Maryl

Maryl Overview

Introducing Maryl, a modern programming language designed for high efficiency and ease of use. Aimed at both beginners with a high level syntaxe and experienced developers with a common syntaxe, Maryl

Key Feature

- the learning curve.
- **Scalability**: Engineered to handle small scripts to large-scale applications efficiently.

> ⓘ Maryl empowers developers to focus on innovation, offering tools to support diverse programming needs in a single cohesive package : Made by developer for developer

Accessibility

Maryl also excels in accessibility by providing features that create an inclusive coding environment. One key feature is the ability to customize alert colors, which is particularly useful for developers with color vision deficiencies, such as color blindness. This allows users to adapt the interface to their needs, ensuring that important information is easily distinguishable, ultimately enhancing their coding experience.

USER MANUAL

How To Use Maryl

This quickstart guide will help you get up and running with Maryl

Step 1: Install a Maryl

You can install Maryl in the following way :

```
git clone git@github.com:MartinFillon/glados.git
cd glados/maryl
make
```

Step 2: Write Your First Program

Create a new file named `add.mrl` and open it in your favorite text editor. Write the following code:

```
int add1(int a, int b){
    return a + b;
}

int start(){
    int a = 2;
    int y = 6;
    return add1(y, 2);
}
```

Step 3: Run the Program

Open a terminal and navigate to the directory where you saved `add.mrl`. Compile the program using `glados`:

```
./glados build add.mrl # -c output_file
./glados run out.masm arg1 arg2 arg3 ...# or the other filename you used
```

Run the executable to see the output

Step 4: Experiment with Variables

Create a new file to include variables and if :

```
int start(){
    int a = 3;
    int b = 4;
    a = b + 1;
    if (a == 5) {
        return 2;
    } else {
        return 3;
    }
    return 0;
}
```

run the program again to see the output.

Step 5: Learn More

Explore more features of Maryl by reading tutorials and documentation. Practice writing different programs to strengthen your understanding.

By following this quickstart guide, you should be able to write, compile, and run basic Maryl programs. Happy coding!

Formal Grammar Description (BNF)

Backus-Naur Form (BNF) is a formal notation used to describe the syntax of programming languages

Grammar Notation

Example of BNF

Below is an example of a simple BNF grammar defining a basic arithmetic expression:

```
<expression> ::= <term> | <term> "+" <expression>
<term> ::= <factor> | <factor> "*" <term>
<factor> ::= <number> | "(" <expression> ")"
<number> ::= "0" | "1" | "2" | ... | "9"
```

This grammar describes expressions involving addition and multiplication of single-digit numbers, including the use of parentheses for grouping.

- `<...>`: Non-terminal symbols to be expanded
- `::=`: Defined as
- `|`: Or (alternative choices)
- `*`: Zero or more occurrences
- `+`: One or more occurrences
- `?`: Optional (zero or one occurrence)
- Literals: Enclosed in quotes (e.g., "+" or "(")
- Grouping: Square brackets `[...]`
- Comments: Indicated by

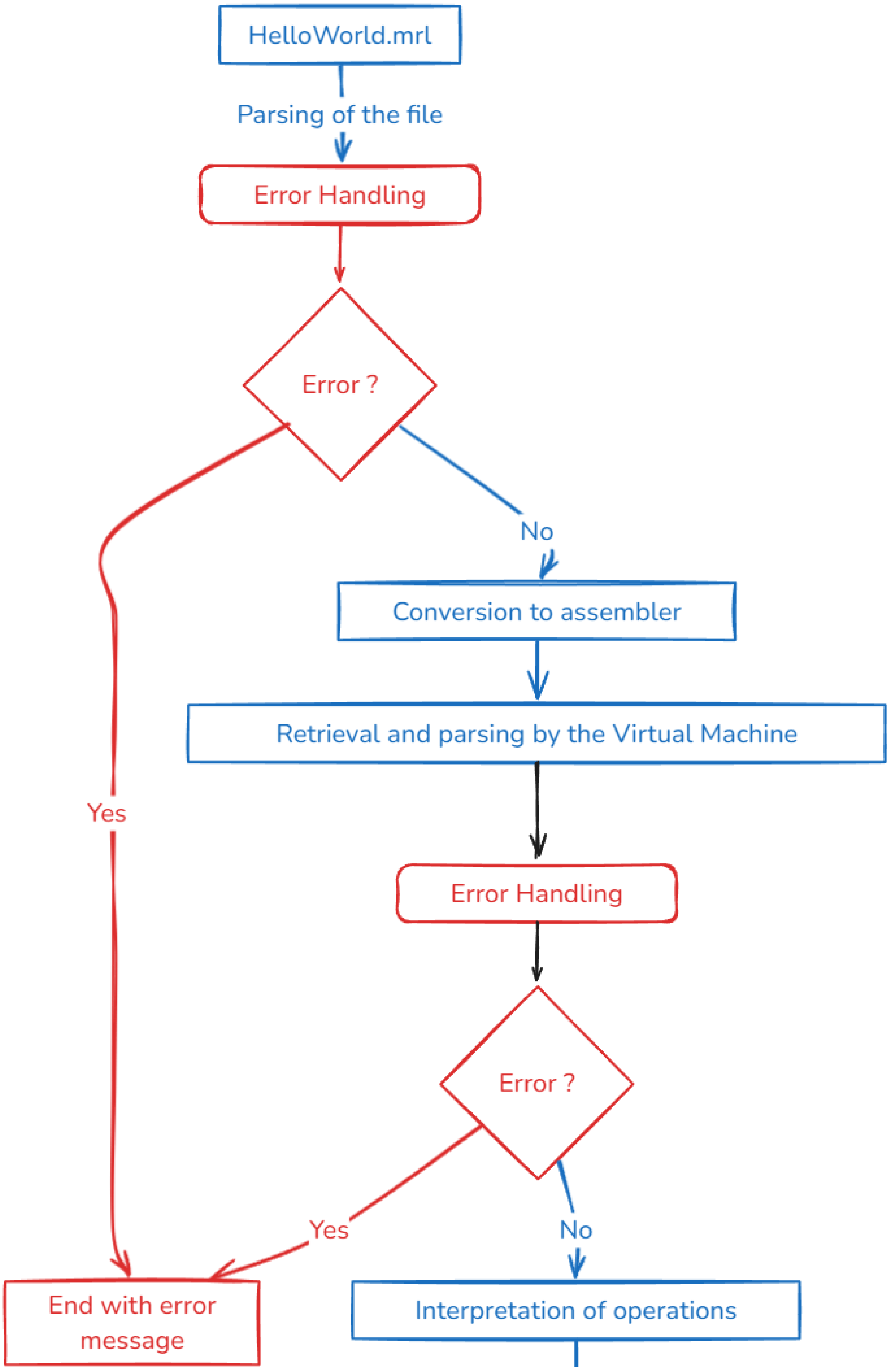
Maryl's grammar description (BNF)

```
<type> ::= "int"
        | "float"
        | "double"
        | "string"
        | "char"
        | "bool"
        | "void"
        | <list-type>
        | <const-type>
        | <struct-type>

<list-type> ::= "[]" <type>
<const-type> ::= "const" <type>
<struct-type> ::= "struct" <identifier> <block>
<expression> ::= <variable>
             | <literal>
             | <function-call>
             | <binary-expr>
             | <prefix-expr>
             | <postfix-expr>
             | <ternary-expr>
             | <grouped-expr>
             | <assignment>
             | <label>

<label> ::= <identifier> ":"
<variable> ::= <identifier>
<identifier> ::= <letter> <alphanumeric>*
             | "_" <alphanumeric>+

<literal> ::= <integer>
          | <double>
          | <bool>
          | <string>
          | <char>
          | <list>

<integer> ::= <digit>+
<double> ::= <digit>* "." <digit>+
<bool> ::= "true" | "false"
<string> ::= '"' <char>* '"'
<char> ::= "'" <char> "'"
<list> ::= "[" <expression> ("," <expression>)* "]"
<struct> ::= "{" <expression> ("," <expression>)* "}"

<function-call> ::= <identifier> "(" <expression-list>? ")"
<expression-list> ::= <expression> ("," <expression>)*
```

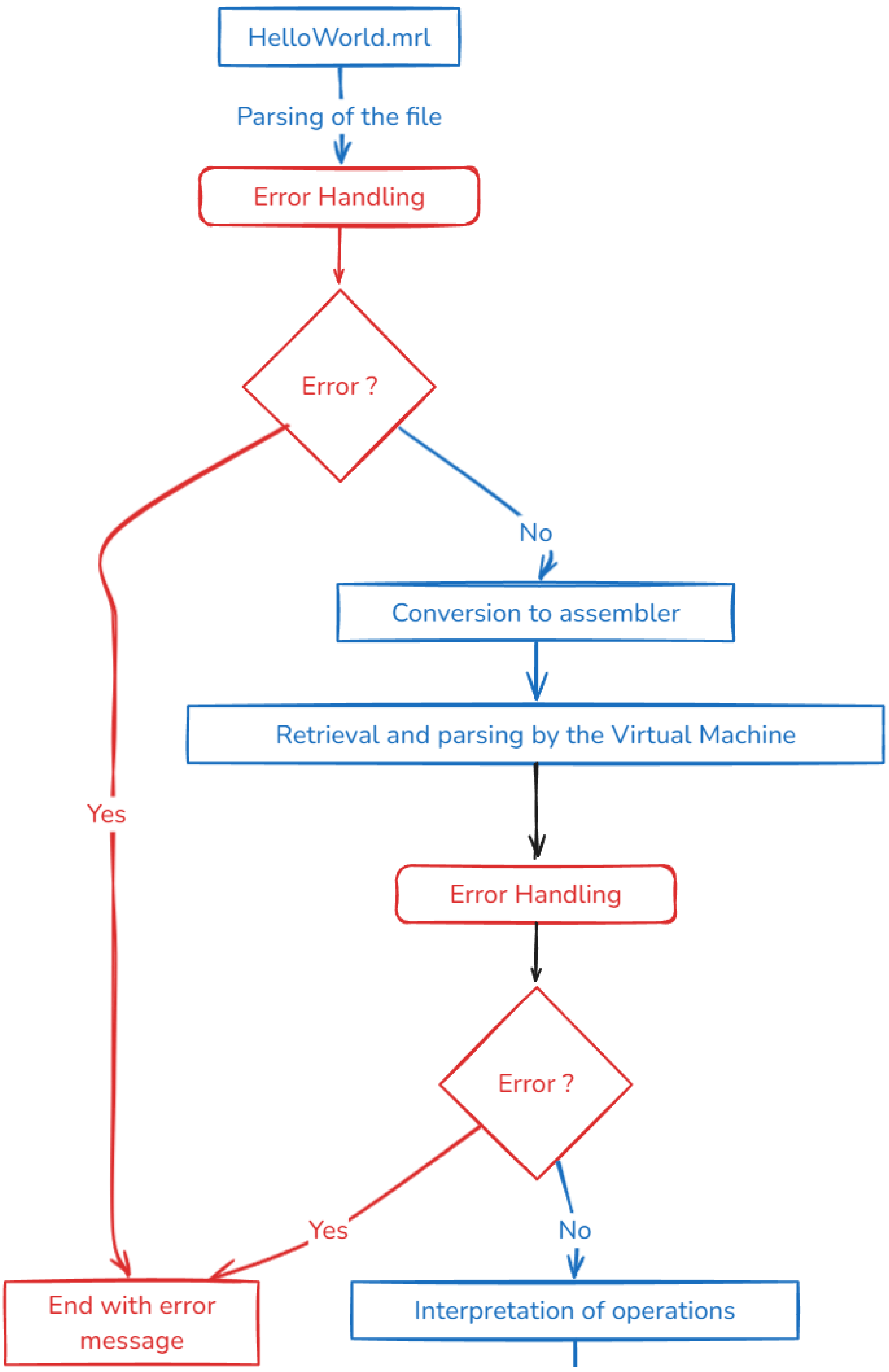
```
<binary-expr> ::= <expression> <binary-operator> <expression>
<binary-operator> ::= "+" | "-" | "|" | "&" | ">>" | "<<" | "^" | "*" | "

<prefix-expr> ::= <prefix-operator> <expression>
<prefix-operator> ::= "!" | "-" | "++" | "--" | "~"

<postfix-expr> ::= <expression> <postfix-operator>
<postfix-operator> ::= "++" | "--"

<ternary-expr> ::= <expression> "?" <expression> ":" <expression>

<grouped-expr> ::= "(" <expression> ")"
<assignment> ::= <variable> <assign-operator> <expression>
<assign-operator> ::= "=" | "+=" | "-=" | "**=" | "*=" | "/=" | "%=" | "|
<statement> ::= <declaration>
             | <expression-statement>
             | <if-statement>
             | <while-statement>
             | <return-statement>
             | <break-statement>
             | <continue-statement>

<declaration> ::= <variable-declaration> | <function-declaration>
<variable-declaration> ::= <type> <identifier> ("=" <expression>)? ";"
<function-declaration> ::= <type> <identifier> "(" <parameter-list>? ")"
<parameter-list> ::= <type> <identifier> ("," <type> <identifier>)*
<function-body> ::= <block>
<block> ::= "{" <statement>* "}"

<expression-statement> ::= <expression> ";"

<if-statement> ::= "if" "(" <expression> ")" <block> <else-if>* <else>?
<else-if> ::= "else if" "(" <expression> ")" <block>
<else> ::= "else" <block>

<while-statement> ::= "while" "(" <expression> ")" <block>

<return-statement> ::= "return" <expression>? ";"

<break-statement> ::= "break" ";"
<continue-statement> ::= "continue" ";"

<import-statement> ::= "import" <string> ";"
```

Compilation Process

The entire Maryl compilation process is explained here


HelloWorld.mrl
Parsing of the file
Error Handling
Error ?
Yes
No
Conversion to assembler
Retrieval and parsing by the Virtual Machine
Error Handling
Error ?
Yes
No
End with error message
Interpretation of operations

"Hello World !"


HelloWorld.mrl
Parsing of the file
Error Handling
Error ?
Yes
No
Conversion to assembler
Retrieval and parsing by the Virtual Machine
Error Handling
Error ?
Yes
No
End with error message
Interpretation of operations

1. Input File

- The process begins with a file named `HelloWorld.mrl`.

---

2. Parsing of the File

- The file undergoes a parsing step to ensure it adheres to the required syntax and format.

---

3. Error Handling (Initial Parsing Phase)

- After parsing:
  - **If an error is found**: The error is flagged and sent to the error-handling mechanism.
  - **If no error is found**: The process continues to the conversion step.

---

4. Conversion to Assembler

- The parsed file is converted into assembler code, a low-level representation of the program.

---

5. Virtual Machine Processing

- The assembler code is retrieved and parsed by the Virtual Machine (VM) for further analysis.

---

6. Error Handling (Virtual Machine Phase)

- At this stage:
    - **If an error is encountered**: The error is processed, and the workflow terminates with an error message.
    - **If no error is encountered**: The VM proceeds to interpret the operations.

---

7. Interpretation of Operations

- The Virtual Machine interprets the assembler instructions step by step.

---

8. Final Output

- Upon successful execution, the program outputs the string:
  "Hello World!"

Error Handling Summary

- Errors can occur in two key stages:
    1. **During file parsing**
    2. **During Virtual Machine parsing**
- In both cases, the system gracefully terminates and displays an appropriate error message.

This structured workflow ensures that errors are detected early, enabling efficient debugging and reliable execution.

Add Features on Maryl

Here is all you need to know to implement a feature on Maryl

Adding New Syntax

To add new native elements to the language, you'll have to use *Megaparsec.* Go to `src/Parsing/ParserAst.hs` and follow these steps:

- Parse a new symbol:
  - Create a function that parses your new symbol using *Megaparsec*'s `Parser` (e.g.: pPointer).
  - Call your new function in one of the next steps.
- Add a new type:
  - In the `MarylType` data type, add your new type.
  - To parse it, add it to the list of types in the `types'` function. If the type is more complexe and needs extra informations, add its parsing function to the `types` function directly.
  - Add the translation from `String` to `MarylType` in the `getType` function.
  - Finally, don't forget to update the documentation so people know which type they can use!
- Add a new statement:
  - Statements like `return`, `if`, loops, should be added to the `pTerm` function.
  - Add the newly parsed statement as a new `Ast` type in the structure.
- Add a new assignment (`=`):
  - Just add your new assignment symbol in the `eqSymbol` function.
- Add a new operator:
  - Maryl parsing for operators is done with the built-in `makeExprParser` of *Megaparsec*, so you just have to add your new operator to the `operatorTable`.

Now that you've added your newly parsed value, you have to add it to the evaluation, in the `src/Eval` folder.

Adding Built-in Functions

To add new built-in functions to Maryl, follow these steps:

1. Define your operator in `VirtualMachine/Interpreter.hs`:

```
operatorMyFunction :: [Value] -> VmState [Value]
operatorMyFunction (x:xs) = -- Implement your function logic here that re
operatorMyFunction _ = fail "Invalid arguments"
```

2. Register your operator in the `operators` list:

```
operators = [
    // ...existing operators...
    ("myfunction", Op operatorMyFunction)
]
```

Extending the Syntax

To add new syntax elements to the language:

1. Add a new instruction type in `VirtualMachine/Instructions.hs`:

```
data Inst =
    // ...existing instructions...
    | MyNewInstruction Type
```

2. Create a constructor for your instruction:

```
myNewInst :: Label -> Value -> Instruction
myNewInst l x = Instruction <code> "mynewop" (MyNewInstruction x) l
```

3. Add a parser in `VirtualMachine/Parser.hs`:

```
parseMyNewInst :: Parser Instruction
parseMyNewInst = lexeme (parseInstruction myNewInst "mynewop" parseVal)
```

4. Add your parser to the `keyWords` list:

```
keyWords = [
    // ...existing keywords...
    parseMyNewInst
]
```

5. Implement the execution logic in `VirtualMachine/Interpreter.hs`:

```
execInstruction (Instruction _ _ (MyNewInstruction x) _) =
    -- Implement your instruction logic here
```

Integration with External Libraries

To integrate external libraries with Maryl:

1. Create a new operator that wraps your external function in `VirtualMachine/Interpreter.hs`
2. Use the `io` function to lift IO operations into the VmState monad:

```
operatorExternalFunc :: [Value] -> VmState [Value]
operatorExternalFunc args = do
    result <- io $ yourExternalFunction args
    return [result]
```

3. Register your external function as an operator following the steps in "Adding Built-in Functions"

Or, create a file ending in `.mrl` and its code can be imported in your programs by simply typing:

```
import "library.mrl";
```

Remember to:

- Handle errors appropriately using `fail` or `eitherS`

- Maintain type safety by properly converting between Value and your external types
- Document any new dependencies in the project's cabal file

Adding a New Operator (Evaluation stage)

To add a new operator, follow these steps:

1. **Define the Operator's Functionality**:
    - Implement the evaluation logic in `Eval.Ops`.
    - The function signature should match others, e.g., `evalNewOp :: Memory -> Ast -> Ast -> Either String (Ast, Memory)`.
2. **Register the Operator**:
    - Add the operator to the `defaultRegistry` in `Evaluator.hs`:

    ```
    defaultRegistry =
        Map.fromList
            [ ("+", evalAdd),
              ...
              ("newOp", evalNewOp) -- Add here
            ]
    ```
3. **Specify Return Types**:
    - Update `assocOpExpectation` in `Eval.Ops` to define the valid return types for the new operator:

    ```
    assocOpExpectation =
        Map.fromList [
            ...
            ("newOp", [Int, Double]) -- Add the return types
        ]
    ```

Adding a New Operator (Translation stage)

To support a new operator:

1. **Update `translateOpInst`**:
    - Add the new operator and its corresponding VM instruction:

```
translateOpInst "newOp" = call Nothing "newOp" -- your VM instruct
```

   - If the instruction doesn't exist in `VirtualMachine.Instructions`, define it there first.

2. **Update `isSingleOp`**:
   - Ensure the operator is marked as a single operator if it doesn't represent a compound assignment:

     ```
     isSingleOp "newOp" = True
     ```

3. **Test the Operator**:
   - Write test cases for the operator to verify correct translation and VM behavior.

Adding New AST Node (Evaluation Stage)

1. **Define the Node**:
   - Extend the `Ast` type in `Parsing.ParserAst` to include the new node.
2. **Implement Evaluation Logic**:
   - Add a case for the new node in `evalNode`:

     ```
     evalNode mem (AstNewNode args) =
         -- Logic to evaluate the new node
         ...
     ```

3. **Update Supporting Functions**:
   - If the new node affects structures like `evalAST`, `applyOp`, or memory functions, update those as needed.

Adding New AST Node (Translation Stage)

To support a new AST node:

1. **Extend the AST Definition**:
   - Add the new node to the `Ast` data type in `Parsing.ParserAst`.
2. **Implement Translation Logic**:
   - Add a case for the new node in `translateAST`:

```
translateAST (AstNewNode args) mem =
    (instructions, updatedMem)
```

- Generate appropriate instructions using the `VirtualMachine.Instructions` module.

3. **Update Supporting Functions**:
   - If the new node interacts with specific constructs (e.g., conditionals or loops), update the relevant translation functions like `translateIf` or `translateLoop`.

Adding New Maryl Types

1. **Define the Type**:
   - Extend the `MarylType` enumeration in `Parsing.ParserAst`.
2. **Update Validation**:
   - Update `isValidType` and `isSameType` to include the new type.
3. **Add Usage Logic**:
   - Modify relevant sections like `evalDefinition`, `evalAssignType`, or `evalBinaryRet` to handle the new type.

Language Review from Security Perspective

Language and Inspiration :

- **C:** Inspiration for the basic syntax, while addressing common security pitfalls like manual memory management. For example, the absence of pointers and non-constant global variables eliminates risks such as memory overflows and unpredictable global state changes.
- **Go:** Adoption of clear syntax for types, particularly for lists and anonymous types, which reduces errors caused by ambiguity in data structure declarations and usage.
- **Rust:** The ability to access list elements using multiple indices (e.g., `list[0,1]`) is a convenient *syntactic sugar* feature that adds expressiveness without compromising safety.

Security and Memory Management:

- The inclusion of a **garbage collector** ensures automatic memory management, significantly reducing risks like memory leaks and dangling pointers.
- **Structs are constant by default**, enhancing immutability and preventing unintended modifications, which is especially important for maintaining data integrity and reducing side effects.
- Lack of pointers prevents errors associated with direct memory manipulation.
- Absence of non-constant global variables promotes encapsulation and improves code readability, reducing the risk of unintended side effects.

Maryl draws from the performance and simplicity of languages like C and Go, while incorporating modern mechanisms, such as a garbage collector, to ensure greater security and avoid common programming errors. These choices highlight a focus on robustness, safety, and ease of use by leveraging the strengths of your inspirations and mitigating their weaknesses.

Security Features

In relation with the review

Building on the strength, the following features were implemented to ensure security and robustness:

- **Garbage Collection**: Automatic memory management eliminates risks associated with manual allocation and deallocation, such as memory leaks and dangling pointers.
- **Default Immutable Structs:** Structs are constant by default, preventing accidental or malicious modifications. This immutability ensures data integrity, simplifies debugging, and reduces side effects.
- **No Pointers:** By removing pointers, your language avoids the vulnerabilities associated with direct memory manipulation, such as buffer overflows and segmentation faults.
- N**o Non-Constant Global Variables:** This design choice promotes encapsulation and prevents unpredictable behavior stemming from uncontrolled global state changes.
- **Safe and Explicit Type System**: Inspired by Go, the type system prioritizes clarity and enforces type safety, reducing errors from implicit type conversions or misuse.
- **Syntactic Sugar for Clarity**: Features like list[0,1] access improve code readability and reduce opportunities for subtle bugs without introducing unsafe behaviors.
- VM - errors of builtin are handled by the VM directly

Vm Built-ins functions

This a non-exhaustive list of the available builtin functions.

- Maths operators (add, sub, div, mod, mul) those take 2 numbers as parameters, returns number.
  - equivalent to
    - add = `+`
    - sub = `-`
    - div = `/`
    - mod = `%`
    - mul = `*`
    - pow = `**`
- Comparison operators (less, greater) those take 2 numbers as parameters, returns `bool`.
  - equivalent to
    - less = `<`
    - greater = `>`
- Equality operators (eq, neq) those take 2 Values of any type as parameters, returns `bool`.
  - equivalent to
    - equal = `==`
    - not equal = `!=`
- Boolean operators (and, or, not) those take 2 booleans (or 1 for not) as parameters, returns `bool`.
- Binary operators (band, bor, xor, shiftR, shifR) that takes two integers as parameter, returns integer.
  - equivalent to:
    - binary and = `&`
    - binary or = `|`
    - xor = `^`
    - right shift = `>>`

    - left shift = `<<`

- `set` is an operator that takes a list, an index and a value, and sets the value at that index if it is inside of the list size. (Also works on strings)
- `get` is an operator that takes a list, an index and returns the value at that index. (Also works on strings)
- `push` pushes the var passed as second parameter into the list passed as first parameter. (Will extend the list length)
- `pop` removes the element at the index passed as the second parameter into the list passed as first parameter. (Will shrink the list length)
- `print` - Takes one argument that is not constrained to any built-in types.
- `open` - Takes a filepath as an argument, represented by a string and returns its corresponding file descriptor as an int. Will return -1 on failure.
- `close` - Takes a file descriptor as an argument, represented by an int 0 if it successfully was able to close it.
- `read` - Takes a file descriptor (int) -1 on failure.
- `append` - Take file descriptor (int). Writes data at the end of the file, or returns -1 on failure.
- `write` - Takes a file descriptor (int). -1 on failure.
- `getLine` - Takes a file descriptor (int). -1 on failure
- `readFile` - Takes a filepath (string).
- `writeFile` - Takes a filepath (string).
- `appendFile` - Takes a filepath (string).
- `listPop` - Takes a list and an integer index (int).
- `listPush` - Takes a list, an integer index (int) and a value.
- `error` - error - Takes a string. Logs or displays an error message
- `strcat` - Takes two strings.
- `strlen` - Takes a string.
- `substr` - Takes a string and two numbers.
- `strcmp` - Takes two strings.
- `setField` - setField - Takes a struct, a string and a value.
- `getField` - Takes a struct and a string (as param).

Optimisation

Compiler

DList

dlist
Hackage

List-like types supporting **O(1)** `append` and `snoc` operations.

DList are both incorporated in the *Evaluation* stage and the *Translation* stage.

- As the **evaluation** stage consist of working with a list of `Ast` Nodes, **DList** simplifies the evaluation of the original list of `Ast` passed by the *Parser*.

Each transformed AST (`transformedAst`) is appended to the `DList` using `DList.snoc`.

Unlike `:` in regular lists, which prepends, **`DList.snoc`** appends to the end efficiently in `O(1)O(1)O(1)` time.

When recursion of all `evalNode` finishes, the accumulated **`DList`** is converted to a regular list using **`DList.toList`**.

This conversion is `O(n)O(n)O(n)`, but it only happens once for the entire result, unlike repeated `O(n)O(n)O(n)` appends for regular lists.

- As for the **translation** stage, it consists of initialising a **DList** of Instruction as they're handling a list of `Ast` and recursively appending `Instruction` type at every translation.

Sequentially appending instructions (e.g., with **`DList.append`**) avoids the overhead of repeatedly copying lists.

Recursive functions benefit significantly, as each recursive step appends new instructions.

During translation, instructions are generated and appended incrementally without worrying about performance penalties.

`DList` is converted to a standard list only at the end (when serializing), minimizing overhead.

Using `DList` provides a clear separation of incremental construction (with **`DList.append`**) and final output (via **`DList.toList`**).

Example Usage

Appending with Standard List:

```
let xs = [1, 2, 3]
let ys = [4, 5]
xs ++ ys  -- O(n) where n = length xs
```

Appending with *DList*:

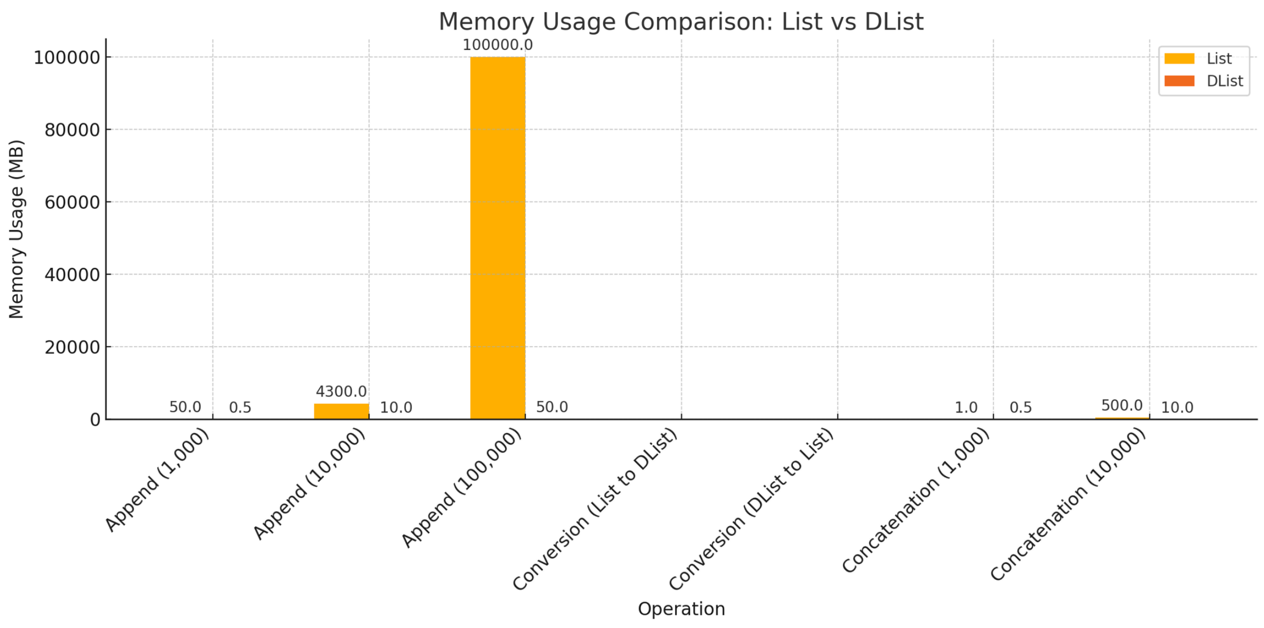
```
import Data.DList

let xs = DList.fromList [1, 2, 3]
let ys = DList.fromList [4, 5]
let zs = xs `DList.append` ys  -- O(1)
DList.toList zs  -- Convert back to [1, 2, 3, 4, 5]
```

Benchmark

Benchmarking results indicate that Haskell's `DList` (difference list) can significantly outperform standard lists (`[]`) in append-heavy operations.

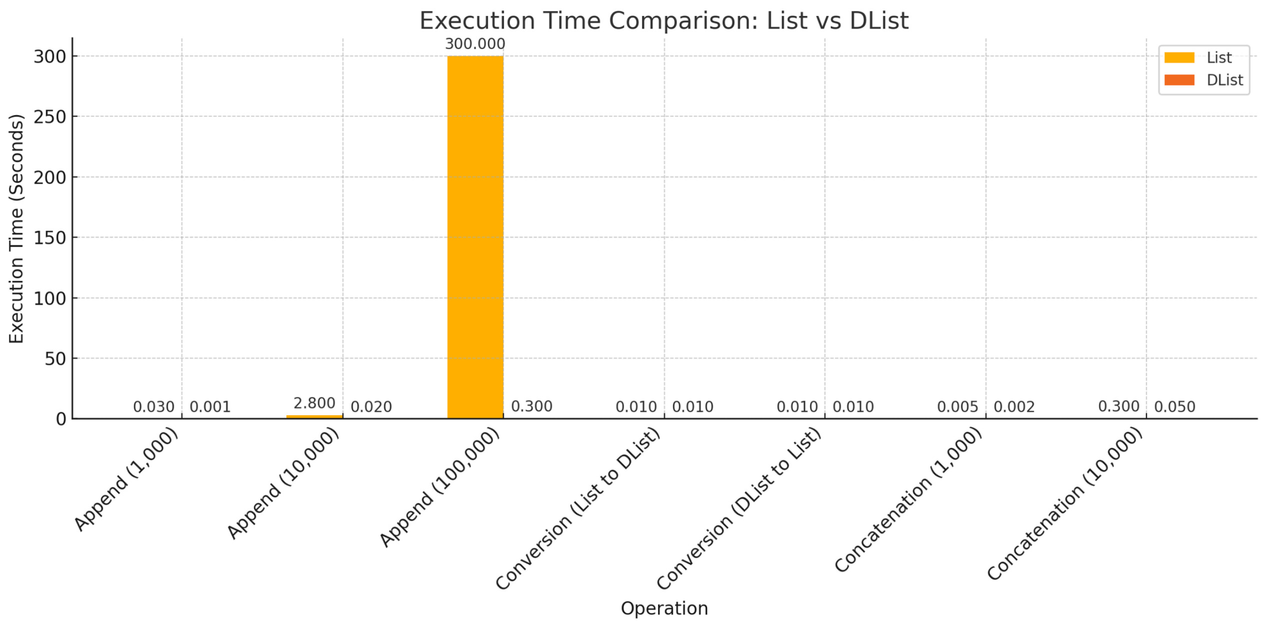
In a micro-benchmark, appending a single character 10,000 times to a standard list took approximately 2.80 seconds and consumed around 4.3 GB of memory. In contrast, performing the same operation using `DList` completed in about 0.02 seconds and used approximately 10 MB of memory.

Source: https://www.cs.umd.edu/class/spring2023/cmsc433/DList.html



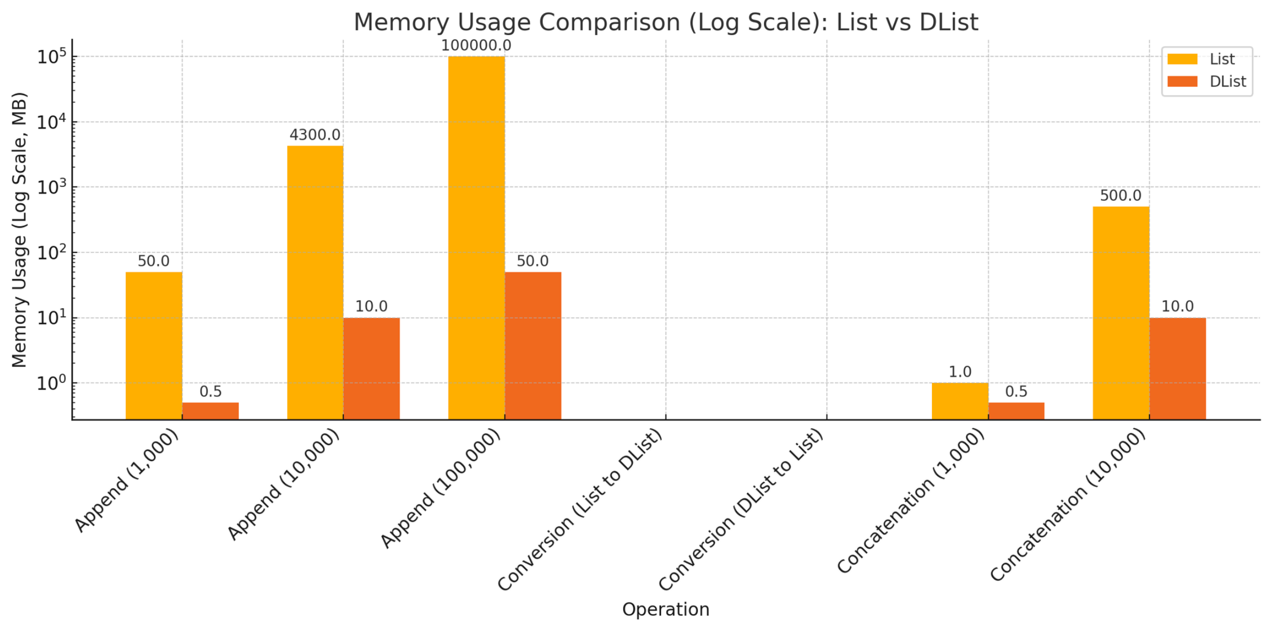
Execution Time Comparison

Shows the significant time advantage of `DList` for append-heavy operations and concatenations, especially as the input size grows.



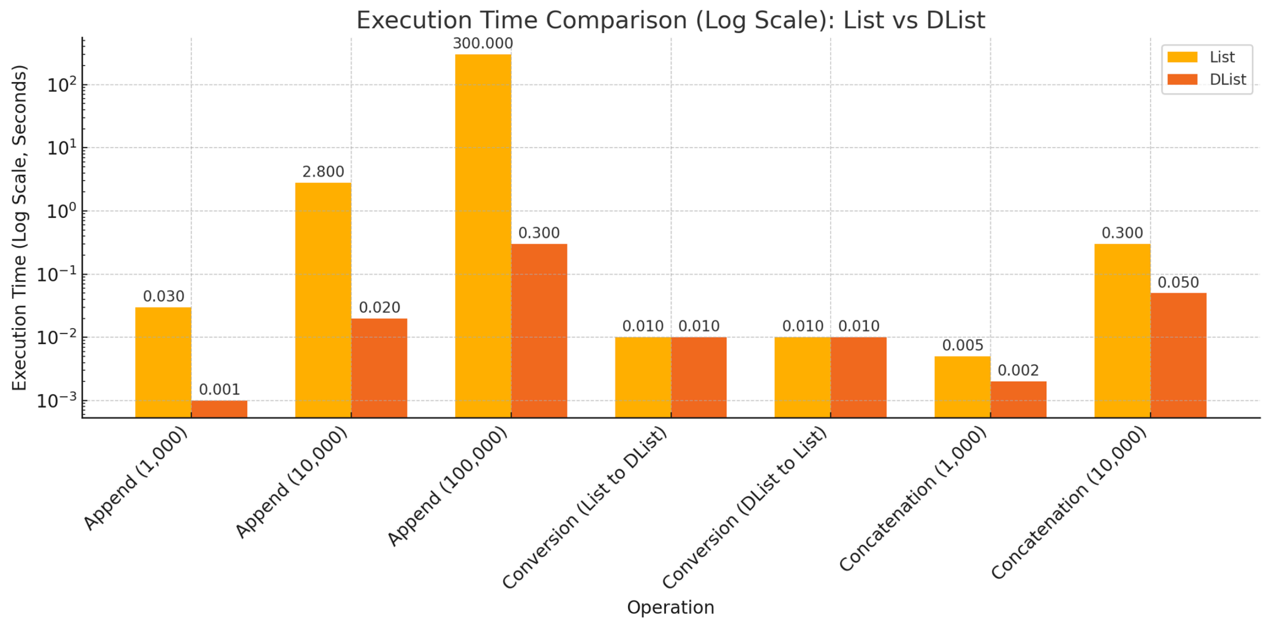
Memory Usage Comparison

Demonstrates how `DList` uses much less memory for similar operations, making it more efficient for large-scale appending and concatenation tasks.



Logarithmic Execution Time Comparison

Clearly shows the massive time efficiency of DList compared to List, especially for larger input sizes. Even small differences in DList times are now visible.



Logarithmic Memory Usage Comparison

Highlights the significantly lower memory usage of DList for append-heavy operations and concatenations.

Source: https://github.com/haskell/core-libraries-committee/issues/236

Inline constant operations

We evaluated the optimization of inline constant math operations by precomputing expressions when input values were constants. This approach involved performing calculations at compile-time, thus reducing runtime overhead and improving execution efficiency. By evaluating these operations before passing instructions to the virtual machine (VM), we minimized the number of instructions executed, leading to more streamlined and performant code.

This technique is particularly advantageous for optimizing performance in scenarios with numerous repetitive computations involving constant values.

Evaluator

The `Evaluator.hs` module is a core component of the Maryl programming language. It evaluates Abstract Syntax Tree (AST) nodes and provides functionality for processing variables, functions, structures, loops, and binary operations. This document provides detailed information about the functionality and usage of the evaluator.

Table of Contents

Overview

The `Evaluator` processes the Maryl AST by evaluating nodes in memory. It supports:

- Variable and constant declarations.
- Function definitions and calls.
- Loop and conditional execution.
- Binary and unary operations.
- Structure definition and instance creation.

The evaluator works closely with the `Memory` module for managing runtime data.

Core Concepts

Memory Management

The evaluator uses a `Memory` type to store variables, structures, and functions. Key memory-related operations include:

- **Reading memory**: Fetching variables or functions by name.
- **Updating memory**: Modifying existing variables or adding new definitions.
- **Freeing memory**: Isolating a memory space (e.g., during function calls).

Supported AST Types

The evaluator processes the following AST types:

- **Variables** (`AstDefineVar`): Define and update variables.
- **Functions** (`AstDefineFunc`, `AstFunc`): Define and call functions.

- **Structures** (`AstDefineStruct`, `AstStruct`): Define and instantiate structures.
- **Binary and Unary Operators** (`AstBinaryFunc`, `AstPrefixFunc`, `AstPostfixFunc`): Perform mathematical and logical operations.
- **Control Flow** (`AstIf`, `AstLoop`): Handle conditionals and loops.

Functionality

Core Functions

`evalNode`

Evaluates a single AST node.

- **Parameters**:
    - `Memory`: The current runtime memory.
    - `Ast`: The AST node to evaluate.
- **Returns**: Either an updated AST with modified memory or an error string.
- **Responsibilities**:
    - Handle variable assignments and updates.
    - Process function calls and loop definitions.
    - Evaluate structure instances and declarations.

`evalAST`

Processes a list of AST nodes sequentially.

- **Parameters**:
    - `Memory`: The current runtime memory.
    - `[Ast]`: A list of AST nodes.

- **Returns**: Either an updated list of evaluated ASTs with modified memory or an error string.
- **Use case**: Execute a program block or function body.

`applyOp`

Applies a binary operation.

- **Parameters**:
    - `Memory`: The current runtime memory.
    - `String`: The binary operator (e.g., `+`, `-`).
    - `Ast`: The left operand.
    - `Ast`: The right operand.
- **Returns**: Either the result of the operation with updated memory or an error string.
- **Use case**: Arithmetic and logical operations.

Binary Operations

The evaluator supports a range of binary operations including addition, subtraction, multiplication, division, and logical comparisons. These operations are implemented using a function registry (`defaultRegistry`) for modularity.

Structure Evaluation

Declaration

When defining a structure (`AstDefineStruct`), the evaluator:

1. Validates the field definitions.
2. Stores the structure in memory.

Instantiation

When creating an instance of a structure (`AstStruct`), the evaluator:

1. Verifies that the structure exists in memory.
2. Validates field values and types.
3. Returns a normalized structure instance.

Error Handling

The evaluator uses the `Either` type for error handling. Errors include:

- **Type mismatches**: When an AST node's type does not match the expected type.
- **Undefined variables or structures**: When a reference is made to a non-existent entity.
- **Invalid operations**: When an operator is applied to unsupported types.

Usage

To use the evaluator, you need to:

1. **Parse the source code** into an AST using the parser module.
2. **Initialize memory** with predefined structures or variables if needed.
3. **Call** `evalAST` with the AST and memory as input.

Example

```haskell
import Eval.Evaluator (evalAST)
import Memory (initializeMemory)
import Parsing.ParserAst (parseAST)

main = do
    let sourceCode = "int x = 10; x = x + 5;"
    case parseAST sourceCode of
        Left parseError -> print parseError
        Right ast ->
            case evalAST initializeMemory ast of
                Left evalError -> print evalError
                Right (result, memory) -> do
                    print result
                    print memory
```

Feel free to contribute or suggest improvements to this evaluator!