# The Flix Programming Language

## Syntax

### Literals

### Control

|  |
| --- |
| if (exp1) exp2 else exp3 |
|  |

### Definitons and Declarations

### Rules

## The Core Datatypes

## Opt

An option Opt is either None or Some(v). *Options cannot be nested[[1]](#footnote-1).*

|  |
| --- |
| null(o: Opt[A]): Bool  Returns true iff o is None. |
| get(o: Opt[A]): A [Unsafe]  Returns v if o is Some(v). Otherwise the semantics is undefined. |
| getWithDefault(o: Opt[A], a: A): A  Returns v if o is Some(v). Otherwise returns a. |
| exists(f: A => Bool, o: Opt[A]): Bool  Returns the value of f(v) if o is Some(v). Otherwise false. |
| forall(f: A => Bool, o: Opt[A]): Bool  Returns the value of f(v) if o is Some(v). Otherwise true. |
| filter(f: A => Bool, o: Opt[A]): Opt[A]  Returns o if o is Some(v) and f(v) is true. Otherwise returns None. |
| map(f: A => B, o: Opt[A]): Opt[B]  Returns Some(f(v)) if o is Some(v). Otherwise returns None. |
| map2(f: (A, B) => C, o1: Opt[A], o2: Opt[B]): Opt[C]  Returns Some(f(v1, v2)) if o1 is Some(v1) and Some(v2).  Otherwise returns None. |
| flatMap(f: A => Opt[B], o: Opt[A]): Opt[B]  Returns f(v) if o is Some(v). Otherwise returns None. |
| flatMap2(f: (A, B) => Opt[C],  o1: Opt[A], o2: Opt[A]): Opt[C]  Returns f(v1, v2) if o1 is Some(v1) and o2 is Some(v2).  Otherwise returns None. |
| toList(o: Opt[A]): List[A]  Returns a one-element list of the value v if o is Some(v).  Otherwise returns the empty list. |
| toSet(o: Opt[A]): Set[A]  Returns a one-element set of the value v if o is Some(v).  Otherwise returns the empty set. |
| withDefault(o1: Opt[A], o2: Opt[A]): Opt[A]  Returns o1 if it is Some(v) otherwise returns o2. |

## List

A list is either the empty list Nil or a cons cell v :: vs.

### Basic List Operations

|  |
| --- |
| Nil: List[A]  The empty list. |
| v :: vs: List[A]  The list with the element v followed by the list vs. |
| null(xs: List[A]): Bool  Returns true iff xs is the empty list, i.e. Nil. |
| head(xs: List[A]): A [Unsafe]  Returns the first element of the list.  If the list is empty the semantics are undefined. |
| tail(xs: List[A]): List[A] [Unsafe]  Returns the list consisting of everything but the first element of the list.  If the list is empty the semantics are undefined. |
| init(xs: List[A]): List[A] [Unsafe]  Returns the list with all the elements except the last.  If the list is empty the semantics are undefined. |
| last(xs: List[A]): A [Unsafe]  Returns the last element of the list.  If the list is empty the semantics are undefined. |
| length(xs: List[A]): Int  Returns the length of the list. |
| append(xs: List[A], ys: List[A]): List[A]  Returns the length of the list. |
| at(position: Int, xs: List[A]): A  Returns the element at position in the list. |
| memberOf(x: A, xs: List[A]): Bool  Returns true if the list xs contains the element x. |
| indexOf(x: A, xs: List[A]): Int  Returns the position of the element x in the list xs.  If the element does not exist in the list the semantics are undefined. |
| find(f: A => Bool, xs: List[A]): Opt[A]  Alias for findLeft. |
| findLeft(f: A => Bool, xs: List[A]): Opt[A]  Optionally returns the first element that satisfies the predicate f. |
| findRight(f: A => Bool, xs: List[A]): Opt[A]  Similiar to findLeft but searches from right to left. |

### List Building

|  |
| --- |
| range(b: Int, e: Int): List[Int]  Returns a list of all integers between b (inclusive) and e (exclusive). |
| repeat(x: A, n: Int): List[A]  Returns a list with the element x repeated n times. |
| scanLeft(f: (B, A) => B, b: B, xs: List[A]): List[B]  Accumulates the result of applying the function f going left to right.  That is, the result is of the form:  b :: f(b, x1) :: f(f(b, x1), x2) ... |
| scanRight(f: (A, B) => B, b: B, xs: List[A]): List[B]  Similar to scanLeft but going right to left. |

### List Transformations

|  |
| --- |
| map(f: A => B, xs: List[A]): List[B]  Returns the result of applying the function f to every element in the list. |
| mapWithIndex(f: (A, Int) => B, xs: List[A]): List[B]  Similar to map but additionally passes the index of every element to f. |
| flatMap(f: A => List[B], xs: List[A]): List[B]  Returns the result of applying the function f to every element in the list and concatenating the result. Aliased as concatMap. |
| reverse(xs: List[A]): List[A]  Returns the reversed list. |
| rotateLeft(n: Int, xs: List[A]): List[A]  Rotates the elements of the list n positions to left. That is, returns a new list where the first n elements in the original list are the last n elements of the new list. |
| rotateRight(n: Int, xs: List[A]): List[A]  Similiar to rotateLeft, but rotates to the right. |
| replace(i: Int, x: A, xs: List[A]): List[A]  Replaces the element at position i with the element x in the list xs.  If the position i does not exist in the list the semantics are undefined. |
| patch(i: Int, n: Int,  xs: List[A], ys: List[A]): List[A]  Replaces n elements at position i in ys with elements from xs. |
| permutations(xs: List[A]): List[List[A]]  Returns all permutations of the list. |
| subsequences(xs: List[A]): List[List[A]]  Returns all subsequences of the list. |
| intersperse(x: A, xs: List[A]): List[A]  Returns the list with the element x inserted between every element. |
| intercalate(xs: List[A], ys: List[List[A]]): List[A]  Returns the concatenation of the list ys with the list xs inserted between every element. |
| transpose(xs: List[List[A]]): List[List[A]] [Unsafe]  Returns the transpose of the list.  If the dimensions of the lists are mismatched the semantics are undefined. |

### List Predicates

|  |
| --- |
| isPrefixOf(xs: List[A], ys: List[B]): Bool  Returns true if the list xs is a prefix of the list ys. |
| isInfixOf(xs: List[A], ys: List[B]): Bool  Returns true if the list xs is an infix of the list ys. |
| isSuffixOf(xs: List[A], ys: List[B]): Bool  Returns true if the list xs is a suffix of the list ys. |

### Fold And Reduce

|  |
| --- |
| fold(f: (B, A) => B, b: B, xs: List[A]): B  Alias of foldLeft. |
| foldLeft(f: (B, A) => B, b: B, xs: List[A]): B  Left-associative fold of the list xs. |
| foldRight(f: (A, B) => B, b: B, xs: List[A]): B  Right-associative fold of the list xs. |
| reduceLeft(f: (A, A) => A, xs: List[A]): A [Unsafe]  Left-associative reduce of the list xs.  If the list is empty the semantics are undefined. |
| reduceLeftOpt(f: (A, A) => A, xs: List[A]): Opt[A]  Safe variant of reduceLeft that returns None if the list is empty. |
| reduceRight(f: (A, A) => A, xs: List[A]): A [Unsafe]  Right-associative reduce of the list xs.  If the list is empty the semantics are undefined. |
| reduceRightOpt(f: (A, A) => A, xs: List[A]): Opt[A]  Safe variant of reduceRight that returns None if the list is empty. |

### Special Folds

|  |
| --- |
| count(f: A => Bool, xs: List[A]): Int  Returns the number of elements in the list that satisfy the predicate f. |
| concat(xs: List[List[A]]): List[A]  Returns the concatenation of the lists in the list xs. |
| exists(f: A => Bool, xs: List[A]): Bool  Returns true iff at least one element in the list satisfies the predicate f.  Returns false if the list is empty. |
| forall(f: A => Bool, xs: List[A]): Bool  Returns true iff every element in the list satisfies the predicate f.  Returns true if the list is empty. |
| and(xs: List[Bool]): Bool  Returns true iff every element in the list is true.  Returns true if the list is empty. |
| or(xs: List[Bool]): Bool  Returns true iff at least one element in the list is true.  Returns false if the list is empty. |

### Sub Lists

|  |
| --- |
| filter(f: A => Bool, xs: List[A]): List[A]  Returns a list of every element in the list xs that satisfy the predicate f. |
| slice(b: Int, e: Int, xs: List[A]): List[A]  Returns the sublist from position b (inclusive) until position e (exclusive). |
| partition(f: A => Bool,  xs: List[A]): (List[A], List[A])  Returns a pair of lists (ys, zs) where are the elements in xs are those that satisfy the predicate f and the elements in zs are those that do not. |
| span(f: A => Bool, xs: List[A]): (List[A], List[A])  Returns a pair of lists (ys, zs) where ys is the longest prefix of xs that satisfies the predicate f and zs is the remainder of xs. Functionally equivalent to (takeWhile(f, xs), dropWhile(f, xs)). |
| drop(n: Int, xs: List[A]): List[A]  Returns the list except for the first n elements.  Returns the empty list if n is larger than the length of the list. |
| dropWhile(f: A => Bool, xs: List[A]): List[A]  Returns the list except for the longest prefix that satisfy the predicate f. |
| take(n: Int, xs: List[A]): List[A]  Returns the first n elements of xs.  Returns the entire list if n is larger than the length of the list. |
| takeWhile(f: A => Bool, xs: List[A]): List[A]  Returns the longest prefix of xs that satisfy the predicate f. |

### Aggregation and Sorting

|  |
| --- |
| sum(xs: List[Int]): Int  Returns the sum of the elements in the list.  Returns zero if the list is empty. |
| product(xs: List[Int]): Int  Returns the product of the elements in the list.  Returns one if the list is empty. |
| min(xs: List[Int]): Int [Unsafe]  Returns the minimum element of a non-empty list.  If the list is empty the semantics are undefined. |
| max(xs: List[Int]): Int [Unsafe]  Returns the maximum element of a non-empty list.  If the list is empty the semantics are undefined. |
| minBy(f: (A, A) => Bool, xs: List[Int]): Int [Unsafe]  Returns the minimum element of a non-empty list according the total order defined by the predicate f. That is, if f(x, y) == true then x is smaller or equal to y. Two elements are equal if f(x, y) = f(y, x).  If the list is empty the semantics are undefined. |
| maxBy(f: (A, A) => Bool, xs: List[Int]): Int [Unsafe]  Similar to minBy but returns the largest element. |
| sort(xs: List[Int]): Int  Returns the elements of the list sorted according to the natural order. |
| sortBy(f: (A, A) => Bool, xs: List[Int]): Int  Returns the elements of the list sorted according to the total order defined by the predicate f. That is, if f(x, y) == true then x is smaller or equal to y. Two elements are equal if f(x, y) = f(y, x). |
| groupBy(f: (A, A) => Bool, xs: List[A]): List[List[A]]  Partitions the list xs into sublists such that for any two elements x and y in a sub list f(x, y) is true. |

### Zipping and Unzipping

|  |
| --- |
| zip(xs: List[A], ys: List[B]): List[(A, B)]  Returns a list where each element at index i is the pair (a, b) where a is the element at position i in xs and b is the element at position i in ys. |
| zipWith(f: (A, B) => C,  xs: List[A], ys: List[B]): List[C]  Like zip but uses the function f to compute each element in the result.  Functionally equivalent to: map(f, zip(xs, ys)). |
| unzip(xs: List[(A, B)]): (List[A], List[B])  Returns a pair where the first component is a list of all first components in xs and the second component is a list of all second components in xs. |

### Two List Operations

|  |
| --- |
| map2(f: (A, B) => C, xs: List[A], ys: List[B]): List[C]  Pairwise applies the function f to the elements of the lists xs and ys. Functionally equivalent to zipWith. |
| flatMap2(f: (A, B) => List[C],  xs: List[A], ys: List[B]): List[C]  Pairwise applies the function f to the elements of the lists xs and ys and concatenates the result in one list. Functionally equivalent to flatten(map2(f, xs, ys)). |
| foldLeft2(f: (C, A, B) => C, c: C,  xs: List[A], ys: List[B]): C  Accumulates the result of applying the function f pairwise to the elements of xs and ys starting with the initial value c going from left to right. Functionally similar to foldLeft(f, c, zip(xs, ys)). |
| foldRight2(f: (A, B, C) => C, c: C,  xs: List[A], ys: List[B]): C  Similar to foldLeft2 but goes from the right to the left. |

### Combined Operations

|  |
| --- |
| concatMap(f: A => List[B], xs: List[A]): List[B]  Alias for flatMap. |
| filterMap(f: A => Opt[B], xs: List[A]): List[B]  Collects the result of applying the partial function f to each element of the list. |
| findMap(f: A => Opt[B], xs: List[A]): Opt[B]  Collects the first result of applying the partial function f to each element of the list. |

### Conversion Operations

|  |
| --- |
| toMap(xs: List[(A, B)]): Map[A, B]  Returns the association list as a map. If the list contains multiple values for the same key, the value appearing latest in the list is used. |
| toSet(xs: List[A]): Set[A]  Returns the list as a set. |

### Order and Lattice Operations

|  |
| --- |
| leq(xs: List[A], ys: List[A]): Bool  Returns true if every element in xs is pairwise smaller or equal to its corresponding element in ys. That is, for any position i in xs, it must be the case that at(i, xs) is smaller or equal to at(i, ys) (if it exists) according to the partial order on A. Functionally equivalent to to zipWith(leq, xs, ys).and where leq is the partial order on A. |
| isAscChain(xs: List[A]): Bool  Returns true iff the elements in xs form an ascending chain. That is, if  according to the partial order on A.  NB: In general, isAscChain is *not* the negation of isDescChain since a list may fail to be a chain due to two elements being incomparable. |
| isDescChain(xs: List[A]): Bool  Returns true iff the elements in xs form a descending chain. That is, if  according to the partial order on A.  NB: In general, isDescChain is *not* the negation of isAscChain since a list may fail to be a chain due to two elements being incomparable. |
| join(xs: List[A]): A  Returns the least upper bound of all element in xs. If the list is empty, returns the bottom element of A. Functionally equivalent to foldLeft(lub, bot, xs) where lub and bot are the least upper bound and bottom element of A, respectively. |
| meet(xs: List[A]): A  Similiar to lub, but with the greatest lower bound and top element. |
| widen(xs: List[A]): A  Similiar to lub, but with the widening operator. |
| narrow(xs: List[A]): A  Similiar to lub, but with the narrowing operator. |
| zipWithJoin(xs: List[A], ys: List[A]): A  Returns the pairwise least upper bound of the two lists xs and ys. Functionally equivalent to zipWith(lub, xs, ys) where lub is the least upper bound of A. |
| zipWithMeet(xs: List[A], ys: List[A]): A  Similiar to zipWithJoin, but with the greatest lower bound. |

## Set

### Basic Operations

|  |
| --- |
| null(s: Set[A]): Bool  Returns true iff s is the empty set. |

## Map

### Basic Operations

|  |
| --- |
| null(m: Map[K, V]): Bool  Returns true iff m is the empty map. |
| get(k: K, m: Map[K, V]): Opt[V]  Optionally returns the value the key k is mapped to in the map m.  Returns None if m has no mapping for k. |
| getWithDefault(k: K, v: V, m: Map[K, V]): V  Returns the value the key k is mapped to in the map m. Returns v if the key has no mapping. |
| memberOf(k: K, m: Map[K, V]): Bool  Returns true iff k is key of the map m. |
|  |

### Insert, Update and Delete Operations

|  |
| --- |
| insert(k: K, v: V, m: Map[K, V]): Map[K, V]  Returns the map m updated with the key k mapped to the value v. |
| insertWith(f: (V, V) => V, k: K,  v: V, m: Map[K, V]): Map[K, V]  Returns the map m updated with the key k mapped to the value v. If the key already exists in the map and is mapped to v2, then the map is updated with the value of f(v, v2). |
| insertWithKey(f: (K, V, V) => V, k: K,  v: V, m: Map[K, V]): Map[K, V]  Returns the map m updated with the key k mapped to the value v. If the key already exists in the map and is mapped to v2, then the map is updated with the value of f(k, v, v2). |
| update(f: V => Opt[V], k: K, m: Map[K, V]): Map[K, V]  Returns ??? |

## Debug

The following *compiler directives* may aid debugging of Flix programs.

*Note: The Flix compiler and runtime is free to ignore these directives.*

|  |
| --- |
| abort!(m: String): Unit  Immediately aborts execution and prints the given message m. |
| print!(a: A): A  Prints the value a and returns it. |
| time!(f: () => A): A  Evaluates the function f while measuring its execution time.  Returns the result of evlauating the function. |

1. This ensures efficient representation of options. [↑](#footnote-ref-1)