨*multiplicative_expression*⟩ % ⟨*cast_expression*⟩

7.2.6 Additive Expressions

The addition (+) and subtraction (-) operators are left associative.

⟨*additive_expression*⟩ ::= ⟨*multiplicative_expression*⟩
  | ⟨*additive_expression*⟩ + ⟨*cast_expression*⟩
  | ⟨*additive_expression*⟩ - ⟨*cast_expression*⟩

7.2.7 Relational Expressions

The relational operators less than (¡), greater than (¿), less than or equal to (¡=) and greater than or equal to (¿=) are left associative.

⟨*relational_expression*⟩ ::= ⟨*additive_expression*⟩
  | ⟨*relational_expression*⟩ ⟨ ¡*additive_expression*⟩

| $\langle$*relational_expression*$\rangle$ $\langle$= ¡*additive_expression*$\rangle$
| $\langle$*relational_expression*$\rangle$ ¿ $\langle$*additive_expression*$\rangle$
| $\langle$*relational_expression*$\rangle$ ¿= $\langle$*additive_expression*$\rangle$

7.2.8 Equality Expression

$\langle$*equality_expression*$\rangle$ ::= $\langle$*relational_expression*$\rangle$
| $\langle$*equality_expression*$\rangle$!= $\langle$*relational_expression*$\rangle$
| $\langle$*equality_expression*$\rangle$ == $\langle$*relational_expression*$\rangle$

7.2.9 Logical AND Expression

The logical and operator (&&) is left associative and returns true if both its operands are not equal to false.

$\langle$*logical_and_expression*$\rangle$ ::= $\langle$*equality_expression*$\rangle$
| $\langle$*logical_and_expression*$\rangle$ && $\langle$*equality_expression*$\rangle$

7.2.10 Logical OR Expression

The logical OR operator (——) is left associative and returns true if either of its operands are not equal to false.

$\langle$*logical_or_expression*$\rangle$ ::= $\langle$*logical_and_expression*$\rangle$
| $\langle$*logical_or_expression*$\rangle$ || $\langle$*logical_and_expression*$\rangle$

7.2.11 Assignment Expressions

The assignment operator (=) is left associative.

$\langle$*assignment_expression*$\rangle$ ::= $\langle$*logical_or_expression*$\rangle$
| $\langle$*unary_expression*$\rangle$ = $\langle$*assignment_expression*$\rangle$

7.2.12 Assignment Lists

Assignment lists consist of multiple assignment statements separated by commas.

⟨*assignment_list*⟩ ::= ⟨*assignment_expression*⟩
  | ⟨*assignment_list*⟩ ::= ⟨*assignment_list*⟩ , ⟨*assignment_expression*⟩

7.2.13 Declarations

Declarations of a variable specify a type for each identifier and a value to be assigned to the identifier. Declarations do not always allocate memory to be associated with the identifier.

⟨*declaration*⟩ ::= let ⟨*storage_class*⟩ ⟨*identifier*⟩ : ⟨*type_name*⟩ =
    ⟨*postfix_expression*⟩
  | var ⟨*storage_class*⟩ ⟨*identifier*⟩ : ⟨*type_name*⟩ = ⟨*postfix_expression*⟩
  | ⟨*declaration*⟩ ⟨*array*⟩

⟨*storage_class*⟩ ::= mutable
  | const

⟨*type_name*⟩ ::= void
  | unit
  | bool
  | int
  | float
  | ⟨*type_name*⟩ ⟨*array*⟩

⟨*array*⟩ ::= [⟨*int_list*⟩]
  | [⟨*array*⟩]

⟨*int_list*⟩ ::= ⟨*integer*⟩ | ⟨*int_list*⟩ , ⟨*integer*⟩

7.2.14 Structure Declaration

Structures are declared by specifying the identifier and then an initialization list of the member variables.

$\langle$*structure declaration*$\rangle$ ::= struct $\langle$*identifier*$\rangle$ { $\langle$*init_list*$\rangle$ }

7.2.15 Function Declaration

Function declarations consist of the keyword fun followed by an identifier and a list of parameters enclosed in parentheses. The list of arguments is followed by a colon and a type name which represents the return type for the function. The arguments list is specified as a comma-separated list of identifier, type pairs.

$\langle$*function_declaration*$\rangle$ ::= fun $\langle$*identifier*$\rangle$ ($\langle$*params_list*$\rangle$) : $\langle$*type_name*$\rangle$

$\langle$*params_list*$\rangle$ ::= $\langle$ $\rangle$
  | $\langle$*identifier*$\rangle$: $\langle$*type_name*$\rangle$
  | $\langle$*params_list*$\rangle$, $\langle$*identifier*$\rangle$: $\langle$*type_name*$\rangle$

7.3 Statements

Statements are executed sequentially in all cases except when explicit constructs for parallelization are used. Statements do not return values.

$\langle$*statement*$\rangle$ ::= $\langle$*expression_statement*$\rangle$
  | $\langle$*branch_statement*$\rangle$
  | $\langle$*compound_statement*$\rangle$
  | $\langle$*iteration_statement*$\rangle$
  | $\langle$*return_statement*$\rangle$

7.3.1 Expression Statements

Expression statements are either empty or consist of an expression. These effects of one statement are always completed before the next is executed. This guarantee is not valid in cases where explicit parallelization is used. Empty expression statements are used for loops and if statements where not action is to be taken.

$\langle$*expression_statement*$\rangle$ ::= $\langle$ $\rangle$
  | $\langle$*expression*$\rangle$;

7.3.2 Statement Block

A statement block is a collection of statements declarations and statements. If the declarations redefine any variables that were already defined outside the block, the new definition of the variable is considered for the execution of the statements in the block. Outside the block, the old definition of the variable is restored.

$\langle$*block*$\rangle$ ::= { $\langle$*compound_statement*$\rangle$ }
  | { $\langle$*block*$\rangle$ $\langle$*compound_statement*$\rangle$}

$\langle$*compound_statement*$\rangle$ ::= $\langle$*declaration*$\rangle$
  | $\langle$*statement*$\rangle$
  | $\langle$*compound_statement*$\rangle$; $\langle$*declaration*$\rangle$
  | $\langle$*compound_statement*$\rangle$; $\langle$*statement*$\rangle$

Branch Statements

Branch statement are used to select one of several statement blocks based on the value of an expression.

$\langle$*branch_statement*$\rangle$ ::= if ($\langle$*expression*$\rangle$) $\langle$*statement*$\rangle$ fi
  | if ($\langle$*expression*$\rangle$) $\langle$*statement*$\rangle$ else $\langle$*statement*$\rangle$ fi

7.3.3 Loop Statements

Loop statements specify the constructs used for iteration.

$\langle$*loop_statement*$\rangle$ ::= while ($\langle$*expression*$\rangle$) $\langle$*statement*$\rangle$
  | for ($\langle$*identifier*$\rangle$ = $\langle$*expression*$\rangle$ to $\langle$*expression*$\rangle$)
  | for ($\langle$*assignment_expression*$\rangle$ = $\langle$*expression*$\rangle$ to $\langle$*expression*$\rangle$ by $\langle$*expression*$\rangle$)

| parallel for (⟨*assignment_expression*⟩ = ⟨*expression*⟩ to ⟨*expression*⟩)
| parallel for (⟨*assignment_expression*⟩ = ⟨*expression*⟩ to ⟨*expression*⟩ by ⟨*expression*⟩)
| parallel shared(⟨*identifier_list*⟩) for (⟨*assignment_expression*⟩ = ⟨*expression*⟩ to ⟨*expression*⟩)
| parallel shared(⟨*identifier_list*⟩) for (⟨*assignment_expression*⟩ = ⟨*expression*⟩ to ⟨*expression*⟩ by ⟨*expression*⟩)
| parallel threads(⟨*expression*⟩) for (⟨*assignment_expression*⟩ = ⟨*expression*⟩ to ⟨*expression*⟩)
| parallel threads(⟨*expression*⟩) for (⟨*assignment_expression*⟩ = ⟨*expression*⟩ to ⟨*expression*⟩ by ⟨*expression*⟩)
| parallel threads(⟨*expression*⟩) shared(⟨*identifier_list*⟩) for (⟨*assignment_expression*⟩ = ⟨*expression*⟩ to ⟨*expression*⟩)
| parallel threads(⟨*expression*⟩) shared(⟨*identifier_list*⟩) for (⟨*assignment_expression*⟩ = ⟨*expression*⟩ to ⟨*expression*⟩ by ⟨*expression*⟩)

⟨*identifier_list*⟩ ::= ⟨*identifier*⟩
| ⟨*identifier_list*⟩ , ⟨*identifier*⟩

7.3.4 Jump Statements

Jump statements are used to break out of a loop or to skip the current iteration of a loop.

⟨*jump_statement*⟩ ::= break
| continue

7.3.5 Return Statements

Return statements are used to denote the end of function logic and the also to specify the value to be returned by a call to the function in question.

⟨*return_statement*⟩ ::= return
| return ⟨*expression*⟩

7.4 Function Definitions

Function definitions consist of a function declaration followed by a statement block.

⟨*function_definition*⟩ ::= ⟨*function_declaration*⟩ ⟨*block*⟩

7.5 Type Conversions

Implicit type conversions are carried out only for compatible types. The implicit casting occurs during assignment or when a value is passed as a function argument. The four types of conversions that are supported are summarized in the table below.

Int to Float	Float variable has the same value as integer
Float to Int	Integer has largest integral value less than the float
Bool to Int	Integer has value 1 if true else 0
Int to Bool	Bool is true if Int is not equal to 0 and false otherwise

7.6 Scope

The scope of a variable is defined as the statement block within which the variable has been declared. If a variable with a particular identifier has been declared and the identifier is re-used within a nested block, the original definition of the identifier is suspended and the new one is used until the end of the block.

7.7 Preprocessor

Before the source for a LePix program is compiled, the program is consumed by a preprocessor, which expands macro definitions and links libraries and other user-defined to the current file, as specified by appropriate preprocessor directives. We support define macros and import macros.

Define macros create an alias for a value or expression, while ifdef and ifndef macros are used to check if a particular alias has already been assigned. Import directives are used to link files/libraries with the current program.

⟨*preprocessor_directive*⟩ ::= #define ⟨*identifier*⟩ ⟨*expression*⟩
  | #ifdef ⟨*identifier*⟩
  | #ifndef ⟨*identifier*⟩
  | #import ”file˙name”
  | #import “⟨*file_name*⟩”
  | #import ''⟨*file_name*⟩''
  | #import string ⟨*file_name*⟩
  | #endif

7.8 Grammar Listing

⟨*primary_expression*⟩ ::= ⟨*identifier*⟩
  | ⟨*integer-constant*⟩
  | ⟨*float-constant*⟩
  | (expression)

⟨*postfix_expression*⟩ ::= ⟨*primary_expression*⟩
  | ⟨*postfix_expresion*⟩ (argument_list)
  | ⟨*postfix_expression*⟩ [expression]
  | ⟨*postfix_expression*⟩ . identifier ⟨*argument_list*⟩ ::= ⟨⟩
  | ⟨*argument_list*⟩ , ⟨*postfix_expression*⟩

⟨*unary_operator*⟩ ::= ˜
  | !
  | -
  | *

⟨*unary_expression*⟩ ::= ⟨*unary_operator*⟩ ⟨*postfix_expression*⟩

⟨*cast_expression*⟩ ::= ⟨*unary_expression*⟩
  | ⟨*unary_expression*⟩ as ⟨*type_name*⟩

⟨*multiplicative_expression*⟩ ::= ⟨*cast_expression*⟩
  | ⟨*multiplicative_expression*⟩ * ⟨*cast_expression*⟩

| ⟨*multiplicative_expression*⟩ / ⟨*cast_expression*⟩
| ⟨*multiplicative_expression*⟩ % ⟨*cast_expression*⟩

⟨*additive_expression*⟩ ::= ⟨*multiplicative_expression*⟩
| ⟨*additive_expression*⟩ + ⟨*cast_expression*⟩
| ⟨*additive_expression*⟩ - ⟨*cast_expression*⟩

⟨*relational_expression*⟩ ::= ⟨*additive_expression*⟩
| ⟨*relational_expression*⟩ ⟨ ¡*additive_expression*⟩
| ⟨*relational_expression*⟩ ⟨= ¡*additive_expression*⟩
| ⟨*relational_expression*⟩ ¿ ⟨*additive_expression*⟩
| ⟨*relational_expression*⟩ ¿= ⟨*additive_expression*⟩

⟨*equality_expression*⟩ ::= ⟨*relational_expression*⟩
| ⟨*equality_expression*⟩ != ⟨*relational_expression*⟩
| ⟨*equality_expression*⟩ == ⟨*relational_expression*⟩

⟨*logical_and_expression*⟩ ::= ⟨*equality_expression*⟩
| ⟨*logical_and_expression*⟩ && ⟨*equality_expression*⟩

⟨*logical_or_expression*⟩ ::= ⟨*logical_and_expression*⟩
| ⟨*logical_or_expression*⟩ || ⟨*logical_and_expression*⟩

⟨*assignment_expression*⟩ ::= ⟨*logical_or_expression*⟩
| ⟨*unary_expression*⟩ = ⟨*assignment_expression*⟩

⟨*assignment_list*⟩ ::= ⟨*assignment_expression*⟩
| ⟨*assignment_list*⟩ ::= ⟨*assignment_list*⟩ , ⟨*assignment_expression*⟩

⟨*declaration*⟩ ::= let ⟨*storage_class*⟩ ⟨*identifier*⟩ : ⟨*type_name*⟩ = ⟨*postfix_expression*⟩
| var ⟨*storage_class*⟩ ⟨*identifier*⟩ : ⟨*type_name*⟩ = ⟨*postfix_expression*⟩
| ⟨*declaration*⟩ ⟨*array*⟩

⟨*storage_class*⟩ ::= mutable
| const

⟨*type_name*⟩ ::= bool
| int
| float
| ⟨*type_name*⟩ ⟨*array*⟩

```
⟨array⟩ ::= [ ⟨int_list⟩ ]
  |  [ ⟨array⟩ ]

⟨int_list⟩ ::= ⟨integer⟩ | ⟨int_list⟩ , ⟨integer⟩

⟨structure declaration⟩ ::= struct ⟨identifier⟩ { ⟨init_list⟩ }

⟨function_declaration⟩ ::= fun ⟨identifier⟩ ( ⟨params_list⟩ ) : ⟨type_name⟩

⟨params_list⟩ ::= ⟨ ⟩
  |  ⟨identifier⟩ : ⟨type_name⟩
  |  ⟨params_list⟩ , ⟨identifier⟩ : ⟨type_name⟩

⟨statement⟩ ::= ⟨expression_statement⟩
  |  ⟨branch_statement⟩
  |  ⟨compound_statement⟩
  |  ⟨iteration_statement⟩
  |  ⟨return_statement⟩

⟨expression_statement⟩ ::= ⟨ ⟩
  |  ⟨expression⟩ ;

⟨block⟩ ::= { ⟨compound_statement⟩ }
  |  { ⟨block⟩ ⟨compound_statement⟩}

⟨compound_statement⟩ ::= ⟨declaration⟩
  |  ⟨statement⟩
  |  ⟨compound_statement⟩ ; ⟨declaration⟩
  |  ⟨compound_statement⟩ ; ⟨statement⟩

⟨branch_statement⟩ ::= if ( ⟨expression⟩ ) ⟨statement⟩ fi
  |  if ( ⟨expression⟩ ) ⟨statement⟩ else ⟨statement⟩ fi

⟨loop_statement⟩ ::= while ( ⟨expression⟩ ) ⟨statement⟩
  |  for ( ⟨identifier⟩ = ⟨expression⟩ to ⟨expression⟩ )
  |  for ( ⟨assignment_expression⟩ = ⟨expression⟩ to ⟨expression⟩ by
     ⟨expression⟩ )
  |  parallel for ( ⟨assignment_expression⟩ = ⟨expression⟩ to ⟨expression⟩ )
  |  parallel for ( ⟨assignment_expression⟩ = ⟨expression⟩ to ⟨expression⟩
     by ⟨expression⟩ )
  |  parallel shared(⟨identifier_list⟩) for ( ⟨assignment_expression⟩ =
     ⟨expression⟩ to ⟨expression⟩ )
```

| parallel shared($\langle$*identifier_list*$\rangle$) for ($\langle$*assignment_expression*$\rangle$ = $\langle$*expression*$\rangle$ to $\langle$*expression*$\rangle$ by $\langle$*expression*$\rangle$)
| parallel threads($\langle$*expression*$\rangle$) for ($\langle$*assignment_expression*$\rangle$ = $\langle$*expression*$\rangle$ to $\langle$*expression*$\rangle$)
| parallel threads($\langle$*expression*$\rangle$) for ($\langle$*assignment_expression*$\rangle$ = $\langle$*expression*$\rangle$ to $\langle$*expression*$\rangle$ by $\langle$*expression*$\rangle$)
| parallel threads($\langle$*expression*$\rangle$) shared($\langle$*identifier_list*$\rangle$) for ($\langle$*assignment_expression*$\rangle$ = $\langle$*expression*$\rangle$ to $\langle$*expression*$\rangle$)
| parallel threads($\langle$*expression*$\rangle$) shared($\langle$*identifier_list*$\rangle$) for ($\langle$*assignment_expression*$\rangle$ = $\langle$*expression*$\rangle$ to $\langle$*expression*$\rangle$ by $\langle$*expression*$\rangle$)

$\langle$*identifier_list*$\rangle$ ::= $\langle$*identifier*$\rangle$
| $\langle$*identifier_list*$\rangle$, $\langle$*identifier*$\rangle$

$\langle$*jump_statement*$\rangle$ ::= break
| continue

$\langle$*return_statement*$\rangle$ ::= return
| return $\langle$*expression*$\rangle$