# Extend

# Language Reference Manual

Ishaan Kolluri(isk2108), Kevin Ye(ky2294) Jared Samet(jss2272), Nigel Schuster(ns3158)

December 10, 2016

# Contents

# 1. Introduction to Extend

Extend is a domain-specific programming language used to designate ranges of cells as reusable functions. It is a dynamically-typed, statically-scoped, declarative language that uses lazy evaluation to carry out computations. Once computed, all values are immutable. In order to offer the best performance, Extend compiles down to LLVM.

Extend's syntax is meant to provide clear punctuation and easily understandable cell range access specifications, while borrowing elements from languages with C-style syntax for ease of development. Despite these syntactic similarities, the semantics of an Extend program have more in common with a spreadsheet such as Microsoft Excel than imperative languages such as C, Java or Python.

# 2. Structure of an Extend Program

An Extend program consists of one or more source files. A source file can contain any number of import directives, function definitions, global variable declarations, and external library declarations, in any order.

## 2.1. Import Statements

Import statements in Extend are written with `import`, followed by the name of a file in double quotes, and terminated with a semicolon. The syntax is as follows:

```
import "string.xtnd";
```

Extend imports act like `#include` in C, except that multiple imports of the same file are ignored. The imports are all aggregated into a single namespace.

## 2.2. Function Definitions

Function definitions comprise the bulk of an Extend program. In short, a function consists of a set of variable declarations, formula assignments, and a return expression. Each variable consists of cells; the values of each cell are, if necessary, calculated according to formulas which each apply

to a specified subset of the cells. Each cell value, once calculated, is immutable. A couple examples follow for context; functions are described in detail in section 5.

```
isNumber(x) {
  return type(x) == "Number";
}

sum_column([m,1] rng) {
  /* Returns the sum of the values in the column, skipping any values that are non-
      numeric */
  [m,1] running_sum;
  running_sum[0,0] = #rng;
  running_sum[1:,0] = running_sum[[-1],] + (isNumber(#rng) ? #rng : 0);
  return running_sum[-1];
}
```

## 2.3. Global Variables

In essence, global variable declarations function as constants in Extend. They are written with the keyword `global`, followed by a variable declaration in the combined variable declaration and assignment format described in section 5.3.1. As with local variables, the cell values of a global variable, once computed, are immutable. A few examples follow:

```
global pi := 3.14159265359;
global num_points := 24;
global [num_points,1]
  circle_x_vals := cos(2 * pi * row() / num_points),
  circle_y_vals := sin(2 * pi * row() / num_points);
```

## 2.4. External Library Declarations

An external library is declared with the `extern` keyword, followed by the name of an object file in double quotes, followed by a semicolon-delimited list of external function declarations enclosed by curly braces. A library declaration informs the compiler of the functions' names and signatures and instructs the compiler to link the object file when producing an executable. An external function declared as `foo` will call an appropriately written C function `extend_foo`. An example follows:

```
extern "mylib.o" {
  foo(arg1, arg2);
  bar();
}
```

This declaration would cause the compiler to link `mylib.o` and would make the C functions `extend_foo` and `extend_bar` available to Extend programs as `foo` and `bar` respectively. The required signature and format of the external functions is specified precisely in section 5.7.

### 2.5. `main` function

When a compiled Extend program is executed, the `main` function is evaluated. All computations necessary to calculate the return value of the function are performed, after which the program terminates. The `main` function must be a function of a single argument, conventionally denoted `args`, which is guaranteed to be a 1-by-n range containing the command line arguments.

### 2.6. Scoping and Namespace

For functions and for global variables, there is a single namespace that is shared between all files composing an Extend program, and they are visible throughout the entire program. Functions declared in external libraries share this namespace as well. For a local variable, the scope is the entire body of the function in which it is defined. Functions may declare local variables sharing a name with a global variable; inside that function, the name will refer to the local variable.

```
isNumber(x) {
  return type(x) == "Number";
}

sum_column([m,1] rng) {
  /* Returns the sum of the values in the column, skipping any values that are non-
      numeric */
  [m,1] running_sum;
  running_sum[0,0] = #rng;
  running_sum[1:,0] = running_sum[[-1],] + (isNumber(#rng) ? #rng : 0);
  return running_sum[-1];
}
```

## 3. Types and Literals

Extend has three primitive data types, **Number**, **String**, and **Empty**, and one composite type, **Range**.

### 3.1. Primitive Data Types

A **Number** is an immutable primitive value corresponding to a double-precision 64-bit binary format IEEE 754 value. Numbers can be written in an Extend source file as either integer or floating point constants; both are represented internally as floating-point values. There is no separate type representing an integer.

A **String** is a immutable primitive value that is internally represented a C-style null-terminated byte array corresponding to ASCII values. A String can be written in an Extend source file as a sequence of characters enclosed in double quotes, with the usual escaping conventions. Extend does not allow for slicing of strings to access specific characters; access to the contents of a string will only be available through standard library functions.

The **Empty** type can be written as the keyword `empty`, and serves a similar function to `NULL` in SQL; it represents the absence of a value.

| **Primitive Data Types** | **Examples** |
|---|---|
| Number | `42 or -5 or 2.71828 or 314159e-5` |
| String | `"Hello, World!\n" or "foo" or ""` |
| Empty | `empty` |

## 3.2. Ranges

Extend has one composite type, **Range**. A range borrows conceptually from spreadsheets; it is a group of cells with two dimensions, described as rows and columns. Each cell is assigned a formula that either evaluates to a Number, a String, `empty`, or another Range. Cell formulas are described in detail in section 5.3. A range can either be declared as described in section 5.2 or with a range literal expression. Ranges can be nested arbitrarily deeply and can be used to represent (immutable) lists, matrices, or more complicated data structures.

### 3.2.1. Range Literals

A range literal is a semicolon-delimited list of rows, enclosed in curly brackets. Each row is a comma-delimited list of numbers, strings, or range literals. A few examples follow:

```
legal_ranges() {
  r1 := {"Don't"; "Panic"}; // two rows, one column
  r2 := {"Don't", "Think", "Twice"}; // one row, three columns
  r3 := {1,2,3;4,5,6;7,8,9}; // three rows, three columns
  r4 := {"Hello";0,1,2,3,4}; // two rows, five columns
  r5 := {{{{{1}}}}}; // one row, one column
  r7 := {-1.5,-2.5,{-2,"nested"},-3.5}; // one row, four columns
  return 0;
}
```

# 4. Expressions

Expressions in Extend allow for arithmetic and boolean operations, function calls, conditional branching, extraction of contents of other variables, string concatenation, and determination of the location of the cell containing the expression. The sections for boolean and conditional operators refer to truthy and falsey values: the `Number` 0 is the only falsey value; all other values are truthy. As `empty` represents the absence of a value, it is neither truthy nor falsey.

## 4.1. Arithmetic Operators

The arithmetic operators listed below take one or two expressions and return a number, if both expressions are Numbers, or `empty` otherwise. Operators grouped within the same inner box have the same level of precedence, and are listed from highest precedence to lowest precedence. All of the binary operators are infix operators, and, with the exception of exponentiation, are left-associative. Exponentiation, bitwise negation, and unary negation are right-associative. All of the unary operators are prefix operators. The bitwise operators round their operands to the nearest signed 32-bit integer (rounding half to even) before performing the operation and evaluate to a Number.

| Operator | Description | Definition |
|---|---|---|
| `~` | `Bitwise NOT` | Performs a bitwise negation on the binary representation of an expression. |
| `-` | `Unary negation` | A simple negative sign to negate expressions. |
| `**` | `Power` | Returns the first expression raised to the power of the second expression |
| `*` | `Multiplication` | Multiplies two expressions |
| `/` | `Division` | Divides first expression by second. |
| `%` | `Modulo` | Finds the remainder by dividing the expression on the left side of the modulo by the right side expression. |
| `«` | `Left Shift` | Performs a bitwise left shift on the binary representation of an expression. |
| `»` | `Right Shift` | Performs a sign-propagating bitwise right shift on the binary representation of an expression. |
| `&` | `Bitwise AND` | Performs a bitwise AND between two expressions. |
| `+` | `Addition` | Adds two expressions together. |
| `-` | `Subtraction` | Subtracts second expression from first. |
| `\|` | `Bitwise OR` | Performs a bitwise OR between two expressions. |
| `^` | `Bitwise XOR` | Performs a bitwise exclusive OR between two expressions. |

```
easy() {
        return 3 - (-3) ** 2 %5; //-1
}
g_eazy() {
        return (((1 << 2 | 1) << 2) | 1); //42
}
```

## 4.2. Boolean Operators

These operators take one or two expressions and evaluate to `empty`, 0 or 1. Operators grouped within the same inner box have the same level of precedence and are listed from highest precedence to lowest precedence. All of these operators besides logical negation are infix, left-associative operators. The logical AND and OR operators feature short-circuit evaluation. Logical NOT is a prefix, right-associative operator. For Strings, the boolean operators <, <=, >, and >= implement case-sensitive lexicographic comparison.

| Operator | Description | Definition |
|---|---|---|
| `!` | `Logical NOT` | Evaluates to 0 or 1 given a truthy or falsey value respectively. `!empty` evaluates to `empty`. |
| `==` | `Equals` | Always evaluates to 0 if the two expressions have different types. If both expressions are primitive values, evaluates to 1 if they have the same type and the same value, or 0 otherwise. If both expressions are ranges, evaluates to 1 if the two ranges have the same dimensions and each cell of the first expression == the corresponding cell of the second expression. `empty` == `empty` evaluates to 1. Strings are compared by value. |
| `!=` | `Not equals` | `x != y` is equivalent to `!(x == y)`. |
| `<` | `Less than` | If the expressions are both Numbers or both Strings and the first expression is less than the second, evaluates to 1. If the expressions are both Numbers or both Strings and the first expression is greater than or equal to the second, evaluates to 0. Otherwise, evaluates to `empty`. |
| `>` | `Greater than` | Equivalent rules about typing as for <. |
| `<=` | `Less than or equal to` | Equivalent rules about typing as for <. |
| `>=` | `Greater than or equal to` | Equivalent rules about typing as for <. |
| `&&` | `Short-circuit Logical AND` | If the first expression is falsey or `empty`, evaluates to 0 or `empty` respectively. Otherwise, if the second expression is truthy, falsey, or `empty`, evaluates to 1, 0, or `empty` respectively. |
| `\|\|` | `Short-circuit Logical OR` | If the first expression is truthy or `empty`, evaluates to 1 or `empty` respectively. Otherwise, if the second expression is truthy, falsey, or `empty`, evaluates to 1, 0, or `empty` respectively. |

```
somethings_false() {
        return !1 != !1 || 4 <= 3;
}
somethings_true() {
        return empty || empty <= !3 || 5 > 3;
}
```

## 4.3. Conditional Expressions

There are two types of conditional expressions: a simple ternary if-then-else expression and a `switch` expression which can represent more complex logic.

### 4.3.1. Ternary Expressions

A ternary expression, written either as `cond-expr ? expr-if-true : expr-if-false` or, equivalently, `if(cond-expr, expr-if-true, expr-if-false)` evaluates to `expr-if-true` if `cond-expr` is truthy, or `expr-if-false` if `cond-expr` is falsey. If `cond-expr` is `empty`, the expression evaluates to `empty`. Both expr-if-true and expr-if-false are mandatory. `expr-if-true` is only evaluated if `cond-expr` is truthy, and `expr-if-false` is only evaluated if `cond-expr` is falsey. If `cond-expr` is `empty`, neither expression is evaluated.

### 4.3.2. Switch Expressions

A `switch` expression takes a optional condition, and a list of cases and expressions that the overall expression should evaluate to if the case applies. In the event that multiple cases are true, the expression of the first matching case encountered will be evaluated. An example is provided below:

```
switch_example(foo) {
  return switch (foo) {
        case 2: "foo is 2";
        case 3,4: "foo is 3 or 4";
        default: "none of the above";
  };
}

alternate_format(foo) {
  return switch {
        case foo == 2:
                "foo is 2";
        case foo == 3, foo == 4:
                "foo is 3 or 4";
        default:
                "none of the above";
  };
}
```

The format for a `switch` statement is the keyword `switch`, optionally followed by pair of parentheses containing an expression `switch-expr`, followed by a list of case clauses enclosed in curly braces and delimited by semicolons. A case clause consists of the keyword `case` followed by a comma-separated list of expressions `case-expr1 [, case-expr2, [...]]`, a colon, and an expression `match-expr`, or the keyword `default`, a colon, and an expression `default-expr`. If `switch-expr` is omitted, the `switch` expression evaluates to the `match-expr` for the first case where one of the `case-exprs` is truthy, or `default-expr` if none of the `case-exprs` apply. If `switch-expr` is present, the `switch` expression evaluates to the `match-expr` for the first case where one of the `case-exprs` is equal (with equality defined as for the == operator) to `switch-expr`, or `default-expr` if none of the `case-exprs` apply.

The `switch` expression can be used to compactly represent what in most imperative languages would require a long string such as `if (cond1) {...} else if (cond2) {...}`. The `switch` operator is internally converted to an equivalent (possibly nested) ternary expression; as a result, it features short-circuit evaluation throughout.

## 4.4. Additional Operators

There are four additional operators available to determine the size and type of other expressions. In addition, the infix + operator is overloaded to perform string concatenation.

| Operator | Description | Definition |
|---|---|---|
| `size(expr)` | `Dimensions` | Evaluates to a Range consisting of one row and two columns; the first cell contains the number of rows of `expr` and the second contains the number of columns. If `expr` is a Number, a String, or Empty, both cells will contain 1. |
| `type(expr)` | `Value Type` | Evaluates to "Number", "String", "Range", or "Empty". |
| `row()` | `Row Location` | No arguments; returns the row of the cell that is being calculated |
| `column()` | `Column Location` | No arguments; returns the column of the cell that is being calculated |
| `+` | `String concatenation` | "Hello, " + "World!\n" == "Hello, World!\n" |

Given `foo[5,5]`, then `foo[2,5] = row() * 2 + col()` will evaluate to 9.

## 4.5. Function Calls

A function expression consists of an identifier and an optional list of expressions enclosed in parentheses and separated by commas. The value of the expression is the result of applying the function to the arguments passed in as expressions. For more detail, see section 5.

## 4.6. Range Expressions

Range expressions are used to select part or all of a range. A range expression consists of a bare identifier, a bare range literal, or an expression and a selector. If a range expression has exactly 1 row and 1 column, the value of the expression is the value of the single cell of the range. If it has more than 1 row or more than 1 column, the value of the expression is the selected range. If the range has zero or fewer rows or zero or fewer columns, the value of the expression is `empty`. If a range expression with a selector would access a row index or column index greater than the number of rows or columns of the range, or a negative row or column index, the value of the expression is `empty`.

### 4.6.1. Slices

A slice consists of an optional integer literal or expression `start`, a colon, and an optional integer literal or expression `end`, or a single integer literal or expression `index`. If `start` is omitted, it defaults to 0. If `end` is omitted, it defaults to the length of the dimension. A single `index` with no colon is equivalent to `index:index+1`. Enclosing `start` or `end` in square brackets is equivalent to the expression `row() + start` or `row() + end`, for a row slice, or `column() + start` or `column() + end` for a column slice. The slice includes `start` and excludes `end`, so the length of a slice is `end - start`. A negative value is interpreted as the length of the dimension minus the value. As mentioned above, the value of a range that is not 1 by 1 is a range, but the value of a 1 by 1 range is essentially dereferenced to the result of the cell formula.

### 4.6.2. Selections

A selection expression consists of an expression and a pair of slices separated by a comma and enclosed in square brackets, i.e. [row_slice, column_slice]. If one of the dimensions of the range has length 1, the comma and the slice for that dimension can be omitted. If the comma is present but a slice is omitted, that slice defaults to [0] for a slice corresponding to a dimension of length greater than one, or 0 for a slice corresponding to a dimension of length one.

### 4.6.3. Corresponding Cell

A very common selection to make is the cell in the "corresponding location" of a different variable. Since this case is so common, #var is syntactic sugar for var[,]. As a result, if var has more than column and more than one row, #var is equivalent to var[row(),column()]. If var has multiple rows and one column, it is equivalent to var[row(),0]. If var has one row and multiple columns, it is equivalent to var[0,column()]; and if var has one row and one column, it is equal to var[0,0].

### 4.6.4. Selection Examples

```
selection_examples() {
  foo[0,2] /* This evaluates to the cell value in the first row and third column. */
  foo[0,:] /* Evaluates to the range of cells in the first row of foo. */
  foo[:,2] /* Evaluates to the range of cells in the third column of foo. */
  foo[:,[1]] /* The internal brackets denote RELATIVE notation.
  In this case, 1 column right of the column of the left-hand-side cell. */

  foo[3,] /* Equivalent to foo[3,[0]] if foo has more than one column
  or foo[3,0] if foo has one column */

  foo[5:, 7:] /* All cells starting from the 6th row and 8th column to the bottom right
      */

  foo[[1]:[2], 0:[7]]
  /* Selects the rows between the 1st and 2nd row after LHS row, and
     all the columns up to the 7th column to the right of the LHS column */

  /* In this example, each cell of bar would be equal to the cell
   * in foo in the equivalent location plus 1. */
  [5,5] foo;
  [5,5] bar := #foo + 1; // #foo = foo[[0],[0]]

  /* In this example, bar would be a 3x5 range where in each row,
   * the value in bar is equal to the value in foo in the same column.
   * In other words, each row of bar would be a copy of foo. */
  [1,5] foo; // foo has 1 row, 5 columns
  [3,5] bar := #foo; // #foo = foo[0,[0]]

  /* In this example, the values of baz would be
   * 11, 12, 13 in the first row;
   * 21, 22, 23 in the second row;
   * 31, 32, 33 in the third row. */
```

```
  foo := {1,2,3}; // 1 row, 3 columns
  bar := {10;20;30}; // 3 rows, 1 column
  [3,3] baz := #foo + #bar; // Equivalent to foo[0,[0]] + bar[[0],0]
}
```

### 4.7. Precedence Expressions

A precedence expression is used to force the evaluation of one expression before another, when that order of operation is required for functions with side-effects. It consists of an expression `prec-expr`, the precedence operator `->`, and an expression `succ-expr`. The value of the expression is `succ-expr`, but the value of `prec-expr` will be calculated first and the result ignored. All functions written purely in Extend are free of side effects. However, some of the external functions provided by the standard library, such as for file I/O and plotting, do have side effects.

## 5. Functions

The bulk of an Extend program consists of functions. Although Extend has some features, such as immutability and lazy evaluation, that are inspired by functional languages, its functions are not *first class objects*. By default, the standard library is automatically compiled and linked with a program, but there are no functions built into the language itself.

### 5.1. Format

As in most programming languages, the header of the function declares the parameters it accepts. The body of the function consists of an optional set of variable declarations and formula assignments, which can occur in any order, and a return statement, which must be the last statement in the function body. All variable declarations and formula assignments, in addition to the return statement, must be terminated by a semicolon. This very simple function returns whatever value is passed into it:

```
foo(arg) {
  return arg;
}
```

## 5.2. Variable Declarations

A variable declaration associates an identifier with a range of cells of the specified dimensions, which are listed in square brackets before the identifier. For convenience, if the square brackets and dimensions are omitted, the identifier will be associated with a single cell. In addition, multiple identifiers, separated by commas, can be listed after the dimensions; all of these identifiers will be separate ranges, but with equal dimension sizes. The dimensions can be specified as any valid expression that evaluates to a Number, which will be rounded to the nearest signed 32-bit integer. If either dimension is zero or negative, or if the expression does not evaluate to a Number, a runtime error causing the program to halt will occur.

```
[2, 5] foo; // Declares foo as a range with 2 rows and 5 columns
[m, n] bar; // Declares bar as a range with m rows and n columns
[3, 3] ham, eggs, spam; // Declares ham, eggs and spam as distinct 3x3 ranges
baz; // Declares baz as a single cell
```

## 5.3. Formula Assignment

A formula assignment assigns an expression to a subset of the cells of a variable. Unlike most imperative languages, this expression is not immediately evaluated, but is instead only evaluated if and when it is needed to calculate the return value of the function. A formula assignment consists of an identifier, an optional pair of slices enclosed in square brackets specifying the subset of the cells that the assignment applies to, an =, and an expression, followed by a semicolon. As with the expressions specifying the dimensions of a range, these slices specifying the cell subset can contain arbitrary expressions, as long as the expression taken as a whole evaluates to a Number, which will be rounded to the nearest signed 32-bit integer. Negative numbers are legal in these slices, and correspond to (dimension length + value).

```
[5, 2] foo, bar, baz; // Declares foo, bar, and baz as distinct 5x2 ranges
foo[0,0] = 42; // Assigns the expression 42 to the first cell of the first row of foo
foo[0,1] = foo[0,0] * 2; // Assigns (foo[0,0] * 2) to the 2nd cell of the 1st row of foo
bar = 3.14159; // Assigns pi to every cell of every row of bar
baz[1:-1,0:1] = 2.71828; // Assigns e to cells (1,0) through (3,1), inclusive, of baz

/* The next line assigns foo[[-1],0] + 2 to every cell in
   both columns of foo, besides the first row */
foo[1:,:] = foo[[-1],0] + 2;
```

The last line of the source snippet above demonstrates the idiomatic Extend way of simulating an imperative language's loop; foo[4,0] would evaluate to 42+2+2+2+2 = 50 and foo[4,1] would

evaluate to (42*2)+2+2+2+2 = 92.

### 5.3.1. Combined Variable Declaration and Formula Assignment

For convenience, a variable declaration and a formula assignment to all cells of that variable can be combined on a single line by inserting a `:=` and an expression after the identifier. Multiple variables and assignments, separated by commas, can be declared on a single line as well. All global variables must be defined using the combined declaration and formula assignment syntax.

```
/* Creates two 2x2 ranges; every cell of foo evaluates to 1 and every cell of
   bar evaluates to 2. */
[2,2] foo := 1, bar := 2;
```

### 5.3.2. Formula Assignment Errors

If the developer writes code in such a way that more than one formula applies to a cell, a runtime error will occur if the cell's value is required to compute the return expression. If there is no formula assigned to a cell, the cell will evaluate to `empty`.

## 5.4. Parameter Declarations

Parameters can be declared with or without dimensions. If dimensions are declared, they can either be specified as integer literals or as identifiers. If a dimension is specified as an integer literal, the program will verify the dimension of the argument before beginning to evaluate the return expression; if it does not match, a runtime error will occur causing the program to halt. If it is specified as an identifier, that variable will contain the dimension size and will be available inside the function body. If the same identifier is repeated in the function declaration, the program will verify that every parameter dimension with that identifier has equal dimension size; if they differ, a runtime error will occur causing the program to halt. A few examples follow:

```
number_of_cells([m,n] arg) {
  return m*n; // m and n are initialized with the dimensions of arg
}

die_unless_primitive([1,1] arg) {
  return 0; // If arg is not a primitive value, a runtime error will occur
}

num_cells_if_column_vector([m,1] arg) {
  // If arg has one column, return number of cells; otherwise runtime error
  return m;
}
```

```
die_unless_square([m,m] arg) {
  return 0; // Runtime error if number of rows != number of columns
}

num_cells_if_same_size([m,n] arg1, [m,n] arg2) {
  // If arguments are the same size, return # of cells, otherwise runtime error
  return m*n;
}
```

## 5.5. Application on Ranges

Extend gives the developer the power to easily apply operations in a functional style on ranges.

For example, the following function performs cell wise addition:

```
foo([m,n] arg1, [m,n] arg2) {
  [m,n] bar := #arg1 + #arg2;
  return bar;
}
```

This function normalizes a column vector to have unit norm:

```
normalize_column_vector([m,1] arg) {
  [m,1] squared_lengths := #arg * #arg, normalized := #arg / vector_norm;
  vector_norm := sqrt(sum(squared_lengths));
  return normalized;
}
```

## 5.6. Lazy Evaluation and Circular References

All cell values and variable dimensions are evaluated lazily if and when they are needed to calculate the return expression. Using lazy evaluation ensures that the cell values are calculated in a valid topological sort order and allows for detection of circular references; internally this is accomplished by constructing a function for each formula which is called the first time the cell's value is needed, and marking the cell as "in-progress" once it starts being evaluated and as "complete" once the value has been calculated. The only guarantees the language places on the order of cell evaluation are: (1) It will be a valid topological ordering; (2) In conditional expressions and in short-circuiting operator expressions, only the relevant conditional branches will be evaluated; and (3) In an expression using the precedence operator, the preceeding expression will be evaluated before the succeeding expression. A range selection consisting of multiple cells will not cause the constituent cells to be evaluated; however, selection of a single cell will cause that cell's value to be evaluated. If a program is written in such a way as to cause a circular dependency of one cell on another, and the return expression is dependent on that cell's value, a

runtime error will occur. For example, in the following function:

```
maybeCircular(truth_value) {
  x := x;
  return truth_value ? x : 0;
}
```

A runtime error will occur if maybeCircular(1) is called; but if maybeCircular(0) is called, the function will simply return 0.

### 5.7. External Libraries

Using the following library declaration:

```
extern "mylib.o" {
  foo(arg1, arg2);
  bar();
}
```

will make the functions foo (taking two arguments) and bar (taking zero arguments) available within Extend. In LLVM, the compiler will declare external functions extend_foo and extend_bar as functions of two and zero arguments respectively. All arguments must have the type value_p, and the function must have return type value_p, declared in the Extend standard library header file. In other words, the C file compiled to generate the library must have defined:

```
value_p extend_foo(value_p arg1, value_p arg2) {
  /* function body here; */
}

value_p extend_bar() {
  /* function body here; */
}
```

## 6. Standard Library Reference

## 7. Example Program

```
import "./samples/stdlib.xtnd";

main([1,n] args) {
  /* Get a working copy */
  return 0;
}
```