Extend

Language Reference Manual

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

December 6, 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, global variable declarations, function definitions, 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. 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 same form as a variable declaration within a function as described in section 5.2. As with local variables, the value of a global variable, once computed, is 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);
```

2.3. Function Definitions

Function definitions are described in detail in section 5.

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 7.2.

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 contains a formula that either evaluates to a Number, a String, 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 allows 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.

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 bitwise right shift on the binary representation of an expression.
&	`Bitwise AND`	Performs a bitwise AND between two expressions. If both expressions have a 1 at the same digit, the resultant expression will have a 1 there; otherwise, it is 0.
+	`Addition`	Adds two expressions together.
-	`Subtraction`	Subtracts second expression from first.
|	`Bitwise OR`	Performs a bitwise OR between two expressions. If at least one of the expressions has a one at the same digit, the resultant expression will have a 1 there; otherwise, it is 0.
^	`Bitwise XOR`	Performs a bitwise exclusive OR between two expressions. If exactly one of the expressions has a one in the same digit, the resultant expression will have a 1 there; otherwise it is 0.

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.

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.
`!=`	`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 first, evaluates to 1. If the expressions are both Numbers or both Strings and the first expression is greater than or equal to the first, evaluates to 0. Otherwise, evaluates to empty.
`>`	`Greater than`	Evaluates to 1 if the first expression is greater than the first; equivalent rules about typing as for <.
`<=`	`Less than or equals to`	Evaluates to 1 if the first expression is less than or equal to the second; equivalent rules about typing as for <.
`>=`	`Greater than or equals to`	Evaluates to 1 if the first expression is less than or equal to the second; 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.

4.3. Conditional Operators

The conditional operators can be used to create expressions that evaluate to a specified expression based on the truthiness of a specified expression. There are two types of conditional expressions: a simple if-then-else expression written using the ternary operator **? :** and `switch` expressions which can represent more complex logic.

4.3.1. Ternary Expressions

A ternary expression, written as `cond-expr ? expr-if-true : expr-if-false` works like an if-else conditional statement. It evaluates to `expr-if-true` if `cond-expr` is numeric and truthy, or `expr-if-false` if `cond-expr` is numeric and 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:

```
[1,1] foo := 3;
return switch (foo) {
        case 2: "foo is 2";
        case 3,4: "foo is 3 or 4";
        default: "none of the above";
}

/* Equivalently: */
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 value 1 is assumed. The `switch` expression evaluates to the `match-expr` for the first case where one of the `case-exprs` is truthy, if `switch-expr` is omitted, or equal (with equality defined as for the == operator) to `switch-expr`, if `switch-expr` is present, 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 (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 containing the formula
`column()`	`Column Location`	No arguments; returns the column of the cell containing the formula
`+`	`String concatenation`	"Hello, " + "World!\n" == "Hello, World!\n"

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 formula 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`]. It can also be written as the hash symbol # and an expression. As mentioned earlier, the composite **range** type has the ability to slice in both an absolute and relative fashion. 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. #expr is syntactic sugar for expr[,]; for a range with more than column and more than one row, it denotes the position in the RHS expression corresponding to the position of cell on the left-hand-side of the formula assignment. For a range with only one column, it denotes the single cell in the same row as the cell on the left-hand-side of the assignment. For a range with only one row, it denotes the single cell in the same column as the cell on the left-hand-side of the assignment. These possibilities mean that there are multiple ways to slice ranges, which are illustrated below.

```
foo[1,2] /* This evaluates to the cell value in the second row and third column. */
foo[1,:] /* Evaluates to the range of cells in the second 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[1,] /* Equivalent to foo[1,[0]] if foo has more than one column
or foo[1,0] 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], [5:7]]
/* Selects the rows between the 1st and 2nd row after LHS row, and
   between 5th and 7th column from LHS column */

#foo
/* If foo has >=2 rows and >= 2 columns, equivalent to foo[[0],[0]];
   If foo has 1 row and multiple columns, equivalent to foo[0,[0]]; etc. */
```

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. The only functions with side effects in Extend are the built-in file I/O functions described in section 6.3 or user-defined functions that call those built-in functions; an example is located in that section.

5. Functions

Functions lie at Extend's core; however, they are not *first class objects.* Since it can be verbose to write certain operations in Extend, the language will feature a small number of built-in functions and and a comprehensive standard library. An important set of built-in functions will handle I/O (see section 6.3). Besides the built-in file I/O functions, all functions in Extend are free of side effects.

5.1. Format

Every function in Extend follows the same format, but allows some optional declarations. As in most programming languages, the header of the function declares the parameters it accepts and the dimensions of the return value. 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:

```
[1,1] foo([1,1] arg) {
  return arg;
}
```

The leading `[1,1]` marks the return dimensions. `foo` is the function name. In parentheses the function arguments are declared, again with dimensions of the input. The body of the function follows, which in this case is only the return statement.

5.2. Variable Declaration

A variable declaration associates an identifier with a range 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 1x1 range, and if only a single dimension is listed instead of two, the identifier will be associated with a range consisting of one row and the specified number of columns. 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 either as literal integers or as expressions that evaluate to integers.

```
[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
baz; // Declares baz as a 1x1 range
[10] ham, eggs, spam; // Declares ham, eggs and spam as distinct 1x10 ranges
```

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. The slices specifying the cell subset can contain arbitrary expressions, as long as the expression taken as a whole evaluates to an integer.

```
[5, 2] foo, bar;
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

/* The next line assigns foo[[-1],0] + 1 to every cell in
   both columns of foo, besides the first row */
foo[1:,0:1] = 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. Although this may appear wasteful, intermediate values can be garbage collected once they are no longer needed to calculate the function's return value.

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.

```
/* 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, this causes a compile-time error if the compiler can detect it or a runtime error if the compiler cannot detect it in advance and the cell is evaluated. If there is no formula assigned to a cell, the cell will evaluate to `empty`.

5.4. Dimension Assignment

Extend will feature gradual typing for function declarations. This will enable users with a weak experience in typing to use the language, while allowing more sophisticated developers to enforce type checking at compile time. In addition, it allows the developer to return ranges whose size is an unpredictable or complex function of the inputs.

To avoid specifying the precise return dimensions, an underscore can be used. This marks a variable range. Thus our function now looks like this:

```
[_,1] foo([5,5] arg1, [1,1] arg2) {
  return arg1[0:arg2 ,0];
}
```

Here we are selecting a range from arg1 that depends on the value of arg2 and can therefore not be known ahead of time.

5.5. Parameter Declarations

If a parameter is declared with an identifier for the dimensions, instead of an integer literal, that identifier will contain the dimension size of the argument inside the function. In addition, expressions consisting solely of other identifiers are allowed, and will cause a run-time error if the sizes of the arguments are not consistent.

However Extend will feature even more options to specify ranges. If a certain operation should be applied to a range of numbers of unknown size, the size can be inferred at runtime and match the return size:

```
[m,1] foo([m,1] arg) {
  return arg[0:m, 0] + 1;
}
```

This function will add 1 to each element in arg. Notice, that `m` is used across the function as a variable identifier to apply the operation to the range.

Summarizing, we have 3 ways of specifying a return range:

Type	Symbol Example	Description
Number	3	A number is the simplest descriptor. It specifies the absolute return size
Expression	bar * 2	An expression that can be anything, ranging from a simple arithmetic operation to a function call. To use this, any identifier used, must also be present as a range descriptor in a function parameter.
Underscore	_	This marker is unique, since it is a wildcard. While the other options aim to be specific, the underscore circumvents declaring the range size.

5.6. Application on Ranges

Extend gives the developer the power to easily apply operations in a functional style on ranges. As outlined in the section above, there are various ways to apply functions to ranges. A feature unique to Extend is the powerful operation on values and ranges. To apply a function on a per cell basis, the corresponding variable needs to be preceded by "#". The following function applies cell wise addition:

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

While both function above result in the same value, and only show the syntactical difference. If we wanted each cell to to be the square root divided by the sum of the input we have the following:

```
[m] foo([m] arg) {
  [m] bar := sqrt(#arg) / sum(arg);
  return bar;
}
```

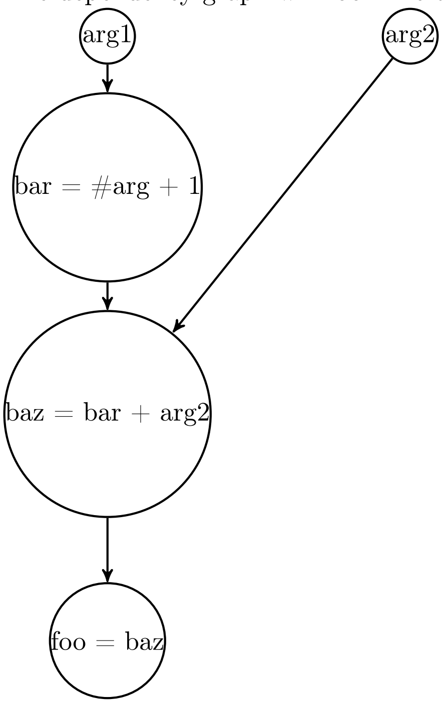
Notice that `arg` is only once preceded by `#`.

5.7. Dependencies Illustrated

The dependency resolution is another asset that sets Extend apart from other languages. Most languages compile ordinarily and execute the given commands sequentially. Extend builds a dependency graph. The advantage of this is that only relevant code segments will be executed. Given the function

```
[m,n] foo([m,n] arg1, [m,n] arg2) {
  [m,n] bar := #arg1 + 1;
  [m,n] faz := #arg1 + 3;
  [m,n] baz := bar + arg2;
  return baz;
}
```

The dependency graph will look like this:



Notice that `faz` does not appear in the graph, because it is not relevant for the return value. Ultimately this graph enables Extend to find the leaves, evaluate code paths in the best configuration and even in parallel.

6. Built-In Functions

There are a small number of built-in functions that allow operations that would otherwise be impossible to provide through user-defined functions. These are `isEmpty()`, since comparing `empty` with any other value simply returns `empty`; `size()`, `row()`, and `column()`, to determine the dimensions of a range or the location of a cell within a range; the file I/O functions `open()`, `close()`, `read()`, and `write()`, which have side effects; and the serialization and deserialization functions `toString()` and `fromString()`.

6.1. isEmpty

Since `empty` cannot be compared to any other value using the boolean operators, the built-in function `isEmpty(expr)` can be used to determine whether the supplied expression evaluates to `empty`. It returns 1 if the supplied argument is empty and 0 otherwise.

6.2. Dimension and Position Functions

The built-in function `size(expr)` returns a 1x2 range containing the number of rows and columns, in that order, of the value of that expression. `size(empty)` returns `{0, 0}`. The built-in functions `row()` and `column()` return the row index or column index of the cell in which they are evaluated. Examples include:

```
/* The 5x5 identity matrix */
[5,5] id := row() == column() ? 1 : 0;

/* A 1x10 range in which the first 5 cells evaluate to "left"
   and the next 5 cells evaluate to "right" */
[1,10] left_half := column() < 5 ? "left" : "right";
```

6.3. File I/O

Although the anticipated use cases of Extend generally do not include highly interactive programs, the language has built-in functions that allow the developer to read from and write to files, including standard input, output, and error. These functions are the only part of the language with side effects; as a result, the dependencies between expressions referencing the file I/O functions should be carefully analyzed by the developer to ensure that the program behaves as intended. The precedence operator -> can be used to create an artificial dependency between expressions to enforce the correct order of evaluation.

6.3.1. File Pointers

The built-in `open` and `close` functions open and close file pointers for reading and writing. An attempt to open a nonexistent file, or a file that the user does not have permission to read and write, will result in a runtime error causing the program to halt, as will an attempt to close a file pointer that is not open. The return value of `open` is a range that can be supplied as the file pointer argument to `close`, `read`, or `write`. The return value of `close` is `empty`. The built-in variables `STDIN`, `STDOUT`, and `STDERR` refer to file pointers that do not need to be opened or closed.

6.3.2. Reading and Writing

The built-in `read`, `readline`, and `write` functions read from and write to an open file pointer. `read` takes a maximum number of bytes and a file pointer as arguments and returns a 1-by-n range, where n is the lesser of the number of bytes actually read and the maximum number of bytes requested. If the maximum number of bytes requested is `empty`, the entire contents of the file are returned. `readline` takes a file pointer as argument and returns a 1-by-n range, where n is the number of bytes between the current position of the file pointer and the first newline encountered or EOF, whichever occurs first. The newline, if present, is included in the returned range. The arguments to `write` are a 1-by-n range and a file pointer and the return value is `empty`.

6.3.3. Example using the precedence operator

```
main(args) {
  return 0;
}

bmi() {
  STDIN := 0;
  STDOUT := 1;
  q1 := write("What is your height in inches?\n", STDOUT);
  height := q1 -> parseFloat(readline(STDIN));
  q2 := height -> write("What is your weight in pounds?\n", STDOUT);
  weight := q2 -> parseFloat(readline(STDIN));
  return weight * 703 / height ** 2;
}
```

6.4. Serialization and Deserialization

The built in functions `toString(expr)` and `fromString(s)` will serialize an expression to a string and vice versa. If expr is a range, `toString()` will evaluate the value of every cell in the range, proceeding from left to right within a row and from top to bottom within the range, and will produce a string that could be used as a range literal in a source file. `fromString()` will do the reverse. Note that these functions do not comprise an eval() function; toString() will only have numbers in its result, and fromString() will not deserialize a string containing anything besides literal values. They are provided mainly for convenience in loading and parsing complex datasets. It is possible that these two functions will be provided as part of the standard library rather than as built-in functions.

7. Entry Point, External Libraries, Scoping and Namespace

7.1. `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. If the function declaration includes parameters, the first argument will be a 1-by-n range containing the command line arguments. Any other parameters, if declared, will evaluate to `empty`.

7.2. External Libraries

7.3. Scoping and Namespace

8. Example Program

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

main([1,n] args) {
/*  seqFP := open(args[0]);
  seq1 := readline(seqFP)[:-2]; // discard newline
  seq2 := seq1 -> readline(seqFP)[:-2]; */
```

```
  [10,1] seq1;
  [10,1] seq2;
  seq1[0,0] = "A";
  seq1[1,0] = "A";
  seq1[2,0] = "B";
  seq1[3,0] = "B";
  seq1[4,0] = "C";
  seq1[5,0] = "C";
  seq1[6,0] = "D";
  seq1[7,0] = "D";
  seq1[8,0] = "E";
  seq1[9,0] = "E";
  seq2[0,0] = "A";
  seq2[1,0] = "A";
  seq2[2,0] = "B";
  seq2[3,0] = "C";
  seq2[4,0] = "C";
  seq2[5,0] = "D";
  seq2[6,0] = "D";
  seq2[7,0] = "E";
  seq2[8,0] = "E";
  seq2[9,0] = "F";
  alignment := computeSequenceAlignment(seq1, seq2, 1, -1, -3);
/*  output := write(alignment[:,0], STDOUT) ->
            write(alignment[:,1], STDOUT) ->
            close(seqFP); */
  return
    printf(1, toString(alignment[:,0]) + "\n") ->
    printf(1, toString(alignment[:,1]) + "\n") ->
    0;
}

[_,2] computeSequenceAlignment([m,1] seq1, [n,1] seq2,
  matchReward, mismatchPenalty, gapPenalty) {

  [m, n] scoreFromMatch, scoreFromLeft, scoreFromTop;
  [m, n] path, step;
  seq2T := transpose(seq2);
  [m+1,n+1] score;

  score[0, 0] = 0;
  score[1:,0] = score[[-1],] + gapPenalty;
  score[0,1:] = score[,[-1]] + gapPenalty;
  score[1:,1:] = nmax(scoreFromMatch[[-1],[-1]],
      nmax(scoreFromLeft[[-1],[-1]], scoreFromTop[[-1],[-1]]));

  scoreFromMatch = #score + ((#seq1 == #seq2T) ? matchReward : mismatchPenalty);
  scoreFromLeft = score[[1],] + gapPenalty;
  scoreFromTop = score[,[1]] + gapPenalty;

  step = (#scoreFromMatch >= #scoreFromLeft) ?
      ((#scoreFromMatch >= #scoreFromTop) ? "D" : "T") :
      ((#scoreFromLeft  >= #scoreFromTop) ? "L" : "T");

  path[-1,-1] = 1;
  path[-1,:-1] = (step[,[1]] == "L" && isNumber(path[,[1]])) ? 1 + path[,[1]] : empty;
  path[:-1,-1] = (step[[1],] == "T" && isNumber(path[[1],])) ? 1 + path[[1],] : empty;
  path[:-1,:-1] = switch {
    case step[[1],[1]] == "D" && isNumber(path[[1],[1]]):
      1 + path[[1],[1]];
    case step[,[1]] == "L" && isNumber(path[,[1]]):
      1 + path[,[1]];
    case step[[1],] == "T" && isNumber(path[[1],]):
      1 + path[[1],];
  };

  pathLen := path[0,0];
  [m, 1] seq1Positions := pathLen - max(path[,:]);
  [1, n] seq2PositionsT := pathLen - max(path[:,]);
  seq2Positions := transpose(seq2PositionsT);
  [pathLen, 1] seq1Loc := match(seq1Positions, row());
  [pathLen, 1] seq2Loc := match(seq2Positions, row());

  [pathLen, 2] results;
```

```
  results[:,0] = seq1[#seq1Loc];
  results[:,1] = seq2[#seq2Loc];

  return
    printf(1, "seq1:\n" + toString(seq1) + "\n\n") ->
    printf(1, "seq2:\n" + toString(seq2) + "\n\n") ->
    printf(1, "seq2T:\n" + toString(seq2T) + "\n\n") ->
    printf(1, "scoreFromMatch:\n" + toString(scoreFromMatch) + "\n\n") ->
    printf(1, "scoreFromLeft:\n" + toString(scoreFromLeft) + "\n\n") ->
    printf(1, "scoreFromTop:\n" + toString(scoreFromTop) + "\n\n") ->
    printf(1, "score:\n" + toString(score) + "\n\n") ->
    printf(1, "step:\n" + toString(step) + "\n\n") ->
    printf(1, "path:\n" + toString(path) + "\n\n") ->
    printf(1, "pathLen:\n" + toString(pathLen) + "\n\n") ->
    printf(1, "seq1Positions:\n" + toString(seq1Positions) + "\n\n") ->
    printf(1, "seq2Positions:\n" + toString(seq2Positions) + "\n\n") ->
    printf(1, "seq1Loc:\n" + toString(seq1Loc) + "\n\n") ->
    printf(1, "seq2Loc:\n" + toString(seq2Loc) + "\n\n") ->


    results;
}
```