blisp - A Simple LISP Interpreter

Brian Hoffpauir
United States
bhoffpauirmail@gmail.com

***Abstract*—This report introduces `blisp`, a Scheme-like LISP programming language and interpreter written in Java. The `blisp` language implements core features found in LISP dialects such as atoms, lists, symbols, and lambdas. It supports both an interactive REPL environment and script execution modes, enabling users to evaluate expressions and build reusable code. While `blisp` is inspired by Scheme, it offers its own design choices, including a flexible API for embedded scripting. This document details the design and implementation of `blisp`, along with examples of usage and core language features like special forms and macros. Section VIII provides a complete reference for the `blisp` language.**

***Index Terms*—LISP, interpreters, languages, design, Java, APIs, libraries.**

I. Introduction

(BLISP)

Fig. 1. `blisp` logo.

```
(DEFINE (HELLO X)
  (PRINTLN "Hello, " X))

(HELLO "blisp")
```

`blisp` is a simple, Scheme-like **LISP** programming language library and interpreter written in Java. `blisp` attempts to implement the features that are fundamental to all **LISP** dialects such as atoms, lists, special forms, and lambdas. The language not adhere to any standard put forth by other **LISP** dialects, but it does attempt to closely resemble Scheme in both appearance and behavior. The language library implements a flexible **API** that can evaluate expressions or even support embedded scripting. The interpreter program provides a means to execute `blisp` programs or present the user with an interactive **REPL**.

TABLE I
LISP Progamming Languages

Language	Description
Common Lisp	Many features, support for multiple paradigms, multiple fast implementations.
Scheme	Minimalist functional **LISP** with focus on **DSL**s.
Racket	Multi-paradigm, Scheme-like **LISP** with an academic focus.
Clojure	Functional **LISP** on the JVM with Java interop support.

Table I lists several of the most common **LISP** programming languages and their features. blisp aims to provide the **LISP** ecosystem with a **LISP** dialect suitable for standalone scripts and easily embedding into Java applications.

II. Design

`blisp` inherits many of the design features found in other dialects of **LISP**, most notably Scheme. The following subsections describe the features of `blisp` and how they form the language.

A. Atoms

In `blisp`, atoms are the fundmental units of a program. Atoms represent the atomic/scalar values that can be represented in the language. Examples of atoms in `blisp` include: numbers, strings, booleans, characters, and symbols. Atoms are immutable after creation, and thus their values will not change during the course of a program. However, what can change is the atomic value bound to a particular symbol in the environment. This is discussed further in Section II-B.

B. Symbols

Symbols are the identifiers to which values may be bound. When the `blisp` interpreter encounters a token that represents a symbol, it looks up the value of this symbol within the current execution environment. `blisp` supports symbol binding in global and lexical environments. The `define` and `let` constructs are used to bind symbols to values in the global and lexical environments respectively. Symbols in `blisp` are case-insensitive. The special symbol, `nil`, can be used to represent the absence of a value. The `nil` symbol is equivalent to the empty list `()`.

In `blisp`, scopes are known as the environment. The global environment persists throughout execution and initially contains the default operations builtin to the language. When special forms such as `lambda` or `let` are used, lexical scopes are created for the parameters or let bindings respectively. See Section II-D and Section II-E for more information on lambdas and special forms respectively.

C. Lists

One data structure which is an inherent characteristic of **LISP** dialects is the list. `blisp` is no different in that all code structures are held internally in lists that resemble trees with other expressions as branches. In `blisp`, any symbol placed at the beginning of a list is looked up in the environment and used to invoke the lambda or special form bound to that

symbol with the arguments being the remainder of the list. `blisp` differs from some **LISP** dialects in that there is no `cons` pair construct. As such, there are also no supported `car` (access left element in a cons pair) and `cdr` (access right element) operations.

D. Lambdas

In `blisp`, lambdas are anonymous functions that offer fundamentals support for code reuse. Lambdas take the following form:

`(LAMBDA (`$param_1$ `...` $param_n$`)` $expr_1$ `...` $expr_n$`)`

where the last expression in the lambda body, exprn, it the value returned by invoking a lambda. The λ (Unicode character U+03BB) can be used in place of the `lambda` symbol. Lambdas can be bound to symbols, however it is more common to use the `define` construct. When lambdas are bound to symbols in `blisp`, they are then known as procedures. Lambdas are treated as first-class values, and can be bound to symbols or passed anonymously as arguments to other procedures. The syntax for invoking a procedure is as follows:

`(`$procsym$ arg_1 `...` arg_n`)`

E. Special Forms and Macros

In `blisp`, special forms are constructs that resemble lambdas or procedures, but do not conform to the traditional evaluation rules. In fact, the syntax used to create lambdas is itself a special form. What makes the `lambda` construct a special form is how its arguments are evaluated. Any procedure-like construct that does not evaluate its arguments in the typical greatest-depth-first and left-to-right fashion is a special form. Special forms such as `if` are implemented within the language library. Another similar feature in `blisp` are macros. The sole distinction between macros and special forms are that macros are not implemented within the language library, but with the `defmacro` construct instead.

III. Implementation

`blisp` is implemented as a language **API** library and an interpreter program that utilizes the library. The language **API** provides a interface the user can use to tokenize, parse, and evaluate expressions using the rules of the language. The choice language for the implementation of `blisp` is Java.

A. Language API Library

The `blisp` language **API** uses the object oriented paradigm as well as generic programming facilities. The API is composed of four main components: the tokenizer, parser, environment, and evaluator. Any functioning embedded application of `blisp` will use these four components.

1) The Tokenizer: The tokenizer component translates a input string of source code into a list of tokens. This list of tokens is then subsequently given as input to the parser. The tokenizer ignores white-space and skips over single-line comments (beginning with a `;` character and continuing until the end of the line). The syntax of the individual tokens is preserved in the tokenization stage. For example, tokenized strings are quoted and characters maintain the `\X` syntax.

Algorithm 1 Tokenizer Algorithm

Require: Input string `input`
Ensure: List of tokens `tokens`
Initialize `currentIndex` ← 0
Create empty list `tokens`
while `currentIndex` < `length(input)` **do**
  `ch` ← `input[currentIndex]`
  if `ch` is whitespace **then**
    `currentIndex` ← `currentIndex + 1`
    Continue
  end if
  if `ch` is `';'` **then**
    Skip characters until end of line
    Continue
  end if
  if `ch` is `''` **then**
    Add `''` to `tokens`
    `currentIndex` ← `currentIndex + 1`
    Continue
  end if
  if `ch` is `'('` or `')'` **then**
    Add `ch` to `tokens`
    `currentIndex` ← `currentIndex + 1`
    Continue
  end if
  if `ch` is `'"'` **then**
    Add result of `readString()` to `tokens`
    Continue
  end if
  if `ch` is `'`
  `'` **then**
    Add result of `readCharacter()` to `tokens`
    Continue
  end if
  if `ch` is a letter or symbol start **then**
    Add result of `readSymbolOrNumber()` to `tokens`
    Continue
  end if
  Throw `UnknownTokenException` if `ch` is invalid
end while
return `tokens`

2) The Parser: The parser component creates tree code structures from the output produced by the tokenization stage. In `blisp`, all non-atomic code expressions are represented internally as lists of objects, which may also be lists. These

trees of expressions can be nested to an arbitrary depth, or as much as the **JVM**'s stack space permits (the implementation is recursive). The parser steps through the output of the tokenization stage and creates internal representations of lists and atomic values as they are encountered. Tokens representing atoms are converted from the language syntax into values that can be used for language operations.

Algorithm 2 Parser Algorithm

```
1: Function parse(tokens)
2: if tokens is empty then
3:    Throw IllegalArgumentException
4: end if
5: return parseExpression()
1: Function parseExpression()
2: if no more tokens then
3:    Throw NoSuchElementException
4: end if
5: token ← next token
6: if token is "'" then
7:    return parseExpression()
8: else if token is "(" then
9:    list ← empty list
10:   while token is not ")" do
11:      list.add(parseExpression())
12:   end while
13:   return ListAtom(list)
14: else if token is ")" then
15:   Throw UnbalancedParenthesisException
16: else
17:   return parseAtom(token)
18: end if
1: Function parseAtom(token)
2: if token starts with "
   " then
3:    return CharacterAtom(token)
4: end if
5: if token starts with "" then
6:    return StringAtom(token)
7: end if
8: if token is a valid symbol then
9:    return SymbolAtom(token)
10: end if
11: if token is a valid number then
12:   return NumberAtom(token)
13: end if
14: Throw LispRuntimeException("Invalid
   token")
```

3) The Environment: In `blisp`, scope is referred to as the environment. The environment component is where symbols are bound to values. The builtin language procedures are all placed in the global environment and defined in the language API. The symbols used by special forms are reserved and cannot be rebound, unlike the language procedures. The `let` special form is used to create local environments (lexical scopes).

4) The Evaluator: The evaulator component takes the expressions produced at the parsing stage and evaluates them by resolving symbols within a given environment. When given certain atomic values such as characters, string, and numbers, the interpreter simply evaluates those atoms to themselves. Symbols are evaluated by resolving their value binding in the environment. Special forms are also processed in the evaluator.

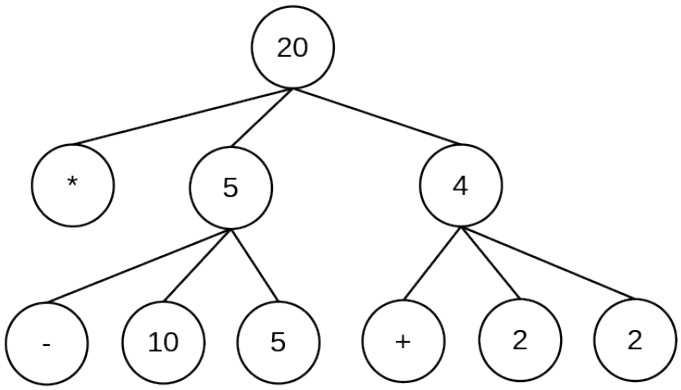

Fig. 2. The above tree represents the evaluation of the expression: (* (- 10 5) (+ 2 2)).

Upon encountering a list, the evaluator checks to see if the first element of the list is a symbol that corresponds to a special form or is bound to procedure in the environment. If the symbol represents a special form, then the unevaluated arguments are passed to its implementation. Special forms typically exert their own control over evaluation. If the symbol is bound to a procedure, then the arguments are evaluated from left to right and applied to the procedure call. Figure 2 presents how the expression `(* (- 10 5) (+ 2 2))` is evaluated. This expression is equivalent to the following arithmetic: $(10 - 5) * (2 + 2)$.

The evaluator makes heavy use of generic programming and pattern matching internally.

B. Standalone Interpreter

The standalone interpreter program uses the **API** library to provide a **REPL** style interface to the user. In **REPL** mode `blisp` prompts the user for an expression that it will subsequently evaluate. Upon encountering an error a dedicated exception will be thrown and a message will be displayed, otherwise the correct result of evaluating the expression will be shown. The interpreter also supports a script file execution mode. In this mode, the interpreter steps through each expression in a source code file and evaluates them in order from top to bottom. Evaluation stops upon reaching the end of the script file. See Figure 3 for an illustration.

Algorithm 3 blisp Interpreter Execution

Require: Command-line arguments `args`
Ensure: Execution of the interpreter in REPL or script mode

Initialize `options, mode, showTokens, showParser, showStackTrace, extendedPrint`
parseArguments(args)
if `mode` is `REPL` **then**
  displayREPLInfo()
end if
Create global environment `env`
initializeBuiltIns(env)
while `running` **do**
  if `mode` is `REPL` **then**
    displayPrompt()
  end if
  `line` ← readInput()
  if `line` is `EOF` **then**
    if `mode` is `SCRIPT_AND_REPL` **then**
      switchToREPLMode()
    else
      Break
    end if
  end if
  Append `line` to `expression`
  if `expression` has unmatched parentheses **then**
    Continue
  end if
  `tokens` ← Tokenizer.tokenize(expression)
  if `showTokens` **then**
    displayTokens(tokens)
  end if
  `parsedExpr` ← Parser.parse(tokens)
  if `showParser` **then**
    displayParsedExpression(parsedExpr)
  end if
  `result` ← Evaluator.evaluate(parsedExpr, env)
  if `mode` is `REPL` **then**
    print(result)
  end if
  Reset `expression`
end while
Return exit code

Below is an illustration of a typical **REPL** session with the `blisp` interpreter:

```
blisp v1.0 on Linux (amd64) version 6.6.17
Type "help" or "license" for more information.
Press Ctrl+D or type "(exit)" to exit this
    REPL.
>>> (+ 2 2)
4.0
>>> (DEFINE A (+ 2 2))
4.0
>>> (* A 2)
8.0
>>> ; Declare and immediately invoke a lambda:
>>> ((LAMBDA (X Y) (+ X Y)) A 2)
6.0
```

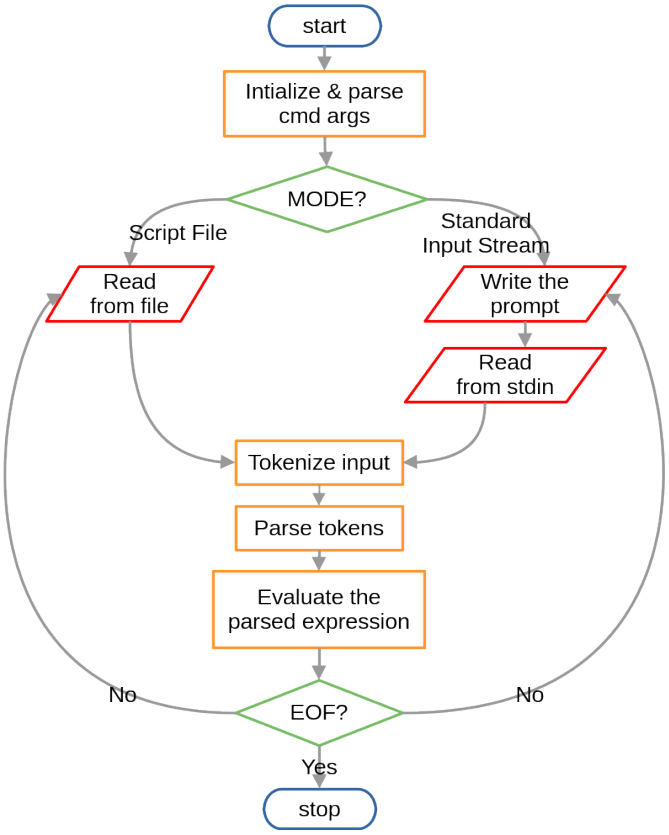

Fig. 3. The following flowchart represents the program flow of the interpreter (programs using `blisp` for embedded scripting will behave similarly).

The user is first presented with interpreter version information and the prompt. In this session, the first expression evaluate produces a value of 2. The second expression binds the result of evaluating the expression `(+ 2 2)`. The third expression multiplies the value bound to `a` by 2. The third input entered by the user is a single line comment. In `blisp`, any characters following a semi-colon are ignored by the parser. Finally, the fourth expression provides a lambda definition as the first argument in a list, and immediately invokes it with the arguments `a` and `2`. Recall from Section II-C, that the symbol at the start of a list is evaluated and the remaining elements are invoked as arguments to the procedure or special form referred to by the symbol.

To use the interpreter in the script execution mode the user must pass the path of the script file as a command-line argument to the interpreter program. For example, the following invocation of the interpreter program passes the path to the script file `test.blisp`[1]:

```
java -jar target/blisp-1.0-SNAPSHOT.jar
    scripts/test.blisp
```

[1].blisp is the choice extension for `blisp` source code files

The interpreter can also open a **REPL** after a script's execution with the `-i | --interactive` flag, thus the environment created by the script will still be accessible.

```
java -jar target/blisp-1.0-SNAPSHOT.jar
    scripts/test.blisp --interactive
```

Running the above command will have the following behavior:

```
  4.0
  20000.0
  5.0
  10.0 20.0 test true
  >>> (ADD 2 2) ; REPL session starts here
      (symbol bindings from script still
      available)
  4.0
  >>> (SUB 10 5)
  5.0
  >>> (EXIT) ; Leave REPL session
```

The interpreter also supports a simple **REPL** mode with use of the `-i | --interactive` flag:

```
java -jar target/blisp-1.0-SNAPSHOT.jar -i
```

IV. Using the Language

Those familiar with other **LISP** dialects will be at home when using `blisp`. While you have already seen some usage of the `blisp` interpreter in Section III-B, this section presents more involved examples. See Section VIII for a comprehensive language reference.

A. Example I - Core Features

This example aims to showcase most of `blisp`'s important features. The `example_01.blisp` script file for this example is located in the `scripts/` directory at the root of the repository. A similar convention will be used for all other examples in this section.

```
; Atoms
(PRINTLN 10 100.01 101e4) ; Numbers
(PRINTLN "this is a string" "t") ; Strings
(PRINTLN \A \B \C) ; Characters
(PRINTLN TRUE) ; This is a boolean
(PRINTLN NIL) ; nil is a special symbol

; Lists
(PRINTLN (LIST 1 2 3))

; Symbol bindings
(DEFINE A 100) ; Bind 100 to the symbol a
(PRINTLN A)

; Lambdas
(DEFINE ADD (LAMBDA (A B) (+ A B))) ; Create
    and bind lambda to symbol add
(DEFINE SUB (LAMBDA (A B) (- A B))) ; Can use
    unicode to create a lambda

; Procedures
; Bind MUL to procedure with arguments A & B
    to the body expr
(DEFINE (MUL A B)
  (* A B))

(PRINTLN (MUL 4 4)) ; Calls mul with two
    arguments, prints 16

; Bulitin procedures
(PRINTLN (+ 2 2 4 9 10 22)) ; Arithmetic
    operations

(PRINTLN (MOD 242 5)) ; Math procedures

(PRINTLN (LIST? (LIST 1 2 3))) ; Atom type
    predicates

; Special forms
(IF (= 2 2)
  (PRINTLN "True")
  (PRINTLN "False"))

; Prints "test" and returns 10
(PRINTLN (BEGIN (PRINTLN "test")
                10))
```

B. Example II - Procedures & Recusion

This example showcases how recursion works in `blisp`. This example is from `example_02.blisp`.

```
; Recursion (nth factorial)
(DEFINE (FACT N)
  (IF (= N 1)
    1
    (* N (RECUR (- N 1)))))
; Procedure to display the nth factorial
(DEFINE (SHOW-FACT N)
  (PRINTF "fact(%d) = %d\n" N (FACT N)))

(SHOW-FACT 1)
(SHOW-FACT 2)
(SHOW-FACT 3)
(SHOW-FACT 4)
(SHOW-FACT 5)

; Recursion (nth number in the Fibonacci
    sequence)
(DEFINE (FIB N)
  (IF (< N 2)
    N
    (+ (RECUR (- N 1)) (RECUR (- N 2)))))

(PRINTF "fib(0) = %d\n" (FIB 0))
(PRINTF "fib(1) = %d\n" (FIB 1))
(PRINTF "fib(2) = %.0f\n" (FIB 2))
(PRINTF "fib(3) = %.0f\n" (FIB 3))
(PRINTF "fib(4) = %.0f\n" (FIB 4))
(PRINTF "fib(5) = %.0f\n" (FIB 5))
```

C. Example III - List Processing

This example showcases the list processing features of `blisp`. This example is from `example_03.blisp`. The list processing procedures `MAP`, `FILTER`, and `REDUCE` utilize the first-class properties of lambdas in `blisp`. For example, the first argument of the `MAP` procedure is a lambda that takes an element from the list at a time. The return value of this lambda is the transformed element in the new list created

by the mapping operation. `FILTER` takes a predicate lambda which takes an element argument and returns a boolean value to indicate whether or not the element should be in the list created by the filter operation. `REDUCE` performs a left fold operation with the second argument as the initial value. The first argument is the reduction lambda which takes the accumulator variable and current iteration element as arguments. It returns the new value of the accumulation variable.

```
(DEFINE LST1 (LIST 1 2 3 4 5))
(PRINTLN "List 1: " LST1)

; List transformations using the map procedure
(PRINTLN "Mapping over List 1: " (MAP (LAMBDA
    (N) (+ N 2)) LST1))

; List transformations using the filter
    procedure
(PRINTLN "Filtering List 1: " (FILTER (LAMBDA
    (N) (> N 2)) LST1))

; Accumulating a result using the reduce
    procedure (sum)
(PRINTLN "Reducing List 1: " (REDUCE + 0
    LST1))

; Summing the even numbers up to 10
(REDUCE + 0 (RANGE 0 (INC 10) 2))
```

V. Conclusion

Developmening a Scheme-like LISP dialect like `blisp` demonstrates the feasibility of building a flexible, embedded scripting language in Java. `blisp` supports the core features of Scheme, such as atoms, symbols, lists, lambdas, and special forms. The interpreter supports both **REPL** and script execution modes, providing users with an environment to experiment with and evaluate **LISP**-like code. The language library provides a means of integration into new or existing Java applications, offering users the ability to evaluate expressions and build reusable embedded code. While `blisp` stays true to key concepts found in **LISP** dialects, it also introduces some custom design choices that distinguish it from other implementations, making it a practical tool for learning and embedding **LISP**-like functionality in Java applications.

VI. Future Work

Although `blisp` has achieved many of its initial goals, there are several areas for further improvement and development. First, adding support for more advanced data structures, such as vectors or hash maps, would enhance its usability. Maps are crucial and foundational in **LISP** dialects such as Clojure. Additionally, implementing proper **TCO** would improve performance for recursive functions, aligning `blisp` closer to Scheme standards. Tail recursion is property of procedures which execute an interative process in constant space, even if they have a recursive implementation [1]. While the methods of implementation are somewhat limited due to the **JVM**'s own lack of **TCO**, a solution might be able to use imperative loops under the hood. Another potential area of

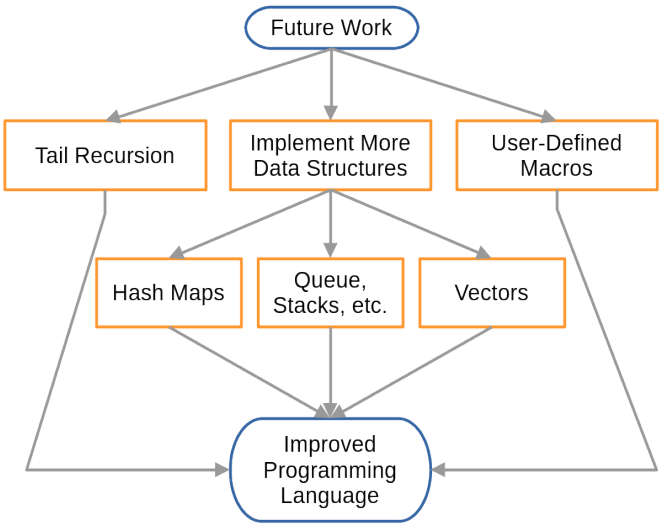

Fig. 4. Future work areas for `blisp`.

improvement is extending the macro system to allow for more complex code expansion and optimizations. Furthermore, expanding the library to include more built-in procedures, as well as incorporating features like concurrency or multi-threading, would make `blisp` suitable for larger projects with greater performance and functionality demands. A module/package system would help reduce name collisions and increase the viability of using `blisp` in large embedded projects. Finally, creating a more robust error handling and debugging system would assist users in developing more complex scripts and embedded applications.

VII. Acronyms

This section describes any acronyms that were used in this document.

API Application Programming Interface
DSL Domain-Specific Language
JAR Java Archive
JVM Java Virtual Machine
LISP LISt Processing
REPL Read Evaluate Print Loop
TCO Tail-Call Optimiziation

References

[1] H. Abelson and Gerald Jay Sussman, "Structure and Interpretation of Computer Programs, second edition." MIT Press, 1996.

VIII. Language Reference

This section presents a list of all the features and operations of the `blisp` language and a syntactical/behavioral description for each.

A. Atoms

An atom in `blisp` is a simple, indivisible element. These can be numbers, booleans, or strings.

Syntax: Atoms are directly represented as themselves.

Description: Atoms in `blisp` include:

- **Symbols**: Identifiers to which values can be bound.
- **Numbers**: Can be integers or floating-point numbers.
- **Booleans**: Represent truth values, with `true` for true and `false` for false.
- **Strings**: A sequence of characters enclosed in double quotes.
- **Characters**: A single character preceeded by a backslash.

Numbers will be given a distinct type, integer or double, based on the literal you use to declare them. In arithmetic operations, these underlying types will be promoted to doubles as necessary.

Examples:

```
sym      ; A symbol
42       ; A number (integer)
42.99    ; A number (floating-point)
42e+9    ; A number (scientific notation)
true     ; A boolean true
false    ; A boolean false
"hello"  ; A string
\\A      ; A character
\\0x41   ; A character (Unicode A)
```

B. Symbols

Symbols in `blisp` are identifiers that refer to values or procedures.

Syntax: `symbol`

Description: Symbols are used to name variables or procedures in the environment. They can consist of letters, numbers, and certain special characters, but cannot start with a number.

Examples:

```
x       ; A symbol referring to a variable
my-proc ; A symbol referring to a procedure
+       ; A symbol for the addition operator
```

C. Lists

Lists are one of the core data structures in `blisp`. They consist of a sequence of elements enclosed in parentheses.

Syntax: (`element`$_1$ `element`$_2$ `...` `element`$_n$)

Description: A list can contain atoms, symbols, or other lists (making it a recursive structure). The first element in a list typically represents a function or special form, and the remaining elements are treated as arguments.

Examples:

```
(1 2 3)                 ; A list of numbers
(+ 1 2 3)               ; A list
    representing a procedure call
(define (square x) (* x x)) ; A list
    defining a procedure
```

Operations:

- `first`: Returns the first element of a list.
- `rest`: Return the remainder of a list.
- `last`: Returns the last element of a list.
- `nth`: Returns the n^{th} element in a list.
- `cons`: Constructs a new list by prepending an element to an existing list.
- `count`: Returns the number of elements in a list.

Examples:

```
(first '(1 2 3))       ; Returns 1
(rest '(1 2 3))        ; Returns (2 3)
(last '(1 2 3))        ; Returns 3
(nth 1 '(1 2 3))       ; Returns 2
(cons 0 '(1 2 3))      ; Returns (0 1 2 3)
(count (list 1 2 3))   ; Returns 3
```

D. Special Forms

Special forms are language constructs or control structures that don't follow the normal evaluation rules.

`define`

Syntax 1: (`define` sym $value$)

Syntax 2: (`define` ($procsym$ $param_1$ `...` $param_n$) $exprs$)

Description: Binds value to the symbol sym in the current environment (Syntax 1). Bind a procedure to the symbol $procsym$ that has parameters $param_1$ through $param_n$ with $exprs$ as the procedure body (Syntax 2).

Example:

```
(define a 2) ; Binds the value 2 to the
    symbol a
(define (add x y) (+ x y)) ; Create a
    procedure with 2 args bounds to the
    symbol add
```

`lambda`

Syntax: (`lambda` ($param_1$ `...` $param_2$) $expr_1$ `...` $expr_n$)

Invocation Syntax: ((`lambda` ($params$) $exprs$) arg_1 `...` arg_n)

Description: Create a lambda with parameters $param_1$ through $param_2$.

Example:

```
(lambda (a b) (+ a b)) ; Returns the
    lambda
((lambda (a b) (+ a b)) 2 2) ; Invoke the
    lambda with arguments 2 & 2
```

`if`

Syntax: (`if` $cond$ $expr_1$ $expr_2$)

Description: Evaluates $cond$, if cond is true $expr_1$ is evaluated and returned, otherwise $expr_2$ is evaluated and returned.

Example:

```
(if true 10 20) ; Returns 10
(if false 10 20) ; Returns 20
(if (= 2 2)
  (begin (println "true")
         10)
  false) ; Prints "true", returns 10
```

`begin`
Syntax: `(begin` $expr_1$ `...` $expr_n$`)`
Description: Evaluates $expr_1$ through $expr_n$ in order, returns $expr_n$.
Example:

```
(begin (println "test") 2 3) ; Prints
    "test", returns 3
```

`quote`
Syntax: `(quote` $expr$`)`
Description: Return $expr$ as an unevaluated list or atom.
Example:

```
(quote (list 1 2 3)) ; Returns (list 1 2
    3)
(quote 2) ; Returns 2
```

E. Procedures

Procedures are named functions. All procedures have defined parameters and can return any atomic value. All procedures can refer to themselves using the `recur` symbol. This symbol permits procedures to call themselves recursively.

Naming Conventions:

- `proc-name`: Typical procedure name using hypen-snake-case.
- `predicate?`: Use with procedures that return a Boolean value.
- `make-foo`: Use for procedures that create a structured value.
- `foo->bar`: Use for procedures that convert values (Ex: number to string).

`inc`
Syntax: `(inc` n`)`
Description: Increase the value of n by one and return.
Example:

```
(inc 5) ; Returns 6
```

`dec`
Syntax: `(dec` n`)`
Description: Decrease the value of n by one and return.
Example:

```
(dec 5) ; Returns 4
```

`+`
Syntax: `(+` arg_1 `...` arg_n`)`
Description: Adds the provided arguments. Each argument must evaluate to a number. The result is the sum of all the arguments.
Example:

```
(+ 1 2 3) ; Returns 6
```

`-`
Syntax: `(-` arg_1 `...` arg_n`)`
Description: Subtracts the provided arguments. Each argument must evaluate to a number. The result is the difference of all the arguments.
Example:

```
(- 8 4 2) ; Returns 2
```

`*`
Syntax: `(*` arg_1 `...` arg_n`)`
Description: Multiply the provided arguments. Each argument must evaluate to a number. The result is the product of all the arguments.
Example:

```
(* 2 2 2) ; Returns 8
```

`/`
Syntax: `(/` arg_1 `...` arg_n`)`
Description: Divide the provided arguments. Each argument must evaluate to a number. The result is the combined quotient of all the arguments.
Example:

```
(/ 4 2) ; Returns 2
```

`mod`
Syntax: `(mod` $numer$ $denom$`)`
Description: Return the remainder of dividing $numer$ by $denom$. Numbers are floored to integers.
Example:

```
(mod 4 2) ; Returns 0
```

`list`
Syntax: `(list` arg_1 `...` arg_n`)`
Description: Create a list formed from arg_1 through arg_n.
Example:

```
(list 1 2 3 4 5) ; Returns (1 2 3 4 5)
```

`first`
Syntax: `(first` lst`)`
Description: Return the first element in the list lst.
Example:

```
(first (list 1 2 3 4 5)) ; Returns 1
```

`rest`
Syntax: `(rest` lst`)`
Description: Return the tail end of the list lst.
Example:

```
(rest (list 1 2 3 4 5)) ; Returns (2 3 4
    5)
```

`last`
Syntax: `(last` lst`)`
Description: Return the last element in the list lst.
Example:

```
(last (list 1 2 3 4 5)) ; Returns 5
```

`nth`
Syntax: `(nth` lst n`)`
Description: Returns the n^{th} element in the list lst (zero-based).
Example:

```
(nth (list 1 2 3 4 5) 4) ; Returns 5
```

`count`
Syntax: `(count` lst`)`
Description: Returns the number of elements in a list (single-depth).
Example:

```
(count (list 1 2 3 4 5)) ; Returns 5
```

`map`
Syntax: `(map` $proc$ lst`)`
Description: Return the elements in lst after applying the $proc$ procedure or function to each element.
$proc$ **Description:** ($proc$ $elem$) $\implies$ `Any value`
Example:

```
(map inc (list 1 2 3 4 5)) ; Returns (2 3
    4 5 6)
```

`filter`
Syntax: `(filter` $pred$ lst`)`
Description: Return a list of elements in lst that satisfy the predicate $pred$.
$pred$ **Description:** ($pred$ $elem$) $\implies$ `true/false`
Example:

```
(filter number? (list 1 "test" \A 2)) ;
    Returns (1 2)
```

`reduce`
Syntax: `(reduce` $proc$ $initial$ lst`)`
Description: Return the result of successively applying the $proc$ procedure to each element of lst. The $initial$ argument will set the first value used by the $prev$ variable in the $proc$ procedure.
$proc$ **Description**: ($proc$ $prev$ $next$) $\implies$ `result`
Example:

```
(reduce + 0 (list 1 2 3 4 5)) ; Returns 15
```

`range`
Syntax 1: `(range` end`)`
Syntax 2: `(range` $start$ end`)`
Syntax 3: `(range` $start$ end $step$`)`
Description: `1` returns the list of numbers starting at 0 upto end. `2` returns the list of numbers from $start$ upto end. `3` is indentical to `2` except that $step$ defines the rate at which numbers are added to the range.
Example:

```
(range 3) ; Returns (0 1 2)
(range 0 3) ; Returns (0 1 2)
(range 0 3 1) ; Returns (0 1 2)
```

`=`
Syntax: `(=` arg_1 `...` arg_n`)`
Description: Return `true` if all arguments are equal, `false` otherwise.
Example:

```
(= 2 2 2 2 2) ; Returns true
```

`not=`
Syntax: `(not=` arg_1 `...` arg_n`)`
Description: Inverse of =.

`<`
Syntax: `(<` arg_1 `...` arg_n`)`
Description: Return `true` if all arguments are less than their right neighbor, `false` otherwise.
Example:

```
(< 2 3 4 5) ; Returns true
```

`>`
Syntax: `(>` arg_1 `...` arg_n`)`
Description: Return `true` if all arguments are greater than their right neighbor, `false` otherwise.
Example:

```
(> 5 4 3 2) ; Returns true
```

`<=`
Syntax: `(<=` arg_1 `...` arg_n`)`
Description: Return `true` if all arguments are less than or equal to their right neighbor, `false` otherwise.
Example:

```
(<= 2 3 4 4) ; Returns true
```

`>=`
Syntax: `(>=` arg_1 `...` arg_n`)`
Description: Return `true` if all arguments are greater than or equal to their right neighbor, `false` otherwise.
Example:

```
(>= 4 4 3 2) ; Returns true
```

`symbol?`
Syntax: `(symbol?` sym`)`
Description: Return `true` if sym is a symbol, `false` otherwise.

`number?`
Syntax: `(number?` n`)`
Description: Return `true` if n is a number, `false` otherwise.

`boolean?`
Syntax: `(boolean?` b`)`
Description: Return `true` if b is a boolean, `false` otherwise.

`string?`
Syntax: `(string?` s`)`
Description: Return `true` if s is a string, `false` otherwise.

`char?`
Syntax: `(char?` ch`)`
Description: Return `true` if ch is a character, `false` otherwise.

`list?`
Syntax: `(list?` lst`)`
Description: Return `true` if lst is a list, `false` otherwise.

`print`
Syntax: `(print` arg_1 `...` arg_n`)`
Description: Print each argument to `stdout`.
Example:

```
(print "hello" 99 true) ; Prints "hello
    99 true"
```

`sprintf`
Syntax: `(sprintf` fmt arg_1 `...` arg_n`)`
Description: Format each arg_1 through arg_n using fmt as the format string. The format string syntax is directly used by Java's `String.format(`fmt`, ...)`. Returns the formatted string.
Format Specifiers Supported by Atom

- Number – %f for doubles or %d for integers
- Boolean – %b
- String – %s
- Character – %c
- Symbol – %s

Example:

```
(sprintf "%s %.2f %b" "test" 99.13 true)
    ; Prints "test 99.13 true"
```

`printf`
Syntax: `(print` fmt arg_1 `...` arg_n`)`
Description: Print each arg_1 through arg_n to `stdout` using fmt as the format string. `sprintf` is used in the implementation. Returns `nil`.
Example:

```
(printf "%s %.2f %b" "test" 99.13 true) ;
    Prints "test 99.13 true"
```

`println`
Syntax: `(println` arg_1 `...` arg_n`)`
Description: Same as `print`, but with an added newline.

`exit`
Syntax: `(exit [`$exitcode$`])`
Description: Exit the program with the optionally specified $exitcode$.