# The Flix Programming Language

## Syntax

### Control

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

## 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 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. |
| range(Int, Int)  Returns a list |
| repeat(A, Int) |
| permutations(List[A]): List  fdsfds |
|  |
| map(f: A => B, xs: List[A]): List[B]  Foo |
| flatMap(f: A => List[B], xs: List[A]): List[B] |
|  |
|  |
|  |

### Index Operations

### Sub Lists

|  |
| --- |
| filter(f: A => Bool, xs: List[A]): List[A]  Returns a list of every element x in xs for which f(x) is true. |
| slice(b: Int, e: Int, xs: List[A]): List[A]  Returns the sublist from position b (inclusive) until position e (exclusive). |
| take(n: Int, xs: List[A]): List[A]  Returns the first n elements of xs. |
| takeWhile(f: A => Bool, xs: List[A]): List[A]  Returns the longest prefix of xs for which the predicate f holds. |
| drop(n: Int, xs: List[A]): List[A]  fdsffs |
| dropWhile(f: A => Bool, xs: List[A]): List[A]  fsfa |

### 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. |

### 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)