Extend

Language Reference Manual

Ishaan Kolluri, Kevin Ye, Jared Samet, Nigel Schuster

October 26, 2016

Contents

1. Introduction to Extend

Extend is a domain-specific programming language used to designate ranges of cells as reusable functions. It abstracts dependencies between cells and models a dependency graph during compilation. In order to offer great performance for any size of datasets, Extend compiles down to LLVM.

Extend's syntax is meant to provide clear punctuation and easily understandable cell range access specifications, while maintaining the look of modern functional programming languages. Given Extend's functionality resonates well with spreadsheets, it borrows syntactical elements from programs such as Microsoft Excel.

2. Structure of an Extend Program

Extend is predominantly composed of function declarations. In order to run the program, the **main** function will be executed. To illustrate the scope of the language, the OCaml grammar is attached below:

```
/* Ocamlyacc parser for Extend */

%{
open Ast
%}

%token LSQBRACK RSQBRACK LPAREN RPAREN LBRACE RBRACE HASH
%token COLON COMMA QUESTION GETS ASN SEMI PRECEDES UNDERSCORE
%token SWITCH CASE DEFAULT
%token PLUS MINUS TIMES DIVIDE MOD POWER LSHIFT RSHIFT
%token EQ NOTEQ GT LT GTEQ LTEQ
%token LOGNOT LOGAND LOGOR
%token BITNOT BITXOR BITAND BITOR
%token EMPTY RETURN IMPORT GLOBAL
%token <int> LIT_INT
%token <float> LIT_FLOAT
%token <string> LIT_STRING
%token <string> ID
%token EOF

%right QUESTION
%left PRECEDES
%left LOGOR
%left LOGAND
%left EQ NOTEQ LT GT LTEQ GTEQ
%left PLUS MINUS BITOR BITXOR
%left TIMES DIVIDE MOD LSHIFT RSHIFT BITAND
%right POWER
%right BITNOT LOGNOT NEG
%left HASH LSQBRACK

%start program
%type <Ast.program> program

%%

program:
    imports globals func_decls EOF { (List.rev $1, List.rev $2, List.rev $3) }

imports:
    /* nothing */ {[]}
  | imports import {$2 :: $1}

import:
    IMPORT LIT_STRING SEMI {$2}

globals:
    /* nothing */ {[]}
  | globals global {$2 :: $1}

global:
    GLOBAL vardecl {$2}

func_decls:
    /* nothing */ {[]}
  | func_decls func_decl {$2 :: $1}

func_decl:
    ID LPAREN func_param_list RPAREN LBRACE opt_stmt_list ret_stmt RBRACE
    { {
      name = $1;
      params = $3;
      body = $6;
      ret_val = ((None, None), $7)
    } }
  | ret_dim ID LPAREN func_param_list RPAREN LBRACE opt_stmt_list ret_stmt RBRACE
    { {
      name = $2;
      params = $4;
      body = $7;
      ret_val = ($1, $8);
    } }

opt_stmt_list:
```

```
    /* nothing */ { [] }
  | stmt_list { List.rev $1 }

stmt_list:
    stmt { [$1] }
  | stmt_list stmt { $2 :: $1 }

stmt:
    vardecl { $1 } |  assign { $1 }

ret_stmt:
    RETURN expr SEMI {$2}

vardecl:
    var_list SEMI { Vardecl((None, None), List.rev $1) }
  | dim var_list SEMI { Vardecl($1, List.rev $2) }

var_list:
    ID varassign { [ ($1, $2)] }
  | var_list COMMA ID varassign { ($3, $4) :: $1}

varassign:
    /* nothing */ { None }
  | GETS expr { Some $2 }

assign:
    ID lhs_sel ASN expr SEMI { Assign($1, $2, Some $4) }

expr:
    expr rhs_sel        { Selection($1, $2) }
  | HASH expr           { Selection($2, (None, None)) }
  | op_expr             { $1 }
  | ternary_expr        { $1 }
  | switch_expr         { $1 }
  | func_expr           { $1 }
  | range_expr          { $1 }
  | expr PRECEDES expr  { Precedence($1, $3) }
  | LPAREN expr RPAREN  { $2 }
  | ID                  { Id($1) }
  | LIT_INT             { LitInt($1) }
  | LIT_FLOAT           { LitFlt($1) }
  | LIT_STRING          { LitString($1) }
  | EMPTY               { Empty }

op_expr:
    expr PLUS expr      { BinOp($1, Plus, $3) }
  | expr MINUS expr     { BinOp($1, Minus, $3) }
  | expr TIMES expr     { BinOp($1, Times, $3) }
  | expr DIVIDE expr    { BinOp($1, Divide, $3) }
  | expr MOD expr       { BinOp($1, Mod, $3) }
  | expr POWER expr     { BinOp($1, Pow, $3) }
  | expr LSHIFT expr    { BinOp($1, LShift, $3) }
  | expr RSHIFT expr    { BinOp($1, RShift, $3) }
  | expr LOGAND expr    { BinOp($1, LogAnd, $3) }
  | expr LOGOR expr     { BinOp($1, LogOr, $3) }
  | expr BITXOR expr    { BinOp($1, BitXor, $3) }
  | expr BITAND expr    { BinOp($1, BitAnd, $3) }
  | expr BITOR expr     { BinOp($1, BitOr, $3) }
  | expr EQ expr        { BinOp($1, Eq, $3) }
  | expr NOTEQ expr     { BinOp($1, NotEq, $3) }
  | expr GT expr        { BinOp($1, Gt, $3) }
  | expr LT expr        { BinOp($1, Lt, $3) }
  | expr GTEQ expr      { BinOp($1, GtEq, $3) }
  | expr LTEQ expr      { BinOp($1, LtEq, $3) }
  | MINUS expr %prec NEG  { UnOp(Neg, $2) }
  | LOGNOT expr           { UnOp(LogNot, $2) }
  | BITNOT expr           { UnOp(BitNot, $2) }

ternary_expr:
  /* commented out optional part for now */
    expr QUESTION expr COLON expr %prec QUESTION { Ternary($1, $3, $5) }

switch_expr:
    SWITCH LPAREN switch_cond RPAREN LBRACE case_list RBRACE { Switch($3, List.rev $6) }
```

```
switch_cond:
    /* nothing */ { None }
  | expr { Some $1 }

case_list:
    case_stmt { [$1] }
  | case_list case_stmt { $2 :: $1 }

case_stmt:
    DEFAULT COLON expr SEMI { (None, $3) }
  | CASE case_expr_list COLON expr SEMI { (Some (List.rev $2), $4) }

case_expr_list:
    expr { [$1] }
  | case_expr_list COMMA expr { $3 :: $1 }

func_expr:
    ID LPAREN opt_arg_list RPAREN { Call($1, $3) }

range_expr:
    LBRACE row_list RBRACE { LitRange(List.rev $2) }

row_list:
    col_list {[List.rev $1]}
  | row_list SEMI col_list {$3 :: $1}

col_list:
    expr {[$1]}
  | col_list COMMA expr {$3 :: $1}

opt_arg_list:
    /* nothing */ {[]}
  | arg_list { List.rev $1 }

arg_list:
    expr {[$1]}
  | arg_list COMMA expr {$3 :: $1}

lhs_sel:
    /* nothing */                          { (None, None) }
  | LSQBRACK lslice RSQBRACK               { (Some $2, None) }
  | LSQBRACK lslice COMMA lslice RSQBRACK { (Some $2, Some $4) }

rhs_sel:
    LSQBRACK rslice RSQBRACK               { (Some $2, None) }
  | LSQBRACK rslice COMMA rslice RSQBRACK { (Some $2, Some $4) }

lslice:
    /* nothing */                          { (None, None) }
  | lslice_val                             { (Some $1, None) }
  | lslice_val COLON lslice_val            { (Some $1, Some $3) }
  | lslice_val COLON                       { (Some $1, Some DimensionEnd) }
  | COLON lslice_val                       { (Some DimensionStart, Some $2) }
  | COLON                                  { (Some DimensionStart, Some DimensionEnd) }

rslice:
    /* nothing */                          { (None, None) }
  | rslice_val                             { (Some $1, None) }
  | rslice_val COLON rslice_val            { (Some $1, Some $3) }
  | rslice_val COLON                       { (Some $1, Some DimensionEnd) }
  | COLON rslice_val                       { (Some DimensionStart, Some $2) }
  | COLON                                  { (Some DimensionStart, Some DimensionEnd) }

lslice_val:
    expr { Abs($1) }

rslice_val:
    expr { Abs($1) }
  | LSQBRACK expr RSQBRACK { Rel($2) }

func_param_list:
    /* nothing */ { [] }
  | func_param_int_list { List.rev $1 }
```

```
func_param_int_list:
    func_sin_param { [$1] }
  | func_param_int_list COMMA func_sin_param { $3 :: $1 }

func_sin_param:
    ID { ((None, None), $1) }
  | dim ID { ($1, $2) }

dim:
    LSQBRACK expr RSQBRACK { (Some $2, None) }
  | LSQBRACK expr COMMA expr RSQBRACK { (Some $2, Some $4) }

ret_dim:
    LSQBRACK ret_sin RSQBRACK { ($2, None) }
  | LSQBRACK ret_sin COMMA ret_sin RSQBRACK { ($2,$4) }

ret_sin:
    expr { Some $1 }
  | UNDERSCORE { Some Wild }
```

3. Types and Literals

3.1. Primitive Data Types

Extend has two primitive data types, **numbers** and **ranges**. In the vein of Javascript, numbers are essentially 64-bit floating point numbers in Extend. Ranges are elaborated on in the next section. The user can choose to represent numbers in the following different ways:

Char

A **char** literal is essentially a size 1 numerical range. At evaluation, the number in the range will be compared with its ASCII equivalent.

String

A **string** literal is a range of numbers of size n, where n is the length of the string. The string 'hello' can be represented internally as [104, 101, 108, 108, 111].

Integer

A **integer** can be represented as a size 1 numerical range as well. However, it retains its numerical value upon evaluation.

Float

A **float**, like Javascript numbers, can be represented as 64 bit, where the fraction is stored in bits 52 to 62.

Below is a snippet illustrating programmatic declarations for each of the above types.

```
/* Integer */
num = 5;
/* Char */
chr = 'A'
/* String */
str = 'Hello'
/* Float */
num = 1.5;
```

3.2. Ranges

Ranges are a data type unique to the Extend language. It borrows conceptually from spreadsheets; a range is a group of cells with dimensions represented as rows and columns. Each range is either one or two-dimensional. A range is composed of cells, and cells are comprised of functions that can have dependencies on the values of other cells. A range is written as follows:

```
/* This is a left-handed range, used to assign a value. */
[1,2]foo; /*Range with 1 row and 2 columns */
```

3.2.1. Range Slicing

Extend somewhat mimics Python in its range slicing syntax; however, it offers the ability to slice a range in both absolute and relative terms.

```
foo[1,2] /* This evaluates to the cell value at row 1, column 2. */
foo[1,] /* Evaluates to the range of cells in row 1. */
foo[,2] /* Evaluates to the range of cells in column 2.*/
foo[,[1]] /* The internal brackets denote RELATIVE notation.
In this case, 1 column right of the one currently being operated on. */
foo[5:, 7:] /* 5th row down, and 7th column from the absolute origin.
foo[[1:2], [5:7]]
/* Selects the rows between the 1st and 2nd row from current row */
/* Selects the columns between 5th and 7th column from current column */
```

3.2.2. The Underscore Symbol

The underscore(_) symbol allows the dimension of the range to have an unspecified size. For example, in function signatures, using the underscore allows the return value to have various possible dimensions. An example is illustrated below:

```
[1,_]foo;
/* A range with 1 row and an unknown, variable number of columns. */
```

4. Expressions

Expressions in Extend allows for arithmetic and boolean operations, function calls, conditional branching, and extraction of contents of other variables. In Section 2 "Structure of an Extend Program," it shows the OCAML grammar for Extend and exactly what an expression can be under "expr:". In this section, Extend expressions are broken down into the following 5 categories: Operators, Booleans, Variable Declaration, Variable Assignment, and Conditionals.

4.1. Operators

Extend has three different operator types: **Multiplication**, **Addition**, and **Unary**. The precedence of the operators are listed in order from the highest starting in section 4.1.1. and each subsequent subsection is of lower precedence than the previous subsection.

4.1.1. Multiplication, Quotient, and Modulo

The multiplication, division, and modulo operators are of equal and highest precedence when it comes to operations. The expressions for all three operators are arithmetic types.

Multiplication

The binary '*' operator computes the **multiplication** of two expressions.

```
Syntax:
expr * expr
```

Quotient

The binary '/' operator computes the **quotient** of two expressions. The quotient operator will perform the quotient as a decimal operation. This means that if you assign both expressions are of type integer, then the result will be an integer. For example, the quotient of 5/2 will compute to 2 and not 2.5. This can be solved by making one or both expressions to type float. Another thing to note is that if the second expression or divisor is 0, then it will yield an undefined value.

```
Syntax:
expr / expr
```

Modulo

The binary '%' operator computes the **modulo** of two expressions. The modulo operator finds the remainder from dividing the expression to the left of the modulo symbol, the dividend, with the expression to the right of the modulo symbol, the divisor. For the modulo operator to return a correct value, the expression to the left of the modulo symbol or dividend must be greater than 0.

```
Syntax:
expr % expr
```

4.1.2. Addition and Subtraction

The Addition and Subtraction operators are of equal and second highest precedence after Multiplication, Division, and Modulo when it comes to operations. The expressions for all three operators are also arithmetic types.

Addition

The binary operator '+' computes the addition of two expressions.

```
Syntax:
expr + expr
```

Subtraction

The binary operator '-' computes the subtraction of two expressions.

```
Syntax:
expr - expr
```

4.1.3. Unary

The unary operators operate on a single expression. Extend uses three unary operators: Minus, Logical Complement, and Bitwise Complement.

Minus

The unary Minus operator '-' can be used to negate numbers of type integer, decimal and floating-point.

```
Syntax:
-expr
```

Logical Complement

The unary Logical Complement Operator '!', also known as negation, can be used to negate a boolean type.

```
Syntax:
!expr
```

Bitwise Complement

The unary Bitwise Complement Operator '~' can be used to to get the bitwise complement of an expression. This operator can only be used on an integral type.

```
Syntax:
~expr
```

4.2. Booleans

Extend supports boolean comparisons between two expressions. The operators that compare two expressions and return a boolean value are Equal To, Not Equal To, Greater Than, Less Than, Greater Than Or Equal To, Less Than Or Equal To, And, and Or.

```
/* this checks if the two expressions are equal     */
expr == expr
/* this checks if the two expressions are not equal */
expr != expr
/* this checks if the first expression is greater   */
/* than the second expression                       */
expr > expr
/* this checks if the first expression is less than */
/* the second expression                            */
expr < expr
/* this checks if the first expression is greater   */
/* than or equal to the second expression           */
expr >= expr
/* this checks if the first expression is less than */
/* or equal to the second expression                */
expr <= expr
/* this checks if the first expression is true AND  */
/* if the second is true                            */
expr && expr
/* this checks if the first expression is true OR  */
/* if the second is true                            */
expr || expr
```

4.3. Conditionals

Conditionals statements and conditional expressions allow you to control your program to do different actions in different situations. Extend uses the operators Switch, Case, and Default to do this. A Switch statement evaluates an expression, checks each Case statement for a match and executes the statements in that Case statement. If none of these cases match, then it will execute the Default case. An example of this is shown below:

```
[1,1] foo = 3;
return switch {
        case foo = 2:
                "foo is 2";
        case foo = 3:
```

```
                "foo is 3";
        case foo = 4:
                "foo is 4";
        default:
                "foo is none of the cases"
}
```

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 6.4). 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;
```

5.4. 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;
}
```

If we want to apply a function to the whole range at once we drop the leading symbol. Thus matrix addition takes the following shape:

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

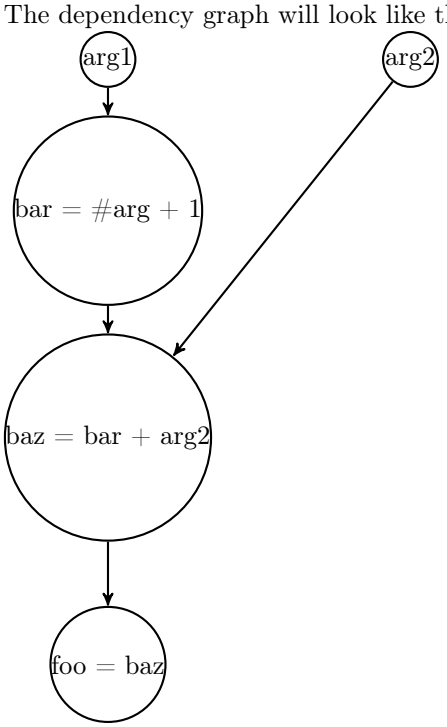
Notice that `arg` is only once preceded by `#`.

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

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. 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.

6.4. 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.

6.4.1. 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.

6.4.2. 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`.

6.4.3. Example using the precedence operator

```
bmi() {
  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;
}
```

7. Entry Point

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`.

8. Example Program

```
main([1,n] args) {
  seqFP := open(args[0]);
  seq1 := readline(seqFP)[:-2]; // discard newline
  seq2 := seq1 -> readline(seqFP)[:-2];
  alignment := computeSequenceAlignment(seq1, seq2, 1, -1, -3);
  output := write(alignment[:,0], STDOUT) ->
            write(alignment[:,1], STDOUT) ->
            close(seqFP);
  return output;
}

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

  [m, n] scoreFromMatch, scoreFromLeft, scoreFromTop;
  [m, n] step, path;
  [1,n] 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) ? DDD : TTT) :
      ((#scoreFromLeft  >= #scoreFromTop) ? LLL : TTT);
```

```
  path[-1,-1] = 1;
  path[-1,:-1] = (step[,[1]] == LLL && !isEmpty(path[,[1]])) ? 1 + path[,[1]] : empty;
  path[:-1,-1] = (step[[1],] == TTT && !isEmpty(path[[1],])) ? 1 + path[[1],] : empty;
  path[:-1,:-1] = switch () {
    case step[[1],[1]] == DDD && !isEmpty(path[[1],[1]]):
      1 + path[[1],[1]];
    case step[,[1]] == LLL && !isEmpty(path[,[1]]):
      1 + path[,[1]];
    case step[[1],] == TTT && !isEmpty(path[[1],]):
      1 + path[[1],];
  };

  pathLen := path[0,0];
  [m, 1] seq1Positions := pathLen - rmax(path[,:]);
  [1, n] seq2PositionsT := pathLen - rmax(path[:,]);
  [n, 1] seq2Positions := transpose(seq2PositionsT);
  [pathLen, 1] resIdx := colRange(0, pathLength);
  [pathLen, 1] seq1Loc := match(resIdx, seq1Positions);
  [pathLen, 1] seq2Loc := match(resIdx, seq2Positions);

  [pathLength, 2] results;
  results[:,0] = seq1[seq1Loc];
  results[:,1] = seq2[seq2Loc];

  return results;
}
```